



Databázové systémy

IDS

Studijní opora

doc. Ing. Jaroslav Zendulka
Ing. Ivana Rudolflová

Verze: 18. 7. 2006

Tato publikace je určena výhradně jako podpůrný text pro potřeby výuky. Bude užita výhradně v přednáškách výlučně k účelům vyučovacím či jiným vzdělávacím účelům. Nesmí být používána komerčně. Bez předchozího písemného svolení autora nesmí být kterákoliv část této publikace kopírována nebo rozmnožována jakoukoliv formou (tisk, fotokopie, mikrofilm, snímání skenerem či jiný postup), vložena do informačního nebo jiného počítačového systému nebo přenášena v jiné formě nebo jinými prostředky.

Tento učební text vznikl za podpory projektu „Zvýšení konkurenceschopnosti IT odborníků – absolventů pro Evropský trh práce“, reg. č. CZ 04.1.03/3.2.15.1/0003. Tento projekt je spolufinancován Evropským sociálním fondem a státním rozpočtem České republiky.

Veškerá práva vyhrazena

© Jaroslav Zendulka, Ivana Rudolflová, Brno 2005, 2006

Obsah


















1.	Úvod.....	3
1.1.	Informace o organizačních záležitostech předmětu	3
1.2.	Informace o obsahových a metodických záležitostech předmětu	5
2.	Relační model dat.....	6
2.1.	Úvod.....	6
2.2.	Struktura relační databáze	7
2.3.	Integritní omezení v relační databázi	12
2.4.	Relační algebra	20
3.	Návrh relační databáze	33
3.1.	Úvod.....	33
3.2.	Konceptuální modelování	35
3.3.	Transformace konceptuálního modelu na tabulky relační databáze	65
3.4.	Normalizace schématu databáze	86
3.4.1.	Úvod do teorie závislostí.....	87
3.4.2.	Normální formy a proces normalizace schématu	95
4.	Organizace dat v databázi na fyzické úrovni	110
4.1.	Úvod.....	110
4.2.	Databázové soubory	113
4.3.	Správa vyrovnávací paměti	120
4.4.	Indexování a hašování.....	121
4.4.1.	Indexování.....	123
4.4.2.	Hašování.....	138
4.4.3.	Shlukování.....	142
4.4.4.	Bitmapový index	145
4.5.	Fyzický návrh databáze	147
4.6.	Organizace dat na fyzické úrovni u serveru Oracle 10g	151
5.	Transakční zpracování.....	158
5.1.	Úvod.....	158
5.2.	Transakce	159
5.3.	Transakce v jazyce SQL.....	161
5.4.	Zotavení po chybách a poruchách.....	164
5.5.	Řízení souběžného přístupu	174
5.6.	Zotavení souběžných transakcí	193
5.7.	Zotavení a souběžný přístup v SQL	194
6.	Trendy v databázových technologiích.....	199
6.1.	Úvod.....	199
6.2.	Databáze založené na objektech.....	201
6.3.	Přístup k databázím z WWW	206
6.3.1.	PHP (Personal Home Page).....	207
6.3.2.	Internet Information Server a ASP (Active Server Pages).....	210
6.3.3.	Přístup k databázím z jazyka Java.....	211
	Dodatek A Řešení testových otázek.....	217

1. Úvod

1.1. Informace o organizačních záležitostech předmětu

V této studijní opoře jsou použity následující piktogramy:

Tab. 1.1 Použité piktogramy

Ikona	Navrhovaný význam	Ikona	Navrhovaný význam
	Počítačové cvičení, příklad		Správné řešení
	Otázka, příklad k řešení		Obtížná část
	Příklad		
	Slovo tutora, komentář		
	Potřebný čas pro studium, doplněno číslicí přes hodiny		Důležitá část
	Reference		Cíl
			Definice
	Souhrn		Zajímavé místo (levá, pravá varianta)
			Rozšiřující látka, informace, znalosti. Nejsou předmětem zkoušky.



Tato studijní opora nepokrývá celou náplň předmětu, nýbrž zahrnuje pouze témata předmětu, která se ukazují pro studenty jako obtížnější zvládnutelné (např. teorie relačního modelu dat) a taková, která jsou klíčová a v nichž studenti často chybují (např. návrh relační databáze). Každá kapitola obsahuje několik řešených příkladů a na závěr kapitoly, případně i rozsáhlejší podkapitoly, je vždy uvedena řada otázek a příkladů, jejichž samostatné vyřešení se očekává od vás. Navíc je potom uvedeno několik testových otázek odpovídajících z hlediska náročnosti otázkám u závěrečné písemné zkoušky. Pokud není uvedeno jinak, jsou u těchto testových otázek nabízeny dvě odpovědi, z nichž je správná nejvýše jedna. Tj. buď není správná žádná nebo je správná jedna z nabízených odpovědí. Správné řešení testových otázek je uvedeno v dodatku na konci studijní opory.

U zbývajících témat obsahu předmětu se předpokládá, že je zvládnete s využitím doporučené studijní literatury a konzultací.



Časový rozsah předmětu Databázové systémy je 39 hodin přednášek a 13 hodin práce na projektu za semestr.

Projekt navazuje na poslední projekt řešený v předmětu Úvod do softwarového inženýrství (IUS), jehož cílem bylo vytvořit základ konceptuálního modelu

reprezentujícího požadavky na zvolenou aplikaci. Pokračování tohoto projektu spočívá v dopracování konceptuálního modelu (model případů použití a ER diagram či diagram tříd), návrhu tabulek relační databáze, naplnění tabulek vzorkem dat a naprogramování požadovaných funkcí. Databázovým prostředím pro uložení dat je databázový server Oracle a vývojovým prostředím Oracle Forms Developer a Oracle Reports Developer a alternativním prostředím vývojové prostředí SQL Windows a databáze SQL Base firmy Gupta. Výsledky projektu budou odevzdávány ve třech etapách:

- odevzdání konceptuálního modelu a databázového schématu,
- odevzdání SQL skriptu pro vytvoření tabulek databáze a naplnění vzorkem dat.
- odevzdání finálních verzí souborů tvořících kompletní projekt.

Navržená databáze projektu může být využita i k samostatnému procvičení dotazování v SQL.

Kreditová hodnota předmětu je 5 kreditů. **Časová náročnost** zvládnutí obsahu předmětu je 140 hodin. Z toho 50 hodin připadá na řešení projektu a domácí úlohy, zbytek na zvládnutí přednášené látky. Časová náročnost témat obsažených v této studijní opoře je následující:

Relační model dat	10 hod.
Návrh relační databáze	20 hod.
Organizace dat na fyzické úrovni	10 hod.
Transakční zpracování	10 hod.
Trendy rozvoje databázových technologií	10 hod.



Podmínkou **získání zápočtu** a tedy i připuštění ke zkoušce je získání nejméně 25 bodů za **bodované aktivity** v průběhu semestru, kterými jsou:

projekt	36 bodů
z toho konceptuální model a databázové schéma	6 bodů
SQL skript pro vytvoření a naplnění tabulek	5 bodů
...realizace projektu	25 bodů
půlsestrální test	14 bodů

Na závěrečnou písemnou zkoušku lze získat zbývajících 50 bodů, tedy za celý předmět 100 bodů.

Klasifikace je podle klasifikační stupnice uvedené ve čl. Studijního a zkušebního řádu VUT - viz <http://www.fit.vutbr.cz/info/predpisy/szvut04.pdf>.

Podrobnější informace k požadavkům na projekty, domácí úlohu, termíny a konzultací a další aktuální informace budou přístupné na adrese <http://www.fit.vutbr.cz/study/courses/IDS/private/index.html>.

1.2. Informace o obsahových a metodických záležitostech předmětu



Cíl: Cílem předmětu je zvládnutí základů teorie relačních databázových systému a získání praktických dovedností s použitím databázových technologií na úrovni potřebné pro návrh databáze, tvorbu databázových aplikací a správu databázových systémů.

Anotace: Základní pojmy databázových systémů (DBS). Konceptuální modelování. Teorie relačního modelu dat. Návrh relační databáze z konceptuálního modelu. Normalizace schématu databáze a její využití při návrhu relační databáze. Jazyk SQL. Transakční zpracování. Architektury DBS: klient/server, vícevrstvé architektury. Základy činností administrátora databáze: bezpečnost a integrita dat, úvod do fyzického návrhu databáze, optimalizace výkonnosti, zotavení po poruchách, řízení souběžného přístupu. Trendy v rozvoji databázových technologií. Řešení databázové aplikace s využitím moderního vývojového a databázového prostředí.

Prerekvizitní znalosti: Předmět Databázové systémy předpokládá znalosti získané v předmětu Úvod do softwarového inženýrství (IUS) a to zejména v oblasti základních kroků vývoje programových systémů a modelů jejich životního cyklu. Dále předpokládá znalost pojmů souvisejících s množinami, relacemi, zobrazeními a grafy z předmětu Diskrétní matematika (IDA), znalost základů vyhledávacích stromů a hašování z předmětu Algoritmy (IAL) a znalost základů programování z předmětu Základy programování (IZP). Ve všech případech jde o povinné předměty bakalářského studijního programu Informační technologie.

Základní literatura:



- [Pok02] Pokorný, J.: Dotazovací jazyky. Učební texty Univerzity Karlovy v Praze. UK v Praze. 255 s. 2002. ISBN 80-246-0497-3.
- [Pok98] Pokorný, J.: Databázová abeceda. Science, Veletiny. 240 s. 1998. ISBN 80-86083-02-0.
- [Sil05] Silberschatz, A., Korth H.F, Sudarshan, S.: Database System Concepts, Fifth Edition. McGraw-Hill. 1168 p. 2005. ISBN 0-07-295886-3.
- [Oracle] Oracle10g Database Online Documentation. (Release 2 (10.2)). Oracle Corporation. Dokumenty dostupné na adrese http://www.oracle.com/pls/db102/portal.portal_db?selected=1.

Doporučená literatura:



- [Dat03] Date, C.J.: An Introduction to Database Systems, Eighth Edition. Addison Wesley. 1024 p. 2003. ISBN 0-321-19784-4.

2. Relační model dat



Cíl: V této kapitole se seznámíte se základy teorie, na níž jsou relační databázové systémy postaveny.

Anotace: Relace, tabulka, atribut, n-tice, atomická hodnota, normalizovaná relace, kandidátní klíč, primární klíč, hodnota NULL, cizí klíč, referenční integrita, relační algebra, selekce, projekce, spojení, relační kalkul.

Prerekvizitní znalosti: V této kapitole se předpokládá znalost základů teorie množin, matematického pojmu relace a pojmu algebra. Dále se předpokládá znalost základních pojmů databázových technologií, které se vysvětlují na úvodní přednášce předmětu, jako jsou perzistentní data, databáze, systém řízení báze dat (SŘBD), databázový systém, datový model, integrita dat, integritní omezení, procedurální a neprocedurální databázový jazyk.



Odhad doby studia: 10 hodin.

2.1. Úvod



Existence teorie relačního modelu dat, která vznikla dříve, než byly implementovány první relační databázové systémy, je jedním z významných odlišností relačních systémů od databázových systémů vycházejících z jiného datového modelu. Takové systémy začaly vznikat ve druhé polovině 60. let minulého století (nejznámější jsou síťové a hierarchické) v souvislosti s výrazným růstem požadavků na vývoj datově intenzivních aplikací v 60. letech. Pro tyto systémy bylo typické, že nejprve vznikla nějaká implementace a teprve dodatečně se hledala teorie, která by tuto implementaci podepřela. Podobná situace platí i pro databázové systémy postavené na jiném než relačním základě (např. objektově-orientované), které vznikly až v pozdějším období. V tomto smyslu mají relační systémy výjimečné postavení a tento fakt určitě přispěl i k tomu, že se relační databázové systémy prosadily na přelomu 70. a 80. let v konkurenci s výše zmíněnými systémy hierarchickými a síťovými.

Vysvětlení základů této teorie je cílem této kapitoly. Jejich pochopení vám pomůže lépe porozumět přednostem relačních databází, ale také omezením, která z relačního modelu dat vyplývají.

Základem relačních databázových systémů se stala publikace pracovníka firmy IBM E.F.Codda s názvem „A relational data model for large shared data banks“, která vyšla v roce 1970 v časopisu Communications of the ACM. V této publikaci byl zaveden zcela nový datový model pro ukládání perzistentních dat s cílem dosažení datové nezávislosti, tj. nezávislosti aplikačních programů na změnách ve struktuře databáze a použitých přístupových metodách. V dalších letech se teorie relačního modelu dále rozvíjela a začala implementace prvních relačních systémů.

Připomínáme, že relační model dat spadá v klasifikaci datových modelů, jak byla zavedena na úvodní přednášce, mezi databázové modely, tj. zahrnuje:

- *Definici logické struktury dat v relační databázi*, tj. říká, jak jsou data v relační

databázi strukturovaná na logické úrovni bez ohledu na vlastní implementaci. Oddělení logické struktury od implementace (ta v té době ani neexistovala) bylo jedním z hlavních přínosů článku E.F.Codda.

- *Definici obecných integritních pravidel*, tj. omezení kladená na data, která musí platit v každé relační databázi bez ohledu na to, zda obsahuje např. informace o studentech a jejich studijních výsledcích či o zboží a jeho objednávkách a dodávkách.
- *Formální dotazovací jazyk*, který umožňuje formulovat dotazy nad daty uloženými v relační databázi. I tato matematická podpora pro manipulaci s daty se stala významným přínosem relačního modelu, protože umožnila na jedné straně vznik neprocedurálních dotazovacích jazyků mezi které patří i jazyk SQL, na druhé straně zavádí operace nad daty v relační databázi, které lze použít k jejich implementaci.

Kromě teorie relačního modelu dat obsahoval výše uvedený článek teorii na podporu návrhu logické struktury relační databáze s cílem omezení redundance dat. I tato teorie představovala významný přínos, neboť umožnila nahradit intuitivní přístup k návrhu struktury databáze přístupem systematickým.

V této kapitole studijní opory se zaměříme na samotný relační model dat. Teorii na podporu návrhu se budeme zabývat v kapitole 3.4.



Protože teorie relačního modelu představuje formální, matematický pohled na relační databáze, používá odpovídající matematickou terminologii. Příkladem může pojem „relace“, kterým se označuje databázová tabulka, což je naopak termín běžně používaný databázovými odborníky v běžné řeči. Jedním z cílů této kapitoly proto bude i vysvětlení korespondence těchto běžně používaných termínů a jim odpovídajících matematických pojmů. Protože v dalších kapitolách s výjimkou kapitoly 3.4 pojednávající o již zmíněné teorii na podporu návrhu relační databáze budeme používat běžnou (neformální) databázovou terminologii, omezíme se na použití formálních matematických pojmů pouze v několika základních definicích. Ve vysvětlujícím textu budeme ve všech kapitolách používat neformální, ale běžně používanou terminologii relačních databází..

2.2. Struktura relační databáze



V souvislosti s relačními databázemi se při výkladu struktury takových databází nejčastěji setkáme s konstatováním, že data jsou v relační databázi strukturovaná do *tabulek*. Pojem tabulka ale nevyhovuje exaktnímu matematickému vyjadřování. V teorii relačního modelu se používá pro označení takové tabulky pojem *relace*, který s určitým rozšířením odpovídá pojmu relace, jak jej znáte z diskrétní matematiky. V dalším výkladu budeme ilustrovat základní pojmy relační datové struktury na následujícím příkladu:



Příklad 2.1

Uvažujme příklad databáze studijní části informačního systému naší fakulty, ve které budeme chtít ukládat mimo jiné informace o studentech. Předpokládejme pro jednoduchost, že u každého studenta budeme chtít znát jen jeho křestní jméno, příjmení, datum narození a adresu trvalého bydliště. Dále předpokládejme, že každý student má na fakultě přiřazeno jednoznačné přihlašovací jméno (login), které používá pro přihlašování se do počítačové sítě.



Vlastnosti, jejichž hodnoty u studenta chceme registrovat, tedy jsou LOGIN, JMÉNO, PŘÍJMENÍ a ADRESA. Tyto vlastnosti se v terminologii relačního modelu dat nazývají *atributy*. Každý atribut může nabývat hodnot z nějaké množiny přípustných hodnot, např. jméno z množiny jmen, adresa z možných adres. Takové množiny (obory hodnot) atributů se zde nazývají *domény*. Informace charakterizující jednoho konkrétního studenta pak dává do vztahu (relace) konkrétní hodnoty z příslušných domén. Jestliže například student Jan Novák má přiřazeno přihlašovací jméno xnovak00 a bydlí na adrese Cejl 9 Brno, pak je charakterizován hodnotami ('xnovak00', 'Jan', 'Novák', 'Cejl 9 Brno'). Taková skupina hodnot, které k sobě patří na základě nějakého vztahu (v našem případě charakterizují určitého studenta), se v teorii relačního modelu označuje jako *n-tice* a hodnota *n-tice* příslušející určitému atributu se označuje jako hodnota atributu, např. Již asi tušíte, že v relační databázi budou takovéto hodnoty uloženy v tabulce, jejíž sloupce budou odpovídat atributům a řádky *n-ticím*. Budeme mít tedy tabulku, kterou bychom si mohli označit STUDENT se sloupci LOGIN, JMÉNO, PŘÍJMENÍ a ADRESA, a vždy jedním řádkem reprezentujícím jednoho studenta. V terminologii relačního modelu dat bychom řekli, že máme relaci STUDENT s atributy LOGIN, JMÉNO, PŘÍJMENÍ a ADRESA.

Podívejme se na uvedený příklad formálněji využitím pojmů z teorie množin a relací. Nechť D_{LOGIN} , $D_{JMÉNO}$, $D_{PŘÍJMENÍ}$ a D_{ADRESA} jsou množiny (v našem příkladě domény atributů LOGIN, JMÉNO, PŘÍJMENÍ a ADRESA), potom relace $R_{STUDENT}$ na množinách D_{LOGIN} , $D_{JMÉNO}$, $D_{PŘÍJMENÍ}$ a D_{ADRESA} je definována jako podmnožina kartézského součinu těchto množin, tj. $R_{STUDENT} \subseteq D_{LOGIN} \times D_{JMÉNO} \times D_{PŘÍJMENÍ} \times D_{ADRESA}$, tj. $R_{STUDENT} = \{(V_{LOGIN}, V_{JMÉNO}, V_{PŘÍJMENÍ}, V_{ADRESA}) : v_{LOGIN} \in D_{LOGIN} \wedge v_{JMÉNO} \in D_{JMÉNO} \wedge v_{PŘÍJMENÍ} \in D_{PŘÍJMENÍ} \wedge v_{ADRESA} \in D_{ADRESA}\}$. Prvky relace jsou tedy *n-tice* tvořené hodnotami z domén.



Příklad 2.2

Uvažujme náš příklad a tyto konkrétní hodnoty domén:

$D_{LOGIN} = \{xnovak00, xnovak01, xcerny00, xzelen05, xmodry02, \dots\}$

$D_{JMÉNO} = \{Eva, Jan, Pavel, Petr, Zdeněk, \dots\}$

$D_{PŘÍJMENÍ} = \{Adam, Černý, Novák, Modrý, Zelená, \dots\}$

$D_{ADRESA} = \{Cejl 9 Brno, Purkyňova 99 Brno, Brněnská 15 Vyškov, \dots\}$

Relace $R_{STUDENT}$ by potom mohla vypadat například následovně:

$R_{STUDENT} = \{(xcerny00, Petr, Černý, Brněnská 15 Vyškov), (xnovak00, Jan, Novák, Cejl 9 Brno), (xnovak01, Pavel, Novák, Cejl 9 Brno)\}$.

Tuto relaci bychom mohli znázornit tabulkou:

LOGIN	JMÉNO	PŘÍJMENÍ	ADRESA
xcerny00	Petr	Černý	Brněnská 15 Vyškov
xnovak00	Jan	Novák	Cejl 9 Brno
xnovak01	Pavel	Novák	Cejl 9 Brno



Pokud tuto tabulku pojmenujeme, například STUDENT, dostáváme základní a jedinou datovou strukturu, se kterou se setkáváme v relační databázi při pohledu na data na logické úrovni (abstrakce implementace).



Z uvedeného příkladu jsou zřejmé některé důležité skutečnosti. Jednak doména atributu zahrnuje všechny možné hodnoty, kterých daný atribut může nabývat. Ne všechny tyto hodnoty se ale musí v tabulce vyskytovat. Množina hodnot atributu, které se v daném

okamžiku v tabulce vyskytují, se někdy označuje jako *aktivní doména*. V praxi se pojem doména redukuje na pojem datový typ, jak jej znáte z programovacích jazyků. V našem příkladě bychom například definovali, že hodnoty atributu *login* jsou znakové řetězce délky 8 znaků, hodnoty atributu *jméno* znakové řetězce proměnné délky apod. Tabulku můžeme chápat jako tvořenou záhlavím, které se také označuje jako *schéma relace*, a tělem. Matematickému pojmu relace potom odpovídá tělo tabulky. Je zřejmé, že počet řádků tabulky i jejich obsah se obecně v čase mění. V našem případě příchod nového studenta na fakultu bude znamenat přidání nového řádku (vznikla nová n-tice dané relace), změna adresy bude znamenat změnu hodnoty atributu adresa v daném řádku apod.

Zatím jsme předpokládali, že prvky domén mohou být libovolné hodnoty a to i z hlediska složitosti. Relační model dat ale klade v tomto smyslu na hodnoty domén poměrně přísné omezení – domény mohou obsahovat pouze tzv., *atomické* označované také jako *skalární*, hodnoty. Znamená to, že hodnota musí představovat z hlediska svého významu nedělitelný celek. Uvažujme například doménu atributu adresa. Obsahuje atomické hodnoty? Odpověď na tuto otázku můžeme dát pouze na základě znalosti významu daného atributu. Jestliže adresa je pro nás nedělitelnou informací, u které nerozlišujeme název ulice, číslo atd., pak jde o hodnoty atomické. Pokud bychom ale naopak potřebovali takové složky adresy rozlišovat, nepůjde o hodnoty atomické, nýbrž složené a odpovídající doména a atribut se potom označují jako *složené* (složené z několika atributů, resp. hodnot). Prakticky by to potom znamenalo, že sloupec adresa by byl složením několika samostatných sloupců, např. pro ulici, číslo a město.

Může nastat ale situace ještě složitější. Předpokládejme v tomto okamžiku, že samotné hodnoty adresy jsou atomické, ale že potřebujeme uchovávat historii změn adresy, tj. nebudeme mít v databázi uloženy pouze aktuální adresy, nýbrž i adresy předchozí, pokud došlo ke změně, a to i s rokem změny.

Příklad 2.3

x+y

Uvažujme náš příklad a zmodifikujme si doménu adresy, aby zahrnovala i historii změn. Předpokládejme, že kromě adresy budeme ukládat i rok ukončení pobytu na dané adrese (u aktuální adresy bude tato hodnota prázdná). Potom bude doména atributu adresa podstatně složitější, protože jeho hodnotami budou obecně množiny dvojic s významem (bydliště, rok):

$D_{ADRESA} = \{ \{(Cejl\ 9\ Brno,\)\}, \{(Purkyňova\ 99\ Brno,\)\}, \{(Brněnská\ 15\ Vyškov,\)\}, \{(Cejl\ 9\ Brno,\ 2005\), (Letná\ 20\ Praha,\)\}, \dots \}$

Pokud bychom studenty uvedené v tabulce příkladu Příklad 2.2 a situaci, kdy se student Jan Novák přestěhoval do Prahy, vypadala by naše tabulka takto:

LOGIN	JMÉNO	PŘÍJMENÍ	ADRESA	
xcerny00	Petr	Černý	BYDLIŠTĚ	ROK
			Brněnská 15 Vyškov	
xnovak00	Jan	Novák	bydliště	rok
			Cejl 9 Brno	2005
			Letná 20 Praha	
xnovak01	Pavel	Novák	bydliště	rok
			Cejl 9 Brno	



Je zřejmé, že teď doména adresy obsahuje dokonce jako své prvky množiny nějakých n-tic hodnot jiných domén, tedy vlastně *zanořené relace*. Atribut definovaný na takové

složité doméně se potom označuje jako *vícehodnotový*.

Jak už bylo uvedeno, relační model dat připouští pouze domény obsahující skalární hodnoty. Pokud bychom chtěli mít hodnoty z příkladu Příklad 2.3 uloženy v relační databázi v jedné tabulce, která by splňovala požadavky relačního modelu, pak by musela vypadat následovně:

x+y

Příklad 2.4

LOGIN	JMÉNO	PŘÍJMENÍ	BYDLIŠTĚ	ROK
xcerny00	Petr	Černý	Brněnská 15 Vyškov	
xnovak00	Jan	Novák	Cejl 9 Brno	2005
xnovak00	Jan	Novák	Letná 20 Praha	
xnovak01	Pavel	Novák	Cejl 9 Brno	

Relace (tabulka) na doménách obsahujících pouze skalární hodnoty se, pokud chceme tuto skutečnost zdůraznit, označuje jako *normalizovaná*, říkáme také, že taková relace je v *1. normální formě (1NF)*. Naopak relace na složených či vícehodnotových doménách (která ve skutečnosti není relací ve smyslu relačního modelu dat) se označuje jako *nenormalizovaná* a proces převodu nenormalizovaných relací na normalizované splňující vlastnosti dobrého návrhu se označuje jako *normalizace*. My jsme zde provedli pouze základní krok, abychom splnili podmínku skalárních hodnot. Pokud se na tabulku v příkladu Příklad 2.4 podíváte pozorněji, zjistíte, že taková tabulka není dobře navržena, protože se v ní některé informace opakují. Říkáme, že obsahuje *redundanci*. Konkrétně, pokud nějaký student změní adresu, opakuje se v tabulce informace o tom, jak se tento student jmenuje. A již z úvodní přednášky víte, že naší snahou při návrhu databáze je takovou redundanci vyloučit. Jak to uděláme, se podrobněji dozvíte v kapitolách 3.3 a 3.4.

Důvodem pro přísný požadavek atomických hodnot v tabulce je zejména fakt, že operace s normalizovanou tabulkou jsou podstatně jednodušší než s tabulkou nenormalizovanou. Pokud bychom například uvažovali, že potřebujeme uložit do nenormalizované tabulky z příkladu Příklad 2.4 informaci o bydlišti studenta, bude operace závislá na tom, zda jde o studenta nového nebo již existujícího. V prvním případě má operace podobu vložení jednoho řádku do tabulky STUDENT, včetně vytvoření zanořené tabulky ve sloupci ADRESA a vložení jednoho řádku do ní. Ve druhém případě se vloží jeden řádek do zanořené tabulky ve sloupci ADRESA řádku příslušného studenta. Jak bude vypadat odpovídající operace pro normalizovanou tabulku podle příkladu Příklad 2.4 ponecháme jako domácí cvičení.

?

Poté, co jsme si neformálně vysvětlili nejdůležitější pojmy související s částí relačního modelu dat, ukážeme si jednu z formálních definic relace (ve smyslu relačního modelu dat), se kterými se můžete v literatuře setkat. Použijeme definici obdobnou definicím v základní literatuře [Pok02] a [Sil05].

DEF

Definice 2.1 Relace

Nechť D_1, D_2, \dots, D_n jsou množiny atomických hodnot označované jako *domény*. Relace (databázová) na doménách D_1, D_2, \dots, D_n je dvojice $\mathbf{R} = (R, R^*)$, kde $R = R(A_1:D_1, A_2:D_2, \dots, A_n:D_n)$ je *schéma relace*, kde A_i ($A_i \neq A_j$ pro $i \neq j$) značí jméno atributu definovaného na doméně D_i , a $R^* \subseteq D_1 \times D_2 \times \dots \times D_n$ je *tělo relace*. Počet atributů n relace se označuje *stupeň (řád) relace*, kardinalita těla relace $m = |R^*|$ se označuje *kardinalita relace*.

Schéma relace zapisujeme často zjednodušeně ve tvaru $R(A_1, A_2, \dots, A_n)$, tj. pouze uvedením jména relace a jmen atributů. Tento způsob vyjádření budeme používat i my, pokud budeme chtít popsat tvar relace, resp. databázové tabulky. Například schéma tabulky STUDENT z příkladu Příklad 2.2 bychom zapsali ve tvaru STUDENT(LOGIN, JMÉNO, PŘÍJMENÍ, ADRESA).

Označením relace jako databázové jsme pouze chtěli vyjádřit fakt, že nejde o pojem zcela totožný s matematickým pojmem relace. Tomu přesně odpovídá pouze tělo naší (databázové) relace. V dalším ale budeme používat označení relace výhradně pro databázové relace, pokud nebude uvedeno jinak.

Jestliže si definici přečtete pozorně, zjistíte některé další vlastnosti relace, mezi které patří zejména tyto:

- některé atributy mohou být definovány na stejné doméně, jména atributů však musí být v jednom schématu jednoznačná,
- protože tělo relace je množina n -tic, je neuspořádané,
- ze stejného důvodu neexistují duplicitní n -tice, tj. stejné řádky v tabulce.

Až se budeme učit databázový jazyk SQL, ukážeme si, do jaké míry jsou tyto vlastnosti relace v praxi dodrženy.



E.F.Codd definoval relaci ještě trochu jinak a s takovou definicí se v literatuře také můžete setkat – např. v doporučené [Dat03]. Tyto definice se liší od naší v tom, že navíc chápou schéma relace a tedy i prvky n -tic jako množiny, tedy neuspořádané.

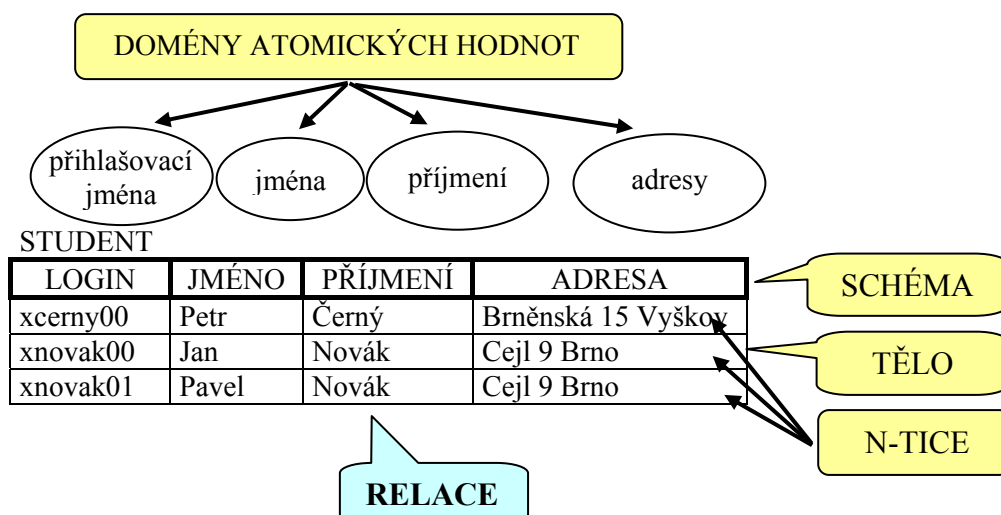


V této podkapitole jsme si vysvětlili, jak definuje relační model dat strukturu relační databáze. Využívá k tomu matematického pojmu relace, který se používá pro označení množiny obsahující n -tice hodnot tzv. atributů definovaných na množinách hodnot, které se zde nazývají domény. Každá taková n -tice potom obsahuje pro každý atribut takové hodnoty z odpovídajících domén, které k sobě patří na základě určitého vztahu. My jsme k ilustraci použili relaci, kterou jsme si nazvali SUDENT, s atributy LOGIN, JMÉNO, PŘÍJMENÍ a ADRESA. Protože jedním z možných způsobů reprezentace relace je výčet jejích prvků, tj. oněch n -tic, v podobě tabulky, používá se v běžné databázové terminologii pro označení relace termín tabulka, n -ticím relace potom odpovídají řádky tabulky a atributům jména sloupců.

Dále jsme si vysvětlili, že tabulka v relační databázi může obsahovat pouze tzv. atomické hodnoty. To jsou hodnoty, které jsou z hlediska významu jednoduché, dále nedělitelné. Taková tabulka se nazývá normalizovaná. Tuto vlastnost jsme ilustrovali na příkladě atributu ADRESA, jehož hodnoty mohou být považovány za jednoduché nebo složené podle toho, jak adresu chápeme a hodláme s ní pracovat.

V této části jsme si vysvětlili ještě jeden důležitý pojem a sice schéma relace (tabulky). Rozumíme jím popis tvořený názvem relace, výčtem jmen atributů a definicí odpovídajících domén. V následující kapitole si tento pojem ještě trochu rozšíříme a později při výkladu SQL si ukážeme, jakým příkazem tabulku s daným schématem vytvoříme.

Souhrnně jsou základní pojmy relační struktury dat ilustrovány na **Obr. 2.1**.



Obr. 2.1 Základní pojmy relační struktury dat

Závěrem můžeme shrnout, že data jsou v relační databázi na logické úrovni strukturována do tabulek. Uživatel takové databáze (tedy i aplikace, které k ní přistupují), vnímá relační databázi jako kolekci časově proměnných



V souvislosti s podkapitolou, kterou právě končíme, je namístě upozornit ještě na jednu skutečnost. Relační databáze se nazývají relační, protože data v nich jsou organizována v souladu s relačním modelem dat a ten se nazývá relační, protože data jsou podle tohoto modelu logicky strukturována na základě relací (tabulek). V některé literatuře se můžete dočíst, že se tyto databáze nazývají relační proto, že v nich existují relace mezi tabulkami. Toto není ten důvod. My už víme, že relace v relačním modelu označuje vlastně tabulku.

2.3. Integritní omezení v relační databázi

V minulé kapitole jsme si vysvětlili, jak jsou data v relační databázi logicky organizována. Už víme, že relační databáze je tvořena kolekcí tabulek. Protože typicky existují souvislosti mezi informacemi reálného světa uloženými v těchto tabulkách, musíme mít možnost i tyto souvislosti v nějaké podobě do databáze uložit. U předrelačních systémů se tyto souvislosti uchovávaly v podobě ukazatelů mezi uloženými záznamy. Nevýhodou tohoto způsobu byla zpravidla závislost aplikací na cestách k datům definovaných těmito ukazateli. V případě relačního modelu navrhl E.F.Codd jiný způsob, který si vysvětlíme na následujícím příkladě.



Příklad 2.5

Předpokládejme, že v databázi informačního systému naší fakulty potřebujeme uložit nejen informace o studentech, ale i o projektech, které studenti řeší. Budeme pro jednoduchost předpokládat, že daný projekt může řešit jen jeden student. Za předpokladu, že potřebujeme uchovávat název projektu, zadání projektu a informaci o tom, který student projekt řeší, by naše databáze mohla obsahovat kromě nám již známé tabulky STUDENT (LOGIN, JMÉNO, PŘÍJMENÍ, ADRESA) ještě tabulku PROJEKT (ČÍSLO, NÁZEV, ZADÁNÍ, LOGIN), jejíž sloupec ČÍSLO obsahuje jednoznačné číslo projektu, NÁZEV udává název projektu, ZADÁNÍ obsahuje zadání projektu a sloupec LOGIN říká, který student daný projekt řeší. Uvažujme následující obsah tabulek:

STUDENT

LOGIN	JMÉNO	PŘÍJMENÍ	ADRESA
xcerny00	Petr	Černý	Brněnská 15 Vyškov
xnovak00	Jan	Novák	Cejl 9 Brno
xnovak01	Pavel	Novák	Cejl 9 Brno

PROJEKT

ČÍSLO	NÁZEV	ZADÁNÍ	LOGIN
1	Internetový obchod	Realizujte ...	xnovak01
2	OO databáze	Seznamte se s ...	xcerny00
3	Řízené gramatiky	Připravte ...	xnovak01
4	Lexikální analyzátor	Naprogramujte ...	

Už víme, že jeden řádek tabulky STUDENT reprezentuje jednoho konkrétního studenta a jeden řádek tabulky PROJEKT reprezentuje jeden konkrétní projekt. Je zřejmé, že požadavek, abychom věděli, kdo který projekt řeší, vyžaduje uložení takové informace do databáze. Vztah „řeší“ vyžaduje vytvoření vazby mezi odpovídající dvojicí řádků (kdo řeší projekt). Jak bylo uvedeno výše, předrelační systémy toto řešily použitím ukazatelů, tj. fyzickou vazbou. V relačním modelu dat se taková vazba řeší jako logická, vytvořená na základě rovnosti hodnot v určitých sloupcích odkazovaného a odkazujícího se řádku. V příkladu Příklad 2.5 je vazba reprezentující vztah „řeší“ vytvořena na základě rovnosti hodnot sloupců LOGIN v tabulce STUDENT a LOGIN v tabulce PROJEKT. Například hodnota xnovak01 v řádku projektu číslo 1 se shoduje s hodnotou v řádku tabulky STUDENT pro Pavla Nováka, vytváří tak logickou vazbu (odkaz) mezi těmito dvěma řádky a současně reprezentuje informaci, že projekt číslo 1 řeší Pavel Novák s přihlašovacím jménem xnovak01.

Aby bylo možné takové vazby vytvářet, je potřeba vyřešit dva problémy. Jednak musí existovat možnost jednoznačné identifikace odkazovaného řádku, jednak musí existovat v odkazující se tabulce sloupec, jehož hodnoty budou vazbu vytvářet. Relační model řeší tyto dva problémy pomocí tzv. *klíčů*, přesněji pro identifikaci se používají tzv. *primární klíče* a pro odkazy tzv. *cizí klíče*. Budou to sloupce, pro jejichž hodnoty budou platit určitá omezení, která relační model dat definuje.

My už víme z úvodní přednášky, že omezení kladená na data uložená v databázi se nazývají *integritní omezení*. Tato omezení musí být splněna, mají-li být data v databázi správná. Integritní omezení můžeme rozdělit do dvou skupin:

- obecná
- specifická.

V prvním případě jde o integritní omezení, která musí platit v každé databázi daného typu (u nás relační) bez ohledu na konkrétní aplikační zaměření. Právě taková integritní omezení definuje relační model pro sloupce, které budou plnit roli primárních a cizích klíčů. Půjde tedy o omezení, která musí takové sloupce splňovat bez ohledu na to, zda jde o relační databázi informačního systému fakulty nebo například modulu plánování výroby.

Na druhé straně každá konkrétní aplikační oblast zpravidla má svá specifická omezení na data. V našem příkladu by mohlo existovat omezení na hodnoty přihlašovacího jména, které by říkalo, že znakový řetězec je tvořen šesti písmeny a dvěma číslicemi, Písmena jsou počátečními znaky příjmení, případně doplněné počátečními znaky křestního jména.

Když jsme si vysvětlovali, jak vypadá tabulka relační databáze, viděli jsme, že její tělo je tvořeno množinou řádků (neuspořádanou). Jednotlivé řádky proto nemají žádné adresy nebo pořadová čísla, prostřednictvím kterých by bylo možné se na ně odkazovat. Přitom už víme, že musí existovat možnost určit přesně jeden řádek, na který se nějaký jiný řádek odkazuje nebo se kterým se má provést nějaká operace. Jedinou možností, jak identifikovat konkrétní řádek nějaké tabulky, je prostřednictvím hodnoty v nějakém sloupci (případně kombinaci sloupců), která bude v tomto sloupci v dané tabulce unikátní. Tj. nebudou existovat v této tabulce dva řádky se stejnou hodnotou v takovém sloupci). Za tohoto předpokladu můžeme hodnoty v takovém sloupci použít k odkazům na řádek tabulky. Takový sloupec, který budeme používat k „adresaci“ řádků tabulky, budeme nazývat *primárním klíčem* tabulky. V příkladě Příklad 2.5 je primárním klíčem v tabulce STUDENT sloupec LOGIN a v tabulce PROJEKT sloupec ČÍSLO.



Tabulku relační databáze můžeme považovat za určitou podobu *asociativní paměti*, tj. paměti ze které se informace nevybírají uvedením adresy, ze které se má informace přečíst, nýbrž podle obsahu, tj. uvedením nějaké hodnoty, která je součástí uložené informace, kterou chceme získat. V případě tabulky neuvádíme nějakou adresu nebo číslo řádku, nýbrž hodnotu primárního klíče. Chceme-li vyhledat informace o konkrétním studentovi, zadáme jeho přihlašovací jméno, protože sloupec LOGIN je v tabulce STUDENT primárním klíčem.

Sloupců (případně složených), jejichž hodnoty jsou v dané tabulce unikátní, může být více. Proto zavádí relační model dat kromě pojmu primární klíč ještě i pojem *kandidátní klíč*. Pokud v tabulce existuje více kandidátních klíčů, potom vybereme jeden z nich a ten bude sloužit jako primární klíč. Uvidíme, že na hodnoty ve sloupci, který je primárním klíčem, je kladeno přísnější integritní omezení než na sloupec kandidátního klíče.

Podívejme se na tyto dva pojmy a odpovídající integritní omezení podrobněji. V následujících definicích použijeme opět v souladu s terminologií relačního modelu pojmy relace, atribut a n-tice.



Definice 2.2 Kandidátní klíč (candidate key)

Atribut *CK* relace **R** se nazývá *kandidátním klíčem*, když splňuje tyto dvě časově nezávislé vlastnosti:

- Hodnoty atributu *CK* v relaci **R** jsou *unikátní (jednoznačné)*, tj. neexistují žádné dvě n-tice relace se stejnou hodnotou tohoto atributu.
- Atribut *CK* je vzhledem k jednoznačnosti hodnot v **R** *minimální (neredukovatelný)*, tj. je-li *CK* složeným atributem, nelze vypustit z něho žádnou složku, aniž by přestala být splněna unikátnost hodnot.

V souvislosti s touto definicí je třeba vysvětlit dvě věci. Jednak požadavek časové nezávislosti obou vlastností, který znamená, že vlastnosti musí platit v každém okamžiku. Nestačí tedy například, aby hodnoty byly unikátní jenom v daném stavu tabulky, např. hodnoty ve sloupci NÁZEV tabulky PROJEKT příkladu Příklad 2.5. Musí to být vlastnost daného sloupce vyplývající z jeho významu. Byla by to pravda pouze tehdy, pokud by platilo pravidlo, že název projektu musí být v rámci všech projektů, o nichž budeme uchovávat hodnoty v naší databázi, jednoznačný. To ale v tomto případě asi neplatí.

Druhá poznámka se týká požadavku minimality atributu a zajišťuje, aby jako kandidátní klíče byly uvažovány jen nejjednodušší možné atributy. Pokud bychom například uvažovali v naší tabulce STUDENT složený sloupec (LOGIN, PŘÍJMENÍ) a význam obou sloupců, jak byl uveden, pak hodnoty v takto složeném sloupci, tj. kombinace hodnot, např. (xcerny00, Černý), budou zcela určité v tabulce unikátní. Tento složený sloupec ale nebude kandidátním klíčem v tabulce STUDENT, protože ho lze zjednodušit (zredukovat). Přece už samotné hodnoty ve sloupci LOGIN jsou v tabulce unikátní, a proto hodnoty sloupce PŘÍJMENÍ jsou zde zbytečné. Uvažujme ale jiný příklad.

Příklad 2.6

x+y

Uvažujme tabulku ZKUŠEBNÍ_ZPRÁVA (LOGIN, R_CISLO, ZKRATKA, AK_ROK, BODY), jejíž jeden řádek uchovává informaci, kolik bodů v daném akademickém roce získal daný student z daného předmětu. LOGIN je jednoznačné přihlašovací jméno studenta, R_CISLO je rodné číslo studenta, ZKRATKA je jednoznačná zkratka předmětu, AK_ROK udává akademický rok a BODY je získaný počet bodů. Pokud platí, že student si může v daném akademickém roce zapsat předmět pouze jednou (nemusí ale uspět), pak tato tabulka bude obsahovat dva kandidátní klíče a to (LOGIN, ZKRATKA, AK_ROK) a (R_CISLO, ZKRATKA, AK_ROK). V obou případech jde o kandidátní klíče složené a v obou případech je nutná celá trojice hodnot, aby byla zajištěna unikátnost hodnot.

?

Tabulka předchozího příkladu obsahuje redundanci. Vidíte ji?

?

Z definice relace (Definice 2.2) vyplývá, že obsahuje vždy alespoň jeden kandidátní klíč. Z čeho toto tvrzení plyne a jak tento kandidátní klíč vypadá?

Když jsme si přesněji definovali pojem kandidátní klíč, můžeme již definovat přesněji i pojem primární klíč.

DEF

Definice 2.3 Primární klíč (primary key)

Primárním klíčem je jeden z kandidátních klíčů (vybraný), zbývající kandidátní klíče se nazývají *alternativní* (někdy také *sekundární*).

Pokud v tabulce existuje jediný kandidátní klíč, pak je současně klíčem primárním. Je-li kandidátních klíčů více, potom je primárním klíčem některý z nich. Relační model dat nestanoví v takovém případě způsob výběru. V praxi volíme takový kandidátní klíč, který je nejjednodušší. Běžnou praxí je i to, že pokud by byly v tabulce kandidátní klíče, tvořené sloupci, jejichž hodnoty skutečně potřebujeme ukládat, příliš složité, zavádí se speciální sloupec s jednoduchým oborem hodnot (např. celá čísla), který bude sloužit jako primární klíč.

Protože hodnoty primárního klíče slouží k „adresaci“ řádků tabulky, je jeho postavení mezi ostatními kandidátními klíči výjimečné. Viděli jsme, že hodnoty sloupce, který je kandidátním klíčem, musí splňovat důležité integritní omezení – musí být unikátní. V případě primárního klíče toto nestačí. Musíme vzít v úvahu, že hodnota ve sloupci tabulky může chybět. To je v případě, kdy hodnotu neznáme, neexistuje nebo z nějakého jiného důvodu není zadána. Až budeme probírat jazyk SQL, uvidíme, že pro takovou neexistující hodnotu zavádíme označení NULL. Řekneme si také více o

chování takové neexistující hodnoty a uvidíme, že je její chování určitým způsobem zvláštní.

Uvažujme, že by v naší tabulce STUDENT v jednom nebo několika řádcích chyběla hodnota ve sloupci LOGIN, který je zde primárním klíčem. Z hlediska vztahu v realitě by to znamenalo, že u těchto studentů neznáme jejich přihlašovací jméno, které jako jediné jednoznačně studenta určuje. Pokud bychom něco takového připustili, ztratili bychom možnost se na tyto studenty prostřednictvím přihlašovacího jména odkazovat. Proto relační model dat zpřísňuje integritní omezení pro primární klíč v tom smyslu, že jeho hodnota musí být jednoznačná a plně definovaná.



Definice 2.4 Pravidlo integrity entit

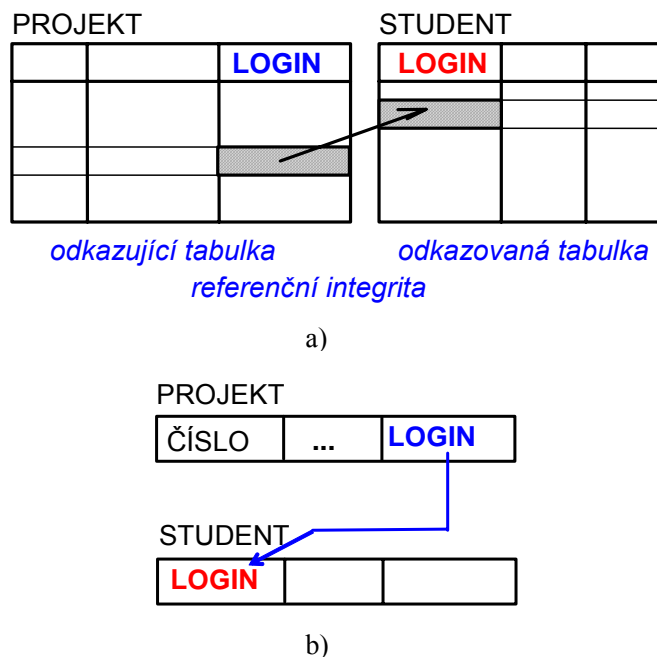
U žádné složky primárního klíče nesmí chybět hodnota.

Uvedená definice, která se někdy označuje jako pravidlo integrity entit, uvažuje obecný případ, kdy je primární klíč složený. Žádná ze složek nesmí být prázdná. Pokud bychom v příkladě Příklad 2.6 zvolili jako primární klíč kombinaci sloupců (LOGIN, ZKRATKA, AK_ROK), pak by hodnota (xcerny00, , '2004/5') nebyla legální, protože není zřejmé, o jaký předmět by mělo jít.

Nyní se vrátíme k problematice vytváření vazeb (logických) mezi řádky tabulek. Už jsme si uvedli, že se vytvářejí prostřednictvím tzv. cizích klíčů. Opět si nejprve neformálně vysvětlíme, co cizí klíč je, a teprve potom uvedeme přesnější definici. Využijeme k tomu Příklad 2.5.

Informaci o tom, který student daný projekt řeší, je vyjádřena hodnotou ve sloupci LOGIN tabulky PROJEKT. Jakých hodnot může tento sloupec nabývat? Hodnota musí jednoznačně určovat některého studenta. Můžeme tedy říci, že oborem hodnot budou přihlašovací jména studentů, tedy bude stejný jako obor hodnot sloupce LOGIN v tabulce STUDENT. V terminologii relačního modelu bychom řekli, že oba atributy jsou definovány na stejné doméně. To ale nestačí. Bude například hodnota xsedyv00 pro náš příklad platná? Asi nebude, protože student s takovým přihlašovacím jménem v tabulce STUDENT neexistuje.

Můžeme tedy říci, že sloupec LOGIN v tabulce PROJEKT slouží k vytvoření vazeb mezi řádky této tabulky a tabulky STUDENT. Takový sloupec se označuje jako *cizí klíč*. Pro jeho hodnoty existuje integritní omezení, které říká, že jsou přípustné pouze takové hodnoty, které se vyskytují ve sloupci, který je kandidátním klíčem v odkazované tabulce. Soulad hodnot cizího a odkazovaného kandidátního klíče se označuje jako *referenční integrita*. Situace pro náš příklad je znázorněna na **Obr. 2.2**.



Obr. 2.2 Vazba mezi řádky tabulek z příkladu Příklad 2.5. a) Vytvoření vazby řádků, b) Typické znázornění v diagramu schématu databáze

Formální definice cizího klíče by mohla vypadat takto:

DEF

Definice 2.5 Cizí klíč (foreign key)

Atribut *FK* relace **R2** se nazývá *cizí klíč*, právě když splňuje tyto časově nezávislé vlastnosti:

- Každá hodnota *FK* je buď plně zadaná nebo plně nezadaná.
- Existuje relace **R1** s kandidátním klíčem *CK* takovým, že každá zadaná hodnota *FK* je identická s hodnotou *CK* nějaké *n*-tice relace **R1**

Z této definice vyplývá opět několik skutečností, které je potřeba zdůraznit. Jednak vidíme, že narozdíl od primárního klíče může být hodnota cizího klíče nezadaná. To je i případ projektu číslo 4 v příkladu Příklad 2.5, kdy buď tento projekt žádný student neřeší nebo z nějakého jiného důvodu není hodnota (možná zatím) zadaná. Cizí klíč, podobně jako kandidátní klíč může být složený. V takovém případě je platná pouze plně zadaná nebo naopak plně nezadaná hodnota, tj. nemohou být některé složky takového složeného klíče zadané a jiné nezadané.

Pokud je hodnota zadaná, pak se musí shodovat s některou hodnotou ve sloupci, který je kandidátním klíčem v odkazované tabulce. Odkazující se řádek se může odkazovat jen na jeden řádek odkazované tabulky. Proto platné zadané hodnoty musí být unikátní v odkazovaném sloupci. A takovou vlastnost, jak už víme, má právě sloupec, který je v odkazované tabulce kandidátním klíčem.

Omezení hodnot opět musí platit v každém časovém okamžiku, přesněji pro každý stav databáze, v němž jsou považována data za správná.

Soulad hodnot cizích klíčů a odkazovaných klíčů kandidátních klíčů, tedy referenční integrita, představuje významný rys relačních databází, neboť „drží databázi pohromadě“. Zajištění referenční integrity je důležitou úlohou SŘBD, resp. programátora. Můžeme tedy zformulovat následující pravidlo pro data v databázi, aby byla správná z pohledu referenční integrity:

**Definice 2.6** Pravidlo referenční integrity

Relační databáze nesmí obsahovat žádnou nesouhlasnou hodnotu cizího klíče.



Přestože integritní omezení pro cizí klíč připouští, aby odkazovaným sloupcem byl nejen primární klíč odkazované tabulky, v praxi jde zpravidla o primární klíč.

Jména sloupců cizího klíče a odkazovaného kandidátního klíče mohou být různá, i když v praxi se velice často používají jména stejná. Podstatné je, aby význam hodnot v obou sloupcích (tedy odpovídající doména) byl stejný. Náš sloupec LOGIN v tabulce PROJEKT se klidně mohl nazývat například STUDENT. Důležité je, že hodnoty v obou sloupcích jsou přihlašovací jména studentů, v tabulce PROJEKT významově rozšířená o fakt, že je to přihlašovací jméno studenta řešícího daný projekt.

Odkazy mezi sloupci tabulek můžeme znázornit orientovaným grafem, jak je patrné z **Obr. 2.2 b)**. Posloupnost takových odkazů může vytvořit cyklus, který se pak označuje jako referenční cyklus. Např. cizí klíč nějaké tabulky T1 se odkazuje na řádky tabulky T2 a ta obsahuje cizí klíč, který se naopak odkazuje na řádky tabulky T1. Může dokonce existovat i tabulka, která obsahuje cizí klíč, který se odkazuje na řádky téže tabulky.

**Příklad 2.7**

Uvažujme tabulky ZAMĚSTNANEC (OS_ČÍSLO, JMÉNO, PŘÍJMENÍ, ODDĚLENÍ, NADŘÍZENÝ) a ODDĚLENÍ (ZKRATKA, NÁZEV, VEDOUcí). OS_ČÍSLO, resp. ZKRATKA jsou primární klíče tabulek. Sloupec ODDĚLENÍ v tabulce ZAMĚSTNANEC je cizím klíčem odkazujícím se na sloupec ZKRATKA tabulky ODDĚLENÍ a udává, ve kterém oddělení zaměstnanec pracuje. Sloupec VEDOUcí v tabulce ODDĚLENÍ je cizím klíčem odkazujícím se na sloupec OS_ČÍSLO tabulky ZAMĚSTNANEC a udává, kdo je vedoucím daného oddělení. Tyto dva odkazy (reference) tvoří cyklus.

Sloupec NADŘÍZENÝ v tabulce ZAMĚSTNANEC je cizím klíčem odkazujícím se na sloupec OS_ČÍSLO téže tabulky. V tomto případě jde o seberefenci.



Častou chybou studentů bývá, že nedokáží vyrovnat se situací, kdy je potřeba použít složený cizí klíč, protože odkazovaný kandidátní klíč je také složený. Uvažujme následující příklad:

**Příklad 2.8**

Uvažujme tabulku ZKUŠEBNÍ_ZPRÁVA (LOGIN, R_CÍSLO, ZKRATKA, AK_ROK, BODY) z příkladu Příklad 2.6. Víme, že jedním ze dvou kandidátních klíčů v této tabulce je složený sloupec (LOGIN, ZKRATKA, AK_ROK). Předpokládejme, že náš informační systém umožňuje provádět opravy zkušební zprávy garantem předmětu, ale takové opravy je třeba v databázi registrovat. Předpokládejme pro tyto účely tabulku OPRAVA (POŘ_ČÍSLO, LOGIN, ZKRATKA, AK_ROK, NOVÉ_BODY). Sloupec POŘ_ČÍSLO je pořadové číslo opravy zkušební zprávy z daného předmětu v daném akademickém roce a hodnota sloupce NOVÉ_BODY udává nově přidělené body. Význam zbývajících tří sloupců je stejný jako v tabulce ZKUŠEBNÍ_ZPRÁVA. Protože je potřeba mít informaci o tom, kterého řádku zkušební zprávy se oprava týká, budou právě tyto tři sloupce tvořit složený cizí klíč v tabulce NOVÉ_BODY, který se bude odkazovat na složený kandidátní klíč (LOGIN, ZKRATKA, AK_ROK).



V tomto příkladě se jedná o jeden cizí klíč, nikoliv trojici cizích klíčů, které se odkazují na odpovídající složky kandidátního klíče. Shodovat se musí celá trojice, nikoliv jenom odpovídající složky.



Skutečnost, že složené primární klíče komplikují do určité míry odkazy (odkazující se tabulka musí zahrnovat všechny sloupce jako cizí klíč) vede, jak již bylo uvedeno, často k praxi zavedení speciálních jednoduchých sloupců s funkcí primárního klíče. V našem příkladě by tabulka ZKUŠEBNÍ_ZPRÁVA mohla navíc obsahovat sloupec ČÍSLO_ZPRAVY, který by obsahoval unikátní číslo řádku zkušební zprávy. SŘBD často podporuje automatické generování takových unikátních čísel. Potom by i tabulka OPRAVA obsahovala jenom odpovídající jednoduchý cizí klíč.

Nyní si můžeme zpřesnit pojem struktura relační databáze, přesněji schéma relační databáze. V předchozí podkapitole jsme si zavedli pojem schématu relace, resp. tabulky. Zatím jsme předpokládali, že schéma relační databáze je potom souhrnem schémat jednotlivých tabulek. Nyní jsme poznali, že důležitým prvkem relační databáze jsou integritní pravidla. Protože podobně jako schémata tabulek popisují i integritní omezení data v databázi, zahrneme je do celkového schématu relační databáze. Formálně bychom mohli schéma relační databáze vyjádřit takto:



Definice 2.7 Schéma relační databáze

Schématem relační databáze nazýváme dvojici (R, I) , kde

$R = \{R_1, R_2, \dots, R_k\}$ je množina schémat relací a $I = \{I_1, I_2, \dots, I_l\}$ je množina integritních omezení.

Někdy jsou lokální integritní omezení rozdělena mezi jednotlivá schémata, tj.

$$R = \{(R_1, I_1), (R_2, I_2), \dots, (R_k, I_k)\}$$

Z integritních omezení nás budou v této souvislosti zajímat zejména omezení týkající se primárních, kandidátních a cizích klíčů a to z toho důvodu, že jejich zajištění je zpravidla přímo podporováno SŘBD. Schématem tabulky budeme tedy rozumět nejen její jméno a výčet sloupců a datových typů, ale i případné vymezení jejího primárního klíče, kandidátních a cizích klíčů, případně dalších integritních omezení vztažených k této tabulce.

Pokud data v databázi splňují všechna integritní omezení, řekneme, že jsou *konzistentní*.



Schéma tabulky jsou data, která popisují vlastnosti samotných dat v tabulce. Taková data, která popisují jiná data, se označují často jako *metadata*. SŘBD je ukládá do systémového katalogu, o kterém jsme se zmínili již na úvodní přednášce při výkladu základních pojmů.



V této podkapitole jsme si vysvětlili, jaká obecná integritní omezení musí platit v každé relační databázi. Jde o dvě integritní omezení. Prvé se týká primárních klíčů a druhé cizích klíčů. Primární klíč je sloupec tabulky (případně složený), který používáme k identifikaci jednotlivých řádků tabulky. Primární klíč je jedním z kandidátních klíčů tabulky. Kandidátním klíčem rozumíme každý sloupec tabulky (případně složený), jehož hodnoty jsou unikátní, tj. nesmí existovat v tabulce dva řádky se stejnou hodnotou v tomto sloupci, a dále musí být minimální (neredukovatelný). Sloupec, který

je primárním klíčem navíc nesmí obsahovat nezadané hodnoty označované často jako NULL. V našem ilustračním příkladu je například jediným kandidátním klíčem a tedy i primárním klíčem tabulky STUDENT sloupec LOGIN.

Cizím klíčem rozumíme sloupec tabulky (opět obecně složený), který slouží k vytváření logických vazeb mezi řádky tabulek. Hodnota v takovém sloupci musí být buď plně zadaná nebo nezadaná. Je-li zadaná, pak se musí shodovat s hodnotou v právě jednom řádku odkazovaného sloupce odkazované tabulky. Tímto sloupcem musí být kandidátní klíč odkazované tabulky. Soulad hodnot cizích klíčů a odkazovaných kandidátních klíčů se nazývá referenční integrita. V našem příkladě je cizím klíčem sloupec LOGIN tabulky PROJEKT a odkazovaným sloupcem sloupec LOGIN tabulky STUDENT.

Zpřesnili jsme si také pojem schéma relační databáze. Zahrnuje jednak schémata jednotlivých tabulek, jednak integritní omezení, která musí být splněna. Pokud data integritní omezení nesplňují, říkáme, že je databáze v nekonzistentním stavu.

Současné SŘBD poskytují zpravidla podporu jak pro integritní omezení týkající se kandidátních a primárních klíčů, tak na podporu zajištění referenční integrity. Odpovídající konstrukce najdeme i v jazyce SQL, kterému se budeme podrobně věnovat později.

2.4. Relační algebra

E.F. Codd již ve svém článku z roku 1970 popsal základní operace nad relací jako jedinou strukturou dat v relační databázi na logické úrovni. Vytvořil tak základ tzv. relační algebry. Množina operací relační algebry se postupně rozšiřovala o další. My zůstaneme u těchto základních operací, jejichž pochopení je důležité pro to, abychom později porozuměli tomu, jak probíhá zpracování zejména dotazů v nějakém databázovém jazyce pro relační databáze, například v SQL.

V kapitole 2.2 jsme viděli, že tělo tabulky relační databáze je definováno jako množina řádků. Je proto přirozené, že součástí relační algebry budou běžné množinové operace, jako je sjednocení, průnik, doplněk a kartézský součin. Budou ale existovat i speciální relační operace, které nám umožní vybrat z tabulky jen určité sloupce, určité řádky a umožní nám spojovat řádky několika tabulek.



Definice 2.8 Relační algebra

Relační algebrou rozumíme dvojici $RA = (R, O)$, kde nosičem R je množina relací a O je množina operací, která zahrnuje:

- tradiční množinové operace (sjednocení, průnik, rozdíl, součin),
- speciální relační operace, mezi které patří projekce, selekce (restrikce), spojení (přirozené).



Důležitou vlastností relační algebry je, že je uzavřená. To znamená, že jak operandy operací, tak jejich výsledky jsou tabulky. Operace jsou unární (nad jednou tabulkou) nebo binární (nad dvěma tabulkami, výsledkem je vždy jedna tabulka).

Všechny množinové operace relační algebry jsou binární. Jejich význam je stejný jako je známe z teorie množin, pouze je potřeba respektovat, že sjednocení, průnik a rozdíl nelze provádět nad libovolnými tabulkami. Když si představíte tabulky STUDENT a PROJEKT z našeho příkladu, pak asi sjednocení řádků těchto tabulek

by sice bylo možné, protože těla tabulek jsou množiny, ale musíme mít na mysli, že tabulka jako relace je tvořena nejen tělem, ale i svým schématem. To by muselo respektovat obecně rozdílná schémata obou tabulek. Proto budeme definovat operace sjednocení, průnik a rozdíl pouze nad tabulkami se stejným schématem, tj. stejnými sloupci.

V případě kartézského součinu už omezení na tvar tabulek, nad kterými operaci provádíme, již není, protože vzniklé řádky zahrnují hodnoty ze všech sloupců obou tabulek jako u klasického kartézského součinu.

DEF

Definice 2.9 Tradiční množinové operace relační algebry

Sjednocením relací **R1** = (*R*, *R1**) a **R2** = (*R*, *R2**) se schématem *R* je relace

$$\mathbf{R1} \text{ union } \mathbf{R2} = (R, R1^* \cup R2^*).$$

Analogicky pro *průnik* (**R1 intersect R2**) a *rozdíl* (**R1 minus R2**).

Kartézským součinem relací **R1** = (*R1*, *R1**) a **R2** = (*R2*, *R2**) je relace

$$\mathbf{R1} \text{ times } \mathbf{R2} = ((R1, R2), R1^* \times R2^*).$$

Slovně bychom mohli například definici pro sjednocení tabulek vyjádřit tak, že výsledkem bude tabulka se stejným schématem jako je schéma tabulek, nad kterými se provádí, a tělo bude tvořeno těmi řádky, které se vyskytují v alespoň jedné z obou tabulek.

x+y

Příklad 2.9

Uvažujme tabulky T1 a T2.

T1

A	B	C
0	a	d
1	a	e
2	b	f

T2

A	B	C
2	c	d
0	a	d
0	a	e

Výsledky operací sjednocení, průniku pak budou

T1 union T2

A	B	C
0	a	d
1	a	e
2	b	f
2	c	d
0	a	e

T1 intersect T2

A	B	C
0	a	d

T1 minus T2

A	B	C
1	a	e
2	b	f

Výsledek operace kartézského součinu (pro rozlišení sloupců tabulek mají ve výsledku sloupce koncovku T1, resp. T2) bude

T1 times T2

AT1	BT1	CT1	AT2	BT2	CT2
0	a	d	2	c	d
0	a	d	0	a	d
0	a	d	0	a	e
1	a	e	2	c	d
...

x+y

Příklad 2.10

Uvažujme následující tabulky

STUDENT

LOGIN	JMÉNO	PŘÍJMENÍ	ADRESA
xcerny00	Petr	Černý	Brněnská 15 Vyškov
xnovak00	Jan	Novák	Cejl 9 Brno
xnovak01	Pavel	Novák	Cejl 9 Brno

STUDENT_Z_BRNA

LOGIN	JMÉNO	PŘÍJMENÍ	ADRESA
xnovak00	Jan	Novák	Cejl 9 Brno
xnovak01	Pavel	Novák	Cejl 9 Brno

STUDENT_MIMO_BRNO

LOGIN	JMÉNO	PŘÍJMENÍ	ADRESA
xcerny00	Petr	Černý	Brněnská 15 Vyškov

Výsledkem sjednocení tabulek STUDENT_Z_BRNA a STUDENT_MIMO_BRNO bude tabulka

STUDENT_Z_BRNA *union* STUDENT_MIMO_BRNO

LOGIN	JMÉNO	PŘÍJMENÍ	ADRESA
xnovak00	Jan	Novák	Cejl 9 Brno
xnovak01	Pavel	Novák	Cejl 9 Brno
xcerny00	Petr	Černý	Brněnská 15 Vyškov

Výsledkem průniku tabulek STUDENT_Z_BRNA a STUDENT bude tabulka

STUDENT_Z_BRNA *intersect* STUDENT

LOGIN	JMÉNO	PŘÍJMENÍ	ADRESA
xnovak00	Jan	Novák	Cejl 9 Brno
xnovak01	Pavel	Novák	Cejl 9 Brno

Výsledkem rozdílu tabulek STUDENT a STUDENT_Z_BRNA bude tabulka

STUDENT *minus* STUDENT_Z_BRNA

LOGIN	JMÉNO	PŘÍJMENÍ	ADRESA
xcerny00	Petr	Černý	Brněnská 15 Vyškov

Všimněte si, že pokud bychom předpokládali, že v tabulce STUDENT_Z_BRNA, resp. STUDENT_MIMO_BRNO z předchozího příkladu jsou informace o všech studentech s trvalým bydlištěm v Brně, resp. mimo Brno, pak můžeme chápat tabulku STUDENT_Z_BRNA *union* STUDENT_MIMO_BRNO jako výsledek dotazu „Najdi všechny studenty fakulty“. Analogicky tabulka STUDENT *minus* STUDENT_Z_BRNA je výsledkem dotazu „Najdi studenty, kteří nebydlí v Brně“.



Už teď jsme došli k důležitému poznatku, že výrazy relační algebry lze použít jako dotazovací jazyk nad relační databází.

Nyní se podívejme na speciální relační operace relační algebry. Už víme, že máme tři takové operace – projekci, selekci (označovanou také jako restrikce) a spojení. Projekce a selekce jsou operace unární, spojení je operací binární.

Výsledkem projekce bude tabulka, která bude obsahovat jenom některé ze sloupců původní tabulky. Formální definice by mohla vypadat takto:

DEF**Definice 2.10** Projekce

Projekce relace $R = (R, R^*)$ na atributy X, Y, \dots, Z je relace

$$R[X, Y, \dots, Z]$$

se schématem (X, Y, \dots, Z) a tělem zahrnujícím všechny n -tice $t = (x, y, \dots, z)$ takové, že v R^* existuje n -tice t' s hodnotou atributu X rovnou x , Y rovnou y , ... Z rovnou z .

 $x+y$ **Příklad 2.11**

Uvažujme tabulku

T				
A	B	C	D	E
0	a	x	3	1
1	b	y	6	2
0	b	y	4	2

Projekce tabulky T na tabulku se sloupci B, C, E bude tabulka

T[B, C, E]		
B	C	E
a	x	1
b	y	2



Všimněte si, že výsledná tabulka může mít méně řádků než tabulka původní. Jak to vysvětlíte?

 $x+y$ **Příklad 2.12**

Uvažujme tabulku STUDENT z příkladu Příklad 2.10

STUDENT

LOGIN	JMÉNO	PŘÍJMENÍ	ADRESA
xcerny00	Petr	Černý	Brněnská 15 Vyškov
xnovak00	Jan	Novák	Cejl 9 Brno
xnovak01	Pavel	Novák	Cejl 9 Brno

a její projekci na sloupce LOGIN, JMÉNO, PŘÍJMENÍ

STUDENT[LOGIN, JMÉNO, PŘÍJMENÍ]

LOGIN	JMÉNO	PŘÍJMENÍ
xcerny00	Petr	Černý
xnovak00	Jan	Novák
xnovak01	Pavel	Novák

která je výsledkem dotazu „Najdi přihlašovací jméno, křestní jméno a příjmení všech studentů“.

Výsledkem operace selekce bude narozdíl od projekce tabulka, která bude zahrnovat všechny sloupce, ale obecně jen některé řádky. Budou to ty řádky, které vyhovují zadané podmínce.

DEF

Definice 2.11 Selektce

Nechť θ je operátor porovnání dvou hodnot ($<$, $>$, $<=$, $=$, atd.). θ selektce (restrikce) relace $R = (R, R^*)$ na attributech X a Y je relace

$$R \text{ where } X \theta Y,$$

která má stejné schéma jako relace R a obsahuje všechny n -tice $t \in R^*$, pro které platí $x \theta y$, kde x je hodnota atributu X a y hodnota atributu Y v n -tici t . Na místě atributu X , resp. Y může být konstanta. Pak jde o θ selekci na atributu X , resp. Y .

x+y

Příklad 2.13

Uvažujme tabulku

T

A	B	C	D	E
0	a	x	3	1
1	b	y	6	2
0	b	y	4	0

Selektce tabulky T pro podmínku $A < B$ bude tabulka

T where $A < B$

A	B	C	D	E
1	b	y	6	2
0	b	y	4	2

Selektce tabulky T pro podmínku $A = 0$ bude tabulka

T where $A = 0$

A	B	C	D	E
0	a	x	3	1
0	b	y	4	2

Selektce umožňuje podle definice pouze výběr řádků na základě jednoduché podmínky porovnání hodnot ve sloupci, resp. porovnání s konstantou. Není ale problém rozšířit tvar podmínky tak, aby bylo možné používat logické spojky *and*, *or* a negaci *not*, jak jsme zvyklí u logických výrazů. Využijeme k tomu množinových operací relační algebry.

Například pro použití spojky *and* pro spojení podmínek c_1 a c_2 v selekci tabulky T můžeme psát

$$T \text{ where } c_1 \text{ and } c_2 \equiv (T \text{ where } c_1) \text{ intersect } (T \text{ where } c_2)$$

Protože logický výraz $c_1 \text{ and } c_2$ nabývá hodnoty TRUE pouze při současném splnění obou podmínek, bude výsledná tabulka obsahovat pouze ty řádky T, které splňují jak c_1 , tak c_2 . A ty jsou právě výsledkem průniku selektce T pro c_1 a pro c_2 .

x+y

Příklad 2.14

Uvažujme tabulku STUDENT z příkladu Příklad 2.10

STUDENT

LOGIN	JMÉNO	PŘÍJMENÍ	ADRESA
xcerny00	Petr	Černý	Brněnská 15 Vyškov
xnovak00	Jan	Novák	Cejl 9 Brno
xnovak01	Pavel	Novák	Cejl 9 Brno

Dotazu „Který student má přihlašovací jméno xnovak00?“ by odpovídal výraz relační algebry STUDENT *where* LOGIN = 'xnovak00' s výsledkem

STUDENT *where* LOGIN = 'xnovak00'

LOGIN	JMÉNO	PŘÍJMENÍ	ADRESA
xnovak00	Jan	Novák	Cejl 9 Brno



V předchozím příkladu jsme uzavřeli řetězec znakové konstanty (xnovak00) do apostrofů, abychom ji odlišili od názvů tabulek a sloupců. Tento způsob budeme v podobných situacích používat i v dalších příkladech.

Poslední ze speciálních relačních operací je spojení, přesněji přirozené spojení. Umožňuje nám spojovat řádky dvou tabulek na základě stejné hodnoty ve stejné pojmenovaných sloupcích. Formálně bychom ji mohli definovat takto:



Definice 2.12 Přirozené spojení (natural join)

Nechť $R1 = (R1, R1^*)$ je relace se schématem $R1(X1, X2, \dots, Xm, Y1, Y2, \dots, Yn)$ a $R2$ relace se schématem $R2(Y1, Y2, \dots, Yn, Z1, Z2, \dots, Zk)$. Uvažujme složené atributy $X=(X1, X2, \dots, Xm)$, $Y=(Y1, Y2, \dots, Yn)$ a $Z=(Z1, Z2, \dots, Zk)$.

Potom *přirozené spojení* relací $R1$ a $R2$ je relace

$R1$ join $R2$

se schématem (X, Y, Z) a tělem zahrnujícím všechny n -tice $t = (x, y, z)$ takové, že v $R1^*$ existuje n -tice t' s hodnotou x atributu X a hodnotou y atributu Y a v $R2^*$ existuje n -tice t'' s hodnotou y atributu Y a hodnotou z atributu Z .

Definice vypadá složitě, ale když ji pozorně přečtete a zamyslíte se nad ní, zjistíte, že vyjadřuje matematicky přesněji to, co už o této operaci víme, že výsledná tabulka bude zahrnovat všechny kombinace řádků obou tabulek, které mají stejné hodnoty ve stejné pojmenovaných sloupcích. Všimněte si také, že výsledná tabulka zahrnuje všechny sloupce obou tabulek s tím, že stejné pojmenované sloupce tam jsou pouze jedenkrát.

**Příklad 2.15**

Uvažujme tabulky T1 a T2

T1

A	B	C
0	a	d
1	a	e
2	b	f

T2

C	D	E
e	1	0
d	1	1
d	0	1

Výsledkem jejich přirozeného spojení bude tabulka

T1 *join* T2

A	B	C	D	E
0	a	d	1	1
0	a	d	0	1
1	a	e	1	0

Spojení probíhá na základě rovnosti hodnot ve sloupci C. Prvý řádek tabulky T1 se proto spojí s druhým a třetím řádkem tabulky T2 a vzniknou tak prvé dva řádky výsledné tabulky. Třetí řádek výsledku vznikne spojením druhého řádku tabulky T1 a prvního řádku tabulky T2. Poslední řádek tabulky T1 se nespojí s žádným řádkem tabulky T2, a proto se jeho hodnoty ve výsledné tabulce neobjeví.

Spojení tabulek je velice důležitou operací relační algebry. V kapitole 2.3 jsme si vysvětlili, že mezi tabulkami relační databáze jsou často logické vazby na základě shody hodnoty cizího klíče a hodnoty primárního klíče odkazované tabulky. Na základě těchto hodnot se potom nejčastěji řádky tabulek spojují v případech, kdy nelze získat výsledek dotazu operací nad jednou tabulkou.

x+y

Příklad 2.16

Uvažujme tabulky STUDENT a PROJEKT z příkladu Příklad 2.5

STUDENT

LOGIN	JMÉNO	PŘÍJMENÍ	ADRESA
xcerny00	Petr	Černý	Brněnská 15 Vyškov
xnovak00	Jan	Novák	Cejl 9 Brno
xnovak01	Pavel	Novák	Cejl 9 Brno

PROJEKT

ČÍSLO	NÁZEV	ZADÁNÍ	LOGIN
1	Internetový obchod	Realizujte ...	xnovak01
2	OO databáze	Seznamte se s ...	xcerny00
3	Řízené gramatiky	Připravte ...	xnovak01
4	Lexikální analyzátor	Naprogramujte ...	

Předpokládejme, že chceme znát přihlašovací jméno, jméno a příjmení studenta, resp. studentů (může jich být více?), kteří řeší projekt s názvem Internetový obchod. Tento dotaz ale nejsme schopni vyřešit operacemi nad jednou tabulkou. Název projektu, který nás zajímá, je v tabulce PROJEKT, zatímco jméno a příjmení studenta v tabulce STUDENT. Potřebujeme proto spojit informace uložené v těchto dvou tabulkách. Výraz relační algebry, který zodpoví daný dotaz tedy bude obsahovat operaci přirozeného spojení, selekci (zajímají nás pouze řešitelé projektu Internetový obchod) a projekci (zajímají nás pouze hodnoty ve sloupcích LOGIN, JMÉNO, PŘÍJMENÍ). Mohl by mít tvar:

((STUDENT *join* PROJEKT) *where* NÁZEV = ' Internetový obchod ')
[LOGIN, JMÉNO, PŘÍJMENÍ]

Ukážeme si postupně, jak vznikne výsledek. Mezivýsledek po přirozeném spojení bude tabulka zahrnující všechny sloupce tabulek STUDENT a PROJEKT (sloupec LOGIN ale jenom jedenkrát) a bude mít tři řádky.

STUDENT *join* PROJEKT

LOGIN	JMÉNO	PŘÍJMENÍ	ADRESA	ČÍSLO	NÁZEV	...
xcerny00	Petr	Černý	Brněnská 15 Vyškov	2	OO databáze	...
xnovak01	Pavel	Novák	Cejl 9 Brno	1	Internetový obchod	...
xnovak01	Pavel	Novák	Cejl 9 Brno	3	Řízené gramatiky	...

Následná selekce vybere z této tabulky řádek týkající se projektu s názvem Internetový obchod.

((STUDENT *join* PROJEKT) *where* NÁZEV = ' Internetový obchod ')

LOGIN	JMÉNO	PŘÍJMENÍ	ADRESA	ČÍSLO	NÁZEV	...
xnovak01	Pavel	Novák	Cejl 9 Brno	1	Internetový obchod	...

A provedeme projekci této tabulky na sloupce, jejichž hodnoty nás zajímají.

((STUDENT *join* PROJEKT) *where* NÁZEV = ' Internetový obchod ') [LOGIN, JMÉNO, PŘÍJMENÍ]

LOGIN	JMÉNO	PŘÍJMENÍ
xnovak01	Pavel	Novák



Možná vás u předchozího příkladu napadlo, že jsme výsledek získali zbytečně komplikovaně. Například jsme zbytečně spojovali řádky tabulek STUDENT a PROJEKT, které se na požadovaném výsledku nijak nepodílejí. Efektivnějšího dosažení výsledku bylo možné dosáhnout jiným pořadím prováděných operací. Pokud vás toto napadlo, pak máte pravdu a zasloužíte pochvalu. Praktické využití této vlastnosti relační algebry si vysvětlíme v závěru této podkapitoly.



Relační algebra zavádí pouze přirozené spojení řádků tabulek, tj. na základě rovnosti hodnot ve všech stejně pojmenovaných sloupcích obou tabulek. Bylo by ale možné spojovat řádky i na základě obecnějších podmínek, ve sloupcích s různými jmény apod. Proto existují ještě další druhy operace spojení, se kterými se seznámíme při výkladu jazyka SQL.



V literatuře se často používají pro operace relační algebry následující symboly:

$\sigma_{\theta}(R)$	$R \text{ where } \theta$
$\Pi_{X,Y}(R)$	$R[X, Y]$
$R \bowtie S$	$R \text{ join } S$

Nyní se zkusme zamyslet nad tím, jestli jsou všechny operace relační algebry potřebné v tom smyslu, že je nelze vyjádřit pomocí jiných operací relační algebry. Uvažujme například operaci přirozeného spojení tabulek T1(A, B, C) a T2(C, D, E). Zřejmě platí, že

$$T1 \text{ join } T2 = ((T1 \text{ times } T2) \text{ where } C_{T1} = C_{T2}) [A, B, C, D, E]$$

Kartézský součin vytvoří všechny kombinace řádků obou tabulek, selekce vybere ty, které mají stejnou hodnotu ve sloupci C (použili jsme index se jménem tabulky pro rozlišení), a projekce zajistí pouze to, že sloupec C bude ve výsledku jen jedenkrát.

Podobně bychom mohli najít ekvivalentní výraz pro vyjádření operace průniku. U ostatních operací by se nám to nepodařilo. Můžeme tedy udělat závěr, že operace sjednocení, rozdíl, kartézský součin, projekce a selekce stačí k vyjádření všech operací s tabulkami, které jsme si uvedli. Označují se proto jako *minimální množina operací relační algebry*. Proč byly tedy zavedeny zbývající operace? Důvodem je použití v podobných algebrách (průnik) nebo praktický význam (spojení).

Již v úvodu této podkapitoly jsme uvedli, že relační algebra se postupem času vyvíjela. Byly do ní zejména přidávány další operace, např. operace pro přejmenování sloupce tabulky, pro agregační funkce (poznáme je v souvislosti s jazykem SQL), operace přiřazení apod. Potřebu přejmenovat sloupce jsme viděli i u předchozí diskuse o minimální množině operací.

Závěrem této podkapitoly si uvedeme, v čem spočívá význam relační algebry. Proč jsme si ji vůbec uváděli? Význam lze spatřovat v následujících dvou oblastech:

- je referenčním prostředkem pro hodnocení vlastností a porovnání relačních dotazovacích jazyků
- je vhodným základem pro optimalizaci zpracování dotazů.

Prvá oblast představuje význam především pro teorii. Vychází z toho, že výrazy relační algebry tvoří dotazovací jazyk a vyjadřovací síla jiných dotazovacích jazyků se porovnává právě s jazykem výrazů relační algebry. Každý databázový jazyk, kterým lze vyjádřit alespoň totéž, co operacemi relační algebry, se nazývá *relačně úplný* (*relationally complete*).

Druhá oblast naopak představuje význam pro implementaci SŘBD pro relační databáze. Zde se využívá toho, že jazyk výrazů relační algebry je procedurálním dotazovacím jazykem. Definuje tedy operace a jejich pořadí, kterými získáme požadovaný výsledek. V komentáři k příkladu Příklad 2.16 jsme si už uvedli, že výraz, který jsme zvolili pro zodpovězení daného dotazu, tj.

```
((STUDENT join PROJEKT) where NÁZEV = ' Internetový obchod ')\n[LOGIN, JMÉNO, PŘÍJMENÍ]
```

nebude s velkou pravděpodobností nejvýhodnějším pořadím operací, protože operace spojení, která je prováděna jako první, vytvoří mezivýsledek s obecně mnoha řádky, které nás nezajímají, protože se netýkají projektů s názvem Internetový obchod. Výhodnější by byl například výraz

```
(STUDENT join (PROJEKT where NÁZEV = ' Internetový obchod '))\n[LOGIN, JMÉNO, PŘÍJMENÍ]
```

protože ten nejprve vybere z tabulky PROJEKT řádky týkající se projektů, které nás zajímají (v našem příkladu to byl řádek jeden) a pouze ty spojuje. Fakt, že operace spojení bude prováděna s podstatně menším operandem, může umožnit použít efektivní implementaci této operace pracující pouze v operační paměti.

Úlohu nalezení co nejefektivnějšího způsobu provedení příkazu databázového jazyka řeší komponenta SŘBD nazývaná optimalizátor dotazu. Jedním z přístupů, které se při optimalizaci používají, je právě nalezení co nevhodnějšího výrazu relační algebry, který reprezentuje zadaný příkaz, například dotaz v SQL.



Kromě výrazů relační algebry jako procedurálního dotazovacího jazyka vznikly už v 70. letech i neprocedurální formální jazyky pro dotazování nad relační databází. Jsou to tzv. *relační kalkuly*. Jde o dotazovací jazyk na bázi logiky. Obsahuje formule

logiky 1. řádu, tj. takové, ve kterých se vyskytují proměnné, konstanty, operátory porovnání, operátor negace, logické spojky a kvantifikátory. Podle toho, zda jsou proměnné definovány na řádcích tabulek, tj. n-ticích relací, nebo hodnotách ve sloupcích, tj. hodnotách domény atributu se rozlišují dva typy relačního kalkulu:

- n-ticový relační kalkul
- doménový relační kalkul.

Např. v n-ticovém relačním kalkulu by dotaz „Najdi studenty řešící projekt s názvem Internetový obchod.“ z příkladu Příklad 2.16 mohl mít tvar

$$\{s | s \in \text{STUDENT} \wedge \exists p \in \text{PROJEKT} (s.\text{LOGIN} = p.\text{LOGIN} \wedge p.\text{NÁZEV} = \text{'Internetový obchod'}) \}$$

Pro jednoduchost jsme zde předpokládali, že chceme kompletní informaci o studentovi, tedy i adresu.

Uvedený výraz bychom mohli číst: „Výsledkem bude množina řádků tabulky STUDENT (hodnot proměnné s), pro které existuje v tabulce PROJEKT řádek (hodnota proměnné p) takový, že obsahuje stejnou hodnotu ve sloupci LOGIN a ve sloupci NÁZEV je hodnota 'Internetový obchod'.“ Vidíme, že výraz neříká, jakými operacemi se má výsledek získat, definuje pouze vlastnosti požadovaného výsledku. Proto patří jazyk takových výrazů mezi neprocedurální jazyky.

Jestli již znáte jazyk SQL, se kterým se v tomto předmětu budeme seznamovat, víte, že bychom takový dotaz mohli zapsat jako příkaz tvaru

```
SELECT *
FROM STUDENT s, PŘEDMĚT p
WHERE s.LOGIN = p.LOGIN AND p.NÁZEV = 'Internetový obchod'
```

který můžeme považovat za syntakticky jinak zapsaný výše uvedený výraz n-ticového relačního kalkulu. A skutečně můžeme říci, že SQL implementuje n-ticový relační kalkul.



V této kapitole jsme si vysvětlili základní operace nad daty v relační databázi. Uvedli jsme si relační algebru, která tyto operace definuje. Jsou to jednak tradiční množinové operace (sjednocení, průnik, rozdíl a kartézský součin), jednak speciální relační operace, mezi které patří projekce, selekce a spojení. Všechny tyto operace mají jako operandy tabulky (relace, odtud název algebry) a také výsledkem každé operace je tabulka.

Význam množinových operací je takový, jak ho známe z teorie množin s tím, že operace sjednocení, průniku a rozdílu lze provádět pouze s tabulkami, které mají stejné schéma. Operace projekce a selekce jsou unární operace. Prvá vybírá z tabulky zadané sloupce, druhá vybírá řádky splňující zadanou podmínku. Podmínkou může být porovnání hodnot ve sloupcích, porovnání hodnoty ve sloupci s konstantou, negovaná podmínka nebo několik podmínek spojených logickými spojkami pro logický součin a součet.

Důležitou relační operací je spojení, které umožňuje spojovat řádky dvou tabulek. Relační algebra definuje tzv. přirozené spojení, u kterého dochází ke spojení řádků na základě rovnosti hodnot ve stejně pojmenovaných sloupcích. V praxi se nejčastěji tabulky spojují na základě rovnosti hodnot cizího klíče a odkazovaného primárního klíče.

Dále jsme si zavedli pojem minimální množina operací relační algebry, kterou tvoří

sjednocení, rozdíl, kartézský součin, projekce a selekce a vysvětlili jsme si, že jazyk výrazů relační algebry lze chápat jako dotazovací jazyk pro relační databáze.

Relační algebra se používá pro porovnávání vyjadřovací síly dotazovacích jazyků relační databáze. Databázový jazyk, kterým lze vyjádřit alespoň totéž, co operacemi relační algebry, se nazývá relačně úplný. Relační algebra se rovněž využívá při optimalizaci zpracování dotazů. Zde se využívá toho, že jazyk výrazů relační algebry je procedurálním dotazovacím jazykem a je tedy vhodnou reprezentací neprocedurálních databázových jazyků (např. SQL) při hledání efektivního způsobu provedení příkazů.



Informace k relačnímu modelu dat můžete najít v základní literatuře [Pok02] na stranách 17 až 45. Je zde uvedena základní definice relačního modelu dat zahrnující vysvětlení základních pojmů relační datové struktury. Další dvě kapitoly jsou definovány relační algebrou a relačním kalkulům. V [Pok98] je relační model dat stručně popsán na stranách 141 až 144. Učebnice [Sil05] vysvětluje relační model dat na str. 37 až 73. Nejrozsáhleji je relační model popsán v doporučené učebnici [Dat03]. V části pojednávající o relační algebře C.J. Date uvádí i některé další operace, které jsme jen zmínili nebo neuváděli vůbec. Již jsme také uvedli, že autor zde, narozdíl od nás, definuje relaci ne pomocí schématu, nýbrž záhlaví chápaného jako množina (neuspořádaná) atributů. Potom ani tělo relace přesně neodpovídá klasickému matematickému pojetí a n-tice jsou definovány jako množiny dvojic (atribut, hodnota atributu). V obou učebnicích je uvedena řada otázek a příkladů k procvičení.



Cvičení:

- C2.1** Vysvětlete, co znamená, že tabulka v relační databázi musí být normalizovaná, tedy alespoň v tzv. 1NF.
- C2.2** Vysvětlete, jaké výhody přináší omezení, že tabulka relační databáze musí být normalizovaná a jaké naopak přináší problémy.
- C2.3** Vysvětlete, proč se relační databáze nazývají zrovna relační.
- C2.4** Vysvětlete pojem schéma tabulky (relace)?
- C2.5** Může tabulka relační databáze podle teorie relačního modelu obsahovat dva stejné řádky? Svoje tvrzení zdůvodněte.
- C2.6** Vysvětlete rozdíl mezi obecnými integritními omezeními daného databázového modelu a omezeními specifickými a uveďte, jaká obecná integritní omezení musí platit v relačním modelu dat.
- C2.7** Jaké vlastnosti musí splňovat sloupec tabulky, který je kandidátním klíčem?
- C2.8** Když si necháte vypsát obsah nějaké tabulky relační databáze a uvidíte, že se v některém sloupci nevyskytují stejné hodnoty, můžete s jistotou prohlásit, že tento sloupec je kandidátním klíčem dané tabulky? Svoje tvrzení zdůvodněte.
- C2.9** Obsahuje každá tabulka relační databáze podle teorie relačního modelu nějaký kandidátní klíč? Svoje tvrzení zdůvodněte.
- C2.10** Jaké vlastnosti musí splňovat sloupec tabulky, který je primárním klíčem?
- C2.11** Čím se liší vlastnosti sloupce, který je primárním sloupcem, od vlastností kandidátního klíče?
- C2.12** Vysvětlete pojem prázdná hodnota (NULL).
- C2.13** Vysvětlete, proč nesmí sloupec, který je primárním klíčem, obsahovat v žádném řádku tabulky prázdnou hodnotu.
- C2.14** Pokud je primární klíč složený, může obsahovat prázdné hodnoty jen

v některých svých sloupcích, ale ne ve všech současně? Svoje tvrzení zdůvodněte.

C2.15 Jaké vlastnosti musí splňovat sloupec tabulky, který je cizím klíčem?

C2.16 Vysvětlete pojem referenční integrita.

C2.17 Může mít jedna tabulka více cizích klíčů? Svoje tvrzení zdůvodněte.

C2.18 Jaké kandidátní klíče obsahuje tabulka OPRAVA z příkladu Příklad 2.8?

C2.19 Vysvětlete, proč nemohou být jednotlivé sloupce LOGIN, ZKRATKA, AK_ROK cizími klíči v tabulce OPRAVA z příkladu Příklad 2.8 odkazujícími se na stejně pojmenované sloupce v tabulce ZKUŠEBNÍ_ZPRÁVA.

C2.20 Vysvětlete slovně operaci rozdíl relační algebry.

C2.21 Může být výsledkem operace selekce relační algebry neprázdné tabulky tabulka prázdná? Svoje tvrzení zdůvodněte.

C2.22 Uveďte výraz obsahující zavedené základní operace relační algebry, který je ekvivalentní výrazu $T \text{ where } c1 \text{ and } c2$, kde T je tabulka, $c1$ a $c2$ jsou podmínky pro operaci selekce a *and* je logická spojka.

C2.23 Uveďte výraz obsahující zavedené základní operace relační algebry, který je ekvivalentní výrazu $T \text{ where not } c$, kde T je tabulka, c je podmínka pro operaci selekce a *not* značí negaci.

C2.24 Uvažujte následující tabulky

T1

A	B
a	0
a	1
b	2

T2

B	C	D
1	x	0
1	y	0
3	y	1

a použijte je k vysvětlení operace spojení relační algebry.

C2.25 Uvažujte množinové operace relační algebry a pro každou z nich určete, zda je komutativní a asociativní.

C2.26 Může být výsledkem operace projekce relační algebry neprázdné tabulky tabulka prázdná? Svoje tvrzení zdůvodněte.

C2.27 Vysvětlete, co označuje minimální množina operací relační algebry a které operace do ní patří.

C2.28 Vyjádřete průnik dvou tabulek $T1$ *intersection* $T2$ pomocí operace rozdílů.

C2.29 Předpokládejte existenci další operace relační algebry $T \text{ rename } A \text{ as } B$, kde A je jméno sloupce tabulky T a B je nové jméno sloupce. Výsledkem je tabulka se stejným schématem jako původní tabulka T s výjimkou sloupce A , který je přejmenován na B .

Uvažujte tabulky $T1(A, B, C)$ a $T2(C, D, E)$ a výraz relační algebry

$((T1 \text{ rename } C \text{ as } C1) \text{ times } T2) \text{ where } C1=C [A, B, C]$

Najděte ekvivalentní výraz relační algebry, který bude obsahovat jen operaci spojení.

C2.30 Vysvětlete, v čem spočívá význam relační algebry pro optimalizaci zpracování dotazů u relačních SRBD.

C2.31 Vysvětlete, kdy můžeme o nějakém databázovém jazyku říci, že je relačně

úplný.

C2.32 Uvažujte tabulky STUDENT(LOGIN, JMÉNO, PŘÍJMENÍ, ADRESA) a PROJEKT(ČÍSLO, NÁZEV, ZADÁNÍ, LOGIN) z příkladu Příklad 2.5. Vyjádřete slovně dotazy vyjádřené následujícími výrazy relační algebry:

- a) STUDENT *where* JMÉNO = 'Novák' [ADRESA]
- b) (STUDENT join PROJEKT) *where* JMÉNO = 'Novák' [ZADÁNÍ]
- c) (STUDENT join PROJEKT) *where* NÁZEV = 'E-obchod' [JMÉNO]

C2.33 Uvažujte tabulky STUDENT(LOGIN, JMÉNO, PŘÍJMENÍ, ADRESA) a PROJEKT(ČÍSLO, NÁZEV, ZADÁNÍ, LOGIN) z příkladu Příklad 2.5. Vyjádřete výrazy relační algebry následující dotazy:

- a) Kdo (stačí znát přihlašovací jméno) řeší projekt číslo 251?
- b) Kdo (chceme znát přihlašovací jméno, jméno a příjmení) řeší projekt číslo 251?
- c) Které dvojice studentů (stačí znát přihlašovací jména) řeší stejný projekt?



Test

T1. Doména v relačním modelu může obsahovat pouze:

- a) numerické, textové a binární hodnoty
- b) libovolné hodnoty - podmínkou je, že hodnota musí být skalární (atomická)

T2. K porušení referenční integrity může dojít při těchto operacích s odkazující se tabulkou:

- a) vkládání, rušení a modifikaci
- b) rušení a modifikaci

T3. Matematický pojem n -nární relace označuje

- a) kartézský součin n množin
- b) podmnožinu kartézského součinu n množin

T4. Primární klíč tabulky je:

- a) jeden z kandidátních klíčů
- b) nejjednodušší kandidátní klíč

T5. Výsledek operace relační algebry selekce relace **R**

- a) má právě tolik n -tic, co relace **R**
- b) má nejvýše tolik n -tic, co relace **R**

3. Návrh relační databáze



Cíl: V této kapitole se seznámíte s postupy používanými při návrhu tabulek relační databáze.

Anotace: Konceptuální modelování, logický návrh databáze, fyzický návrh databáze, entita, entitní množita, vztah, vztahová množina, atribut, kardinalita vztahu, generalizace/specializace, slabá entitní množina, ER model, ER diagram, transformace ER diagramu na tabulky, teorie závislostí, normalizace tabulky.

Prerekvizitní znalosti: V této kapitole navážeme na některá témata, kterými jste se již zabývali v předmětu Úvod do softwarového inženýrství (IUS), konkrétně na etapy vývoje programových systémů a zejména na modelovací techniky používané při analýze požadavků. Předpokládá se zejména znalost základů datového (ER diagram) a funkčního (diagram datových toků) modelování jako základních technik strukturovaného přístupu a základů diagramu tříd a dalších modelů (zejména modelu případů použití) objektového přístupu k analýze požadavků. Dále se předpokládá znalost pojmů zobrazení a funkce z matematiky a znalost relačního modelu dat v rozsahu kapitoly 2 této studijní opory.



Odhad doby studia: 10 hodin.

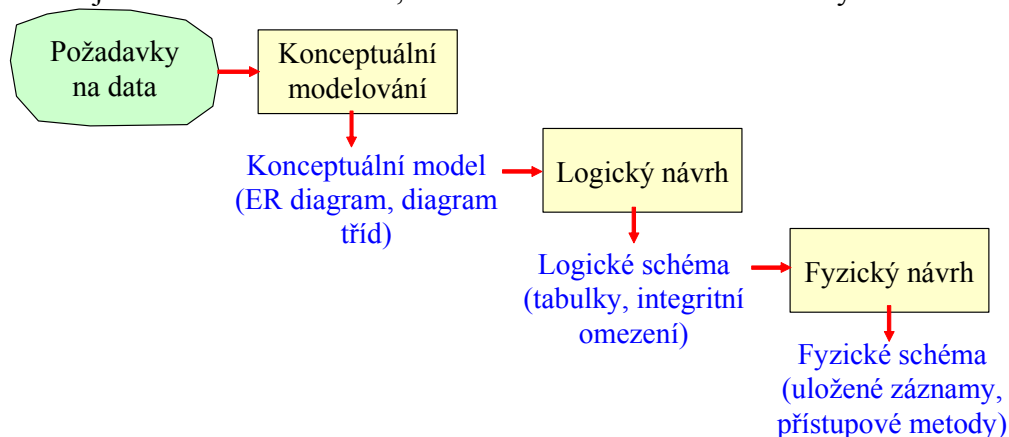
3.1. Úvod

Již ve svém článku, ve kterém E.F.Codd prezentoval základy relačního modelu dat uvedl také základy návrhu relační databáze. Ukázal, jakým způsobem by se reprezentovaly záznamy hierarchické či síťové databáze v databázi relační. Postupně se tyto základy rozvinuly do postupu, který se, podobně jako samotný relační model, opírá o silné matematické zázemí. Základní myšlenka spočívá v tom, že kvalita návrhu tabulek relační databáze je vyjádřena pomocí tzv. *normálních forem*. Základními kritérii kvality přitom je vyloučení redundance a co nejjednodušší kontrola dodržení integritních omezení, která vyplývají ze závislostí mezi informacemi uloženými v tabulce. Pokud použijeme náš ilustrační příklad modulu studijní agendy informačního systému fakulty, potom správně navržená tabulka, ve které budou informace o studentech, bude taková, kde informace o jednom studentovi bude na jednom řádku. Pokud se jistá informace o studentovi (např. jak se jmenuje student s přihlašovacím jménem xnovak00) může společně s identifikací studenta opakovat na několika řádcích nějaké tabulky, pak taková tabulka není navržena správně a uvidíme, že se toto projeví tím, že tabulka bude v nižší normální formě, než je vyžadována pro dobrý návrh. Postup zkvalitňování návrhu tabulek využitím normálních forem se nazývá *normalizace*.

Z předmětu Úvod do softwarového inženýrství víte, že v 70. letech minulého století se dostalo do popředí zájmu hledání metodologií, metod a nástrojů na podporu tvorby rozsáhlých softwarových systémů. Velký důraz se zde klade na využití modelování v průběhu vývoje. Protože takové systémy velice často pracují s perzistentními daty ukládanými do databáze spravované systémem řízení báze dat, vznikaly metody a

odpovídající nástroje pro modelování požadavků na data ukládaná v databázi. Teorie relačního modelu dat, implementace prvních relačních SRBD v polovině 70. let a postupná převaha relačních databází ovlivnily i vznikající metody vývoje programů.

Mezi základní kroky, kterými procházíme při vývoji softwarového systému patří analýza požadavků, návrh, konstrukce (programování), testování a předání do užívání. Tyto kroky mohou po sobě bezprostředně následovat nebo mohou obsahovat cykly v závislosti na použitém modelu životního cyklu. Pokud se zaměříme pouze na kroky související s návrhem databáze, můžeme rozlišit tři základní kroky znázorněné na



Obr. 3.1 Základní kroky návrhu relační databáze

Konceptuální modelování, které se v databázové terminologii označuje někdy také jako konceptuální návrh, patří do etapy analýzy požadavků. Jeho cílem je analyzovat požadavky na data, která budou uložena v databázi. Z hlediska klasifikace datových modelů, které jsme si uvedli na úvodní přednášce, patří mezi modely založené na objektech (konceptech) aplikační domény, pro kterou softwarový systém vyvíjíme. Jestliže budeme vyvíjet modul studijní agendy fakultního informačního systému, budeme modelovat data reprezentující studenty, projekty, předměty a vztahy mezi nimi. Někdy se tato úroveň modelování také označuje jako *sémantické modelování*.

Výsledkem konceptuálního modelování je konceptuální model. Významným krokem v oblasti konceptuálního modelování bylo v polovině 70. let zavedení tzv. *entitně-vztahového (entity-relationship) modelu*, který se postupně stal základním modelem reprezentujícím požadavky na data uložená v databázi, z něhož vychází návrh tabulek relační databáze. Prezентujeme ho zpravidla v podobě ER diagramu. Jde o model, který je představitelem modelovacích technik strukturované analýzy. Podobnou roli plní v případě objektově-orientované analýzy *diagram tříd*.



Uvědomujete si rozdíl mezi relačním modelem dat a konceptuálním modelem? Relační model dat popisuje obecnou strukturu, integritní omezení a operace nad daty v relační databázi bez ohledu na konkrétní aplikační doménu. Naproti tomu konceptuální model modeluje data konkrétní aplikační domény, např. fakulty.

Dalším krokem je *logický návrh*. Jeho cílem je navrhnout strukturu databáze (tedy strukturu jednotlivých tabulek) tak, aby především v databázi bylo možné reprezentovat veškeré požadované informace, neexistovala redundance a kontrola integritních omezení vyplývajících ze závislostí mezi hodnotami uloženými v databázi byla co nejjednodušší. Výsledkem logického návrhu je schéma relační databáze, označované často jako *logické schéma databáze*. Máme-li vytvořen konceptuální

model, typicky v podobě ER diagramu, existují pravidla, jak tento model transformovat na schéma relační databáze.

Posledním krokem je *fyzický návrh*. Jeho výsledkem je *fyzické schéma*. Cílem je navrhnout fyzické uložení tabulek, které jsou výsledkem logického návrhu, využitím prostředků konkrétního SŘBD tak, aby bylo dosaženo co nejlepších výkonnostních parametrů. Na fyzické úrovni mívají relační databáze složitější strukturu než na úrovni logické, kde jsou jedinou strukturou podle relačního modelu dat tabulky. Organizaci na fyzické úrovni relační model neurčuje, je to již záležitost jeho implementace. Používají se zde takové pojmy jako tabulkový prostor, segment apod. Důležitou součástí organizace dat v databázi na fyzické úrovni je podpora SŘBD pro efektivní přístup k datům tabulek. Mezi tyto přístupové metody patří především *indexování a hašování*, se kterými se seznámíme v kapitole 4. Pod fyzickým schématem si tedy můžete představit zejména informaci o umístění tabulek ve fyzické struktuře databáze a o přístupových metodách k záznamům tabulek.

V této kapitole se zaměříme na logický návrh databáze. O fyzickém návrhu se dozvíte více v kapitole 4. a n přednášce, kde si řekneme něco o optimalizaci zpracování dotazů.

Z předchozích odstavců je zřejmé, že schéma relační databáze lze získat jedním z následujících dvou způsobů:

- vytvořením konceptuálního modelu a jeho transformací
- použitím normalizace s tím, že na počátku předpokládáme, že všechny informace budou uloženy v jedné tabulce a tu normalizujeme.

Přestože návrh založený pouze na normalizaci je principiálně možný, bylo by jeho použití pro rozsáhlé databáze ve srovnání s návrhem, který využívá konceptuálního modelování, nepraktické. Proto v praxi používáme jako hlavní metodu návrhu právě tento způsob. Normalizace si přesto svůj význam zachovala. Pochopení normálních forem a s nimi souvisejícími zdroji redundance a některých dalších nedostatků návrhu je v každém případě pro návrháře databáze velice užitečné. U složitých systémů může ER diagram představovat abstraktnější pohled (např. hodnoty atributů entit nejsou atomické) a tuto skutečnost je nutné respektovat při transformaci konceptuálního modelu nebo zkontrolovat kvalitu navržených tabulek využitím normalizace. V každém případě zde návrhář znalosti normalizace přímo nebo nepřímo využije.

Z tohoto důvodu si normální formy a normalizaci vysvětlíme i my v této kapitole. Nejprve si v kapitole 3.2 zopakujeme a prohloubíme znalosti datového modelování (ER modelu) a zmíníme zvláštnosti objektového modelování z pohledu následného návrhu relační databáze. Dále si v kapitole 3.3 uvedeme pravidla transformace konceptuálního modelu na schéma relační databáze a konečně kapitola 3.4 bude úvodem do problematiky normálních forem a normalizace.

3.2. **Konceptuální modelování**

Konceptuální modelování je součástí řady metodologií vývoje softwarových systémů. Probíhá ve fázi analýzy požadavků. Jeho cílem je vytvoření modelu konceptů aplikační domény, se kterými bude vyvíjený systém pracovat. Nás budou z pohledu návrhu databáze takového systému zajímat ty koncepty a vztahy mezi nimi, jejichž reprezentace bude uložena v databázi.

Nejznámější a nejčastěji používanou modelovací technikou konceptuálního pro návrh relačních databází je *entitně-vztahové modelování*, jehož výsledkem je *entitně-vztahový diagram* (ER diagram nebo ERD z anglického entity-relationship).



V češtině se používá pro entitně-vztahové modelování, resp. diagram také označení *entitně-relační modelování*, resp. diagram. My budeme preferovat označení entitně-vztahové, aby následně nedocházelo vlivem tohoto označení k již zmíněnému nesprávnému výkladu pojmu relační databáze.

Název této modelovací techniky, která vznikla v polovině 70. let minulého století (za autora je považován P. P. Chen) odráží dva základní prvky vytvářeného modelu

- entity (E - entity)
- vztah (R - relationship)

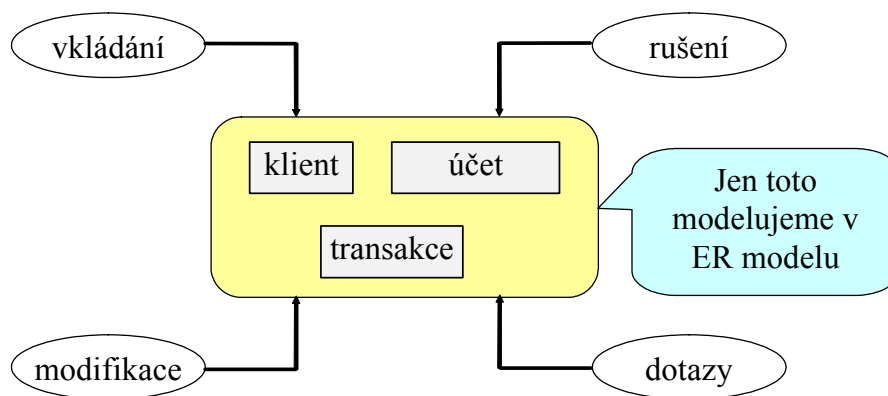
Chápe modelovanou aplikační doménu jako množinu entit, mezi nimiž mohou existovat určité vztahy. Důležitým rysem ER modelu je, že popisuje data „v klidu“. Neukazuje, jaké operace budou s těmito daty prováděny.



Příklad 3.1

Uvažujme, že vyvíjíme informační systém banky. Systém bude pracovat s informacemi o klientech, jejich účtech. Musí umožňovat vkládat informace o nových klientech, účtech, operacích s těmito účty (budeme je označovat jako transakce), musí umožňovat modifikaci některých informací, např. osobní data klienta, stav účtu, musí umožňovat rušení některých informací a samozřejmě i dotazování na data uložená v databázi, např. jaké transakce probíhaly s daným účtem v určitém období.

Předmětem ER modelování budou pouze data reprezentující klienty, účty a transakce s nimi. Naopak požadované operace vkládání, modifikace, rušení a dotazování se na modelu přímo neobjeví – viz Obr. 3.2. Model pouze musí zajistit poskytnutí všech potřebných informací pro zodpovězení požadovaných dotazů.



Obr. 3.2 Předmět ER modelování – data „v klidu“



Ve kterém modelu strukturované analýzy, kam patří ER modelování by se operace s daty objevily?

Entitou budeme rozumět objekt (koncept) aplikační domény světa, který je rozlišitelný od jiných objektů, o níž potřebujeme mít informace v databázi. Například entitou v příkladu Příklad 3.1 by mohl být klient banky s identifikačním číslem 999

nebo účet s číslem účtu 100. V souvislosti s entitami je potřeba si uvědomit dvě věci - modelujeme pouze perzistentní data, která budou uložena v databázi a musí existovat způsob, jak rozlišit dvě entity. My jsme k tomu použili jednak klasifikaci entit do množin entit téhož typu (Klient, resp. Účet), jednak použitím identifikačního čísla klienta a účtu. Množinu entit téhož typu, které sdílí tytéž vlastnosti označujeme jako *entitní množinu* (*entity set*). Existuje řada notací pro kreslení ER diagramů. My budeme používat notaci vycházející z jazyka UML (Unified Modeling Language). Entitní množiny budeme znázorňovat obdélníkem. Každá entitní množina má jméno, které musí být v rámci modelu jednoznačné.

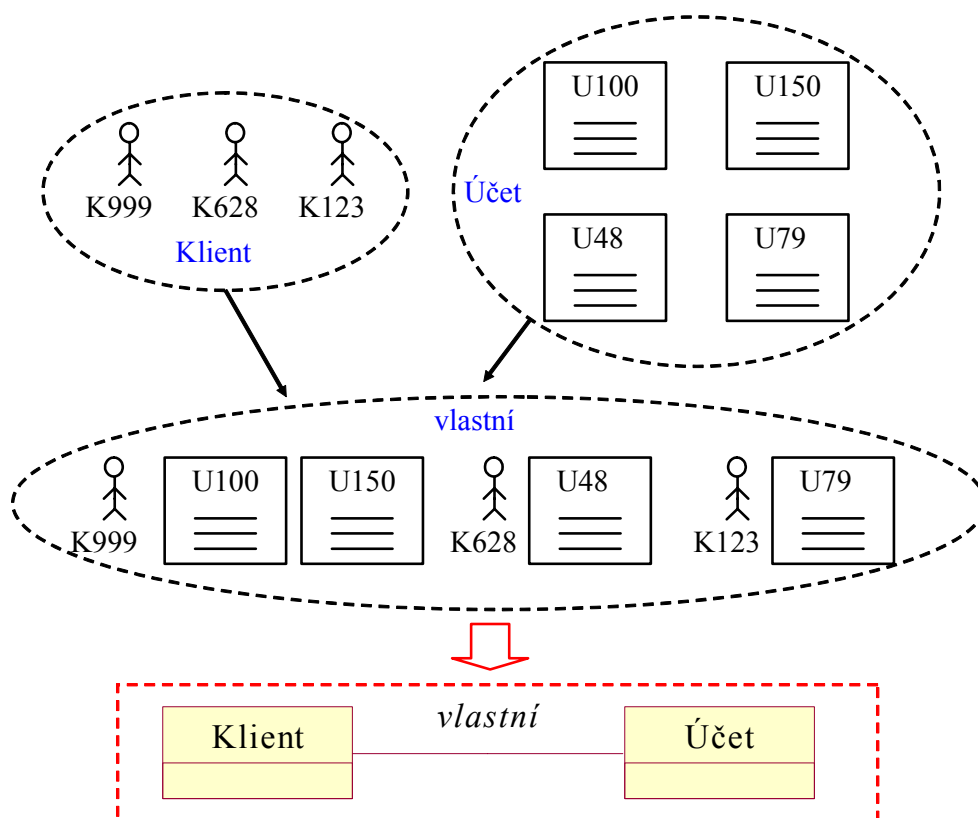
Protože vytvářený model a odpovídající diagram slouží nejen jako podklad pro návrh databáze, ale také pro komunikaci se zákazníkem, pro kterého systém vytváříme, a je součástí návrhové dokumentace, musí být srozumitelný a množství informací, které pro pochopení nelze vyčíst z ER diagramu, by mělo být co nejmenší. Z tohoto důvodu je důležité používání výstižných jmen pro všechny prvky modelu, včetně jmen entitních množin.

Entity mohou být v určitých *vztazích*. Podobně jako u entit klasifikujeme vztahy podle typu a množinu vztahů téhož typu označujeme jako *vztahovou množinu* (*relationship set*). Například jedním konkrétním vztahem v příkladu Příklad 3.1 by mohl být vztah mezi klientem s identifikačním číslem 999 a účtem s číslem účtu 100, který vyjadřuje, že daný klient je vlastníkem daného účtu. Typ daného vztahu a odpovídající vztahovou množinu bychom potom mohli nazvat „vlastní“. V ER diagramu znázorňujeme vztahové množiny úsečkou.

Klasifikace entit a vztahů do entitních a vztahových množin je ukázána na Obr. 3.3.



V literatuře se někdy používá označení entita, resp. vztah i ve významu entitní, resp. vztahové množiny. Vztah entitní množiny a entity můžeme přirovnat ke vztahu třídy a objektu v objektově-orientovaném přístupu, kde třída definuje vlastnosti objektů daného typu a objekt je potom instancí dané třídy. My budeme v této studijní opoře oba pojmy odlišovat, stejně jako to dělají autoři studijní i doporučené literatury.



Obr. 3.3 Klasifikace entit a vztahů do entitních a vztahových množin

Entitní a vztahové množiny definují vlastnosti entit a vztahů daného typu. Vlastnosti entit, jejichž hodnoty je potřeba ukládat do databáze, označujeme jako *atributy*. Například u klienta můžeme potřebovat kromě již zmíněného identifikačního čísla mít informaci o jménu, adrese a telefonu. Podle struktury hodnot atributů můžeme rozlišit atributy

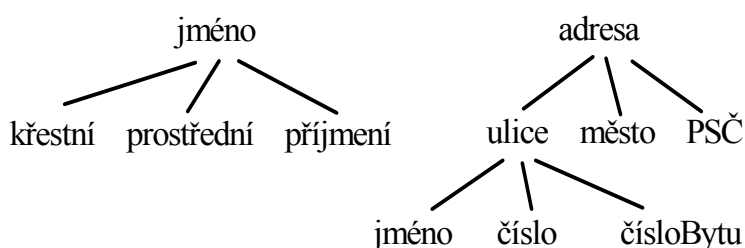
- jednoduché (simple)
- složené (composite)

Rozdíl je stejný jako jsme diskutovali v kapitole 2.2 Struktura relační databáze. *Jednoduchým atributem* nazýváme atribut, který nabývá atomických (skalárních) hodnot, *složený atribut* je složený z několika jiných atributů (jednoduchých či složených).

Příklad 3.2

$x+y$

Uvažujme, že jméno klienta potřebujeme chápat jako složené z křestního jména a příjmení, podobně u adresy potřebujeme rozlišovat ulici, město a PSČ a u ulice navíc kromě názvu i číslo a číslo bytu. Potom oba atributy budou složené – viz Obr. 3.4.

*Entitní množina***Klient***Složené atributy**Složky**Složky***Obr. 3.4** Složené atributy

Jiným hlediskem, podle kterého můžeme klasifikovat atributy, je počet hodnot, jichž může atribut nabývat. Zde rozlišujeme atributy

- jednohodnotové (single-valued)
- vícehodnotové (multiple-valued)

Jednohodnotový atribut může nabývat v každém okamžiku jen jedné hodnoty, zatímco *vícehodnotový atribut* může současně nabývat několika hodnot.

Příklad 3.3 $x+y$

Předpokládejme, že u klienta potřebujeme ukládat v databázi jeho telefonní kontakt a je požadováno, aby náš systém umožňoval uložit všechna telefonní čísla, která nám klient bude ochoten sdělit. V takovém případě atribut telefon bude vícehodnotový – viz Obr. 3.5.

*Entitní množina***Klient***Vícehodnotový atribut*

telefon

Hodnoty atributu

číslo1, číslo2, číslo3, ...

Obr. 3.5 Vícehodnotový atribut

Obě výše uvedená hlediska jsou navzájem nezávislá. Vícehodnotový atribut může nabývat atomických nebo složených hodnot a naopak složky složeného atributu mohou být vícehodnotové. Vztah je stejný jako například mezi polem a záznamem, jak je znáte z programovacích jazyků.

Již víme, že v tabulce relační databáze lze ukládat pouze atomické hodnoty. Až si budeme v příští kapitole vysvětlovat pravidla transformace ER modelu na tabulky relační databáze, uvidíme, že pokud jsou některé atributy složené nebo vícehodnotové, musíme provést úpravy, které povedou na jednohodnotové jednoduché atributy.

Už víme, že entity musí být vzájemně rozlišitelné. Musí proto existovat nějaký atribut entity (případně složený), jehož hodnota bude jednoznačně určovat danou entitu v rámci entitní množiny. Takový atribut se nazývá *identifikátor* nebo také *primární*

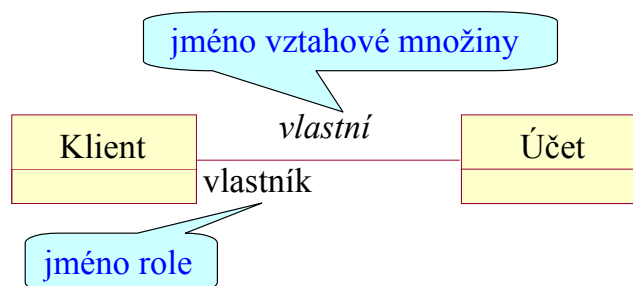
klíč entitní množiny. Uvidíme, že tento atribut se při transformaci na tabulky stane primárním klíčem tabulky, ve které budou informace o entitách uloženy. Analogicky můžeme zavést identifikátor vztahové množiny.

Další vlastností atributu, která je z pohledu návrhu databáze důležitá a která představuje podobně jako identifikátor integritní omezení, je, zda může být jeho hodnota prázdná. Již víme, že pro prázdné hodnoty používáme označení NULL. Přestože pro návrh databáze zpravidla důvod, proč hodnota chybí není důležitý, v některých situacích může být potřeba rozlišovat dva následující případy:

- Hodnota chybí (v angličtině se označuje jako „missing“). V tomto případě hodnota existuje, ale my ji neznáme, Příkladem může být datum narození klienta.
- Hodnotu neznáme (v angličtině se označuje jako „unknown“). V tomto případě ani nevíme, jestli hodnota existuje. Příkladem může být hodnota telefonního kontaktu.

Zvláštním typem atributu je takový, jehož hodnotu můžeme spočítat z hodnot jiných atributů a vztahů. Takový atribut nazýváme *odvozený*. Příkladem může být věk klienta nebo průměrný stav účtu v předchozím roce. Možná vás napadá, k čemu může být takový, když lze jeho hodnotu spočítat. Při analýze požadavků může být užitečné v některých případech takový atribut použít, když chceme zdůraznit, že takovou bude potřeba znát. Záleží potom na rozhodnutí návrháře, zda bude tato spočítaná hodnota uložena v databázi nebo uložena nebude a bude počítána vždy, když bude požadována. Takový atribut naopak může vést na zavedení jiného atributu nebo atributů z jehož hodnot se bude počítat. V našem příkladě by potřeba znát věk klienta by vedla na potřebu uložit datum nebo alespoň rok narození klienta.

Nyní se podívejme podrobněji na vlastnosti vztahů a vztahových množin. Začneme způsobem pojmenování vztahových množin. V první řadě je potřeba říci, že narozdíl od entitních množin vztahové nemusí být vždy pojmenovány. Této možnosti bychom ale měli využívat pouze tehdy, kdy je význam vztahu zřejmý ze jmen entitních množin, jejichž entity ve vztazích daného typu participují. Pokud potřebujeme význam vztahů vyjádřit jménem, můžeme použít buď jméno vztahové množiny nebo jméno role – viz Obr. 3.6.

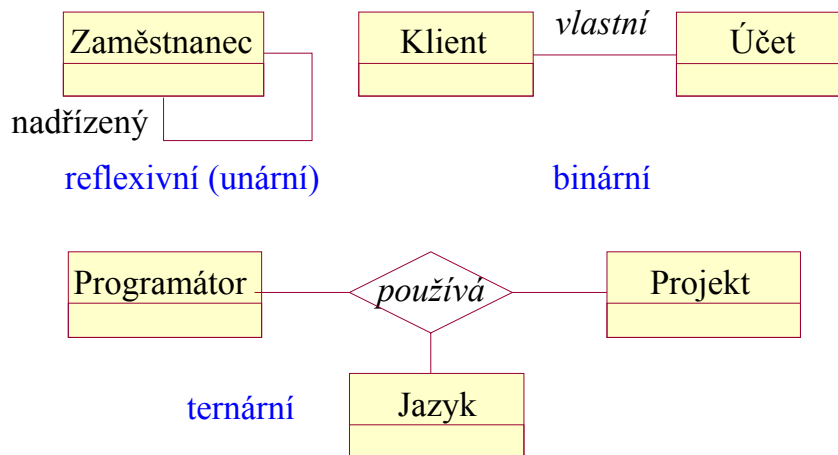


Obr. 3.6 Pojmenování vztahů

Jméno vztahové množiny volíme nejčastěji tak, aby připojením jmen entitních množin vznikla jednoduchá věta vyjadřující význam vztahu daného typu. V Obr. 3.6 je to věta „Klient vlastní účet“. Jméno role je vlastností konce úsečky reprezentující vztahovou množinu v ER diagramu. Vyjadřuje roli, kterou entita množiny na daném konci ve vztahu hraje. V našem příkladě je klient vlastníkem účtu narozdíl např. od ostatních osob, které mohou s účtem provádět omezené manipulace. Jméno role je velmi často totožné s jménem odpovídající entitní množiny (u nás účet). Pak ho samozřejmě

neuvádíme. Vždy použijeme takový způsob pojmenování vztahu, který je pro pochopení jeho významu nejvhodnější. Současně ale pamatujeme na to, že nadbytečná informace může naopak snižovat přehlednost diagramu.

Nejčastějším případem vztahů je ten, kdy jde o vztah mezi dvěma entitami. Počet entit, které do vztahu vstupují, se nazývá *stupeň vztahu*. V případě dvou entit pak hovoříme o vztahu *binárním*, v případě tří o *ternárním*. Vyšší stupeň se vyskytuje výjimečně a už se žádné speciální pojmenování nepoužívá. Zvláštním, ale nijak výjimečným případem, je situace, kdy jde o vztah mezi dvěma entitami téže entitní množiny. Pro ten se používá označení *reflexivní* nebo také *unární*. Příklady jsou uvedeny na Obr. 3.7.



Obr. 3.7 Stupeň vztahu

Význam ternárního vztahu na tomto obrázku je takový, že jistý programátor používá při řešení jistého projektu určitý programovací jazyk.

Další vlastností vztahových množin, která je pro návrh databáze velmi důležitá, je *kardinalita*. Podobně jako jméno role jde o vlastnost „konce“ vztahu. Udává, do kolika vztahů daného typu může jedna entita množiny na tomto konci vstupovat. V případě binárního vztahu musíme tedy určit kardinalitu pro oba konce, u ternárního pro všechny tři.

Příklad 3.4

$x+y$

Uvažujme příklad vztahové množiny vlastní v Obr. 3.7. Předpokládejme takový význam vztahu, že každý účet musí mít právě jednoho vlastníka a jeden klient může vlastnit několik účtů, ale nemusí také vlastnit žádný (např. může jen disponovat s účtem někoho jiného).

Jde o vztah binární, musíme tedy určit kardinalitu pro oba konce. Pro určení kardinality na konci u Klienta (vlastníka účtu) si položíme otázku „Kolik vlastníků může mít jeden účet?“ Odpověď bude „Právě jeden.“ Analogicky pro určení kardinality pro druhý konec bude otázka „Kolik účtů může vlastnit jeden klient?“. Odpověď bude „Žádný, jeden nebo více.“

Kardinalita, jak jsme si ji zavedli, může být vyjádřena intervalem, jehož dolní mez udává minimální počet vztahů daného typu, do nichž entita musí vstupovat, a horní mez udává maximální počet vztahů daného typu, do nichž entita může vstupovat. Pak můžeme hovořit o minimální a maximální kardinalitě. Pro minimální kardinalitu se také používá označení *členství* (*membership*). Obecně může nabývat hodnoty libovolného celého nezáporného čísla, ale nejčastějšími případy jsou ty, kdy minimální

počet je 0 (hovoříme o nepovinném členství), tj. entita nemusí vstupovat do žádného vztahu daného typu, a 1 (povinné členství), kdy musí do vztahu vstupovat.

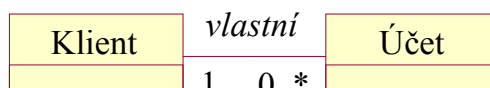
Maximální kardinalita může obecně nabývat hodnoty libovolného přirozeného čísla, ale nejčastěji se rozlišují dva případy – kdy je maximální počet 1 a kdy je maximální počet větší než 1. V druhém případě se používá označení „M“ z anglického „many“.

Pokud není kardinalita uvedena jako interval, potom značí maximální kardinalitu (minimální není uvedena) nebo fakt, že hodnota minimální a maximální kardinality jsou stejné.



Příklad 3.5

ER diagram příkladu Příklad 3.4 doplněný o kardinalitu je uveden na Obr. 3.8

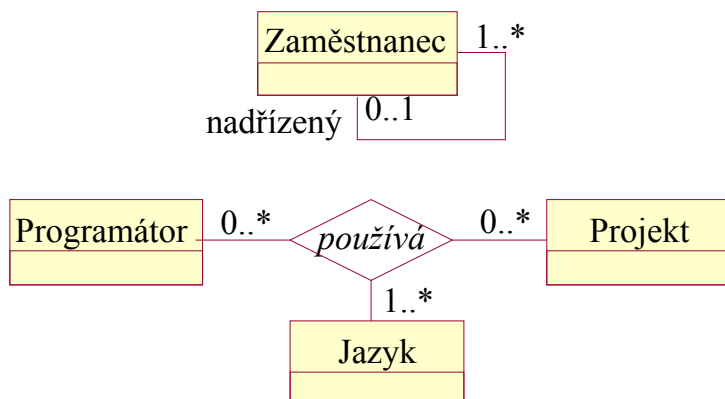


Obr. 3.8 ER diagram příkladu Příklad 3.4 doplněný o kardinalitu

V diagramu je použit pro kardinalitu M symbol „*“, což je notace používaná v jazyce UML.



Vyjádřete slovně význam vztahů na Obr. 3.9 s respektováním uvedené kardinality.



Obr. 3.9 ER diagramy s doplněnou kardinalitou



Kardinalita představuje omezení, které vyplývá z významu vztahu. Proto je pro její správné určení rozhodující pochopení modelovaného vztahu. Pokud není kardinalita zcela zřejmá, je potřeba konzultovat význam vztahu se zákazníkem, pro kterého systém vyvíjíme. V našem příkladu informačního systému banky bychom se například ptali, zda může klient vlastnit více účtů a zda může být klient, který nevlastní žádný účet.

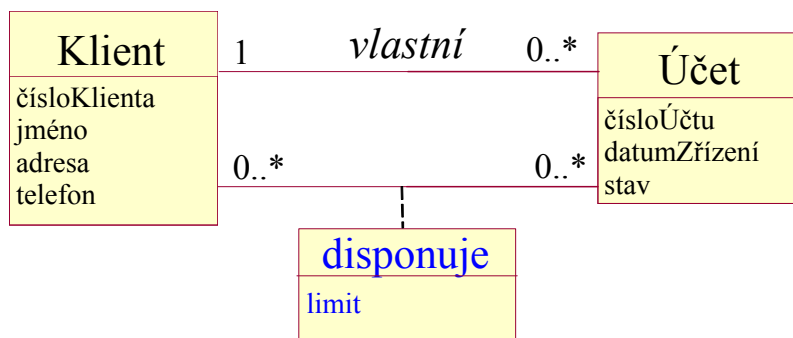
Pro návrh schématu databáze je rozhodující správné určení maximální kardinality. Hodnota minimální kardinality ovlivňuje pouze, zda v nějakém sloupci tabulky může být prázdná hodnota, zatímco maximální kardinalita přímo ovlivňuje strukturu databáze. Z tohoto důvodu také často uvádíme pouze maximální kardinalitu. Potom rozlišujeme binární typy vztahů 1 : 1 („jedna ku jedné“, maximální kardinalita na obou koncích je 1), 1 : M a M : M. Analogicky pro vztah ternární.

Vztahy mohou mít podobně jako entity atributy. V ER diagramu je budeme

znázorňovat jako obdélník asociovaný s úsečkou vztahu.

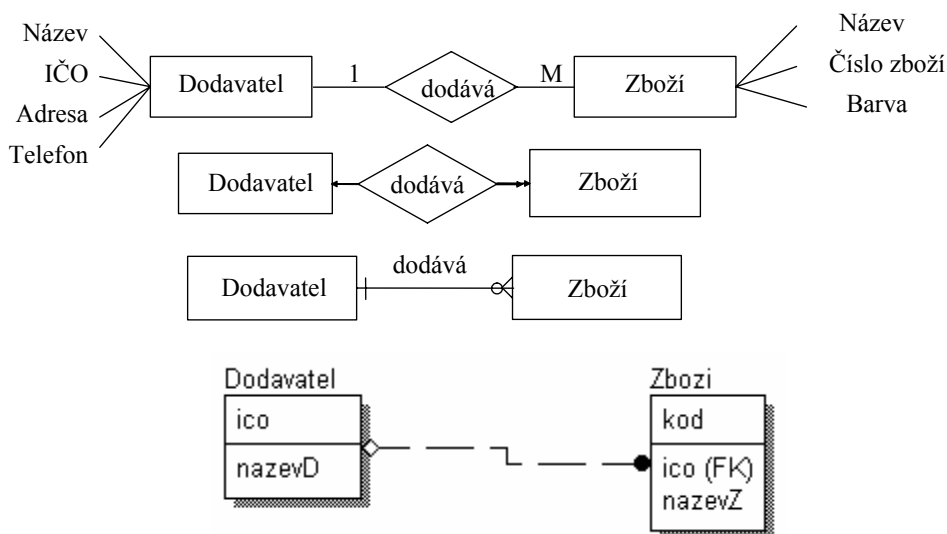
$$x+y$$

Uvažujme v našem příkladu bankovního informačního systému, že kromě vlastníka, který může se svým účtem disponovat neomezeně, mohou s účtem disponovat i další vlastníkem určené osoby (jsou to také klienti banky). Těmto disponujícím osobám ale vlastník může definovat limit pro operace s účtem. Odpovídající ER diagram je na Obr. 3.10.



Obr. 3.10 Atributy vztahu

Již jsme uvedli, že pro kreslení ER diagramu se používají různé notace. Nejvýrazněji se tyto notace odlišují ve způsobu značení kardinality. Některé z používaných notací jsou uvedeny na Obr. 3.11.



Obr. 3.11 Příklad notací pro kreslení ER diagramu

Diagramy ukazují entitní množiny Dodavatel, Zboží a vztahovou množinu dodává s významem, že dodavatel může dodávat více druhů zboží a každé zboží dodává jeden dodavatel. V horní části obrázku je uvedena původní Chenova notace, úplně dole je notace podle standardu IDEF1X. Jak už bylo uvedeno, v současné době se velice často používá notace vycházející z jazyka UML.

Dosud jsme se seznámili s prvky klasického ER modelu. Nyní se podíváme na dvě z rozšíření, která byla zavedena později. Prvým je modelování tzv. slabých entit, což jsou entity, které jsou závislé na jiných entitách. Druhým rozšířením, se kterým se seznámíme, je modelování vztahu, který znáte dobře z objektově-orientovaného

přístupu – generalizace/specializace,

Při datovém modelování často zjistíme, že některé entity nemohou existovat samostatně, že jsou existenčně závislé na jiných entitách. Příkladem v našem bankovním informačním systému mohou být transakce s účty. Každá transakce reprezentuje operaci (např. vklad nebo výběr) s konkrétním účtem. Nemůže existovat samostatně, protože je vždy vázána na nějaký účet. Chceme-li tuto skutečnost v našem modelu zvýraznit, můžeme entitní množinu Transakce modelovat jako *slabou entitní množinu* (*weak entity set*). Existenční závislost slabé entity na jiné entitě, označované jako *identifikující* nebo také *dominantní* se projevuje tím, že její identifikátor je vždy složený a je tvořen jednak identifikátorem identifikující entitní množiny (proto se nazývá identifikující) a atributem, který má tu vlastnost, že jeho hodnota je jednoznačná ne v rámci všech entit slabé entitní množiny, nýbrž jen entit závislých na téže identifikující entitě. Takový atribut se označuje jako *diskriminátor* nebo také *částečný identifikátor*. V našem příkladě by diskriminátorem mohlo být pořadové číslo transakcí, které by byly průběžně číslovány vždy v rámci jednoho účtu. Mohla by tedy existovat řada transakcí s pořadovým číslem např. 1, ale v rámci transakcí týkajících se jednoho konkrétního účtu by taková transakce byla nejvýše jedna. Pokud je identifikátor entitní množiny Účet atribut čísloÚčtu a diskriminátor slabé entitní množiny Transakce atribut pořČíslo, bude identifikátorem množiny Transakce složený atribut (čísloÚčtu, pořČíslo).

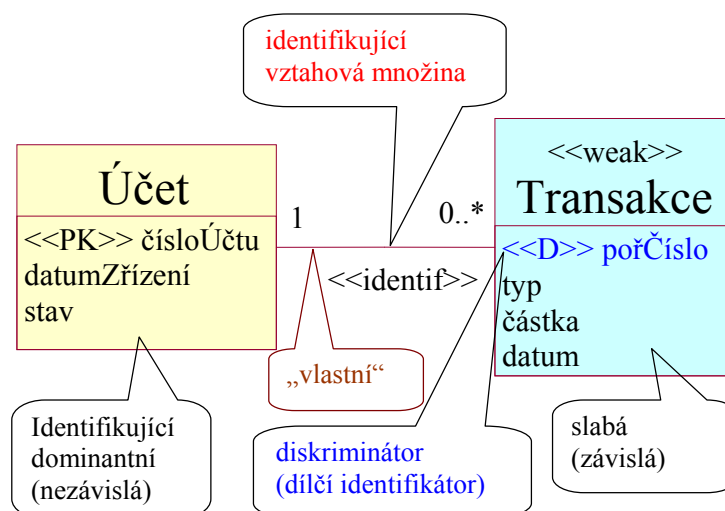
Entitní množina, jejíž entity nejsou takto závislé na jiných entitách, se označuje, pokud chceme tento fakt zdůraznit, jako *silná entitní množina* (*strong entity set*). Takovými entitními množinami by v našem příkladu byly množiny Klient a Účet. Obě mají svůj vlastní identifikátor nezávislý na jiných entitních množinách.

Každá slabá entitní množina je ve vztahu M:1 s příslušnou identifikující množinou. Tento vztah zpravidla nepojmenováváme, implicitní jméno takové vztahové množiny by mohl být „vlastní“ (entita identifikující množiny vlastní žádnou, jednu nebo několik slabých entit). Tato vztahová množina se označuje jako *identifikující*, protože zprostředkuje přenos identifikátoru identifikující množiny do identifikátoru slabé množiny.

Důležité pojmy související se slabými entitními množinami jsou shrnuty na Obr. 3.12. Pro modelování používáme stereotypy <<weak>> pro slabou entitní množinu, <<identif>> pro identifikující vztah a <<D>> pro diskriminátor.

Základní vlastnosti slabé entitní množiny tedy můžeme shrnout do následujících tří bodů:

- je závislá na jiné entitní množině zvané identifikující (pouze jedné!)
- je pomocí této množiny identifikována, tj. identifikátor je vždy složený a obsahuje identifikátor dominantní množiny
- je vždy ve vztahu M:1 s identifikující množinou



Obr. 3.12 Pojmy související se slabými entitními množinami



Jak již bylo uvedeno, slabé entitní množiny používáme, když chceme zdůraznit silnou existenční vazbu na identifikující entitu. Identifikující entitní množinou může být buď silná nebo jiná slabá entitní množina. Při rozhodování, zda při modelování použít slabou entitní množinou byste se měli držet zásady, že nechcete-li závislost zdůraznit nebo jste na pochybách, budete modelovat entitní množinu jako silnou. V našem příkladě bychom mohli zvolit jednoznačné číslování transakcí v rámci všech transakcí místo číslování v rámci transakcí jednoho účtu a tím by entitní množina Transakce byla silnou entitní množinou, vztahová množina na účet by již nebyla identifikující, mohla by se nazývat „patří“ a závislost na účtu by zůstala zachována tím, že každá transakce musí patřit právě jednomu účtu. Tj. minimální i maximální kardinalita je jedna. Jednalo by se tedy obdobu vztahové množiny „vlastní“ mezi množinami Klient a Účet z Obr. 3.8.

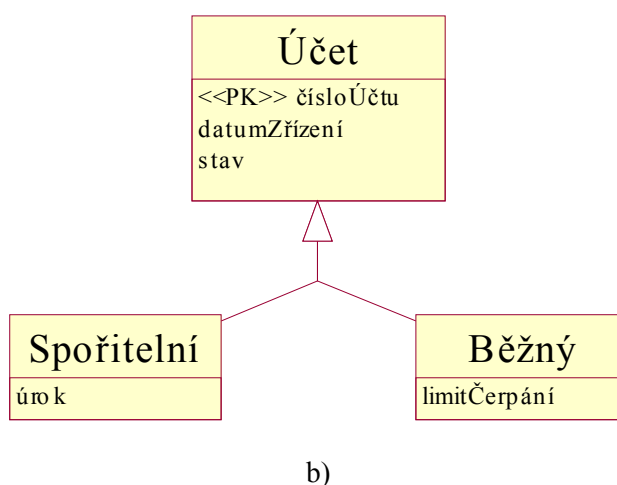
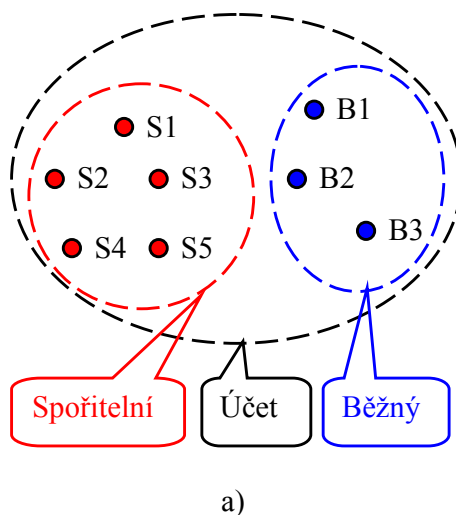
Při datovém modelování se často setkáváme s potřebou modelovat jiný důležitý vztah – vztah *generalizace/specializace*. Jde o vztah dvou entitních množin, kdy jedna rozšiřuje vlastnosti (množinu atributů) množiny druhé.

Příklad 3.6



Uvažujme, že v našem bankovním informačním systému se vyskytují dva typy účtů – spořitelní účet a běžný účet. Oba dva typy mají některé vlastnosti společné, např. číslo účtu, datum zřízení a stav na účtu. Naopak v jiných vlastnostech se liší. Pro jednoduchost uvažujme jako rozdílné vlastnosti jen úrok pro spořitelní a limit čerpání pro běžný účet.

Chceme-li zdůraznit skutečnost, že jak spořitelní tak běžné účty jsou účty s nějakými společnými vlastnostmi, ale mají také nějaké vlastnosti odlišné, můžeme je modelovat jako entity dvou odpovídajících entitních množin Spořitelní a Běžný – viz Obr. 3.13 a), ale také jako entity zobecňující entitní množiny Účet. Protože se jedná o významný typ vztahu, byl zaveden jako rozšíření ER modelu. V ER diagramu budeme pro něj používat stejný symbol jako pro generalizaci/specializaci, resp. dědičnost při objektově-orientovaném modelování v UML – viz Obr. 3.13 b).



Obr. 3.13 Generalizace/specializace. a) Specializace účtů, b) ER diagram

Význam vztahu je stejný jak ho znáte z OO modelování. Entity množiny Spořitelní a Běžný dědí atributy entitní množiny Účet. Pokud se na generalizaci/specializaci na Obr. 3.13 b) podíváte ve směru symbolu šipky generalizace, můžeme říci, že entitní množina Účet je zobecněním množin Spořitelní a Běžný. Naopak v opačném směru můžeme říci, že entitní množiny Spořitelní a Běžný jsou speciálními případy Účtu. Odtud název generalizace/specializace.

Protože specializace dědí i identifikátor generalizující entitní množiny, mají stejný identifikátor, v našem příkladě číslo účtu.

Generalizace/specializace může být zpřesněna použitím dvou navzájem nezávislých charakteristik příslušnost a úplnost. *Příslušnost (membership)* udává, zda mohou některé entity patřit do více než jedné specializační entitní množiny. Rozlišuje se příslušnost *disjunktní (disjoint)*, kdy jsou specializace disjunktní, a *překrývající se (overlapping)*, kdy nemusí být specializační entitní množiny disjunktní. V našem příkladě jde zřejmě o disjunktní specializace – účet je buď spořitelní nebo běžný.

Úplnost (completeness) udává, zda každá entita generalizující entitní množiny musí být zároveň entitou některé ze specializací. Příslušnost může být buď *úplná (total)* nebo *částečná (partial)*. V prvním případě musí každá entita generalizující entitní množiny spadat také do některé specializace, ve druhém mohou existovat entity, které

nelze zařadit do některé ze specializací. V našem příkladu by příslušnost úplná znamenala, že jiný typ účtu než spořitelní nebo běžný v našem systému neexistuje. Naproti tomu částečná příslušnost by říkala, že mohou existovat i jiné typy účtu, které by ale byly charakterizovány pouze atributy entitní množiny Účet.

Uvidíme, že příslušnost a úplnost má vliv na transformaci generalizace/specializace na tabulky relační databáze.

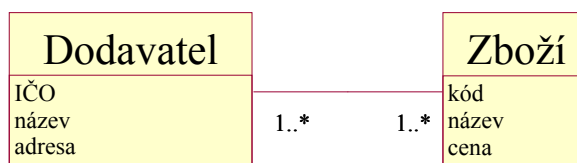


Vztah generalizace/specializace se také v terminologii ER modelování někdy označuje jako ISA vztah z anglického „is a“. Můžeme říci, že Spořitelní „je“ Účet. Analogicky pro Běžný. V ER diagramu se proto můžete někdy setkat se symbolem šipky, jako používáme my, s textem ISA uvnitř.



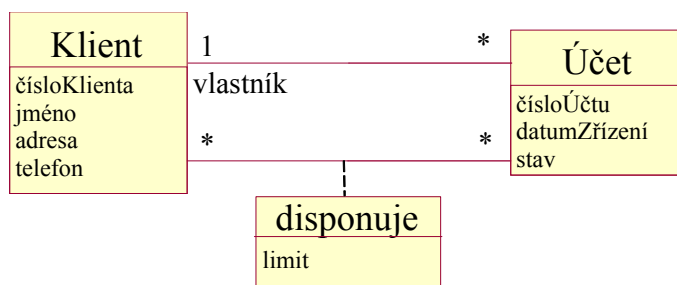
Nyní, kdy už víme, jaké jsou prvky ER modelu, co vyjadřují a jak je budeme kreslit ER diagramu se podíváme na některá důležitá praktická doporučení pro modelování a tvorbu ER diagramu.

V prvé řadě je potřeba znovu zopakovat, co jsme už uváděli u jmen entitních a vztahových množin - námi vytvořený model a odpovídající diagram musí být srozumitelný. Proto musíme používat taková jména, která výstižně vyjadřují význam entitní či vztahové množiny. V případě entitní množiny používáme zpravidla podstatná jména a pro vztahové množiny slovesa a předložky. Také jsme si ukázali možnost použití jmen rolí. Víme již také, že je-li význam vztahové množiny zcela jasný ze jmen entitních množin, kterých se týká, není nutné je uvádět – viz Obr. 3.14.



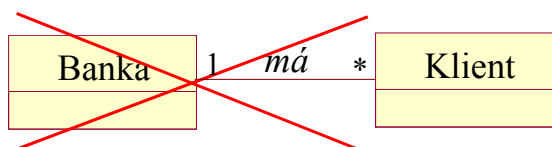
Obr. 3.14 Vztahová množina, kterou není nutné pojmenovat

Naopak pokud existuje několik vztahových množin mezi stejnými entitními množinami, je nutné vztahové množiny pojmenovat nebo použít jmen rolí vždy – viz Obr. 3.15. Pokud bychom v tomto případě nepoužili žádná jména, nebylo by bez doplňujícího vysvětlení jasné, co která vztahová množina znamená. Podobně, kdybychom pojmenovali jenom jednu, např. disponuje, tak nebude zřejmý význam druhé. My jsme pro pojmenování jedné použili jméno vztahové množiny a význam druhé jsme vymezili jménem role.



Obr. 3.15 Pojmenování několika vztahových množin mezi stejnými entitními množinami

Další zásadou, která se často porušuje, je, že entitní množinu, která reprezentuje celý modelovaný systém (v našem příkladu banku) by neměl být v modelu zahrnut – viz Obr. 3.16.



Obr. 3.16 Celkový systém do modelu nepatří

Tato entitní množina by totiž obsahovala pouze jednu entitu a to námi modelovanou banku a ta by byla ve vztahu s významem „má“ či „patří“ s entitami jiných entitních množin. To je samozřejmě zbytečné. Platí zásada, že do modelu zahrnujeme vždy až prvky systému, tedy dalo by se říci prvky „o úroveň níž“. Modelujeme-li banku, pak se v modelu objeví až její klienti, účty apod. Modelujeme-li univerzitu členěnou na fakulty, pak budou v modelu až fakulty, studenti, předměty apod.



Můžete namítat, že informace o bance, resp. univerzitě, jejíž systém vyvíjíme, bude muset být někde v databázi uložena, pokud tyto údaje nechceme „natvrdo“ naprogramovat v kódu aplikace, což by určitě nebylo dobré řešení (proč?). Máte samozřejmě pravdu v databázi určitě bude existovat nějaká tabulka, kde informace tohoto typu budou uloženy. Do konceptuálního modelování je ale nezahrnujeme, nýbrž rozšiřujeme o ně až schéma databáze ve fázi logického návrhu. Určitě by ale měly být i požadavky tohoto typu podchyceny někde v dokumentaci, která je výsledkem analýzy požadavků.

Jiná situace nastane, má-li námi vyvíjený systém pracovat s informacemi o několika bankách, resp. univerzitách. Potom vlastně modelujeme systém bank, resp. univerzit a tam se entitní množina, Banka resp. Univerzita a odpovídající vztahové množiny objeví.

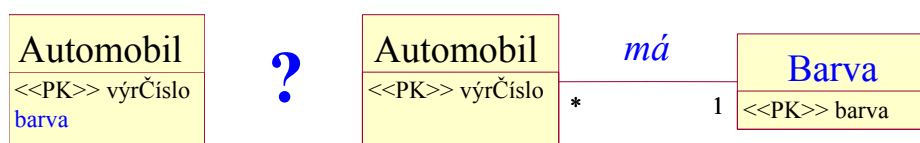
Jiným problémem, před kterým při konceptuálním modelování poměrně často stojíme, je rozhodnutí, zda modelovat nějakou vlastnost entity jako atribut nebo jako vztah na jinou entitu, která ponese hodnotu. Uvažujme následující příklad.



Příklad 3.7

Předpokládejme, že navrhujeme databázi informačního systému výrobce automobilů. Jednou z vlastností, které potřebujeme u každého vyrobeného automobilu uchovávat, je jeho barva.

Řešením může být zavedení atributu barva entitní množiny Automobil, jehož hodnota bude udávat barvu. Jinou možností ale je vytvořit entitní množinu Barva, která bude představovat paletu používaných barev. Její entity budou jednotlivé barvy a barva automobilu potom bude určena vztahem konkrétního automobilu a konkrétní barvy – viz Obr. 3.17.

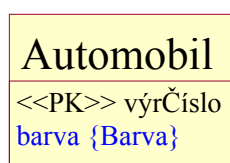


Obr. 3.17 Modelování vlastnosti atributem a vztahem

Jaký je mezi oběma variantami rozdíl? Odpověď na tuto otázku musíme hledat v odpovědi na otázku „Může být barva automobilu libovolná nebo je sortiment barev nějak omezen?“. Pokud může být barva libovolná, pak použijeme atribut. Je-li sortiment omezený, pak jednou z možností, jak toto omezení vyjádřit, je právě

zavedení entitní množiny, která bude uchovávat přípustné hodnoty. Výhodou tohoto řešení je, že na jedné straně je sortiment barev omezen, současně ale lze nové barvy do palety bez problému přidávat. V příští podkapitole uvidíme, že hodnoty palety barev budou uloženy v samostatné tabulce. Takové tabulky a také odpovídající entitní množiny se často označují jako *číselníky*. Příklady takových číselníků mohou být číselník poštovních směrovacích čísel, studijních programů a oborů na fakultě, účetních operací apod. Atribut nesoucí hodnotu je potom buď přímo identifikátorem číselníku nebo, je-li hodnota složitá (např. delší znakový řetězec proměnné délky), zavádí se jako identifikátor jiný atribut.

Při modelování složitých systémů by náš ER diagram mohl obsahovat velké množství číselníků, které by model znepřehledňovaly. V takovém případě lze zavést konvenci, že můžeme použít místo vztahové množiny a číselníku atribut s uvedením omezení obsahujícího název číselníku, např. podle UML ve složených závorkách – viz Obr. 3.18. Složitější číselníky (s více atributy) mohou být potom nakresleny odděleně.



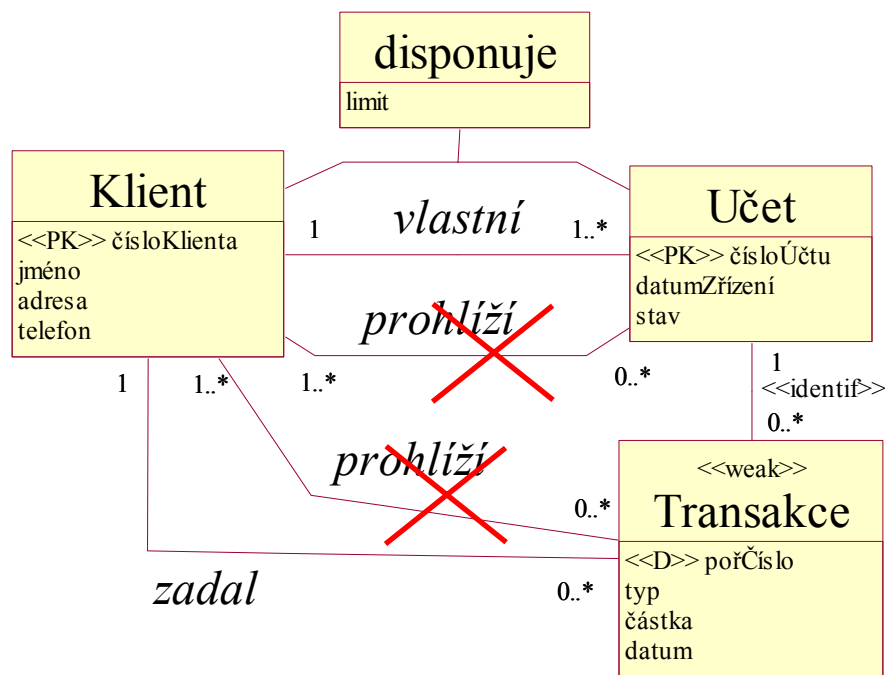
Obr. 3.18 Možná konvence pro číselníky v ER diagramu

Když jsme si vysvětlovali podstatu ER modelu, řekli jsme si, že slouží k modelování dat, která budou uložena v databázi, nikoliv operací s těmito daty. Poměrně častou chybou studentů je, že nejsou schopni takové operace od dat odlišit a modelují je potom jako vztahové množiny. Příčinou bývá to, že v databázi zpravidla uchováváme informace o uživatelích systému (například klientech naší banky), tak o objektech, se kterými provádějí nějaké operace. A ne vždy je nutné o provedené operaci ukládat do databáze informaci.

Příklad 3.8

Uvažuje náš bankovní informační systém a předpokládejme, že banka poskytuje svým klientům i službu zvanou Internetbanking. Tato služba umožní prostřednictvím internetu vlastníkovu účtu a jím pověřeným dalším klientům vzdáleně spravovat daný účet. Pomocí příslušné aplikace mohou mimo jiné prohlížet účty, s nimiž mohou disponovat, a transakce, které s nimi byly prováděny. Dále mohou zadávat příkazy k provedení nových transakcí. Musí být zjistitelné, kdo příkaz zadal. Transakce jsou prováděny hromadně v době menšího zatížení bankovního systému, typicky v noci.

Zadání tohoto příkladu může svádět k nakreslení ER diagramu z Obr. 3.19.



Obr. 3.19 Příklad chybného použití vztahů v ER diagramu

Když se pořádně zamyslíte nad vztahy, které jsou v tomto diagramu modelovány, snadno odhalíte ty, které tam s velkou pravděpodobností nepatří a jsou tedy chybou. Jsou to vztahy *prohlíží* mezi entitami typu Klient a Účet a Klient a Transakce. Pokud budeme vycházet z toho, že v zadání nebyl žádný požadavek na uchovávání informace o tom, co si kdy který klient prohlížel, představuje „prohlíží“ pouze operaci, kterou bude muset aplikace realizující službu Internetbankingu klientovi poskytnout. A právě proto, že v databázi, která je sdílena s ostatními moduly bankovního informačního systému, jsou informace o klientech a účtech, mívají studenti snahu vyjádřit, že klient jako uživatel systému má možnost prohlížet si účty a transakce. Víte, že ER diagram není jediným modelem, který při analýze požadavků používáme. Každý z těchto

modelů představuje jiný pohled na analyzovaný systém. ER diagram ukazuje pouze data, která budou uložena v databázi. Požadavky na operace, které s těmito daty musí být možné provádět, by byly zachyceny jiným modelem. Při použití klasických modelů strukturované analýzy by to byl Data Flow Diagram, můžeme pro tyto účely ale také použít model případů použití (Use Case) jazyka UML, jak ukazuje Obr. 3.20.



Obr. 3.20 Zobrazení operací s daty v diagramu případů použití

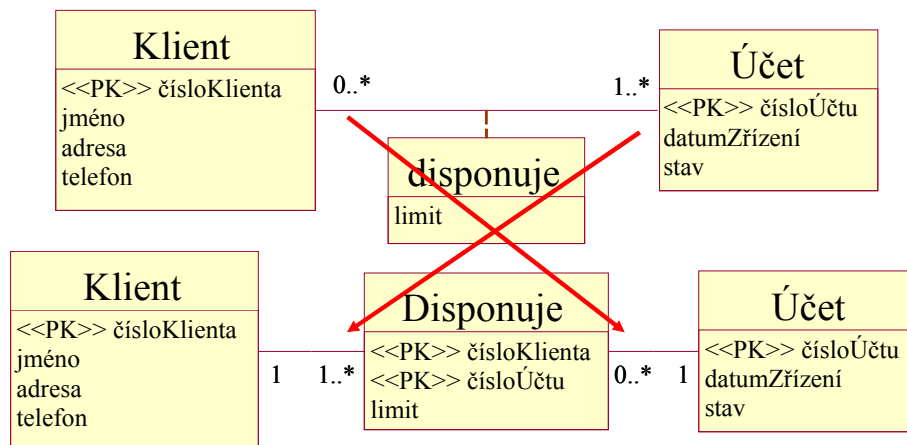
Jiná situace je ale u transakcí provedených na základě příkazu klienta. Protože účtem může disponovat více osob a musí být zjištěné, kdo zadal příkaz k provedení transakce, bude muset být tato informace v databázi uložena. Proto vztahová množina „zadal“ je v ER diagramu správně.



Měli byste si vštípit zásadu, že dříve, než nakreslíte v ER diagramu čáru pro vztahovou množinu, si položíte otázku: „Má být tato informace uložena v databázi?“. Pokud ano, pak je vše v pořádku, pokud ne, pak do diagramu tyto vztahy nepatří. Tuto zásadu bychom měli dodržovat zejména tehdy, jde-li o vztah, kde figurují entity modelující zároveň uživatele systému. U nás to byl Klient.

Když jsme si vysvětlovali pojem kardinalita vztahu, uvedli jsme si, že ER diagram může obsahovat vztahy s kardinalitou M:M. V příští kapitole uvidíme, že zatímco transformace vztahů s kardinalitou 1:1, 1:M a M:1 je přímočará a vyžaduje tabulky pouze pro příslušné entitní množiny, v případě vztahů s kardinalitou M:M musíme použít pro uložení informace reprezentující vztah samostatnou tabulku. Z tohoto důvodu některé metodologie a nástroje pro konceptuální modelování vyžadují, aby byl ER diagram upraven tak, aby neobsahoval žádné vztahy s kardinalitou M:M. Úprava je jednoduchá. Spočívá v zavedení nové entitní množiny (někdy se označuje jako *vazební entitní množina*), která bude reprezentovat původní vztah s kardinalitou M:M. Tato nová entitní množina bude potom mít k entitním množinám, mezi kterými byl vztah s kardinalitou M:M, vztah s kardinalitou 1:M a M:1. Postup ilustruje Obr. 3.21. Abychom přesně ukázali přesně, jak se promítá původní kardinalita do kardinalit nových vztahů, použili jsme poněkud nerealistický příklad. Předpokládáme totiž, že každý klient musí být disponentem (myšleno jiným než vlastníkem) alespoň jednoho účtu.

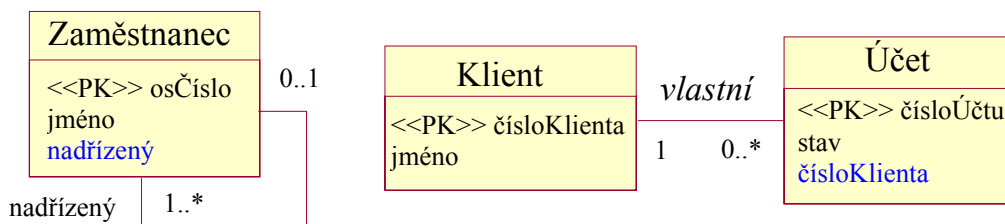
Každá entita nově vytvořené vazební množiny bude reprezentovat jeden původní vztah, tedy který klient může disponovat s kterým účtem. Kardinalita nových vztahů je vždy 1:M a M:1 s tím, že hodnota 1 je vždy na konci u původních entitních množin – vždy jde o jednoho konkrétního klienta a konkrétní účet. Druhé konce nových vztahů potom kopírují původní kardinalitu. Jestliže každý klient mohl disponovat 1 až M účty, pak dvojic (klient, účet), které reprezentují entity vazební množiny může být 1:M. Podobně pro druhou kardinalitu. Jestliže mohlo být pro daný účet 0 až M klientů, kteří ním mohou disponovat, pak může existovat právě takový počet dvojic (klient, účet). Toto pravidlo pro určení kardinality nových vztahů, naznačené v obrázku šipkami, se někdy označuje jako *křížové pravidlo*.



Obr. 3.21 Náhrada vztahu s kardinalitou M:M

Jména vazebních entitních množin se v praxi často tvoří ze jmen entitních množin, které tato vazební spojuje, např. Klient_Účet. Tato jména bychom měli při konceptuálním modelování ale opět používat jen tehdy, kdy je význam naprosto zřejmý.

Vztahy s kardinalitou 1:1, 1:M a M:1 mohou být reprezentovány také atributy, jak jsme již viděli v **Obr. 3.18**. Takový atribut se potom uvádí u entitní množiny s kardinalitou M, resp. u vztahu 1:1 u libovolné z obou entitních množin, případně obou. Z důvodu srozumitelnosti se pro něj používá stejné jméno jako je jméno identifikátoru druhé entitní množiny, jehož hodnot také nabývá, nebo jméno role druhé vztahové množiny. V příští kapitole uvidíme, že takový sloupec skutečně bude v odpovídající tabulce existovat a bude v ní cizím klíčem. V našem příkladu na Obr. 3.21 jsme tyto atributy uvedli u vazební entitní množiny, abychom ukázali, že identifikátory obou původních entitních množin tvoří identifikátor naší vazební množiny. Někteří autoři preferují uvádění takových atributů v ER diagramu. Příklad je uveden na Obr. 3.22.



Obr. 3.22 Příklady atributů reprezentujících vztahy

V prvním příkladu v obrázku je použito jméno role (zde by ani nebylo možné použít jména identifikátoru (proč?)) a ve druhém jméno identifikátoru entitní množiny.



I když je takový atribut v diagramu uveden, nesmí v něm chybět zakreslení odpovídající vztahové množiny. Abychom si uvědomovali, co je vztah a mezi čím a abychom nezapomínali vztahové množiny do diagramu kreslit, nebudeme atributy reprezentující vztahy uvádět. Pokud je přesto uvedete, chyba to nebude.

Dosud jsme se zabývali tím, co tvoří ER model a ER diagram a zásadami, doporučeními a dílčími technikami jejich tvorby. Zatím jsme si ale neuvedli žádný postup, jak pro zadaný problém, vytvořit odpovídající ER diagram. Ukážeme si to nyní. Uvedeme velice jednoduchý postup, který používáme v případě, že máme k dispozici v textové podobě zadání řešeného problému. Zdálo by se, že jde o předpoklad, který v praxi splněn nikdy není. Samozřejmě nikdy nedostaneme na začátku vyčerpávající popis systému, který vyvíjíme. Požadavky na takový systém musíme od zadavatele pracně sbírat během interview s reprezentanty budoucích uživatelů, pomocí dotazníků či jiných technik sběru požadavků. Výsledkem jsou vždy informace, které nějak zpřesňují zadání a tedy informace, které můžeme využít způsobem, který si uvedeme. Postup budeme ilustrovat na následujícím příkladě.



Příklad 3.9

Předpokládejte, že jste dostali za úkol realizovat část informačního systému (IS) fakulty, která bude zahrnovat oblast výuky. Jste ve fázi analýzy a zjistili jste tyto skutečnosti:

- V systému je potřeba uchovávat základní osobní údaje studentů, kterými jsou jméno, adresa a datum narození. Každý student je jednoznačně identifikován svým přihlašovacím jménem, kterým se přihlašuje do fakultní počítačové sítě.
- Studentům jsou nabízeny předměty. Každý předmět má jednoznačnou zkratku, má název, je určitého typu (povinný, volitelný) a má určitý rozsah hodin, kreditovou hodnotu, anotaci.
- Studenti se do předmětů budou přihlašovat prostřednictvím IS. Prohlížíjí si

nabídku předmětů a na vybrané předměty se zapisují. Systém musí hlídat, že celková kreditová hodnota předmětů, které si student v daném akademickém roce zapíše, budou v rozsahu stanoveném předpisy.

- Každý předmět má jednoho garanta z řad učitelů a na jeho výuce se může podílet několik učitelů. U každého učitele, který se na výuce podílí, musí být k dispozici informace o formě výuky (přednáška, cvičení, projekt, ...) a o počtu hodin a studentů, které touto formou vyučuje.
- Pro každý akademický rok se záznam o předmětu vytváří pomocí šablony předmětu, kde jsou základní údaje, jako je název, rozsah, anotace atd., které se ale mohou v jednotlivých letech lišit.
- Systém spravuje rovněž informace o učitelích. Každý učitel má jednoznačné osobní číslo, dále se ukládá jeho jméno, číslo kanceláře a telefon. Garanti předmětu do systému ukládají celkový počet bodů, který studenti z daného předmětu získali. Systém automaticky určí známku.
- Každý učitel je zařazen na nějaký ústav fakulty. Ústav má jednoznačnou zkratku, má název a má vedoucího, kterým je některý z učitelů ústavu.

Postup vychází z jednoduchého jazykového rozboru textu zadání a mohli bychom ho shrnout do následujících kroků:

1. Vyznač v textu kandidáty entitních množin – jsou to typicky podstatná jména, případně ve spojení s přídavnými jmény. Při tom již případně můžeme ignorovat jména, která jsou evidentně irelevantní.
2. U každého označeného kandidáta rozhodni, zda jde skutečně o nějaký objekt modelovaného systému, o kterém bude třeba uchovávat informaci v databázi. Některá jména mohou označovat pouze vlastnosti takových objektů – ta použijeme jako jména atributů příslušných entit. Pro každé vybrané jméno entitní množiny nakresli v ER diagramu odpovídající prvek a doplň hned nebo později nalezená jména atributů. Některá podstatná jména mohou také označovat jména vztahů. Ta uplatníme v kroku 4.
3. Vyznač v textu kandidáty vztahových množin – jsou to typicky slovesa a slovesné vazby.
4. U každého označeného kandidáta rozhodni, zda jde skutečně o vztah, o kterém má být uložena informace v databázi. Dej pozor na slovesa, která označují pouze požadované operace s daty. Pro každé vybrané jméno vztahové množiny nakresli v ER diagramu odpovídající prvek a doplň hned nebo později případné atributy vztahů. Některé vztahové množiny mohou být pojmenovány použitím jmen rolí (podstatné jméno).
5. Pokud se v diagramu vyskytuje ternární nebo dokonce vztah s vyšším stupněm, zvaž, zda ho nelze nahradit vztahy binárními.
6. Pro každou vztahovou množinu v ER diagramu urči kardinalitu. Nezapomeň, že kardinalita se určuje pro každý konec vztahu. Z pohledu návrhu struktury databáze je rozhodující maximální kardinalita, minimální kardinalita může zavádět další integritní omezení. Pokud ze zadání hodnota kardinality není naprosto zřejmá, je nutné význam vztahu zpřesnit dotazem u zadavatele.
7. Pro každou entitní množinu urči, případně doplň, atribut který bude identifikátorem (primárním klíčem). Může to být i atribut složený.
8. Proveď případné další úpravy vedoucí ke zvýšení srozumitelnosti či zpřesnění

významu – zavedení vztahu generalizace/specializace, slabých entitních množin, číselníků apod., je-li to vhodné a účelné.

Postupně provedeme tyto kroky na našem příkladě.

Krok 1 – označení kandidátů entitních množin

Předpokládejte, že jste dostali za úkol realizovat část informačního systému (IS) fakulty, která bude zahrnovat oblast výuky. Jste ve fázi analýzy a zjistili jste tyto skutečnosti:

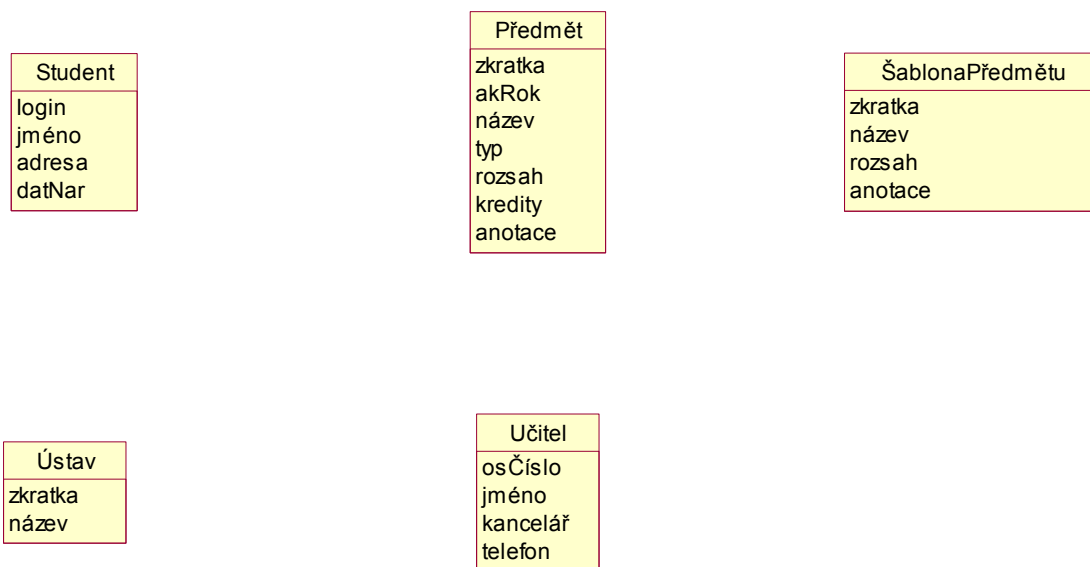
- V systému je potřeba uchovávat základní osobní údaje studentů, kterými jsou jméno, adresa a datum narození. Každý student je jednoznačně identifikován svým přihlašovacím jménem, kterým se přihlašuje do fakultní počítačové sítě.
- Studentům jsou nabízeny předměty. Každý předmět má jednoznačnou zkratku, má název, je určitého typu (povinný, volitelný) a má určitý rozsah hodin, kreditovou hodnotu, anotaci.
- Studenti se do předmětů budou přihlašovat prostřednictvím IS. Prohlížíjí si nabídku předmětů a na vybrané předměty se zapisují. Systém musí hlídat, že celková kreditová hodnota předmětů, které si student v daném akademickém roce zapíše, budou v rozsahu stanoveném předpisy.
- Každý předmět má jednoho garanta z řad učitelů a na jeho výuce se může podílet několik učitelů. U každého učitele, který se na výuce podílí, musí být k dispozici informace o formě výuky (přednáška, cvičení, projekt, ...) a o počtu hodin a studentů, které touto formou vyučuje.
- Pro každý akademický rok se záznam o předmětu vytváří pomocí šablony předmětu, kde jsou základní údaje, jako je název, rozsah, anotace atd., které se ale mohou v jednotlivých letech lišit.
- Systém spravuje rovněž informace o učitelích. Každý učitel má jednoznačné osobní číslo, dále se ukládá jeho jméno, číslo kanceláře a telefon. Garanti předmětu do systému ukládají celkový počet bodů, který studenti z daného předmětu získali. Systém automaticky určí známku.
- Každý učitel je zařazen na nějaký ústav fakulty. Ústav má jednoznačnou zkratku, má název a má vedoucího, kterým je některý z učitelů ústavu.

Záměrně jsme v tomto kroku označili i některá jména, která jsou irelevantní, např. systém (víme, že celý systém jako entitní množinu nemodelujeme), počítačová síť, nabídka předmětů (jde o výsledek nějaké operace s daty). V tomto kroku jsme už částečně prováděli klasifikaci kandidátů – jména potenciálně označující atributy jsme označili, na rozdíl od kandidátů na jména entitních množin čárkovaně.

Krok 2 – zpracování označených kandidátů entitních množin

Již bylo uvedeno, že některá označená jména jsou irelevantní. Některá jména označují totéž, například „předmět“ a „záznam o předmětu“. Podstatné jméno „garant“ označuje roli učitele ve vztahu k předmětu. Někoho by zde možná napadlo i to, že jde o specializaci učitele. K tomuto pohledu se ještě vrátíme za chvíli. Za zmínku stojí jméno „akademický rok“. Mohlo by se třeba zdát, že jde o jméno irelevantní, ale není tomu tak. Jde o atribut a to předmětu. Navíc z odstavce zadání, kde je úloha akademického roku ve vztahu k šabloně zmíněna, je zřejmé, že atributy šablony se

budou opakovat i u předmětu, protože hodnoty se v šabloně mohou rok od roku lišit. Základ ER diagramu po druhém kroku by mohl vypadat podle Obr. 3.23.



Obr. 3.23 ER diagram příkladu Příklad 3.9 po vyznačení entitních množin

Krok 3 – vyznačení kandidátů vztahových množin

Předpokládejte, že jste dostali za úkol realizovat část informačního systému (IS) fakulty, která bude zahrnovat oblast výuky. Jste ve fázi analýzy a zjistili jste tyto skutečnosti:

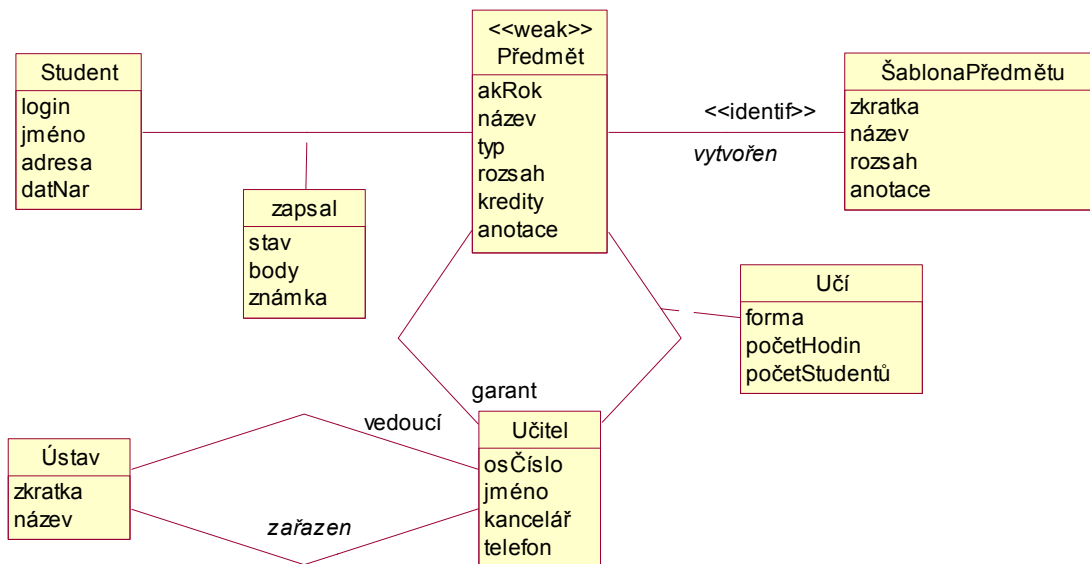
- V systému je potřeba uchovávat základní osobní údaje studentů, kterými jsou jméno, adresa a datum narození. Každý student je jednoznačně identifikován svým přihlašovacím jménem, kterým se přihlašuje do fakultní počítačové sítě.
- Studentům jsou nabízeny předměty. Každý předmět má jednoznačnou zkratku, má název, je určitého typu (povinný, volitelný) a má určitý rozsah hodin, kreditovou hodnotu, anotaci.
- Studenti se do předmětů budou přihlašovat prostřednictvím IS. Prohlížíjí si nabídku předmětů a na vybrané předměty se zapisují. Systém musí hlídat, že celková kreditová hodnota předmětů, které si student v daném akademickém roce zapíše, budou v rozsahu stanoveném předpisy.
- Každý předmět má jednoho garanta z řad učitelů a na jeho výuce se může podílet několik učitelů. U každého učitele, který se na výuce podílí, musí být k dispozici informace o formě výuky (přednáška, cvičení, projekt, ...) a o počtu hodin a studentů, které touto formou vyučuje.
- Pro každý akademický rok se záznam o předmětu vytváří pomocí šablony předmětu, kde jsou základní údaje, jako je název, rozsah, anotace atd., které se ale mohou v jednotlivých letech lišit.
- Systém spravuje rovněž informace o učitelích. Každý učitel má jednoznačné osobní číslo, dále se ukládá jeho jméno, číslo kanceláře a telefon. Garanti předmětu do systému ukládají celkový počet bodů, který studenti z daného předmětu získali. Systém automaticky určí známku.
- Každý učitel je zařazen na nějaký ústav fakulty. Ústav má jednoznačnou

zkratku, má název a má vedoucího, kterým je některý z učitelů ústavu.

Krok 4 – zpracování označených kandidátů vztahových množin

Opět jsme úmyslně označili i některá jména irelevantní, např. „je potřeba uchovávat“ (nejde o žádný vztah mezi entitními množinami, které jsme už identifikovali), „je identifikován“ (jde o vztah atributu a entitní množiny – ten využijeme, až budeme určovat identifikátory vztahových množin). Nutnou podmínkou pro to, aby jméno reprezentovalo námi hledaný vztah je, že musíme z věty zjistit i jména příslušných vztahových množin, např. „záznam o předmětu vytváří pomocí šablony“. Ani to ale ještě nemusí znamenat, že jde o vztah, který máme modelovat. Teď je ten okamžik, kdy si musíme položit otázku: „Jde skutečně o informaci, která má být uložena v databázi?“ Porovnejte v tomto smyslu například „studenti si prohlížejí nabídku předmětů“ a „studenti si zapisují předměty“. Zatímco odpověď na naši otázku bude v prvním případě „Ne“, protože jde pouze o operaci, ve druhém případě bude „Ano“, protože určitě bude potřeba mít uloženou informaci o tom, kdo si co zapsal. Opět by mělo platit, že pokud jsme na pochybách, je potřeba v rámci sběru a analýzy požadavků takovou informaci zpřesnit. V některých případech může být v textu i vhodné jméno pro roli ve vztahu – u nás vedoucí a garant. V tomto kroku jsme již doplnili i atributy vztahů. V tomto kroku také případně zvažujeme modelování některých entitních množin jako slabých. My jsme se takto rozhodli modelovat entitní množinu Předmět, která je závislá na množině ŠablonaPředmětu. Vypustili jsme také díky identifikujícímu vztahu atribut zkratka z atributů entitní množiny Předmět, protože jsme se dohodli, že atributy reprezentující vztahy uvádět nebudeme.

ER diagram po čtvrtém kroku by mohl vypadat podle Obr. 3.24.



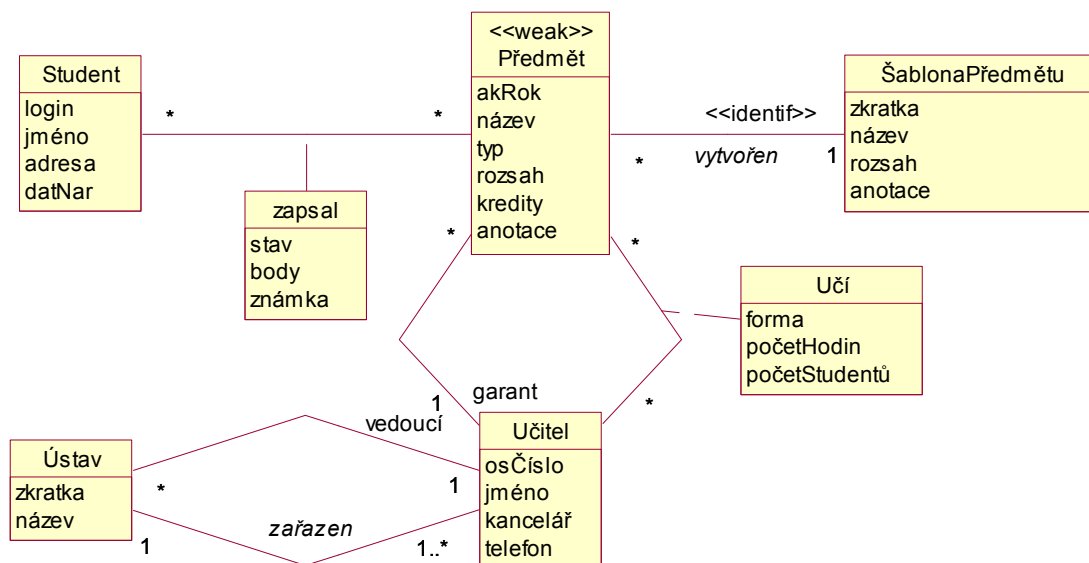
Obr. 3.24 ER diagram příkladu Příklad 3.9 po přidání vztahových množin

Krok 5 – náhrada vztahů vyššího stupně

Všechny vztahy v našem diagramu jsou binární.

Krok 6 – určení kardinality vztahů

Kardinalita vztahů je zřejmá ze zadání. Diagram po tomto kroku je uveden na Obr. 3.25.



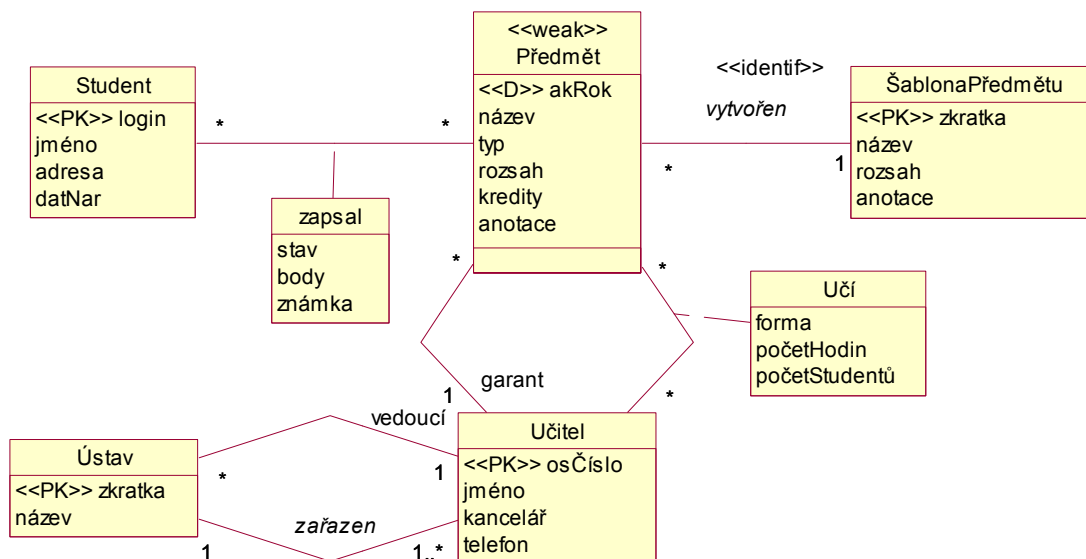
Obr. 3.25 ER diagram příkladu Příklad 3.9 po určení kardinality

Všimněte si, že hodnota minimální kardinality může být v některých případech diskutabilní. Například podle našeho diagramu student nemusí mít zapsaný žádný předmět. To je určitě pravda před zápisem nebo přihlašованиеm do předmětů, ale ne po zápisu. Požadované hlídání celkové kreditové hodnoty by se opět objevilo v některém jiném modelu nebo dokumentu popisujícím požadované kontroly.

Krok 7 – určení identifikátorů entitních množin

Informace o tom, které atributy nabývají u entit jednoznačných hodnot, je uvedena v zadání. Za zmínku stojí pouze identifikátor entitní množiny Předmět. V zadání je uvedeno, že předmět má jednoznačnou zkratku. Zároveň ale ze zadání vyplývá, že v databázi budou informace týkající se nejen jednoho akademického roku. To se projevilo tím, že entity typu Předmět mají jako jeden ze svých atributů akademický rok. Důsledkem je, že hodnota zkratky předmětu už neurčuje jednoznačně předmět v rámci všech předmětů ve všech akademických rocích, o nichž bude informace v databázi. Jednoznačná je pouze hodnota složeného atributu (zkratka, akRok). Naproti tomu u šablony je identifikátorem zkratka, protože u šablony se uchovávají pouze aktuální hodnoty (původní hodnoty po případné modifikaci jsou u předmětu). Proto zde akademický rok jako atribut vůbec nefiguruje.

ER diagram po doplnění identifikátorů entitních množin je na Obr. 3.26.



Obr. 3.26 ER diagram příkladu Příklad 3.9 po určení identifikátorů entitních množin

Krok 8 – další úpravy modelu

V našem příkladu by tyto úpravy mohly zahrnovat následující úvahy:

- Předmět by mohl být modelován jako slabá entitní množina ve vztahu k ŠabloněPředmětu s diskriminátorem akRok.
- Hodnoty atributů typ u Předmětu a forma u Učí by nabývaly nějakých číselníkových hodnot.
- Role garant a vedoucí by mohly svádět k použití generalizace specializace – garant je speciálním případem učitele. Toto ale v tomto případě není správné, protože to platí pouze ve vztahu k nějakému předmětu. Ve vztahu k jinému předmětu už daný učitel být garantem nemusí. V případě vedoucího by použití specializace bylo možné, ale bylo by zbytečné, protože u vedoucího není podle zadání potřeba uchovávat žádné další atributy a specializace by proto nic nepřinášela. I v tomto případě je vhodnější použití pouze role.

Případné závěrečné úpravy ER diagramu z Obr. 3.26 si můžete provést jako domácí cvičení.



Zamyslete se nad tím, jestli bylo možné modelovat požadavek na uchování původních hodnot z šablony (rozsah, kredity, anotace) i jiným způsobem.



Přestože ER model a jeho grafické vyjádření - ER diagram jsou typickými prostředky pro konceptuální modelování při návrhu relační databáze, i tato etapa vývoje byla ovlivněna objektově-orientovaným přístupem k vývoji programů a jeho modelovacími technikami. Můžeme říci, že i zde při konceptuálním modelování můžeme použít objektově-orientovaný přístup s tím, že pro návrh relační databáze využijeme jenom některé jeho techniky. Vy znáte základy objektové orientace a používaných modelovacích technik z předmětu Úvod do softwarového inženýrství. Tam jste se také seznámili se základy jazyka UML a jeho digramy. Když se zamyslete nad tím, jakou roli konceptuální model při návrhu databáze hraje, pak určitě dojdete k závěru, že

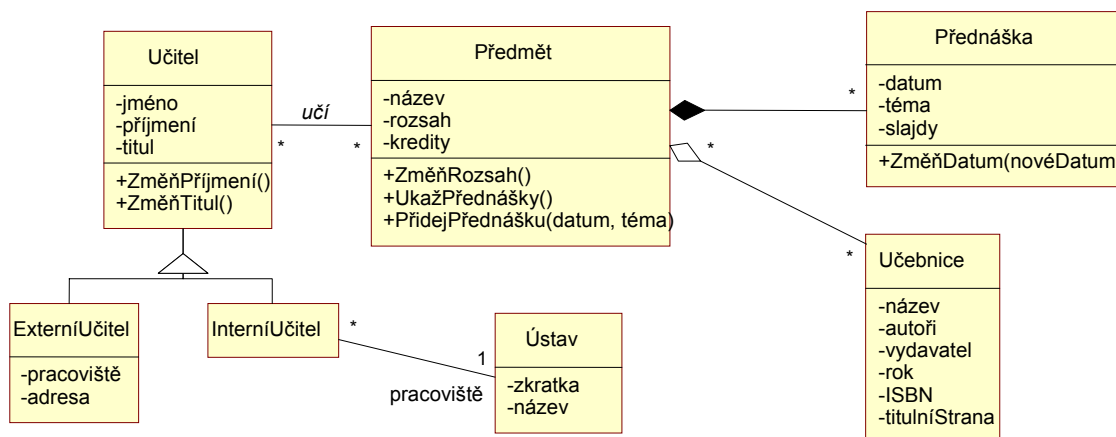
tyto účely použít, je diagram tříd. Samozřejmě diagram tříd nám poskytuje řadu možností, které při použití pro návrh relační databáze přímo nevyužijeme. Vyplývá to z toho, že klasická relační databáze je prostředkem pro ukládání perzistentních strukturovaných dat. Návrh databáze neřeší otázku operací s těmito daty. Vy víte, že toto oddělení pohledu na data a operace s nimi je vlastní klasickému, tj. strukturovanému přístupu n vývoji programů. Objektově-orientovaný přístup naopak data a operace s nimi zapouzdřuje do podoby objektů. Pokud toto vezmeme do úvahy, pak při použití diagramu tříd pro nás budou podstatné třídy, atributy objektů a vztahy mezi třídami, resp. objekty těchto tříd. Z tohoto pohledu najdeme řadu analogií diagramu tříd a ER diagramu. Nejdůležitější z nich shrnuje tabulka Tab. 3.1.

Tab. 3.1 Analogie některých prvků diagramu tříd a ER diagramu

Prvek diagramu tříd	Prvek ER diagramu
třída	entitní množina
objekt	entita
atribut objektu třídy	atribut entity
-	identifikátor entity (primární klíč)
identifikátor objektu (OID)	-
asociace	vztahová množina
vazba (link)	vztah
objekt asociační třídy	vztah s atributy
kompozice (celek/část)	vztah identifikující a slabé ent.množiny
generalizace/specializace/dědičnost	generalizace/specializace

Všimněte si, že v případě identifikátoru, resp. primárního klíče odpovídající prvek neexistuje. Je to dáno způsobem identifikace objektů, resp. entit. Zatímco jedním ze základních vlastností objektové orientace je, že objekty jsou identifikovatelné nezávisle na hodnotách svých atributů, entity jsou identifikovatelné hodnotou primárního klíče, což je jeden z jejich atributů (případně složený). Každý objekt má jednoznačnou hodnotu tzv. identifikátoru objektu (OID), který můžeme chápat jako „zabudovaný“ atribut. Naproti tomu v ER diagramu musíme u každé entitní množiny takový atribut definovat.

Příklad diagramu tříd je uveden na Obr. 3.27.



Obr. 3.27 Příklad diagramu tříd

Z obrázku jsou vidět některé další rozdíly mezi objektově-orientovaným konceptuálním modelem v podobě diagramu tříd a ER modelem. V první řadě jsou to operace, které vyplývají z faktu, že objekt obecně zapouzdřuje data a chování, vyjádřené právě operacemi. My víme, že relační databáze ukládá pouze data, a proto operace z pohledu návrhu databáze pro nás nejsou tak důležité. Existují ale i rozdíly, týkající se atributů, které musíme respektovat při transformaci diagramu tříd na tabulky relační databáze. Poměrně časté je například použití vnořených objektů. Atribut adresa třídy ExterníUčitel by mohl být příkladem takového zanořeného objektu. Jde sice o obdobu složeného atributu, se kterým jsme se setkali i u ER modelu, ale v diagramech tříd se s vnořenými objekty setkáváme častěji. Podobně se můžeme častěji setkat s atributy typu kolekce, které zase odpovídají vícehodnotovým atributům v ER diagramu. V našem příkladu by takovým atributem mohl být atribut autoři u třídy Učebnice. V kapitole 3.3, kde se budeme zabývat transformací konceptuálního modelu na schéma relační databáze, se k těmto odlišnostem ještě vrátíme.

Σ

V této části studijní opory jsme se seznámili s prostředky a technikami, které používáme v první fázi návrhu databáze – při konceptuálním modelování. Zopakovali a rozšířili jste si poznatky týkající se konceptuálního modelování použitím ER modelu, jehož grafickým vyjádřením je ER diagram. Přestože jde o techniku, která vznikla již v polovině 70. let minulého století, je to stále nejpoužívanější způsob konceptuálního modelování pro návrh relační databáze. Zaměřili jsme se proto při rozšíření poznatků, které máte z předmětu Úvod do softwarového inženýrství, zejména na ty vlastnosti ER modelu, které hrají při návrhu schématu databáze klíčovou roli. Po zvládnutí této kapitoly byste měli být schopni jednak porozumět ER diagramu, který vytvořil někdo jiný a vy máte na jeho základě navrhnout schéma databáze, jednak být schopni sami vytvořit správný ER model a odpovídající diagram pro zadaný problém. Druhá úloha je určitě náročnější. K jejímu zvládnutí je třeba, abyste především:

- používali výstižná jména pro prvky modelu, aby byl diagram snadno čitelný a srozumitelný,
- porozuměli rozdílu mezi statickými vztahy, které modelujeme a znázorněním operací s daty, které naopak do ER diagramu nepatří,
- byli schopni správně určit kardinalitu vztahové množiny,
- byli schopni použít generalizaci/specializaci

- rozuměli použití slabých entitních množin a v případě potřeby je byli schopni správně použít,
- rozuměli tomu, co je složený a vícehodnotový atribut,
- uměli v případě potřeby nahradit vztahovou množinu s kardinalitou 1:M vazební vztahovou množinou.

Ukázali jsme si také jednoduchý základní postup, jak na základě textového popisu řešeného problému vytvořit odpovídající ER diagram.

V závěru této podkapitoly jsme si také uvedli, že při použití objektově-orientovaného přístupu ke konceptuálnímu modelování používáme diagram tříd, uvedli jsme si analogie s ER diagramem a naopak nejvýznamnější odlišnosti z pohledu návrhu relační databáze.



V základní literatuře [Pok02] o konceptuálním modelování nic nenajdete. Stručnou informaci o ER modelu lze najít v [Pok98] na stranách 73 až 77. Učebnice [Sil05] vysvětluje ER model na str. 201 až 261. Je zde rovněž zmínka o použití UML pro datové modelování. Rovněž v doporučené učebnici [Dat03] je věnována kapitola ER modelování. V obou těchto učebnicích je uvedena také řada otázek a příkladů k procvičení.

Další užitečnou literaturou k problematice konceptuálního modelování může být kniha Hawryszkiewicz, I.T.: Relational Database Design. An Introduction. Prentice Hall Inc. 1990, kde na str. 85 – 152 lze najít řadu dalších doporučení a modelovacích technik souvisejících s ER diagramy. Sofistikovanější metody konceptuálního modelování, které se uplatní při modelování rozsáhlých systémů, lze najít v knize Batini, C., Ceri, S., Navathe, S., B.: Conceptual Database Design. Benjamin/ Cummings. 1992.

Cvičení:



- C3.1** Vysvětlete pojem konceptuální modelování a jeho pozici v procesu návrhu databáze.
- C3.2** Vysvětlete dva základní přístupy k návrhu databáze – založený na transformaci ERD a využitím normalizace.
- C3.3** Vysvětlete pojmy entita entitní množina, vztah a vztahová množina. Ilustrujte na příkladě.
- C3.4** Vysvětlete pojmy atribut entity a identifikátor entitní množiny. Ilustrujte na příkladě.
- C3.5** Vysvětlete pojmy složený, vícehodnotový atribut a atribut povolující prázdnou hodnotu.
- C3.6** Vysvětlete pojmy stupeň vztahu, kardinalita a členství ve vztahu. Ilustrujte na příkladě.
- C3.7** Charakterizujte výstižně pojem slabá entitní množina a kdy se při konceptuálním modelování používá. Ilustrujte na příkladě.
- C3.8** Charakterizujte výstižně pojem generalizace/specializace a kdy se při konceptuálním modelování používá. Ilustrujte na příkladě.
- C3.9** Vysvětlete zvláštnosti použití diagramu tříd pro konceptuální modelování.
- C3.10** Při analýze požadavků na editor elektrotechnických schémat, jste zjistili, že schéma je tvořeno symboly prvků (např. tranzistor, rezistor) a spoji. Každý prvek je charakterizován řadou atributů (uvažujte jen jméno, typ, pozice), jejichž

hodnoty je třeba uchovávat. Každý prvek má alespoň jeden vývod. I vývody mají řadu atributů (uvažujte jen jméno, typ, pozice). Každý spoj má dva koncové body, kterými může být buď vývod prvku nebo tzv. "propojka", která značí vodivé připojení k jinému spoji ("tečka" ve schématu), sběrnice neuvažujte. Propojky jsou nepojmenované, mezi atributy patří pozice. Spoje jsou pojmenované, případné body zlomu neuvažujte. U spoje musí být informace o koncových bodech, u propojky, na kterém spoji (jednom) leží.

Nakreslete ER diagram, který bude reprezentovat výše uvedené požadavky.

- C3.11** Předpokládejte, že analyzujete požadavky na bankovní informační systém a zjistili jste tyto skutečnosti: Každý klient (jméno, adresa, atd.) vlastní jeden nebo více účtů (číslo, stav, atd.). Informace o vlastníkovi účtu musí být k dispozici. Ke každému účtu může být vydáno několik karet (číslo, doba platnosti, atd.) opravňujících k manipulacím (tzv. transakcím - např. vklad, výběr) s účtem. Majitel karty je klientem banky, nemusí to ale být nutně majitel daného účtu. Rozlišují se dva typy transakcí - lokální (u přepážky pobočky banky) a vzdálená (z bankomatu). U každé transakce je třeba, kromě jiných údajů (číslo, datum, doba, typ operace, částka, atd.) vědět, kterého se týká účtu a odkud byla zadána (v případě lokální jde o informaci o přepážce, v případě vzdálené o informaci o bankomatu a kartě). Systém bude obsahovat informace o všech existujících bankomatech (číslo, adresa, atd.) a přepážkách (pobočka, číslo přepážky, apod.).

Nakreslete ER diagram, který bude reprezentovat výše uvedené požadavky.

- C3.12** Provádíte analýzu jednoduchého systému, který umožní výzkumnému ústavu vést agendu řešených projektů. Systém bude sloužit jenom tomuto jednomu ústavu. Ten je členěn na oddělení (číslo, název), do nichž jsou zaměstnanci zařazeni. Většina ze zaměstnanců (osobní číslo, jméno, rok nástupu do ústavu) je zapojena do řešení nějakého projektu. Každý projekt má jednoznačné číslo, dále má jméno a některé další atributy, které je třeba registrovat. Jeden zaměstnanec může být zapojen do řešení několika projektů a samozřejmě jeden projekt řeší několik řešitelů. Systém musí být schopen uchovávat informaci o době, kterou každý z řešitelů odpracoval na jednotlivých projektech, které řeší. Každý projekt má jednoho zodpovědného řešitele, který může být současně zodpovědný za několik takových projektů. Předpokládejte, že systém pracuje vždy s aktuálními informacemi, tj. změnil-li se zodpovědný řešitel, je třeba znát pouze aktuálního. Systém musí být schopen mimo jiné zodpovědět tyto dotazy:

- Kolik hodin bylo odpracováno na jednotlivých projektech?
- Za kolik projektů zodpovídají jednotlivá oddělení (tj. je z nich zodpovědný řešitel)?
- Na kolika projektech se zodp. vedoucím z jiného oddělení se podílí řešitelé daného oddělení?

Nakreslete ER diagram, který bude reprezentovat výše uvedené požadavky.

- C3.13** Předpokládejte, že analyzujete požadavky na systém, který bude poskytovat počítačovou podporu pro kreslení stavebních výkresů. Z informací, které máte dosud k dispozici, vyplývá, že jednotkou, se kterou bude systém pracovat, bude výkres. Každý výkres bude mít svůj název, autora, datum poslední změny a řadu dalších atributů. Výkres obsahuje jednu nebo více tzv. vrstev (např. vrstva s půdorysem budovy, rozvody plynu, elektřiny apod.), do nichž se umísťují geometrické útvary. Na jedné vrstvě se může nacházet řada geometrických útvarů, ale každý z nich je vždy jen na jedné vrstvě. Každá vrstva má své jméno, které je

jednoznačné v rámci jednoho výkresu.

Geometrické útvary lze rozdělit do dvou skupin – primitivní a složené. Jako primitivní uvažujte bod, lomenou čáru a uzavřenou oblast. Složené útvary vznikají seskupením jiných útvarů (primitivních i složených). Protože musí být k dispozici i operace, která rozloží složený útvar na útvary, jejichž seskupením vznikl, musí být tato informace (tj. které prvky seskupení tvoří) k dispozici.

Jedním z dalších požadavků je, aby systém pracoval s určitými rozšiřitelnými paletami – barev, typů čar a typů výplně. Každá vrstva má potom definovanou implicitní barvu (jedna barva z palety), každý primitivní prvek může mít definovanou jinou barvu (opět z palety). Podobně lomená čára a uzavřená oblast mají definován typ čáry a uzavřená oblast navíc typ výplně – opět z příslušných palet.

Nakreslete ER diagram, který bude reprezentovat výše uvedené požadavky.

- C3.14** Dopravní firma vlastní automobily různých kategorií (uvažujte jen otevřený nákladní, skříňový, krytá dodávka). Zákazníci si objednávají vozidla určitých kategorií na jednotlivé jízdy (vždy s řidičem) nebo na jízdy v určitém období (s řidičem nebo bez řidiče). Jízda je vždy součástí denního plánu jízd. Existují dvě kategorie zákazníků - náhodný zákazník, který dostává účet bezprostředně po ukončení jízdy, a stálý zákazník, který má otevřen u firmy stálý účet, na jehož základě dostává vždy fakturu za určité časové období. Uvažujte, že platba jízdy je doložena platebním dokladem (předpokládejte vždy jeden na jízdu), kterým je buď potvrzení o platbě náhodného zákazníka nebo položka účtu stálého zákazníka.

Při analýze jste identifikovali tyto entitní množiny: Automobil a jeho specializace, Zákazník a jeho specializace, Jízda, Řidič, StálýÚčet, PlánJízd pro daný den, PlatebníDoklad a jeho specializace, Faktura.

Nakreslete ER diagram, který bude reprezentovat výše uvedené požadavky.

- C3.15** Předpokládejte, že analyzujete požadavky na informační systém peněžního ústavu a zjistili jste tyto skutečnosti:

Ústav spravuje mimo jiné účty podnikatelských subjektů (firem). Každý účet má jednoho vlastníka, má jednoznačné číslo a musí být k dispozici informace o částce na účtu (stavu). Jeden podnikatelský subjekt může vlastnit více účtů. Podnikatelským subjektem může být buď právnická osoba (např. s.r.o.) nebo fyzická osoba (podniká na základě živnosti). U každého subjektu je třeba znát obchodní název, adresu a IČO, u právnické osoby navíc datum zápisu do obchodního rejstříku, u fyzické adresu bydliště osoby (může být jiná než sídla firmy). Ke každému účtu právnické osoby existuje seznam lidí, kteří mohou s účtem disponovat (předpokládejte potřebu znát jméno a číslo občanského průkazu). Z účtu lze vybírat peníze, ukládat peníze a převádět na jiný účet (předpokládejte téhož peněžního ústavu). Každá taková operace probíhá na základě bankovního příkazu, který musí obsahovat informaci o datu, typu operace, částce a pro účty právnických osob i informace o disponující osobě, která příkaz zadala. V případě operace převodu na jiný účet se u cílového účtu automaticky vytvoří příkaz s operací „převod z“. U příkazů „převod na“ a „převod z“ je nutné znát cílový, resp. zdrojový účet.

Nakreslete ER diagram, který bude reprezentovat výše uvedené požadavky.

- C3.16** Předpokládejte, že analyzujete požadavky na informační systém katastru nemovitostí. Systém bude pracovat především s informacemi o nemovitostech a jejich vlastnících. Předpokládejte, že každá nemovitost má jednoznačné číslo, má

název a datum zápisu do katastru. Uvazujte dva typy nemovitosti – budovy a pozemky. U budovy je třeba uchovávat informaci o zastavěné ploše a počtu podlaží, u pozemků o rozloze a typu pozemku (stavební parcela, zahrada, ...). U vlastníka musí být k dispozici jméno a adresa. Každá nemovitost může mít několik spoluvlastníků, v takovém případě musí být znám podíl každého z nich. Jeden vlastník může vlastnit více nemovitostí. U budov musí být informace o pozemku, na němž stojí. Předpokládejte, že může být pouze jedna budova na pozemku a že žádná budova nestojí na více než jednom pozemku. Důležitou funkcí systému bude pořizování, tzv. výpisu z katastru nemovitostí, což jsou informace o dané nemovitosti, včetně informace o vlastnících.

Nakreslete ER diagram, který bude reprezentovat výše uvedené požadavky.

- C3.17** Společnost zajišťující dodávky elektřiny do domácností vám dala zakázku na vytvoření informačního systému, který bude mimo jiné pracovat s informacemi o odběratelích a odběrech. Každý odběratel má jednoznačné číslo a uchovává se informace o jeho jménu a adrese. Elektřina je dodávána odběrateli prostřednictvím tzv. odběrních míst (odběratel jich může mít více). Je třeba znát adresu odběrného místa (nemusí být jednoznačná). Každé odběrné místo může zahrnovat několik elektroměrů. U elektroměru je třeba uchovávat informace o jeho čísle a počátečním stavu. Číslo elektroměru je jednoznačné. Společnost má rozděleno území, pokryté dodávkami, na oblasti, za které zodpovídají technici společnosti. Předpokládejte, že za jednu oblast může zodpovídat více techniků a jeden technik může mít na starosti několik oblastí. Pro každé odběrné místo pak musí být k dispozici informace, kteří technici za ně zodpovídají. Informace o technících zahrnuje jejich osobní číslo (jednoznačné) a jméno. Mezi povinnosti těchto techniků patří také provádět odečty stavů elektroměrů v daném místě. K dispozici musí být informace o datu odečtu, stavu elektroměru a kdo odečet provedl.

Každý odběratel provádí zálohové platby ve stanoveném režimu, např. měsíčně. K dispozici musí být informace o datu platby a výši částky. Platby jednoho odběratele jsou průběžně číslovány. Po odečtu stavů elektroměrů se provádí vyúčtování. Předpokládejte nutnost uchovávat pouze informace o období (od, do) a doplatku/nedoplatku.

Nakreslete ER diagram, který bude reprezentovat výše uvedené požadavky.

- C3.18** Analyzujete požadavky na internetovou aplikaci, která bude rozšiřovat možnosti již existující prezentace počítačové firmy na Internetu, včetně nabídky jejích produktů (katalog) o nákup prostřednictvím internetu. Pracovní název aplikace je Internetový obchod.

Zboží, které firma nabízí, je v katalogu rozčleněno do určitých kategorií (např. sestavy, disky, monitory, atd.). Každá kategorie má jednoznačný název a dále má bližší popis. Zboží je zařazeno vždy pouze do jedné z kategorií. Seznam kategorií není pevný, musí být možné ho rozšiřovat. Každé zboží má přidělen jednoznačný kód, má název a aktuální cenu. Zatímco katalog si může prohlížet kdokoli, nákup mohou provádět pouze zákazníci, kteří se u firmy registrovali vyplněním nějakého formuláře s osobními údaji (uvažujte jen jméno, příjmení, adresa, zvolené přihlašovací jméno a heslo). Tyto informace jsou po odeslání formuláře uloženy v databázi a jednou registrovaný zákazník se při příštích nákupech hlásí pouze přihlašovacím jménem a heslem. Vlastní nákup probíhá tak, že zákazník si nejprve vybere kategorii, poté se mu zobrazí veškeré nabízené zboží v této kategorii a on si vybírá. Nákup vybraného zboží vždy potvrdí stiskem nějakého tlačítka u tohoto

zboží a zadáním počtu kusů. Každé takto zakoupené zboží tvoří jednu položku tzv. nákupního košíku. Aplikace musí umožnit zobrazit obsah nákupního košíku, ve kterém může zákazník případně rušit nějaké položky. Na závěr nákupu zákazník potvrdí obsah nákupního košíku a aplikace uloží trvale do databáze veškeré informace o nákupu, včetně data a času nákupu. Protože ceny zboží se mohou měnit, je třeba u každého zakoupeného zboží uložit i informaci o ceně platné v době nákupu.

Fakturaci a sledování plateb již zajišťuje jiná aplikace (účetnictví), která má přístup k datům, vytvořeným aplikací Internetový obchod. Předpokládejte pro účely analyzované aplikace, že u každého nákupního košíku bude uvedeno číslo faktury zaslané zákazníkovi a příznak o zaplacení.



Test

T1. Transformace ERD na tabulky relační databáze patří do etapy

- a) logického návrhu databáze
- b) fyzického návrhu.

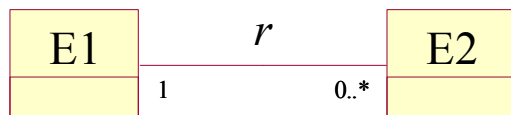
T2. ER diagram jako konceptuální model vzniká, aby reprezentoval

- a) strukturu databázového schématu navrženého na základě požadavků zákazníka vyvíjené databázové aplikace
- b) požadavky zákazníka vyvíjené databázové aplikace na perzistentní data spravovaná aplikací.

T3. Vícehodnotový atribut je atribut, který

- a) je definován na doméně zahrnující více hodnot
- b) může v daném okamžiku nabývat více než jedné hodnoty.

T4. Kardinalita vztahu r v následujícím obrázku říká, že



- a) jedna entita typu E1 nemusí být ve vztahu typu r s žádnou entitou typu E2
- b) jedna entita typu E2 nemusí být ve vztahu typu r s žádnou entitou typu E1

T5. Použijeme-li v ERD generalizaci/specializaci, pak entitní množina nižší úrovně dědí z entitní množiny vyšší úrovně:

- a) všechny atributy a operace
- b) pouze atribut, který je primárním klíčem.

3.3. Transformace konceptuálního modelu na tabulky relační databáze

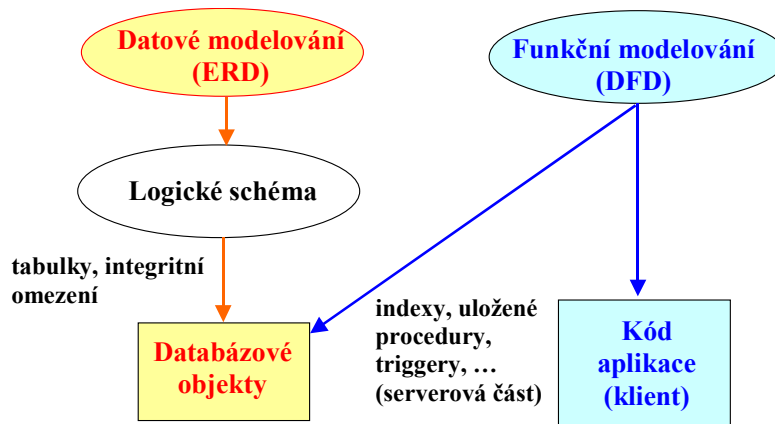
V této kapitole se budeme zabývat krokem logického návrhu relační databáze, tj. ukážeme si, jak na základě konceptuálního modelu v podobě ER diagramu nebo diagramu tříd navrhujeme schéma odpovídající relační databáze. Nejprve si ukážeme postup pro ER diagram. Vysvětlíme si, jaké jsou typické rysy chybného návrhu a uvedeme si pravidla transformace ER diagramu na tabulky, která zajistí, že naše schéma bude dobře navrženo. V závěru se potom také zmíníme o transformaci diagramu tříd. Uvidíme, že pravidla transformace jsou stejná, takže se spíše zaměříme na zdůraznění některých rozdílů diagramu tříd od ER diagramu, na které musíme při transformaci na tabulky pamatovat. Již jsme v tomto smyslu na upozornili v předchozí

kapitole. V obou případech, tj. transformace ER diagramu i diagramu tříd, ale nejprve ještě shrneme úlohu modelů, které vznikají v průběhu analýzy a specifikace požadavků.

V předchozí kapitole jsme se zabývali problematikou konceptuálního modelování, které je prvním krokem při návrhu schématu databáze. Jestliže vytváříme programový systém, který bude ukládat data do databáze (takový systém zde budeme nazývat databázovou aplikací), je návrh schématu databáze jenom jedním dílčím návrhem. V prvé řadě je třeba navrhnout celkovou architekturu aplikace, navrhnout uživatelské rozhraní a navrhnout jednotlivé procedury a funkce, aby byly splněny nejen požadavky na data, která se budou ukládat, ale i na celkovou funkčnost, kterou má aplikace poskytovat.

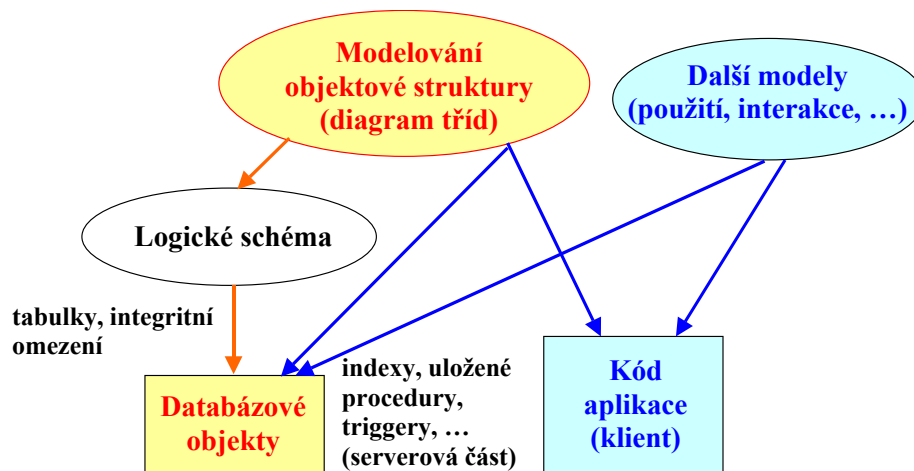
Z úvodní přednášky předmětu víte, že v současné době je naprosto převažující architekturou databázových aplikací je dvouvrstvá nebo vícevrstvá architektura klient-server. V obou případech v takové architektuře najdeme vrstvu databázového serveru a databázového klienta. Na zajištění požadované funkčnosti databázové aplikace se potom podílí obě tyto vrstvy a úlohou návrhu je návrh jak serverové, tak klientské části aplikace. Návrh serverové části zahrnuje návrh schématu databáze, které obsahuje databázové objekty a integritní omezení, která budou kontrolována na straně databázového serveru. Návrh klientské části potom zahrnuje především návrh operací s daty uloženými v databázi, případně i návrh prezentační vrstvy (uživatelského rozhraní). V klasické relační databázi jsou jedinými databázovými objekty tabulky. Moderní relační databáze ale mohou obsahovat kromě tabulek i řadu jiných databázových objektů, které mohou obsahovat i kód, který se provádí na straně databázového serveru. Příkladem objektů tohoto typu jsou databázové triggery nebo uložené procedury a funkce. Podrobněji se s nimi seznámíme později.

V průběhu analýzy a specifikaci požadavků vzniká řada modelů, které reprezentují požadavky na vyvíjenou databázovou aplikaci. Obrázek Obr. 3.28 ukazuje úlohu dvou základních modelů strukturované analýzy – ER diagramu a diagramu datových toků (DFD - Data Flow Diagram) při návrhu databázové aplikace. Z předchozí kapitoly již víme, že výchozím modelem pro logický návrh databáze je ER diagram. DFD je naopak výchozím modelem pro návrh procedur a funkcí, které tvoří kód databázového klienta. Ovlivňuje ale i návrh databáze a to jednak tím, že z něho vychází případný návrh zmíněných databázových objektů obsahujících kód, jednak operace s daty jsou zdrojem návrhových rozhodnutí v oblasti fyzického návrhu databáze, zejména pokud jde o přístupové metody (indexování, hašování apod.).



Obr. 3.28 Úloha modelů strukturované analýzy při návrhu

Analogicky Obr. 3.29 ukazuje úlohu modelů objektově-orientovaného přístupu. Je vidět, že diagram tříd je východiskem logického návrhu, ale protože obsahuje operace, uplatňuje se i při návrhu klientské části. Naopak další z modelů ovlivňují obecně návrh jak serverové tak klientské části aplikace.



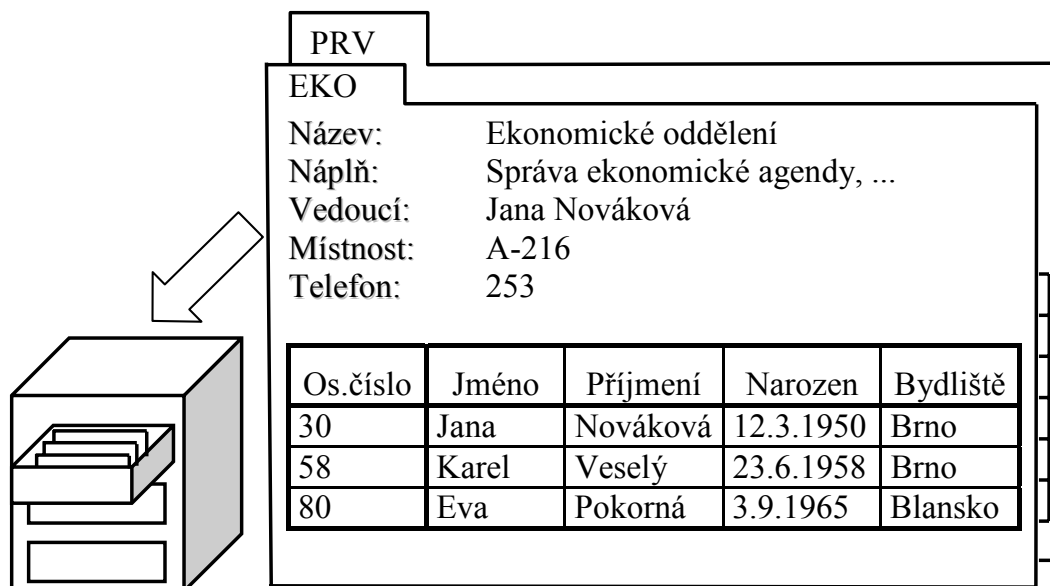
Obr. 3.29 Úloha modelů objektově-orientované analýzy při návrhu

Dříve, než si uvedeme typické nedostatky chybného návrhu tabulek databáze, podívejme se na jednoduchý příklad, na kterém jsme ilustrovali na úvodní přednášce, jak vypadá relační databáze.

$x+y$

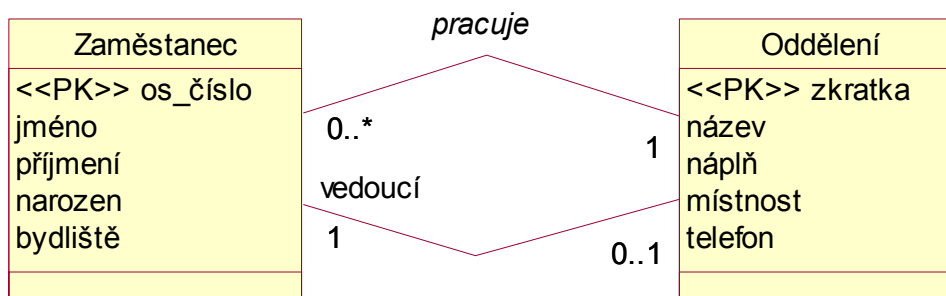
Příklad 3.10

Uvažujme firmu, která má několik oddělení a vede personální agendu v papírové podobě na štítcích, jak je znázorněno na Obr. 3.30. Každý štítek obsahuje základní údaje o oddělení – zkratku, název, náplň, místnost, telefon a kdo je vedoucím oddělení. Pro jednoduchost předpokládáme, že každé oddělení sídlí jen v jedné místnosti, kde je jeden telefon. Dále štítek obsahuje seznam zaměstnanců a jejich základních osobních údajů – osobní číslo, jméno, příjmení, datum narození a místo bydliště.



Obr. 3.30 Štítky agentury z příkladu Příklad 3.10

Předpokládejme, že se firma rozhodla převést tuto agenturu do elektronické podoby. Pokud byste analyzovali požadavky na odpovídající databázovou aplikaci a vycházeli byste z agentury vedené na štítcích, vytvořili byste pravděpodobně jako konceptuální model ER diagram uvedený na Obr. 3.31.

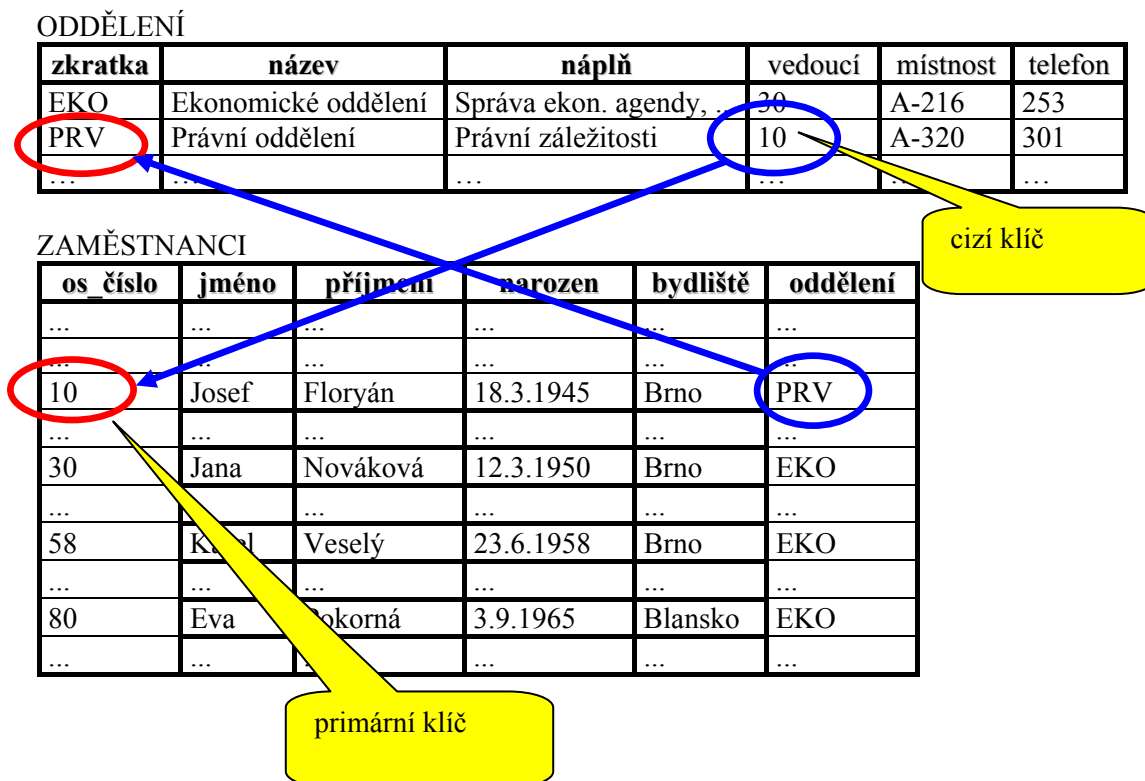


Obr. 3.31 ER diagram příkladu Příklad 3.10



Jak by se změnil konceptuální model, kdyby oddělení firmy mohlo sídlit v několika místnostech a mít několik telefonních čísel?

Relační databáze, kterou podle výše uvedeného diagramu navrhne, je na Obr. 3.32.



Obr. 3.32 Tabulky databáze příkladu Příklad 3.10

Porovnáme-li ER diagram a schéma příslušné databáze, vidíme jasnou korespondenci: entitní množinám odpovídají tabulky a vztahy jsou reprezentovány vazbami mezi řádky tabulek, vytvořenými pomocí cizích klíčů. To jsou dvě z pravidel transformace ER diagramu na tabulky relační databáze, která si uvedeme později. Nejprve si uvedeme typické nedostatky nesprávně navržených tabulek.

Když se znovu podíváme na tabulky databáze na Obr. 3.32 a položíme si otázku, zda jsou navrženy správně, asi intuitivně odpovíme, že ano. Co nás k tomuto závěru vede? Nejvýznamnějším indikátorem tabulky, která je z hlediska pravidel logického návrhu databáze navržena chybně, je, že obsahuje redundanci. Jinými slovy, jistá informace se tam opakuje nebo potenciálně může opakovat. To u našich dvou tabulek nehrozí. Kompletní informace o zaměstnanci je vždy na jednom řádku tabulky, stejně tak informace o oddělení.

Uvažujme nyní jinou verzi návrhu tabulky Zaměstnanci, která vypadá takto:

ZAMĚSTNANCI

os_číslo	jméno	příjmení	oddělení	název
10	Josef	Floryán	PRV	Právní oddělení
30	Jana	Nováková	EKO	Ekonomické oddělení
58	Karel	Veselý	EKO	Ekonomické oddělení
80	Eva	Pokorná	EKO	Ekonomické oddělení

Pomineme fakt, že nám chybí datum narození a místo bydliště zaměstnance, a zaměříme se na sloupce oddělení a název. Asi cítíte, že tyto dva sloupce něco spojuje. Je-li zkratka ekonomického oddělení 'EKO', potom dvojice hodnot ('EKO',

'Ekonomické oddělení') se musí vyskytovat u každého zaměstnance, který je zaměstnán v ekonomickém oddělení. Opakuje se nám v tabulce informace, že název oddělení se zkratkou 'EKO' je 'Ekonomické oddělení', a tedy se v této tabulce vyskytuje redundance. My bychom rádi měli tento fakt uložen pouze na jednom řádku nějaké tabulky.



Studenti často mylně chápou jako redundanci i to, že se opakuje stejná hodnota v jednom sloupci tabulky. To samozřejmě redundance není. Redundance toho typu, který teď diskutujeme, tj. v jedné tabulce, vzniká teprve tehdy, když se opakuje, přesněji může opakovat, informace v alespoň dvou sloupcích, které spolu významově souvisí. Je třeba si uvědomit, že tato souvislost plyne skutečně z významu hodnot v těchto sloupcích, a ne z toho, jestli se zrovna v daném stavu tabulky nějaké hodnoty opakují nebo ne. Jestliže v našem příkladě zkratka oddělení jednoznačně určuje název oddělení a v jednom oddělení může být několik zaměstnanců, pak už na základě těchto informací byste měli být schopni konstatovat, že tabulka není navržena správně, protože vede na redundanci.

Redundance nám vadí ze dvou důvodů. Jednak opakující se informace zabírá zbytečně paměťový prostor, ale především by byla složitá kontrola správnosti dat, tedy odpovídajícího integritního omezení. Před každým vložením nebo modifikací řádku tabulky by SRBD nebo aplikace musel kontrolovat, že ve všech řádcích tabulky s vkládanou hodnotou zkratky oddělení bude i stejná hodnota názvu oddělení.

Z důvodu redundance tedy tabulka není navržena správně. Pokud si situaci zjednodušíme a budeme předpokládat, že zkratka a název oddělení jsou jediné dva údaje, které potřebujeme mít o oddělení v databázi uloženy, pak můžeme zjistit ještě další negativní důsledky chybně navržené tabulky.

Představme si, že vznikne nové oddělení, například vývojové se zkratkou 'VYV'. Pokud bychom chtěli do tabulky vložit informaci o existenci takového oddělení ještě před přiřazením zaměstnanců do tohoto oddělení, pak se nám to nepodaří (přinejmenším podle teorie relačního modelu – proč?). Analogicky pokud by z důvodu nějakých organizačních změn byli třeba přechodně přeřazeni všichni zaměstnanci tohoto oddělení do oddělení jiných, zanikla by i informace o existenci vývojového oddělení.

Hlavní problémy chybného návrhu můžeme tedy shrnout do následujících tří bodů:

- opakující se informace (redundance)
- nemožnost reprezentovat určitou informaci
- složitá kontrola integritních omezení

V kapitole 3.4 uvidíme, že E.F.Codd definoval pro takové závislosti sloupců tabulky, jako jsme viděli u zkratky oddělení a názvu oddělení, pojem *funkční závislost* a společně s dalšími výzkumníky definoval tzv. *normální formy*, které jsou mírou kvality návrhu tabulek databáze. Uvidíme, že výše zmíněné tři nedostatky vyřeší tzv. *Boyce-Coddova normální forma*.

Když víme, jaké jsou problémy chybného návrhu, můžeme říci, že dobrý logický návrh bude takový, který se vyvaruje těchto problémů. To ale nestačí. V matematické terminologii bychom řekli, že to je podmínka nutná, ale ne postačující. Představme si, že informace o tom, ve kterém oddělení zaměstnanec pracuje nebude uložena přímo v tabulce ZAMĚSTNANCI, nýbrž v samostatné tabulce

PRACUJE

os číslo	oddělení
10	PRV
30	EKO
58	EKO
80	EKO

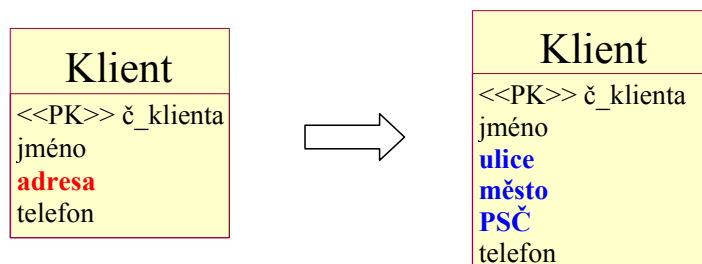
Místo dvou tabulek bude naše databáze tvořena tabulkami třemi tabulkami. Žádná z nich netrpí redundancí. Přesto náš návrh není zcela v pořádku, protože tabulka PRACUJE je zbytečná. Sloupec oddělení, jehož hodnota udává, ve kterém oddělení zaměstnanec pracuje, může totiž být, jak jsme viděli, přímo v tabulce ZAMĚSTNANCI. I při logickém návrhu bychom měli mít na paměti výkonnostní hledisko. Zbytečná tabulka představuje časové ztráty vlivem zbytečného spojování tabulek. Pokud chceme například zodpovědět dotaz „Jaké telefonní číslo má oddělení, kde pracuje Karel Veselý?“ v případě návrhu se dvěma tabulkami spojujeme tabulku ZAMĚSTNANCI s tabulkou ODDĚLENÍ, zatímco v případě návrhu se třemi tabulkami je potřeba postupně spojit tabulky ZAMĚSTNANCI, PRACUJE a ODDĚLENÍ.

Cíle dobrého logického návrhu tedy můžeme shrnout do dvou následujících bodů:

- vyvarování se problémů špatného návrhu,
- zohlednění dalších kritérií, především výkonnostních, zejména nevytvářet zbytečné tabulky, pokud pro to není důvod.

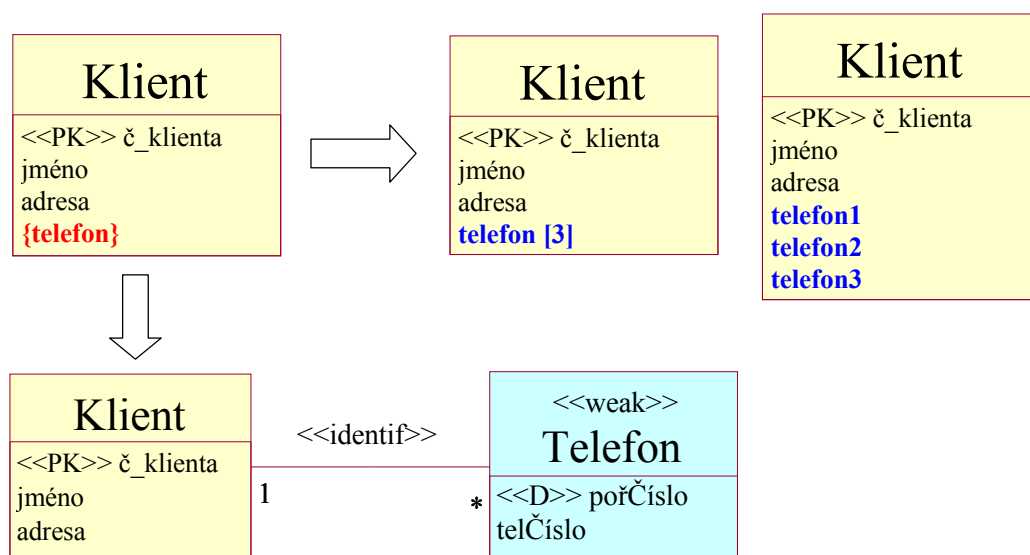
Nyní již můžeme přistoupit k formulaci pravidel transformace ER diagramu na tabulky relační databáze, jejichž dodržení tyto cíle zajistí. Prvým krokem je *odstranění složených a vícehodnotových atributů*. Jde o přípravný krok, který zajistí, že následná transformace povede na normalizované tabulky.

V případě *složeného atributu* je úprava jednoduchá. Spočívá v rozložení atributu na jeho jednotlivé složky (případně opakovaně). Situaci ilustruje Obr. 3.33. Jestliže je adresa atributem složeným, je po úpravě nahrazena svými složkami, v našem případě tedy atributy ulice, město a PSČ.



Obr. 3.33 Odstranění složeného atributu

V případě vícehodnotového atributu je možná jedna ze dvou úprav. Buď se omezí možný počet hodnot nebo se zavede nová entitní množina – viz Obr. 3.34.



Obr. 3.34 Varianty odstranění vícehodnotového atributu

Obrázek ukazuje obě varianty odstranění vícehodnotového atributu telefon entitní množiny Klient. Omezení maximálního počtu možných hodnot znamená, že omezíme počet telefonních čísel, které budeme u klienta ukládat, v obrázku na tři. Nahradíme tak vícehodnotový atribut atributem složeným telefon[3], který následně můžeme nahradit třemi složkami. Toto řešení je jednoduché, ale zavádí určité omezení.

Pokud chceme zachovat obecné řešení, tedy v našem příkladu mít možnost ukládat libovolné množství telefonních čísel klienta, musíme zavést další entitní množinu, jejíž entity budou reprezentovat právě tato čísla. Tuto novou entitní množinu můžeme modelovat jako slabou, závislou na množině Klient. V takovém případě bude atribut pořČíslo jen diskriminátorem. Je ale samozřejmě možné zavést i plnohodnotný identifikátor a modelovat množinu Telefon jako silnou.



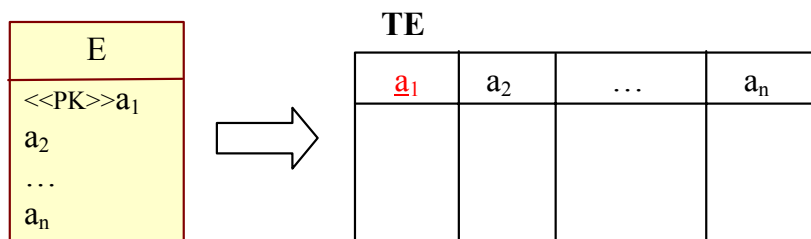
Změnilo by se ještě něco v ER diagramu, pokud bychom modelovali entitní množinu Telefon jako silnou?



Složené a vícehodnotové atributy lze odstraňovat buď skutečně úpravou ER diagramu nebo až v rámci následné transformace na tabulky, resp. po ní. V každém případě musíme mít na paměti, že tabulky, které vzniknou, musí být normalizované.

Nyní již můžeme přistoupit k vlastní transformaci. Jejím základem je transformace entitních množin a transformace vztahových množin. Určitými specifiky se vyznačuje transformace struktur, které jsme si uvedli jako rozšíření ER modelu – slabých entitních množin a generalizace/specializace.

Transformace silných entitních množin, které se nevyskytují ve struktuře generalizace/specializace, je jednoduchá. Každé takové entitní množině odpovídá tabulka navrhované databáze. Sloupce tabulky odpovídají atributům entitní množiny, identifikátor entitní množiny bude primárním klíčem tabulky. Jednotlivé řádky tabulky potom reprezentují jednotlivé entity dané entitní množiny. Situace je znázorněna na Obr. 3.35.

**Obr. 3.35** Transformace silné entitní množiny

Transformace vztahových množin už bude trochu složitější. Budeme zatím uvažovat pouze vztahy binární. V příkladu Příklad 3.10 jsme viděli, že vztahy jsou v databázi reprezentovány vazbami mezi řádky tabulek vytvořenými pomocí cizích klíčů. Tuto vazbu můžeme vytvořit tak, že v tabulce pro jednu entitní množinu přidáme sloupec, který bude cizím klíčem a bude se odkazovat tabulku pro druhou entitní množinu binárního vztahu. Zároveň ale musíme mít na paměti, že hodnota cizího klíče musí být také atomická, v příslušném políčku tabulky nemůže být najednou hodnot více.

Uvažujme, že transformujeme vztahovou množinu „pracuje“ z Obr. 3.31. Identifikátorem entitní množiny ODDĚLENÍ je atribut zkratka, identifikátorem entitní množiny ZAMĚSTNANEC je atribut os_číslo. Jde o vztah s kardinalitou 1:M. Jestli budeme vztahy tohoto typu reprezentovat vztahem vytvořeným pomocí cizího klíče, vzniká otázka, ke které ze dvou tabulek sloupec cizího klíče přidat. Můžeme přidat sloupec k tabulce ODDĚLENÍ, jehož hodnota v řádku by říkala, kteří zaměstnanci v daném oddělení pracují? Určitě ne. Taková hodnota by nebyla atomická, protože zaměstnanců v oddělení může pracovat více – kardinalita protějšího konce vztahu je M. Zkusíme tedy druhou tabulku. Můžeme přidat sloupec v tabulce ZAMĚSTNANEC, jehož hodnota v řádku bude říkat, ve kterém oddělení daný zaměstnanec pracuje? Určitě ano, protože kardinalita protějšího konce je 1 – zaměstnanec může pracovat jenom v jednom oddělení. Můžeme tedy učinit závěr, že vztahové množiny lze transformovat přidáním sloupce cizího klíče k té z obou tabulek, jejíž protějšek má v ER diagramu kardinalitu 1. Výsledkem budou tabulky

ZAMĚSTNANEC

<u>os_číslo</u>	jméno	příjmení	narozen	místo	oddělení

ODDĚLENÍ

<u>zkratka</u>	název	náplň	místnost	telefon

Pokud jde o vztah s kardinalitou 1:1, pak takový sloupec můžeme přidat k libovolné z obou tabulek. V našem příkladě může být u oddělení sloupec, který bude říkat kdo je vedoucím nebo u zaměstnance sloupec, který bude říkat, které oddělení daný zaměstnanec vede. Rozhodnutí, kterou z variant zvolit, závisí zejména na tom, která informace bude potřeba častěji, případně může rozhodovat minimální kardinalita. U nás by obě tato kritéria vedla k první variantě. Jednak určitě bude třeba znát odpověď na otázku: „Kdo je vedoucím daného oddělení?“, jednak ne každý zaměstnanec je vedoucím nějakého oddělení (minimální kardinalita je nula), takže pouze některé hodnoty by byly v odpovídajícím sloupci tabulky Zaměstnanec definované. Současně je potřeba si uvědomit, že ať zvolíme kteroukoliv z obou variant, bude i druhá otázka („Které oddělení vede daný zaměstnanec?“) zodpověditelná. Výsledné schéma po

transformaci tedy bude

ZAMĚSTNANEC

<u>os. číslo</u>	jméno	příjmení	narozen	místo	oddělení

ODDĚLENÍ

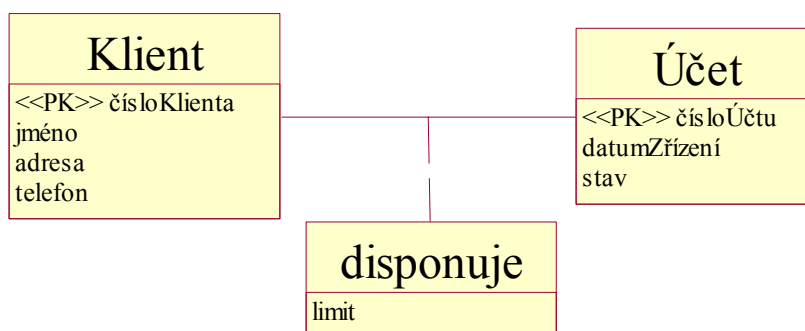
<u>zkratka</u>	název	náplň	místnost	telefon	vedoucí



V zásadě by bylo možné přidat sloupce do obou tabulek. Jaké výhody a naopak nevýhody by takové řešení mělo?

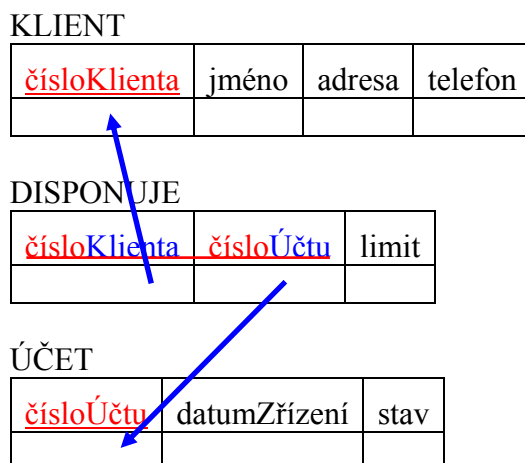
Všimněte si, že i když v praxi velice často používáme pro vkládaný sloupec stejné jméno jako je jméno sloupce odkazovaného, není to nutné. My jsme použili výstižnější jména. Významově ale hodnota ve sloupci vedoucí je osobní číslo vedoucího a ve sloupci oddělení je to zkratka oddělení, kde daný zaměstnanec pracuje.

Odlišná situace vzniká u vztahů s kardinalitou M:M. Zde nemůžeme příslušný vztah reprezentovat přímo, nýbrž musíme použít tzv. vazební tabulku. Uvažujme, že transformuje vztahovou množinu „disponuje“ s kardinalitou M:M mezi entitními množinami Klient a Účet - viz. Obr. 3.36.

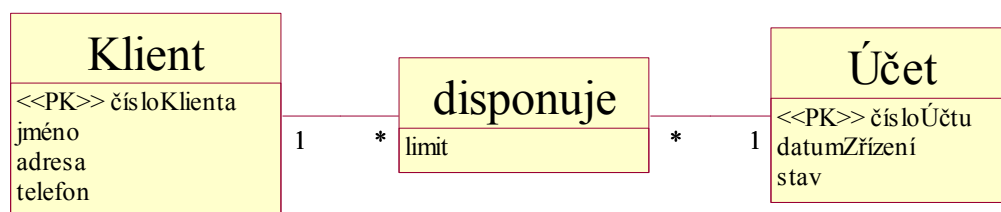


Obr. 3.36 Příklad vztahu s kardinalitou M:M

Entitní množina Klient se transformuje na tabulku KLIENT, analogicky Účet na tabulku ÚČET. Zřejmě ale nemůže být v tabulce KLIENT sloupec, jehož hodnota by říkala, se kterými účty klient může manipulovat, ani v tabulce ÚČET sloupec, jehož hodnota by říkala, kteří klienti mohou s daným účtem manipulovat. V obou případech by hodnoty nebyly atomické. Musíme proto použít další tabulku, která bude ukládat informaci o příslušných vztazích, tedy v našem případě o tom, který klient může manipulovat s kterým účtem. Abychom identifikovali jednoznačně klienta a účet, budou v této tabulce dva sloupce, které budou cizími klíči. Jeden se bude odkazovat na tabulku KLIENT, druhý na tabulku ÚČET. A protože má naše vztahová množina ještě atribut limit, bude v tabulce ještě jeden sloupec pro uložení hodnoty limitu. Výsledkem bude schéma



Když si vzpomenete, co jsme si uváděli o náhradě vztahové množiny s kardinalitou M:M vazební entitní množinou a dvěma vztahy s kardinalitou 1:M a M:1, pak zjistíte, že stejné schéma obdržíme nahrazením vztahové množiny a následnou transformaci vazební entitní množiny a obou vztahů – viz Obr. 3.37.



Obr. 3.37 Použití vazební entitní množiny

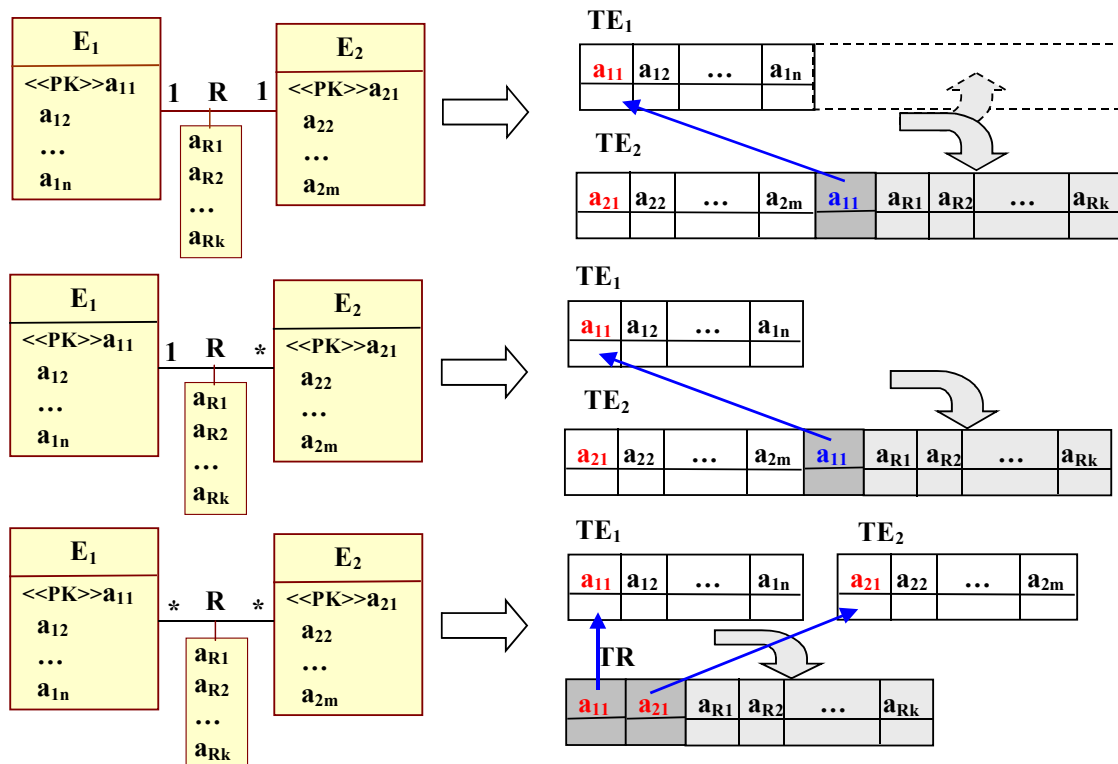


Připomínáme, že pro vazební entitní množiny a vazební tabulky se v praxi často používají jména tvořená jmény entitních množin, resp. tabulek, mezi nimiž vazbu vytváří. V našem příkladě by to mohlo být jméno Klient_Účet, resp. KLIENT_ÚČET.

Nyní už byste měli být schopni transformovat jakoukoliv vztahovou množinu binárních vztahů. Pravidla, která jsme si postupně vysvětlili, jsou schématicky shrnuta na Obr. 3.38. Zde R je transformovaná vztahová množina, a_{Ri} jsou její atributy. Slovně pravidla můžeme shrnout takto:

1. Vztahy s kardinalitou 1:1 reprezentujeme v databázi tak, že rozšíříme jednu z tabulek pro entity, mezi kterými transformované vztahy jsou, o sloupec, který je cizím klíčem odkazujícím se na primární klíč druhé z obou tabulek. Má-li transformovaná vztahová množina nějaké atributy, rozšíříme tabulku s cizím klíčem ještě o odpovídající sloupce.
2. Vztahy s kardinalitou 1:M reprezentujeme v databázi tak, že rozšíříme tabulku pro entity, které mají na svém konci transformovaného vztahu kardinalitu M, o sloupec, který je cizím klíčem odkazujícím se na primární klíč druhé z obou tabulek. Má-li transformovaná vztahová množina nějaké atributy, rozšíříme tabulku s cizím klíčem ještě o odpovídající sloupce.
3. Vztahy s kardinalitou M:M reprezentujeme v databázi tak, že přidáme novou tabulku, která bude obsahovat jako cizí klíče primární klíče tabulek reprezentujících entitní množiny, mezi kterými je transformovaný vztah, a případné sloupce atributů vztahové množiny. Primární klíč této nové tabulky

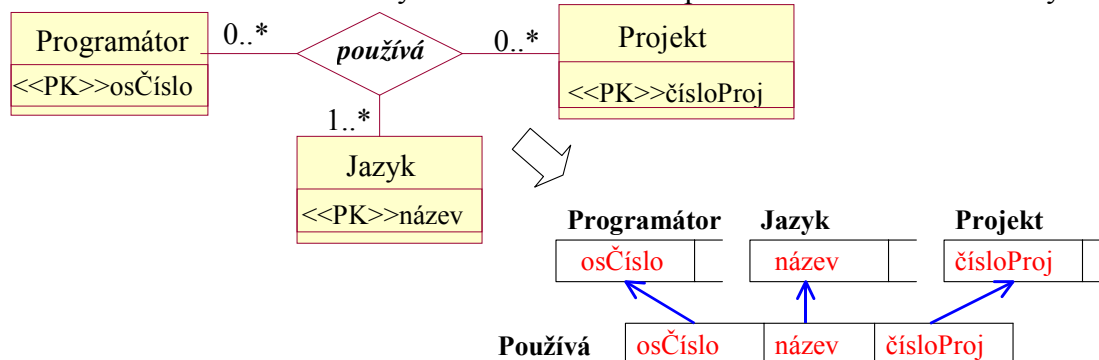
bude složený a bude tvořen primárními klíči obou tabulek a případně ještě jedním nebo několika atributy vztahové množiny, pokud samotná kombinace hodnot primárních klíčů není v tabulce unikátní.



Obr. 3.38 Pravidla transformace vztahových množin

Nyní se podívejme, jak transformovat vztahy vyššího stupně. Přestože i zde závisí na kardinalitě, můžeme říci, že v naprosté většině případů bude potřeba vazební tabulka. Uvažujme příklad ternárního vztahu z Obr. 3.39, který vyjadřuje, že daný programátor používá při řešení daného projektu daný programovací jazyk. Z uvedené kardinality je zřejmé, že programátor musí při řešení projektu používat jeden nebo více jazyků, daný jazyk může při řešení daného projektu používat více programátorů, ale také třeba žádný a že daný programátor může používat daný jazyk ve více projektech, ale také třeba v žádném.

Vazební tabulka bude obsahovat primární klíče všech tří tabulek, které zde budou cizími klíči a současně budou tyto tři cizí klíče tvořit primární klíč vazební tabulky.



Obr. 3.39 Transformace ternárního vztahu

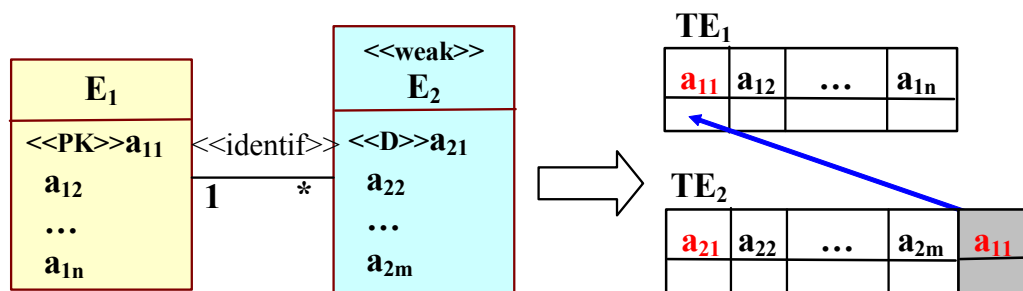


Vidíme, že z hlediska návrhu databáze je zásadní rozdíl mezi vztahem s kardinalitou 1:M a M:M. Z toho by měly pro vás vyplynout následující dva závěry:

1. Je důležité při tvorbě ER diagramu určit správně kardinalitu vztahů.
2. Nejistota, zda se kardinalita vztahu nemůže změnit z 1:M na M:M, může být důvodem pro reprezentace vazební tabulkou vztahu s kardinalitou 1:M. Platíme za to sice nutností další operace spojení, ale změna kardinality nepřinese problémy.

Nyní se podíváme na způsob transformace rozšiřujících prvků ER diagramu. Začneme *slabými entitními množinami*. Z pohledu transformace není ve skutečnosti slabá entitní množina ničím zvláštním s výjimkou primárního klíče odpovídající tabulky. Stačí, když si vzpomeneme, jakou vlastnost má slabá entitní množina z hlediska identifikace jejích entit a jakou roli hraje vztah od identifikující entitní množiny, který má vždycky kardinalitu 1:M. Potom už stačí transformovat entitní množinu a identifikující vztah podle nám známých pravidel. Způsob transformace je znázorněn na Obr. 3.40 a příslušné pravidlo můžeme zformulovat takto:

Slabou entitní množinu reprezentujeme v databázi tabulkou, která bude mít sloupce odpovídající jejím atributům a navíc bude mít sloupec cizího klíče, který se bude odkazovat na hodnoty primárního klíče tabulky reprezentující identifikující entitní množinu. Primární klíč tabulky bude složený a bude tvořen tímto cizím klíčem a diskriminátorem. V obrázku bude tedy primárním klíčem tabulky TE_2 složený sloupec (A_{11}, A_{21}) .

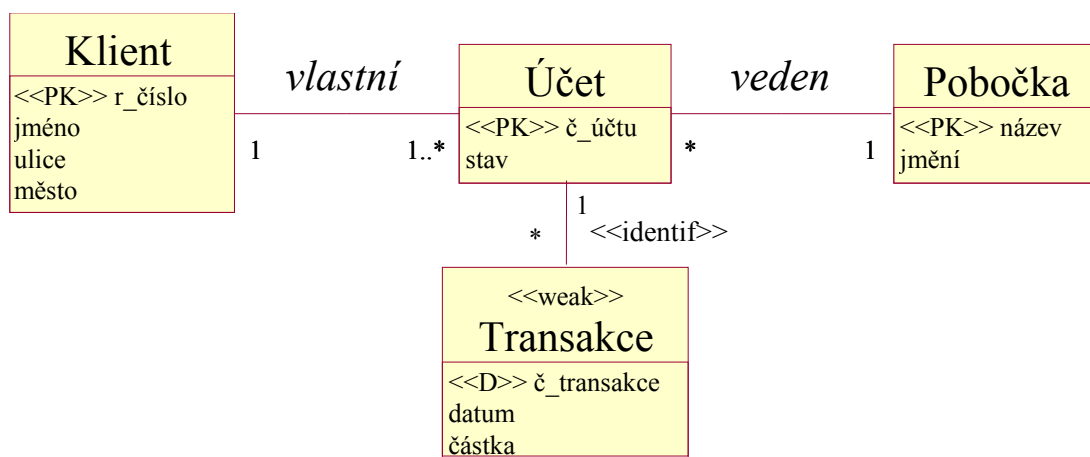


Obr. 3.40 Transformace slabé entitní množiny



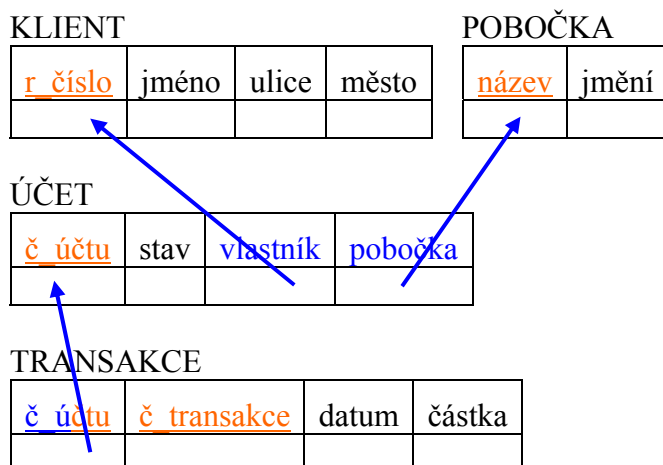
Příklad 3.11

Uvažujme náš jednoduchý systém správy účtů, jehož ER diagram je uveden na Obr. 3.41.



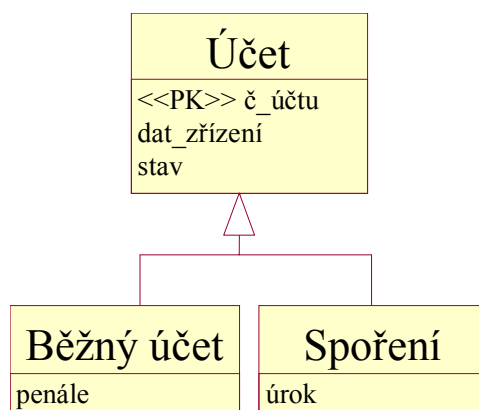
Obr. 3.41 ER diagram jednoduchého systému správy účtů

Předpokládejme, že všechny atributy nabývají atomických hodnot. Budeme tedy pouze transformovat entitní množiny a vztahy na základě nám již známých pravidel. Pouhým pohledem na ER diagram můžeme říci, že naše databáze bude obsahovat čtyři tabulky, protože v diagramu jsou čtyři entitní množiny a není tam žádný vztah s kardinalitou M:M. Můžeme také říci, že tyto tabulky budou mít celkem tři sloupce, které budou cizími klíči, protože v diagramu máme tři vztahové množiny s kardinalitou 1:M. Výsledné schéma tedy bude



Transformace, které jsme dosud poznali byly přímočaré a až na vztah 1:1 také vedoucí na jediné dobré řešení. Nyní se podíváme na transformaci generalizace/specializace kde uvidíme, že se nabízí více řešení, a při výběru nejvhodnějšího musíme vzít do úvahy případně i informace, které nejsou obsaženy v ER diagramu.

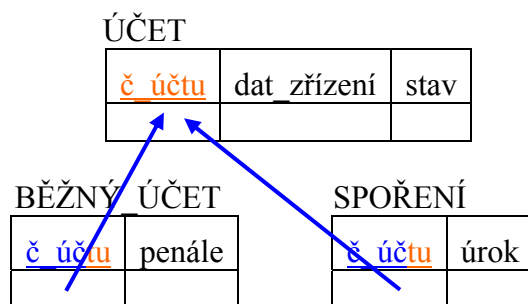
Uvažujme jednoduchý ER diagram obsahující vztah generalizace/specializace z Obr. 3.42. Diagram říká, že existují dva typy účtů – běžný účet a spoření. Oba typy mají některé společné vlastnosti, naopak u některých vlastností, jejichž hodnoty je potřeba ukládat, se liší.



Obr. 3.42 Příklad generalizace/specializace

Pro transformaci této struktury se nabízejí následující varianty:

1. Vytvořit tabulky ÚČET, BĚŽNÝ_ÚČET a SPOŘENÍ. Ve všech bude primárním klíčem sloupec č_účtu, který bude v tabulkách BĚŽNÝ_ÚČET a SPOŘENÍ zároveň cizím klíčem odkazujícím se do tabulky ÚČET. Budou tak provázány části záznamu s hodnotami společných atributů s částmi nesoucími hodnoty atributů specializací. Schéma tedy bude následující:



Kompletní informaci všech běžných účtů bychom získali dotazem, který by měl v podobě výrazu relační algebry tvar `ÚČET join BĚŽNÝ_ÚČET`.

2. Vytvořit pouze tabulky BĚŽNÝ_ÚČET a SPOŘENÍ, které budou obsahovat sloupce pro atributy jak entitní množiny Účet, tak entitní množiny Běžný účet, resp. Spoření. Schéma tedy bude následující:



3. Vytvořit jedinou tabulku ÚČET, ve které by byly informace jak o běžných účtech, tak o spoření. Tabulka by měla sloupce pro všechny atributy všech tří entitních množin v ER diagramu. Rozlišení, zda jde o běžný účet nebo spoření by se opět provádělo testem prázdné hodnoty nebo by se přidal další sloupec jako diskriminátor. Primárním klíčem tabulky by bylo číslo účtu. Schéma by bylo jednoduché

ÚČET

<u>č_úctu</u>	dat_zřízení	stav	penále	úrok

Je zřejmé, že v tomto případě musíme být nějak schopni rozpoznat běžný účet a spoření. Jednou možností je rozlišení na základě prázdné hodnoty ve sloupci penále, resp. úrok. Běžný účet bude mít prázdnou hodnotu ve sloupci úrok a spoření ve sloupci penále.

Kompletní informaci všech běžných účtů bychom získali dotazem, který by měl v podobě výrazu relační algebry tvar $ÚČET \text{ where } úrok \text{ is null}$. Predikát *is null* jsme zde použili pro test prázdné hodnoty.

Jinou možností rozlišení může být zavedení speciálního sloupce (diskriminátoru), nazvaného např. typ, jehož hodnota bude říkat, o jaký typ účtu jde. Schéma by bylo

ÚČET

<u>č_úctu</u>	dat_zřízení	stav	penále	úrok	typ

Rozlišení by mohlo být takové, že hodnota 'B' ve sloupci typ by znamenala, že řádek se týká běžného účtu a hodnota 'S', že jde o spoření. Potom bychom kompletní informaci všech běžných účtů získali dotazem, který by měl v podobě výrazu relační algebry tvar $ÚČET \text{ where typ}='B'$.

4. Vytvořit tabulku ÚČET a tabulku BĚŽNÝ_SPOŘENÍ. Ve druhé tabulce budou sloupce pro atributy jak běžného účtu, tak spoření a navíc tam bude sloupec čísla účtu, který bude v této tabulce primárním i cizím klíčem a opět prováže části záznamu běžného účtu, resp. spoření, které patří k sobě. Schéma tedy bude následující:

ÚČET

<u>č_úctu</u>	dat_zřízení	stav

BĚŽNÝ_SPOŘENÍ

<u>č_úctu</u>	penále	úrok	typ

Rozlišení typu účtu by se provedlo stejným způsobem jako u předchozí varianty. Kompletní informaci všech běžných účtů bychom získali dotazem, který by měl v podobě výrazu relační algebry tvar

$ÚČET \text{ join (BĚŽNÝ_SPOŘENÍ where } úrok \text{ is null)}$, resp.

$ÚČET \text{ join (BĚŽNÝ_SPOŘENÍ where typ}='B')$

při použití sloupce typ jako diskriminátoru.



Zamyslete se nad tím jestli a jaká jsou omezení na hodnoty ve sloupci č_úctu u jednotlivých variant.

Každá z výše uvedených variant má určité výhody a nevýhody a je vhodná za určitých

podmínek. Při rozhodování, kterou vybrat, bychom měli sledovat tyto cíle:

- vyvarovat se redundanci
- zohlednit efektivní provádění častých operací z hlediska spojování tabulek, případně velikosti záznamů

Na splnění prvního cíle má vliv, jestli jsou specializace disjunktí nebo ne. Pokud se budou specializace překrývat, vznikla by u varianty číslo dvě redundance. V našem příkladě by tato situace znamenala, že nějaký účet může být zároveň běžným účtem i spořením. Pak by se u této varianty ukládala řada stejných hodnot do tabulky BĚŽNÝ_ÚČET i SPOŘENÍ.

Tato varianta by byla rovněž nepoužitelná nebo přinejmenším nevhodná v případě, kdy je specializace částečná. V našem příkladě by tato situace znamenala, že mohou být ještě jiné účty, které nejsou ani běžné ani spořitelní a jsou charakterizovány pouze atributy entitní množiny Účet. Varianta číslo dvě by byla nepoužitelná, protože by nebylo kam ukládat informaci o takových účtech. Jediným řešením by bylo zavedení konvence, že informace o nich bude např. společně s běžnými účty, ale toto řešení by určitě nebylo vhodné.

Pro splnění druhého cíle je zejména potřeba zvážit, zda budou často probíhat operace nad společnými atributy bez ohledu na typ účtu nebo naopak vždy budou probíhat operace jen s určitým typem účtu. V prvním případě budou vhodné varianty, kde je samostatná tabulka ÚČET se sloupci společných atributů (varianty číslo 1 a 3) a varianta, kdy je všechno v jedné tabulce (varianta číslo 4). Z těchto dvou možností by varianta číslo 4 mohla být méně vhodná v případě, kdy by běžný účet a spoření měly velké množství specifických atributů.

Varianta 2 bude vhodná z pohledu operací, když budou oddělené operace s běžnými účty a spořením.

Uvedené varianty ilustrované na příkladě účtů můžeme zobecnit do následujících variant transformace struktury generalizace/specializace:

1. Tabulka pro zobecňující entitní množinu (nadtyp) a další tabulka pro každou její specializaci (podtyp). Tabulky podtypů kromě sloupců pro své atributy obsahují i sloupec cizího klíče, který se odkazuje na primární klíč tabulky nadtypu. Tento sloupec je zároveň primárním klíčem tabulky podtypu.

Tato varianta je vhodná pro disjunktí i překrývající se specializace a pro specializaci úplnou i částečnou. Její nevýhodou je použití tří tabulek a tedy potřeba spojování. Naopak výhodou může být oddělení společných atributů, pokud se často provádí operace pouze s nimi a specifických atributů je mnoho. Tato varianta se také může ukázat jako pružná, pokud by existovala možnost budoucího zavedení dalších specializací.

2. Tabulka pro každý podtyp, která obsahuje i atributy nadtypu.

Tato varianta je vhodná pouze v případě disjunktí úplné specializace a to zejména za předpokladu, že není potřeba provádět operace nad společnými atributy nebo alespoň nehrozí zavedení nových specializací v budoucnu (proč tento předpoklad?).

3. Jedna tabulka, ve které je uloženo vše. Obsahuje sloupce pro všechny atributy nadtypu a všech podtypů. Primárním klíčem je identifikátor nadtypu. Rozlišení

jednotlivých podtypů je testem prázdné hodnoty ve sloupci atributu, který je pro daný typ irelevantní nebo zavedením sloupce, který plní funkci diskriminátoru.

Hlavní výhodou této varianty je, že není potřeba spojovat žádné tabulky. Nevýhodou může být velikost tabulky, nutnost vybírat řádky odpovídající určité specializaci (operace selekce relační algebry) a množství prázdných hodnot v případě velkého množství různých atributů disjunktních specializací.

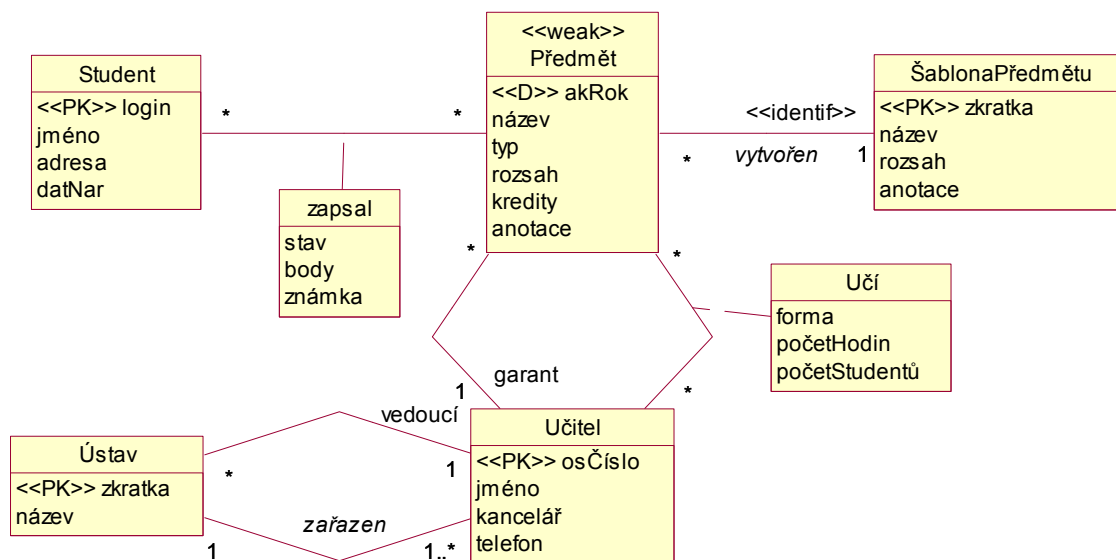
4. Kombinace variant 1 a 3 - tabulka pro nadtyp a společná tabulka pro všechny podtypy. Tato společná tabulka obsahuje sloupce pro všechny atributy podtypů a sloupec cizího klíče, který se odkazuje na primární klíč tabulky nadtypu. Tento sloupec je zároveň primárním klíčem tabulky podtypu. Rozlišení jednotlivých podtypů je testem prázdné hodnoty ve sloupci atributu, který je pro daný typ irelevantní nebo zavedením sloupce, který plní funkci diskriminátoru.

Vlastnosti této varianty vyplývají z toho, že jde o kombinaci variant 1 a 3.

Ted už byste měli být schopni navrhnout na základě ER digramu schéma relační databáze dobrých vlastností. Vyřešíme ale ještě společně dva příklady. Prvým bude transformace ER diagramu, který jsme vytvořili jako výsledek příkladu Příklad 3.9 (informační systém fakulty) v předchozí kapitole. Druhým příkladem bude transformace ER diagramu, který bude obsahovat pouze symbolicky pojmenované prvky. Cílem bude ukázat, že pravidla transformace jsou tak jasná, že bychom měli být schopni navrhnout dobré schéma i v případě, kdy přesně nevíme, co entitní a vztahové množiny modelují.

Příklad 3.12

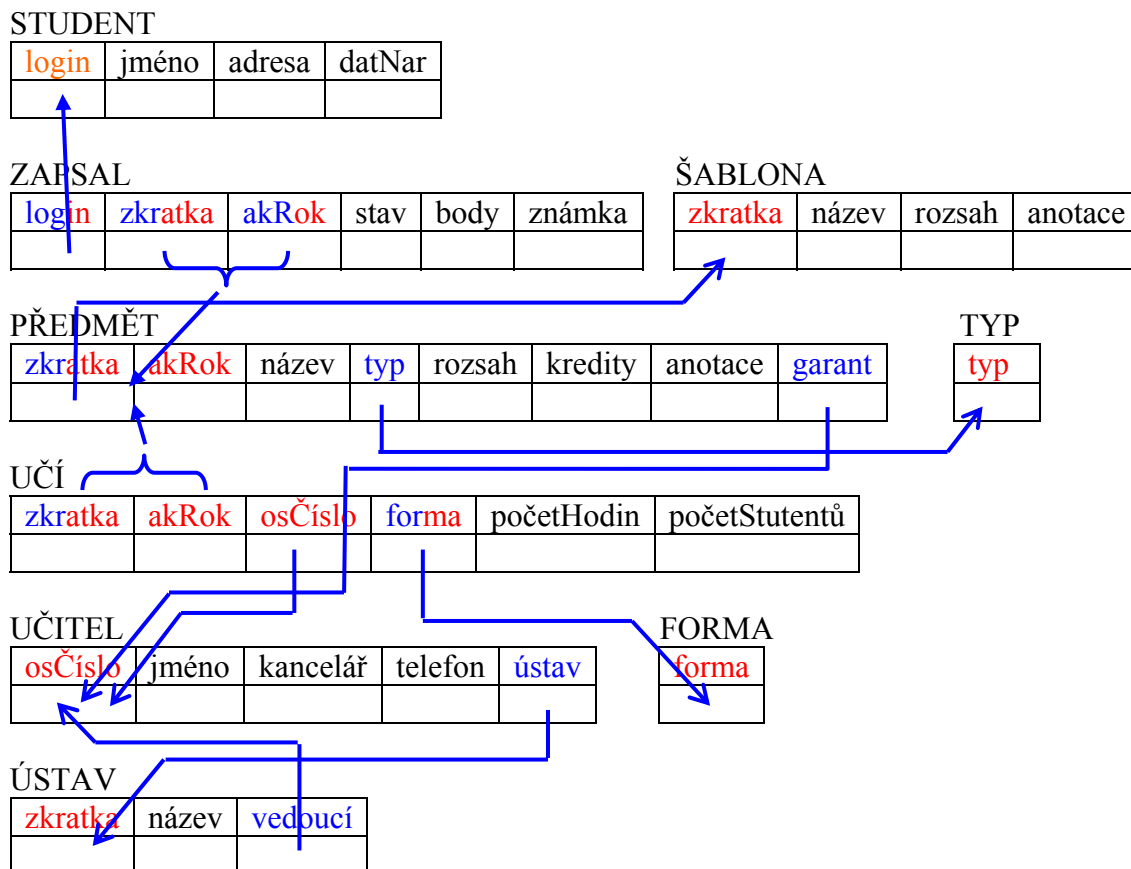
Navrhněte relační databázi informačního systému z příkladu Příklad 3.9. ER diagram je uveden na Obr. 3.43.



Obr. 3.43 ER diagram jednoduchého informačního systému fakulty

Protože diagram obsahuje 5 entitních množin (z toho jedna slabá) a 2 vztahové množiny s kardinalitou M:M, bude výsledné schéma zahrnovat 7 tabulek. Pokud zavedeme číselníky pro typ předmětu a formu výuky, přibudou další dvě tabulky. Dále obsahuje 4 vztahové množiny s kardinalitou 1:M, další čtyři vzniknou náhradou dvou

množin s kardinalitou M:M a dvě zavedením číselníků. Celkem bychom tedy měli mít ve schématu 9 tabulek a 10 vazeb tvořených cizími klíči. Výsledné schéma bude následující:

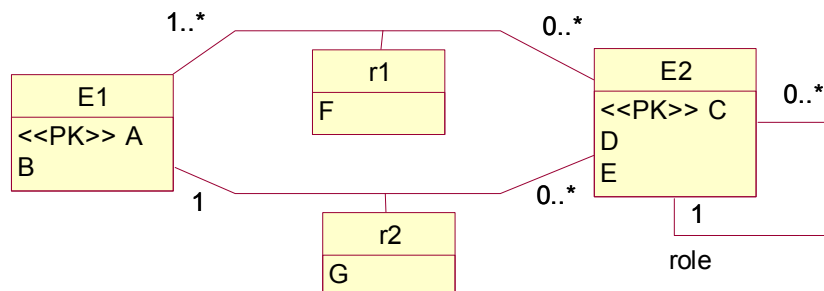


Studenti často chybují při práci se složenými cizími klíči v tom, že místo odkazu šipky vyjadřující odkaz složeného klíče, např. (zkratka, akRok) kreslí šipky pro jednotlivé složky, jako by každá z nich byla cizím klíčem.

Příklad 3.13

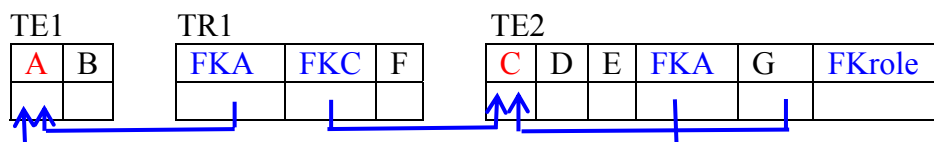
x+y

Navrhněte relační databázi pro ER diagram z Obr. 3.44.



Obr. 3.44 ER diagram s reflexivním vztahem

Diagram obsahuje dvě entitní množiny, dvě vztahové množiny s kardinalitou 1:M a jednu s kardinalitou M:M. Výsledné schéma tedy bude zahrnovat 3 tabulky a 4 vazby prostřednictvím cizích klíčů.



Primární klíč vazební tabulky by byl složený a určitě by zahrnoval sloupce FKA a FKC. U příkladů tohoto typu často činí studentům potíže vypořádat se s reflexivním vztahem.



Viděli jsme, že pravidla transformace jsou jasná a až na transformaci generalizace/specializace také jednoznačná. Proto existují CASE nástroje pro kreslení ER diagramu, které jsou schopny vygenerovat příkazy SQL pro vytvoření tabulek databáze, včetně definic cizích klíčů.

Na závěr této kapitoly upozorníme na způsob transformace diagramu tříd na schéma relační databáze. Protože diagram tříd a ER diagram mají řadu základních vlastností stejných nebo velice blízkých, můžeme říci, že základní pravidla, která jsme si uvedli pro transformaci ER diagramu, platí i pro transformaci diagramu tříd. Jsou však určitá specifika diagramu tříd, která musíme při transformaci zohlednit. Na některá z nich jsme upozornili již v závěru předchozí kapitoly. Jde především o tyto problémy, se kterými je potřeba se vypořádat:

- Třídy definují i operace. Ty při návrhu tabulek neuvažujeme s výjimkou transformace generalizace/specializace. Dále se uplatní při návrhu databázových objektů, které obsahují kód, jako jsou uložené procedury a databázové trigger, a při fyzickém návrhu databáze.
- Objekty jsou identifikované pomocí OID. Pokud není mezi atributy žádný (případně i složený), který by splňoval vlastnosti primárního klíče, zavedeme pro tyto účely další atribut.
- Složené, složité a vícehodnotové atributy. Buď lze použít postup, který jsme si uvedli u ER diagramu nebo využijeme typů pro ukládání velkých binárních (BLOB) nebo znakových (CLOB) objektů, které nabízejí moderní relační systémy.
- Generalizace/specializace je častější. Je třeba pečlivě vybrat nejvhodnější variantu transformace.
- Agregaci lze transformovat jako silnou nebo slabou entitní množinu v ER diagramu.
- Kompozice – modeluje zanořené objekty. Řešení je analogické jako u složených a vícehodnotových atributů.



V této kapitole jsme se naučili navrhnout schéma relační databáze z ER diagramu, případně diagramu tříd. V první řadě jsme si ukázali problémy chybného návrhu, z nichž nejdůležitější je redundance, a stanovili jsme si jako cíl dobrého návrhu se těchto nedostatků vyvarovat a navíc zohlednit i výkonnostní hledisko tím, že nebudeme vytvářet zbytečné tabulky. Potom jsme si definovali základní pravidla transformace. Nejprve je potřeba zkontrolovat, zda atributy všech entitních množin a vztahů nabývají atomických hodnot. Pokud ne, je potřeba provést úpravu diagramu nebo pamatovat na tuto skutečnost při transformaci. Dalším krokem je transformace entitních množin, kdy každá množina je v databázi reprezentovaná tabulkou. Následuje transformace

vztahových množin. Zde jsme si ukázali, že záleží na kardinalitě vztahu. Pokud nejde o vztah s kardinalitou M:M, lze jej reprezentovat pouhou vazbou mezi tabulkami vytvořenou pomocí cizího klíče. Při kardinalitě M:M se neobejdeme bez další tabulky. Ukázali jsme si také transformaci generalizace/specializace, kde jsme viděli, že existuje několik možných variant transformace a výběr nejvhodnější vyžaduje posouzení dalších faktorů, z nichž některé nejsou zjistitelné z ER diagramu.

Pravidla logického návrhu databáze jsou jasná a většinou jednoznačná. Pro daný ER diagram zpravidla vedou na jediné dobré řešení. Naproti tomu ER model v podobě ER diagram, který modeluje řešený problém, může mít řadu různých podob. Kvalita diagramu se promítá i do kvality návrhu databáze. Chyby v ER diagramu, zejména chybné určení kardinality vztahů může vést na chybný návrh databáze. A přímý logický návrh databáze bez použití konceptuálního modelu je možný pouze u menších databází. U rozsáhlých systémů takový postup vede na výsledek, který obsahuje chyby a je obtížně udržitelný. Proto si musíme znovu připomenout význam konceptuálního modelování pro návrh databáze.



Informace k logickému návrhu relační databáze můžete najít v [Pok98] na stranách 191 až 195. Učebnice [Sil05] je transformace ER diagramu velice stručně popsána na str. 241 až 247. Rovněž učebnice [Dat03] věnuje této problematice poměrně malý prostor. Také v doporučené knize Hawryszkiewicz, I.T.: Relational Database Design. An Introduction. Prentice Hall Inc. 1990 je tato problematika prezentována jen na stranách 120 až 130.

I tato skutečnost, že v učebnicích logickému návrhu není věnován velký prostor, svědčí o tom, že jde o problematiku, kde jsou pravidla v podstatě jasně daná.



C3.19 Vysvětlete problémy chybného návrhu databáze a cíl dobrého návrhu. Ilustrujte na příkladě.

C3.20 Vysvětlete způsob transformace entitní množiny při návrhu databáze. Uvažujte i přítomnost složených a vícehodnotových atributů v ER diagramu.

C3.21 Vysvětlete způsob transformace vztahových množin z ER diagramu při návrhu databáze.

C3.22 Vysvětlete způsob transformace slabé entitní množiny z ER diagramu při návrhu databáze.

C3.23 Vysvětlete způsob transformace generalizace/specializace z ER diagramu při návrhu databáze.

C3.24 Navrhněte schéma relační databáze pro příklad C3.10. Nakreslete záhlaví tabulek, určete primární klíče a znázorněte vazby mezi tabulkami.

C3.25 Navrhněte schéma relační databáze pro příklad C3.11. Nakreslete záhlaví tabulek, určete primární klíče a znázorněte vazby mezi tabulkami.

C3.26 Navrhněte schéma relační databáze pro příklad C3.12. Nakreslete záhlaví tabulek, určete primární klíče a znázorněte vazby mezi tabulkami.

C3.27 Navrhněte schéma relační databáze pro příklad C3.13. Nakreslete záhlaví tabulek, určete primární klíče a znázorněte vazby mezi tabulkami.

C3.28 Navrhněte schéma relační databáze pro příklad C3.14. Nakreslete záhlaví tabulek, určete primární klíče a znázorněte vazby mezi tabulkami.

C3.29 Navrhněte schéma relační databáze pro příklad C3.15. Nakreslete záhlaví tabulek, určete primární klíče a znázorněte vazby mezi tabulkami.

C3.30 Navrhněte schéma relační databáze pro příklad C3.16. Nakreslete záhlaví tabulek, určete primární klíče a znázorněte vazby mezi tabulkami.

C3.31 Navrhněte schéma relační databáze pro příklad C3.17. Nakreslete záhlaví tabulek, určete primární klíče a znázorněte vazby mezi tabulkami.

C3.32 Navrhněte schéma relační databáze pro příklad C3.18. Nakreslete záhlaví tabulek, určete primární klíče a znázorněte vazby mezi tabulkami.



Test

T1. Atributy vztahové množiny s kardinalitou 1:M z ER diagramu budou ve správně navržené databázi uloženy:

- a) v samostatné tabulce
- b) v tabulce společně s atributy entitní množiny, u níž je kardinalita 'M'

T2. Atributy vztahové množiny s kardinalitou M:M z ER diagramu budou ve správně navržené databázi uloženy:

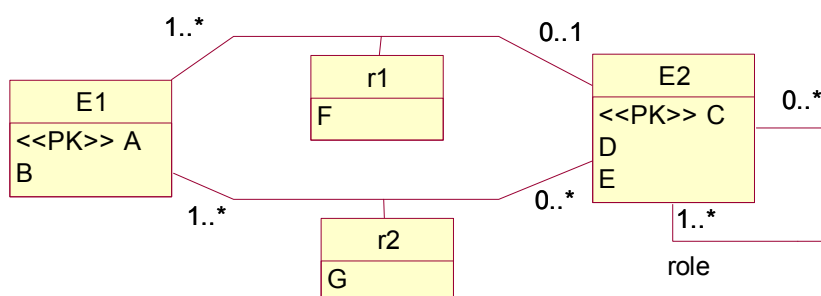
- a) v tabulce společně s atributy některé z obou entitních množin
- b) v každé z obou tabulek pro atributy obou entitních množin

T3. Databáze pro ER diagram z obrázku Obr. 3.45 bude mít:

- a) 3 tabulky
- b) 4 tabulky

T4. Databáze pro ER diagram z obrázku Obr. 3.45 bude mít:

- a) 3 cizí klíče
- b) 4 cizí klíče



Obr. 3.45 Příklad ER diagramu

3.4. Normalizace schématu databáze

Z úvodu ke kapitole 3 již víme, že schéma relační databáze můžeme získat jedním z následujících dvou způsobů:

- vytvořením konceptuálního modelu a jeho transformací
- použitím normalizace s tím, že na počátku předpokládáme, že všechny informace budou uloženy v jedné tabulce a tu normalizujeme.

První způsob jsme si ukázali v předchozí kapitole, nyní se zaměříme na návrh s využitím *normalizace*. Normalizaci můžeme chápat jako postup návrhu databáze, jehož cílem je dostat výsledek, který nebude trpět nedostatky chybného návrhu, jež

jsme si uvedli v předchozí kapitole. Myšlenka normalizace je jednoduchá. Pokud není tabulka navržena dobře, je nutné ji rozdělit na dvě nebo více tabulek jednodušších. Vzniká otázka, jak se pozná, že tabulka není navržena dobře. K ohodnocení kvality návrhu tabulky zavedl E.F.Codd tzv. *normální formy*. V čím vyšší normální formě tabulka je, tím je její návrh kvalitnější. Normální formy definují, jaké vlastnosti tabulka musí mít s ohledem na závislosti mezi jejími atributy. Nejjednodušším typem takových závislostí, kterým se budeme věnovat zejména, jsou závislosti funkční. Příkladem funkční závislosti může být závislost jména studenta na jeho přihlašovací jméno. Pokud platí, že přihlašovací jméno v rámci informačního systému univerzity jednoznačně určuje daného studenta, pak skutečně můžeme říci, že určuje i jeho jméno.

Podobně jako v případě relačního modelu je i kolem normálních forem a normalizace poměrně rozsáhlá teorie. Ta nepochybně přispěla k tomu, že se relační model a relační systémy ve druhé polovině 70. let a v letech 80. minulého století jednoznačně prosadily v konkurenci s tehdy existujícími systémy síťovými a hierarchickými. My se s touto teorií opět seznámíme pouze v rozsahu nezbytném pro pochopení podstaty a pro praktické využití. Nejprve si vysvětlíme základy teorie závislostí a následně jich využijeme při vysvětlení jednotlivých normálních forem a samotného procesu normalizace.

3.4.1. Úvod do teorie závislostí

Teorie závislostí, z níž vychází normalizace schématu relační databáze, se zabývá závislostmi mezi hodnotami v různých sloupcích téže tabulky (tj. závislostmi mezi atributy jedné relace v terminologii relačního modelu). Teorie závislostí rozlišuje tři typy závislostí:

- funkční závislosti (functional dependencies)
- vícehodnotové závislosti, nazývané také multizávislosti (multivalued dependencies)
- závislosti na spojení (join dependencies)

Z praktického hlediska mají největší význam funkční závislosti, kterým se také budeme věnovat nejvíce. Vysvětlíme si také vícehodnotové závislosti. Závislostmi na spojení se zde zabývat nebudeme. Jejich význam je spíše v oblasti teorie než praktického využití.

Protože teorie závislostí, normální formy a normalizace úzce souvisí s teorií relačního modelu dat, používá se zpravidla v učebnicích terminologie relačního modelu. Týká se to zejména používání pojmů relace, atribut relace a n -tice. Budeme se držet této zvyklosti i my s tím, že už máme zažito, že relací zde rozumíme tabulku, atributem její sloupec a n -tici její řádek.



Důležité pro pochopení dalšího obsahu této kapitoly jsou dvě skutečnosti:

1. Zabýváme závislostmi mezi atributy pouze jedné relace. Tj. pro nějakou relaci **R** se schématem $R(A)$, kde A je množina atributů relace **R**, se zabýváme závislostmi mezi atributy množiny A .
2. Závislosti mezi atributy vyplývají z významu (sémantiky) atributů. Abychom tyto závislosti byli schopni odhalit, což je pro normalizaci schématu, jak uvidíme později podstatné, musíme význam atributů dobře pochopit. Musíme rovněž znát a rozumět omezením na data, která vyplývají z reality, kterou

reprezentují. Pokud například atribut LOGIN relace STUDENT je jednoznačně přihlašovací jméno studenta, pak jeho hodnota určuje hodnoty dalších atributů týkající se tohoto konkrétního studenta, např. hodnoty atributů JMÉNO a PŘÍJMENÍ. Podobně pokud má relace STUDENT atribut ADRESA, bude pro nás důležitá znalost toho, zda může mít stejnou adresu několik studentů. Takové vlastnosti atributů nelze odvodit automaticky právě proto, že vyplývají z významu atributů.

Nejjednodušším typem závislosti, kterým se teorie závislostí zabývá, jsou *funkční závislosti*. Funkční závislosti budeme rozumět situaci, kdy hodnota jednoho atributu, např. X , relace jednoznačně určuje hodnotu jiného atributu, např. Y , téže relace. Takovou funkční závislost budeme zapisovat $X \rightarrow Y$.

x+y

Příklad 3.14

Uvažujme příklad relace Klient ($R_ČÍSLO$, JMÉNO, MĚSTO). Nechť je to jedna z tabulek informačního systému banky. Atributy značí postupně rodné číslo, jméno a město bydliště klienta. Předpokládejme, že hodnota rodného čísla jednoznačně určuje klienta. Z této vlastnosti potom vyplývá, že určuje hodnoty ostatních atributů na daném řádku tabulky. Jinými slovy, rodné číslo 600528/0275 by mohlo určovat klienta, který se jmenuje Jan Novák a bydlí v Praze.

Můžeme tedy říci, že hodnota atributu $R_ČÍSLO$ určuje hodnotu atributů JMÉNO a MĚSTO nebo také, že tyto dva atributy na atributu $R_ČÍSLO$ funkčně závisí, což zapíšeme

$$R_ČÍSLO \rightarrow JMÉNO$$

$$R_ČÍSLO \rightarrow MĚSTO$$

Můžeme také psát

$$R_ČÍSLO \rightarrow (JMÉNO, MĚSTO)$$

neboli na rodném čísle závisí hodnota složeného atributu (JMÉNO, MĚSTO), což je určitě pravda.

Pokud bychom chtěli zodpovědět otázku „Závisí některý atribut relace KLIENT funkčně na atributu JMÉNO?“, musíme si uvědomit, že asi může být několik klientů naší banky se stejným jménem. Z tohoto faktu potom také plyne, že tito různí klienti budou mít různá rodná čísla a mohou bydlet v různých městech. Proto například neplatí funkční závislost

$$JMÉNO \rightarrow MĚSTO,$$

což bychom mohli zapsat také jako

$$JMÉNO \nrightarrow MĚSTO.$$

Formálně můžeme funkční závislost definovat takto:

DEF

Definice 3.1 Funkční závislost

Nechť X a Y jsou atributy relace R . Řekneme, že Y *funkčně závisí na* X , zapisujeme

$$X \rightarrow Y,$$

právě když pro libovolné dvě n -tice t_1 a t_2 každého přípustného stavu relace R platí, že je-li x_1 , resp. y_1 hodnota atributu X , resp. Y v n -tici t_1 a x_2 , resp. y_2 hodnota atributu X , resp. Y v n -tici t_2 a $x_1 = x_2$, potom i $y_1 = y_2$.

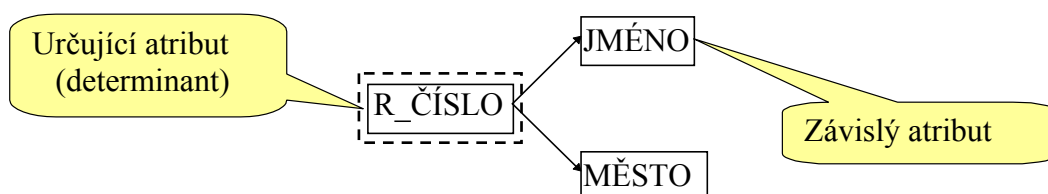
Definice pouze jiným způsobem vyjadřuje fakt, že konkrétní hodnota atributu X určuje právě jednu konkrétní hodnotu atributu Y . Přípustným stavem relace rozumíme každý

takový stav, kdy data splňují všechna integritní omezení vztahující se k dané relaci, tj. kdy data v relaci jsou ve vztahu k realitě, kterou reprezentují, správná.



Funkční závislost můžeme chápat jako binární relaci na množině atributů relace R . Z diskrétní matematiky znáte vlastnosti binárních relací, jako je reflexivita, symetrie apod., které můžeme vyšetřovat, a znáte některé speciální případy binárních relací, např. uspořádání. Zhodnoťte z tohoto pohledu funkční závislost chápanou jako binární relaci.

Názorně můžeme funkční závislosti znázornit pomocí *diagramu funkčních závislostí*. Je to orientovaný graf, jehož uzly jsou atributy dané relace a orientované hrany ukazují funkční závislosti mezi nimi. Pro Příklad 3.14 je diagram funkčních závislostí uveden na Obr. 3.46.



Obr. 3.46 Diagram funkčních závislostí příkladu Příklad 3.14

Obrázek zároveň ukazuje, používanou terminologii, především, že atribut na levé straně zápisu funkční závislosti se označuje *determinant* dané funkční závislosti. Atribut $R_ČÍSLO$ je v diagramu ohraničen čárkovaným obdélníkem. Tak budeme zvýrazňovat kandidátní klíče relace.

Uvidíme, že při normalizaci hraje významnou roli rozlišení atributů, které tvoří kandidátní klíče relace a atributů ostatních. Zavedeme si proto termíny pro odlišení těchto dvou kategorií.



Definice 3.2 Klíčový a neklíčový atribut.

Každý atribut X relace R , který je součástí některého kandidátního klíče relace R , budeme nazývat *klíčovým atributem*. Ostatní atributy budeme nazývat *neklíčovými*.

Nyní se podívejme na praktický důsledek funkčních závislostí mezi atributy. Vyplývá přímo z Definice 3.1. Ta totiž říká, že jestliže platí v relaci funkční závislost jejích atributů $X \rightarrow Y$, pak mají-li být data správná, potom pokud se v relaci vyskytnou dvě n -tice se stejnou hodnotou atributu X , pak musí mít tyto dvě n -tice i stejné hodnoty atributu Y . Ukážeme si takovou situaci na příkladě.

x+y

Příklad 3.15

Uvažujme relaci DISPONUJE našeho bankovního informačního systému. Obsahuje informaci o tom, kteří klienti mohou disponovat s kterými účty a v jaké výši.

DISPONUJE

Č_ÚČTU	R_ČÍSLO	STAV	JMÉNO	LIMIT	MĚSTO
100	600528/0275	100000	Novák	10000	Praha
100	581015/9327	100000	Malá	3000	Brno
130	600528/0275	50000	Novák	5000	Praha
150	450205/3419	150000	Veselý	5000	Ostrava

Pokud si uvědomíme význam atributů, identifikovali bychom tyto funkční závislosti:

$R_ČÍSLO \rightarrow JMÉNO$

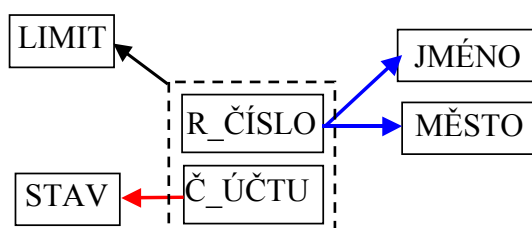
$R_ČÍSLO \rightarrow MĚSTO$, resp. $R_ČÍSLO \rightarrow (JMÉNO, MĚSTO)$

$Č_ÚČTU \rightarrow STAV$

Jestli platí, že s jedním účtem může disponovat více osob a jeden klient může být disponentem několika účtů a že různí klienti mohou mít stanoveny pro disponování s různými účty různé limity a disponenti téhož účtu mohou mít nastaven různý limit, není hodnota atributu LIMIT určena ani samotnou hodnotou atributu Č_ÚČTU, ani samotnou hodnotou atributu R_ČÍSLO. Je určena až kombinací obou těchto hodnot, tj platí funkční závislost

$(Č_ÚČTU, R_ČÍSLO) \rightarrow LIMIT$

Odpovídající diagram funkčních závislostí je uveden na Obr. 3.47. Obrázek zároveň ukazuje, jak budeme v diagramu znázorňovat složené atributy – složky, které složený atribut tvoří, ohraničíme čarou. Pokud je takový složený atribut determinantem nějaké funkční závislosti, vede odpovídající orientovaná hrana od hranice. Je-li závislým atributem, tak tam naopak končí. V našem příkladě je složený atribut (Č_ÚČTU, R_ČÍSLO) zároveň kandidátním klíčem, proto je ohraničující čára čárkovaná.



Obr. 3.47 Diagram funkčních závislostí z příkladu Příklad 3.15

V tabulce je pěkně vidět důsledek funkčních závislostí a naznačuje využití při normalizaci schématu databáze. Z faktu, že jeden klient může disponovat několika účty vyplývá, že v tabulce se ve sloupci R_ČÍSLO může vyskytovat stejné rodné číslo v tolika řádcích, s kolika účty může daný klient disponovat. Z funkční závislosti $R_ČÍSLO \rightarrow (JMÉNO, MĚSTO)$ podle definice plyne, že ve všech takových řádcích musí být také stejná hodnota ve sloupcích JMÉNO a MĚSTO. Například, jak je vidět v tabulce, pokud klient s rodným číslem 600528/0275 může disponovat s účty číslo 100 a 130, budou v tabulce dva řádky s tímto rodným číslem. A protože toto rodné číslo patří panu Novákovi z Prahy, musí být právě tyto hodnoty ve sloupcích JMÉNO

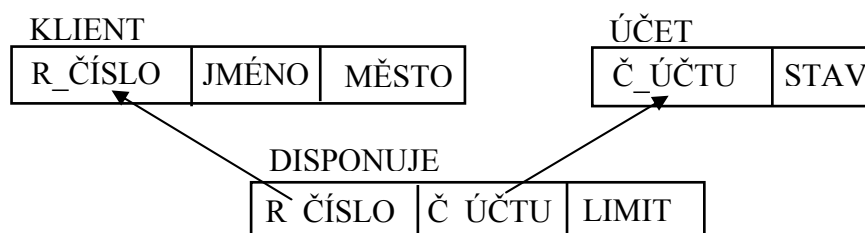
a MĚSTO těchto dvou řádků. Podobný důsledek má funkční závislost Č_ÚČTU → STAV, jak je naznačeno v tabulce červenou barvou. Vidíme, že se nám v naší tabulce opakuje, resp. může opakovat (záleží na konkrétním stavu tabulky) určitá informace, konkrétně, jak se jmenuje a kde bydlí klient s určitým rodným číslem a jaký je stav účtu s jistým číslem účtu. Z úvodní přednášky již víme, že takový jev označujeme za redundanci a z kapitoly 3.3, že je to jev z hlediska logického návrhu databáze nežádoucí. Pokud se například náš pan Novák přestěhuje z Prahy do Brna, musí se tato změna promítnout do všech řádků, kde se nachází informace o jeho bydlišti. Podobně při změně stavu účtu.

Víme, že při správně navržené databázi by taková informace vždy měla být pouze na jednom místě (na jednom řádku nějaké tabulky). Můžeme proto říci, že naše tabulka DISPONUJE není správně navržena. Aby byla data v ní správná, musel by SRBD nebo naše aplikace zajišťovat kontrolu dodržení integritních omezení, která z výše uvedených funkčních závislostí vyplývají. My bychom rádi využili pro kontrolu dvou základních obecných integritních omezení relačního modelu – omezení integrity entit a referenční integrity.

Zkusme porovnat tabulky z příkladů Příklad 3.14 a Příklad 3.15 a odpovídající diagramy funkčních závislostí z Obr. 3.46 a Obr. 3.47 s přihlédnutím k tomu, které atributy představují kandidátní klíče.

V tabulce KLIENT je jediným kandidátním klíčem atribut R_ČÍSLO. Z diagramu funkčních závislostí na Obr. 3.46 je vidět, že kromě funkčních závislostí ostatních atributů na tomto kandidátním klíči nejsou v tabulce žádné jiné funkční závislosti. To signalizuje dobrý návrh, protože pokud je determinantem všech funkčních závislostí kandidátní klíč, nemůže existovat v tabulce více než jeden řádek se stejnou hodnotou tohoto atributu. V našem případě to konkrétně znamená, že informace o klientovi, která je určena jeho rodným číslem, se nachází vždy na jediném řádku této tabulky. V tabulce tedy neexistuje redundance.

U tabulky DISPONUJE z příkladu Příklad 3.15 je situace jiná. Jediným kandidátním klíčem zde je složený atribut (Č_ÚČTU, R_ČÍSLO). Z diagramu funkčních závislostí na Obr. 3.47 je vidět, že zde existují i závislosti na samotných attributech Č_ÚČTU a R_ČÍSLO. Jejich hodnoty se ale samozřejmě v tabulce mohou opakovat. Pouze kombinace čísla účtu a rodného čísla musí být v tabulce jednoznačná. Z toho potom vyplývá redundance, kterou jsme diskutovali výše. Tato tabulka není navržena správně. Správný návrh bude vyžadovat, rozklad tabulky DISPONUJE na tři tabulky. V první s atributy R_ČÍSLO, JMÉNO a MĚSTO bude informace o klientech. Ve druhé s atributy Č_ÚČTU a STAV bude informace o účtech. Teprve ve třetí bude informace o tom, kdo s kterým účtem může disponovat. Tato tabulka bude mít atributy Č_ÚČTU, R_ČÍSLO a LIMIT. Tyto tři tabulky budou provázány prostřednictvím cizích klíčů, jak ukazuje Obr. 3.48. Jak provést tento rozklad, abychom dostali dobré schéma databáze, se naučíme, až si vysvětlíme normalizaci.



Obr. 3.48 Výsledek normalizace tabulky DISPONUJE z příkladu Příklad 3.15



Praktický důsledek funkčních závislostí mezi atributy relace, který se projevuje jako integritní omezení, jež musí být splněno, můžeme zformulovat takto:

Opakuje-li se v relaci stejná hodnota determinantu funkční závislosti, musí se opakovat i odpovídající stejná hodnota závislého atributu. Jestliže je determinantem atribut, který neobsahuje kandidátní klíč, vzniká proto v relaci redundance.

Již jsme si uvedli, že kvalitu návrhu tabulky relační databáze můžeme ověřit určením normální formy, jejíž podmínky tabulka splňuje. Uvidíme, že z praktického hlediska nejdůležitější z řady normálních forem definují požadované vlastnosti tabulek právě s ohledem na funkční závislosti atributů. Pro tyto účely si zavedeme tři speciální typy funkčních závislostí.

Prvním je funkční závislost, kterou označujeme jako *triviální funkční závislost*. Je to funkční závislost, která platí vždy.



Definice 3.3 Triviální funkční závislost

Nechť X a Y jsou atributy relace R takové, že $Y \subseteq X$. Pak pro ně platí funkční závislost $X \rightarrow Y$. Takovou funkční závislost budeme nazývat *triviální funkční závislost*.

Uvažujme složený atribut (JMÉNO, MĚSTO) z příkladu Příklad 3.14. a libovolnou jeho hodnotu, např. ('Novák', 'Praha'). Je zřejmé, že každá taková dvojice jednoznačně určuje hodnoty svých složek. Proto nemohou neplatit triviální funkční závislosti

$$(JMÉNO, MĚSTO) \rightarrow (JMÉNO, MĚSTO)$$

$$(JMÉNO, MĚSTO) \rightarrow JMÉNO$$

$$(JMÉNO, MĚSTO) \rightarrow MĚSTO$$



Pokuste se tvrzení obsažené v definici triviální funkční závislosti dokázat exaktně matematicky.

Dalším typem funkční závislosti, který je důležitý z pohledu normalizace schématu databáze, je tzv. *plná funkční závislost*. Tento typ funkční závislosti se týká situace, kdy je determinantem složený atribut.

Uvažujme funkční závislost $(\check{C}_ÚČTU, R_ČÍSLO) \rightarrow LIMIT$ z příkladu Příklad 3.15. Už jsme si vysvětlili, že při sémantice, kterou předpokládáme, není hodnota atributu $LIMIT$ určena ani samotným číslem účtu, ani samotným rodným číslem disponující osoby. Teprve kombinace těchto dvou hodnot jednoznačně určuje výši limitu pro operace daného klienta s daným účtem. Proto platí výše uvedená funkční závislost, ale neplatí závislosti

$$\check{C}_ÚČTU \rightarrow LIMIT$$

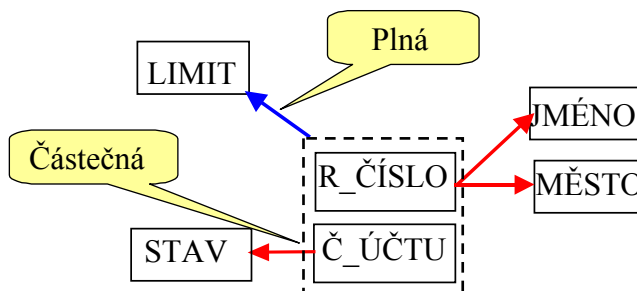
$$R_ČÍSLO \rightarrow LIMIT$$

Potom funkční závislost $(\check{C}_ÚČTU, R_ČÍSLO) \rightarrow LIMIT$ označujeme jako *plnou funkční závislost*. Pro plnou funkční závislost složeného atributu tedy musí platit, že k určení hodnoty závislého atributu nestačí znalost hodnoty jen některé složky determinantu.

Nyní uvažujme závislost $(\check{C}_ÚČTU, R_ČÍSLO) \rightarrow STAV$, která v příkladu Příklad 3.15 platí. Tentokrát ale neplatí, že až kombinace čísla účtu a rodného čísla určí

jednoznačně hodnotu stavu účtu. K tomu stačí i samotná hodnota čísla účtu. Platí tedy funkční závislost $\check{C}_ÚČTU \rightarrow STAV$. K určení hodnoty závislého atributu stačí znalost jedné ze složek determinantu. Proto závislost $(\check{C}_ÚČTU, R_ČÍSLO) \rightarrow STAV$ není plnou funkční závislost. Takovou závislost budeme označovat jako *částečnou funkční závislost*.

Diagram funkčních závislostí příkladu Příklad 3.15 s vyznačením plných a částečných funkčních závislostí je uveden na Obr. 3.49.



Obr. 3.49 Plná a částečné funkční závislosti v příkladu Příklad 3.15

DEF

Definice 3.4 Plná funkční závislost

Nechť X a Y jsou atributy relace \mathbf{R} . Řekneme, že atribut Y je *plně funkčně závislý* na atributu X , právě když je funkčně závislý na X a není funkčně závislý na žádném atributu $Z \subset X$.

!

Praktický důsledek funkčních závislostí na složeném kandidátním klíči, které nejsou plné, jsme diskutovali v souvislosti s příkladem Příklad 3.15. Viděli jsme, že je-li kandidátní klíč relace složený, způsobují částečné funkční závislosti redundanci. Můžeme ho zformulovat do následující podoby:

Je-li kandidátní klíč relace složený a existuje atribut relace, který není na kandidátním klíči závislý plně, vzniká v relaci redundance.

Posledním typem funkční závislosti, který je důležitý pro normalizace schématu relace, je tzv. *tranzitivní závislost*.

Pokud jste zkusili vyšetřit vlastnosti funkční závislosti chápané jako binární relace, zjistili jste, že je tranzitivní. Znamená to, že platí-li pro atributy X , Y a Z nějaké relace \mathbf{R} funkční závislosti $X \rightarrow Y$ a $Y \rightarrow Z$, pak platí také závislost $X \rightarrow Z$. Takovou „nepřímou“ funkční závislost budeme nazývat závislostí tranzitivní.

DEF

Definice 3.5 Tranzitivní funkční závislost

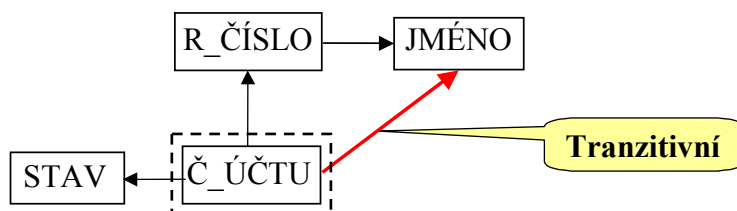
Nechť X a Y jsou atributy relace \mathbf{R} a nechť platí $X \rightarrow Y$, avšak neplatí $Y \rightarrow X$, a nechť existuje atribut Z relace \mathbf{R} , který není v X , ani Y , a platí $Y \rightarrow Z$. Potom říkáme, že atribut Z je *tranzitivně závislý* na atributu X .

Příklad 3.16

Uvažujme relaci ÚČET se schématem ÚČET ($\check{C}_ÚČTU$, STAV, $R_ČÍSLO$, JMÉNO). Atribut $R_ČÍSLO$, resp. JMÉNO značí rodné číslo, resp. jméno vlastníka účtu. Předpokládáme, že každý účet má právě jednoho vlastníka.

Hodnota stavu na účtu je určena číslem účtu, číslo účtu určuje i vlastníka účtu (je jeden), tj. jeho rodné číslo. Rodné číslo pak určuje jméno klienta, který je vlastníkem

účtu. Atribut JMÉNO proto závisí na atributu Č_ÚČTU tranzitivně – viz Obr. 3.50.



Obr. 3.50 Diagram funkčních závislostí příkladu Příklad 3.16

Podívejme se opět na praktický důsledek takové tranzitivní funkční závislosti. Všimněte si, že jde o tranzitivní závislost neklíčového atributu (není součástí žádného kandidátního klíče relace) JMÉNO na kandidátním klíči Č_ÚČTU. Tato závislost je „indukována“ závislostí $R_ČÍSLO \rightarrow JMÉNO$. Tím, že atribut $R_ČÍSLO$ není kandidátním klíčem, může se jeho hodnota v relaci opakovat a stejně se musí opakovat i hodnota atributu JMÉNO. Již byste měli být schopni říci, že tato tranzitivní funkční závislost opět vede na nežádoucí redundanci – v naší tabulce se opakuje informace, jak se jmenuje klient banky s daným rodným číslem.

ÚČET

Č_ÚČTU	STAV	R_ČÍSLO	JMÉNO
100	100000	600528/0275	Novák
120	135000	581015/9327	Malá
130	50000	600528/0275	Novák
150	150000	450205/3419	Veselý

V našem příkladě se opakuje informace o tom, že vlastník účtů číslo 100 a 130 se jmenuje Novák. Opět můžeme říci, že tato tabulka není navržena správně.



Již jsme si naznačili, že při normalizaci nesprávně navržené tabulky provádíme její rozklad na několik tabulek. Zkuste se zamyslet nad tím, jak by asi vypadal správný výsledek zde.



Můžeme nyní zformulovat praktický důsledek tranzitivních funkčních závislostí v relaci:

Existuje-li funkční závislost nějakého atributu relace na neklíčovém atributu, vzniká tranzitivní závislost na kandidátních klíčích relace a v jejím důsledku redundance.



Zkuste vysvětlit předchozí tvrzení, tj. že funkční závislost na neklíčovém atributu vede na tranzitivní závislost na kandidátních klíčích.

Pomůcka: Řešte nejprve vztah neklíčových atributů a kandidátních klíčů.



Možná vás při pohledu na diagramy funkčních závislostí, které jsme vytvořili u ilustračních příkladů, napadlo, že existují i jiné funkční závislosti mezi atributy a ty jsme v diagramech nezakreslovali. Je pravda, že my jsme si v řadě případů výklad zjednodušili. Pokud bychom chtěli být přesní, museli bychom pracovat s pojmy uzávěr množiny funkčních závislostí, pokrytí apod. (viz např. [Sil05]). Například platí-li podle diagramu na Obr. 3.50 funkční závislost $Č_ÚČTU \rightarrow STAV$, pak určitě také platí závislost $(Č_ÚČTU, R_ČÍSLO) \rightarrow STAV$. Mohli bychom říci, že tato závislost z první „logicky vyplývá“.

Naším cílem ale nebylo naučit se teorii závislostí v celé její šíři. Zaměřili jsme se

pouze na základy potřebné pro pochopení normálních forem a procesu normalizace a to zejména z pohledu jejich praktického využití. Zjednodušení se týkalo i diagramů funkčních závislostí. Budeme v nich znázorňovat zejména ty závislosti, které způsobují redundanci, nebo naopak ty, které ukazují, že je relace navržena správně.

Na závěr výkladu základů teorie závislostí zmíním ještě vícehodnotové závislosti. Nebudeme se jimi ale již zabývat tak podrobně.

Viděli jsme, že podstata funkční závislosti spočívá v tom že hodnota jednoho atributu určuje hodnotu atributu druhého. Vícehodnotovou závislost můžeme považovat za obecnější případ závislosti, kdy hodnota jednoho atributu určuje množinu hodnot atributu druhého nezávisle na hodnotách atributů ostatních. Uvažujme následující příklad.

$x+y$

Příklad 3.17

Uvažujme relaci PŘEDMĚT se schématem PŘEDMĚT (NÁZEV, UČITEL, UČEBNICE), kde NÁZEV značí jednoznačný název předmětu, UČITEL je jméno (pro jednoduchost předpokládejme jednoznačné) učitele a UČEBNICE je název (opět uvažujeme jednoznačný) učebnice, která se v předmětu používá. Předpokládejme, že předmět může učit několik učitelů a může se v něm používat několik učebnic s tím, že učebnice jsou pro předmět předepsány a tedy nezávisí na učitelích. Uvažujme několik n-tic této relace pro předmět 'fyzika'.

PŘEDMĚT

NÁZEV	UČITEL	UČEBNICE
fyzika	doc.Kovář	Fyzika I
fyzika	doc.Kovář	Sbírka úloh z fyziky
fyzika	doc.Zelený	Fyzika I
fyzika	doc.Zelený	Sbírka úloh z fyziky

Vidíme, že tento předmět učí dva učitelé a používají se dvě učebnice. Za předpokladu výše uvedené sémantiky platí v této relaci dvě vícehodnotové závislosti

NÁZEV $\rightarrow\!\!\rightarrow$ UČITEL

NÁZEV $\rightarrow\!\!\rightarrow$ UČEBNICE

Hodnota atributu NÁZEV 'fyzika' určuje hodnoty {'doc.Kovář', 'doc.Zelený'} atributu UČITEL a {'Fyzika I', 'Sbírka úloh z fyziky'} atributu UČEBNICE.

Přestože by se zdálo, že z pohledu návrhu tabulky je všechno v pořádku, ve skutečnosti tomu tak není. Opět existuje v tabulce redundance, tentokrát způsobená vícehodnotovou závislostí. Navíc zajištění příslušného integritního omezení je ve srovnání s funkčními závislostmi mnohem složitější.



Zamyslete se nad tím, jak by vypadalo u tabulky PŘEDMĚT vložení informace o další učebnici, která se ve fyzice používá a jak by vypadalo vložení informace o předmětu 'Databázové systémy', který učí tři učitelé a používají se tři učebnice.

3.4.2. Normální formy a proces normalizace schématu

V této kapitole si ukážeme, jak lze využít znalosti o závislostech atributů při logickém návrhu relační databáze.

V kapitole 3.3 jsme si ukázali, že mezi hlavní problémy chybného návrhu patří zejména:

- opakující se informace (redundance)
- nemožnost reprezentovat určitou informaci
- složitá kontrola integritních omezení

Všem třem nedostatkům umožňuje se vyvarovat proces normalizace. V extrémním případě bychom mohli provést kompletní logický návrh pouze použitím normalizace. Víme ale, že při návrhu rozsáhlých databází by takový postup byl značně nepraktický. V praxi normalizaci nejčastěji používáme ke kontrole kvality návrhu a úpravám při zjištění nedostatků schématu databáze navrženého typicky transformací konceptuálního diagramu. Především ale pochopení funkčních závislostí, normálních forem a normalizace výrazným způsobem pomáhá vyvarovat se chyb logického návrhu relační databáze, ať už je prováděn transformací z konceptuálního modelu nebo intuitivně přímo při návrhu jednoduchých databází. Proto by zvládnutí této problematiky mělo patřit mezi atributy dobrého návrháře databáze.

Myšlenka normalizace schématu relačních databází se opírá o dva základní koncepty:

1. Hierarchii tzv. normálních forem, které odrážejí kvalitu logického návrhu schématu relace z hlediska výše uvedených nedostatků.
2. Postupnou dekompozici schématu relace, které vykazuje nedostatky.

Nejčastěji se uvádí šest normálních forem. Každá z nich definuje vlastnosti, které relace musí mít, aby normální formu splňovala. Normální formy tvoří hierarchii. První tři (1NF, 2NF a 3NF) a postup zvaný normalizace zavedl již na začátku 70. let minulého století E.F.Codd. Ten později přidal další normální formu, která se označuje jako Boyce-Coddova (BCNF). Ještě později vznikly 4NF, 5NF. Těchto šest normálních forem tvoří hierarchii v pořadí, jak jsme je uvedli. Znamená to, že má-li relace splňovat k -tou normální formu, musí splňovat podmínky $k-1$ normálních forem a něco navíc.

1NF odráží základní požadavek relačního modelu dat z hlediska složitosti dat – atomické hodnoty jednoduchých atributů. 2NF, 3NF a BCNF definují požadované vlastnosti z hlediska funkčních závislostí atributů. Konečně 4NF, resp. 5NF definují požadované vlastnosti z hlediska vícehodnotových závislostí, resp. závislosti na spojení.

Pokud relace nesplňuje požadovanou normální formu, dekomponujeme ji na několik relací s cílem nedostatky, které brání splnění příslušné normální formy, odstranit. V dalších odstavcích si nejprve vysvětlíme, co rozumíme dekompozicí relace, ukážeme, jaké vlastnosti budeme od dekompozice požadovat, a v kontextu definování jednotlivých normálních forem si postup normalizace vysvětlíme.



Normální forma definuje požadované vlastnosti schématu relace. Proto bychom měli hovořit o normální formě schématu relace a ne normální formě relace. Běžně se ale používá obojí označení a budeme ho používat i my. Označením „relace je v k -té normální formě“ budeme rozumět, že její schéma je v k -té normální formě.

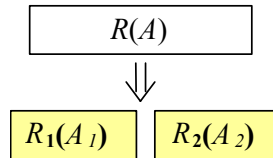


Definice 3.6 Dekompozice

Necht' $R(A)$ je schéma relace **R**. Množina schémat $\{R_1(A_1), R_2(A_2), \dots, R_n(A_n)\}$ je

dekompozicí schématu $R(A)$, jestliže $A = A_1 \cup A_2 \cup \dots \cup A_n$.

Z definice je zřejmé, že dekompozicí rozumíme každý takový rozpad schématu relace na několik schémat, při kterém se žádný atribut původní relace ani neztratí, ani žádný nový nepřibude. Typickou situací v procesu normalizace bude, že při dekompozici nahradíme původní schéma relace dvojicí schémat, jak ukazuje Obr. 3.51.



Obr. 3.51 Typická dekompozice schématu relace při normalizaci

Přestože podle definice obecně může existovat k danému schématu řada dekompozic, ne každá bude pro naše účely přijatelná. Budeme požadovat, aby dekompozice splňovala následující tři vlastnosti:

- bezeztrátovost při zpětném spojení
- zachování závislostí
- odstranění opakování informace (redundance)

Bezeztrátovou dekompozicí (Lossless-Join/Nonloss decomposition) budeme rozumět takovou dekompozici, která zajistí že při rekonstrukci původní relace z nově vzniklých relací operací přirozeného spojení nedojde ke ztrátě informace. Zřejmé to bude z následujícího příkladu.

Příklad 3.18

$x+y$

Uvažujme tabulku ÚČET1 se schématem ÚČET1 ($\underline{\text{Č_ÚČTU}}$, R_ČÍSLO, STAV, POBOČKA, JMĚNÍ), která uchovává informaci o účtech. $\underline{\text{Č_ÚČTU}}$ je jednoznačné číslo účtu, R_ČÍSLO je rodné číslo klienta banky, který může s účtem disponovat, STAV udává množství peněz na účtu, POBOČKA je jednoznačný název pobočky, u které je účet veden, a JMĚNÍ je výše základního jmění této pobočky. Účtem může disponovat několik klientů banky a každý klient může obecně disponovat několika účty.

ÚČET1

$\underline{\text{Č_ÚČTU}}$	R_ČÍSLO	STAV	POBOČKA	JMĚNÍ
100	600528/0275	100000	Jánská	10000000
100	581015/9327	100000	Jánská	10000000
130	600528/0275	50000	Palackého	5000000
150	450205/3419	150000	Palackého	5000000

Jednu z možných dekompozic tvoří dvojice schémat $\text{ÚČET_V_POB}(\underline{\text{Č_ÚČTU}}, \text{R_ČÍSLO}, \text{POBOČKA}, \text{JMĚNÍ})$, $\text{STAV}(\text{R_ČÍSLO}, \text{STAV})$. Odpovídající tabulky, které vzniknou projekcí tabulky ÚČET1, budou:

ÚČET V POB

$\underline{\text{Č_ÚČTU}}$	R_ČÍSLO	POBOČKA	JMĚNÍ
100	600528/0275	Jánská	10000000
100	581015/9327	Jánská	10000000
130	600528/0275	Palackého	5000000
150	450205/3419	Palackého	5000000

STAV

R_ČÍSLO	STAV
600528/0275	100000
581015/9327	100000
600528/0275	50000
450205/3419	150000

Společným sloupcem, na základě jehož hodnot můžeme provést spojení, je R_ČÍSLO. Spojením vznikne tabulka

Č ÚČTU	R ČÍSLO	STAV	POBOČKA	JMĚNÍ
100	600528/0275	100000	Jánská	10000000
100	600528/0275	50000	Jánská	10000000
100	581015/9327	100000	Jánská	10000000
130	600528/0275	50000	Palackého	5000000
130	600528/0275	100000	Palackého	5000000
150	450205/3419	150000	Palackého	5000000

Porovnáním s původní tabulkou ÚČET1 zjistíme, že nám spojením přibýly dva řádky, které v původní tabulce nebyly. Jsou v tabulce vzniklé spojením zvýrazněny. Podívejme se, jak vznikly. V tabulce ÚČET1 byl klient s rodným číslem 600528/0275 veden jako disponent dvou účtů – číslo 100 a 130. Po dekompozici se proto v obou tabulkách objevily dva řádky s tímto rodným číslem. Jejich spojením vznikly kromě původních dvou řádků další dva, které kombinují údaje obsahující mj. číslo účtu s nesprávným stavem (stavem druhého účtu téhož disponenta).

Takovou dekompozici, která nezajistí spojením původní obsah tabulky, označujeme jako *ztrátovou* a je pro nás nepřijatelná. Při normalizaci vyžadujeme dekompozici *bezeztrátovou*. Dá se ukázat, že platí následující věta:

DEF

Věta 3.1 Podmínka bezeztrátové dekompozice

Nechť **R** je relace s množinou atributů **A**. Její dekompozice tvořená relacemi **R1** a **R2** s množinami atributů **A1** a **A2** bude *bezeztrátová*, jestliže platí:
 $A1 \cap A2 \rightarrow A1$ nebo $A1 \cap A2 \rightarrow A2$.

Tj. společný atribut musí být kandidátním klíčem alespoň jedné z relací.

Pro příklad Příklad 3.18 by bezeztrátovou dekompozicí byla dekompozice ÚČET_A_POB (Č_ÚČTU, STAV, POBOČKA, JMĚNÍ) a DISPONUJE (R_ČÍSLO, Č_ÚČTU) – viz Obr. 3.52.



Obr. 3.52 Bezeztrátová dekompozice příkladu Příklad 3.18

Obsah tabulek by byl:

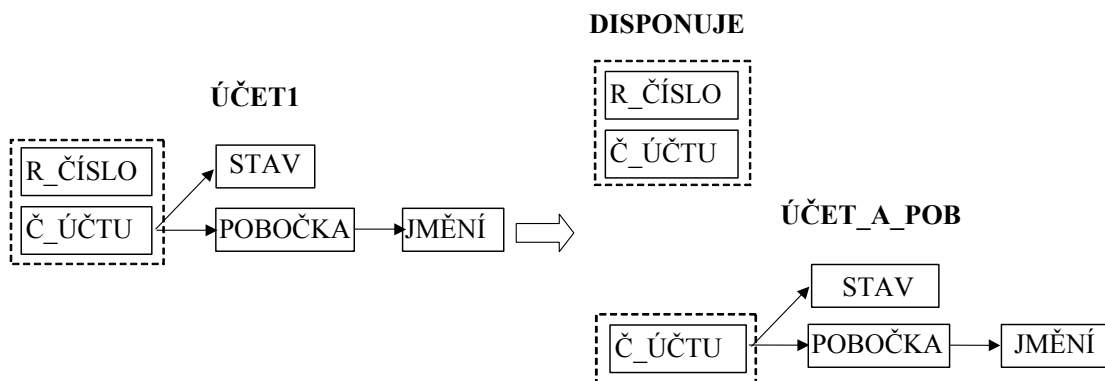
ÚČET_A_POB				DISPONUJE	
Č_ÚČTU	STAV	POBOČKA	JMĚNÍ	Č_ÚČTU	R_ČÍSLO
100	100000	Jánská	10000000	100	600528/0275
130	50000	Palackého	5000000	100	581015/9327
150	150000	Palackého	5000000	130	600528/0275
				150	450205/3419

Je zřejmé, že nyní se každý řádek tabulky DISPONUJE spojí s jedním řádkem tabulky ÚČET_A_POB právě proto, že Č_ÚČTU je v tabulce ÚČET_A_POB kandidátním (v tomto případě i primárním) klíčem. Vzniknou tak původní řádky tabulky ÚČET1.



Zamyslete se nad tím, jestli je tabulka ÚČET1 navržena dobře, nebo ne a proč. Pokud dojdete k závěru, že není, potom se podobně zamyslete nad tím, jestli uvedená dekompozice odstraňuje nedostatky, které jste našli u tabulky ÚČET1.

Nyní se podívejme na druhou z požadovaných vlastností dekompozice – *zachování závislostí*. K vysvětlení opět použijeme Příklad 3.18. Na Obr. 3.53 je znázorněn diagram funkčních závislostí relace ÚČET a relací tvořících její bezztrátovou dekompozici – DISPONUJE a ÚČET_A_POB



Obr. 3.53 Funkční závislosti v relaci ÚČET1 a v relacích DISPONUJE a ÚČET_A_POB

Požadavek zachování závislostí znamená, že při dekompozici budou všechny původní funkční závislosti zachovány a kontrolovatelné v rámci relací tvořících dekompozici. Jinými slovy, nebudou existovat závislosti, jejichž dodržení je možné zkontrolovat pouze operacemi nad více než jednou relací.

Z Obr. 3.53 je vidět, že všechny funkční závislosti, které jsou znázorněny v diagramu pro relaci ÚČET1, jsou zachovány, protože to jsou závislosti nad atributy jedné z relací tvořících dekompozici - relace ÚČET_A_POB.

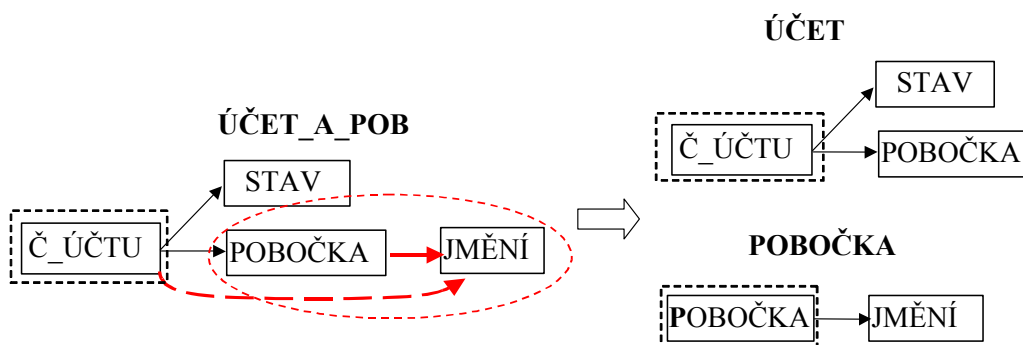
Třetím základním požadavkem na dekompozici je *odstranění opakování informace*, tedy *redundance*. Pokud jste se zamysleli nad schématy tabulek použitými v příkladu Příklad 3.18, měli jste zjistit, že v tabulce ÚČET1 je redundance, neboť se tam opakuje informace jednak o tom, kolik peněz je na kterém účtu a u které pobočky je účet veden, jednak o tom, kolik činí jmění které pobočky. Dekompozicí se situace trochu vylepšila, nicméně v tabulce ÚČET_A_POB se pořád ještě opakuje informace o tom, kolik činí jmění které pobočky.

ÚČET_A_POB

Č_ÚČTU	STAV	POBOČKA	JMĚNÍ
100	100000	Jánská	10000000
130	50000	Palackého	5000000
150	150000	Palackého	5000000

Zkusme se zamyslet nad tím, co je příčinou. Opakování informace způsobuje závislost POBOČKA → JMĚNÍ, přesněji řečeno skutečnost, že tatáž hodnota determinantu této funkční závislosti, tedy atributu POBOČKA, se v tabulce může opakovat, protože atribut POBOČKA není kandidátním klíčem relace. V důsledku uvedené funkční závislosti se pak musí opakovat i odpovídající hodnota atributu JMĚNÍ. Když se podíváme na diagram funkční závislosti z Obr. 3.53, vidíme, že atribut JMĚNÍ závisí na kandidátním klíči relace, kterým je Č_ÚČTU, tranzitivně.

Pokusme se tento problém řešit. Provedeme dekompozici na dvě relace tak, abychom do jedné umístili atributy, jejichž závislosti způsobují problémy, a do druhé zbývající atributy s tím, že zajistíme aby byla dekompozice bezetrátová. Situaci znázorňuje Obr. 3.54.



Obr. 3.54 Tranzitivní závislost v relaci ÚČET_A_POB a její odstranění dekompozicí

Po této úpravě zjistíme, že už je vše v pořádku.

ÚČET			POBOČKA	
Č_ÚČTU	STAV	POBOČKA	POBOČKA	JMĚNÍ
100	100000	Jánská	Jánská	10000000
130	50000	Palackého	Palackého	5000000
150	150000	Palackého		

Informace o účtu je vždy na jednom řádku tabulky, podobně informace o pobočce. V diagramech funkčních závislostí se to projeví tak, že tam jsou plné funkční závislosti všech neklíčových atributů na kandidátních klíčích, které nejsou tranzitivní.

Až si vysvětlíme normální formy, uvidíme, že splnění těchto požadavků odpovídá třetí normální formě. Zároveň si ale vysvětlíme, že úplné odstranění redundance zajistí až tzv. *Boyce-Coddova normální forma*, která řeší ještě některé speciální situace, které mohou nastat. Tato normální forma zajišťuje, že všechny netriviální funkční závislosti jsou dány závislostí na kandidátních klíčích. Protože konkrétní hodnota kandidátního klíče se může v tabulce vyskytovat pouze jedenkrát, nehrozí opakování informace. Lze ale ukázat, že současné splnění požadavku zachování funkčních závislostí a dosažení BCNF není možné ve všech případech. Pokud toto nastane, musíme se rozhodnout, zda upřednostníme snadnější kontrolu integritních omezení plynoucích z funkčních závislostí nebo úplně odstranění redundance.



Cíle logického návrhu založeného na normalizaci lze shrnout do následujících tří bodů:

- bezetrátovost dekompozice
- zachování závislostí
- dosažení minimálně BCNF, resp. 3NF.

Nyní již můžeme přikročit k vysvětlení normálních forem a k ukázce jejich využití v procesu normalizace. Už víme, že normální formy vyjadřují požadavky na schéma relace z hlediska závislostí mezi atributy a že jsou určitou mírou kvality návrhu. Víme také, že normální formy tvoří hierarchii a že 2NF, 3NF a BCNF definují požadované vlastnosti z hlediska funkčních závislostí atributů, 4NF z hlediska vícehodnotových závislostí a 5NF z hlediska závislostí na spojení.

S první normální formou jsme se setkali už u relačního modelu. Odráží skutečnost, že relace relační databáze musí být normalizované, tedy jednoduché atributy musí být definované na doménách obsahujících pouze atomické (skalární) hodnoty.

DEF**Definice 3.7** První normální forma (1NF)

Relace je v *první normální formě*, právě když všechny její jednoduché domény obsahují pouze atomické hodnoty.

Postup normalizace budeme ilustrovat na relaci ÚČET1 (Č_ÚČTU, R_ČÍSLO, STAV, POBOČKA, JMĚNÍ) z příkladu Příklad 3.18.

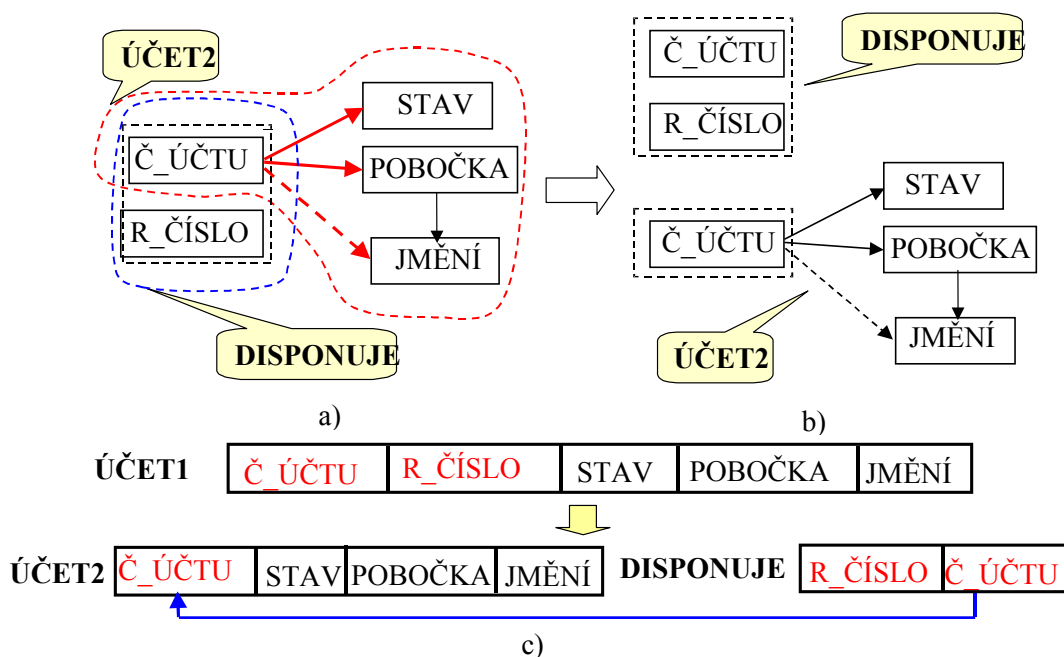
Za předpoklady, že všechny jednoduché atributy této relace nabývají atomických hodnot, což jsme dosud mlčky předpokládali, můžeme říci, že je minimálně 1NF. Myslíme tím, že možná splňuje i požadavky vyšších normálních forem, ale při normalizaci schématu musíme vždy začít nejnižší, kterou je právě 1NF, a postupujeme k normám vyšším, dokud nedosáhneme BCNF, resp. 3NF. V každém kroku provedeme kontrolu, zda relace splňuje danou normální formu. Pokud splňuje, pokračujeme dalším krokem. Pokud ne, provedeme dekompozici s cílem splnění požadavků právě kontrolované normální formy. Tím, že se nám rozpadne dekompozicí schéma relace na schémata několik, musíme v dalších krocích normalizovat stejným postupem každé ze vzniklých schémat.

Druhá a třetí normální forma definují požadavky, které musí splňovat neklíčové atributy z hlediska jejich funkčních závislostí na kandidátních klíších. Druhá vyžaduje plné funkční závislosti a třetí netranzitivní závislosti. Už jsme viděli, že právě tyto závislosti způsobují redundanci.

DEF**Definice 3.8** Druhá normální forma (2NF)

Schéma relace **R** je ve *druhé normální formě*, právě když je v 1NF a každý její neklíčový atribut, je plně funkčně závislý na každém kandidátním klíči relace **R**.

Diagram funkčních závislostí relace ÚČET1 je uveden na Obr. 3.55 a). Je vidět, že relace nesplňuje 2NF, protože neklíčové atributy STAV, POBOČKA a JMĚNÍ funkčně závisí na atributu Č_ÚČTU, který je pouze složkou složeného kandidátního klíče (Č_ÚČTU, R_ČÍSLO). Nejde tedy o plnou funkční závislost, jak vyžaduje 2NF.



Obr. 3.55 a) Diagram funkčních závislostí relace ÚČET1
 b) Diagramy funkčních závislostí relací DISPONUJE a ÚČET2, které vzniknou dekompozicí při převodu do 2NF
 c) Schéma před a po dekompozici

Provedeme tedy dekompozici s cílem odstranit zdroj problémů, tj. tyto závislosti, které nejsou plné. Dostaneme relaci, kterou si označíme např. ÚČET2. Jejím kandidátním klíčem bude Č_ÚČTU, protože určuje hodnoty všech ostatních atributů v této relaci. Schéma druhé relace musí obsahovat zbývající atributy, tj. R_ČÍSLO, a aby byla bezetrátová, i kandidátní klíč druhé z relací tvořících dekompozici, tj. Č_ÚČTU. Nazvěme ji DISPONUJE. Diagramy funkčních závislostí obou nově vzniklých relací jsou na Obr. 3.55 b) a schéma ukazující vazbu mezi novými relacemi na Obr. 3.55 c). Červeně jsou označeny atributy, které tvoří kandidátní klíč (vždy je jenom jeden).

Nyní musíme ověřit, zda nové relace splňují 2NF. Relaci DISPONUJE určitě ano, obsahuje pouze klíčové atributy. Relace ÚČET2 také, funkční závislosti na kandidátním klíči jsou plné. Obě jsou tedy minimálně ve 2NF.

Pro hodnoty relace ÚČET1 z příkladu Příklad 3.18 budou relace vypadat takto:

ÚČET2			
Č_ÚČTU	STAV	POBOČKA	JMĚNÍ
100	100000	Jánská	10000000
130	50000	Palackého	5000000
150	150000	Palackého	5000000

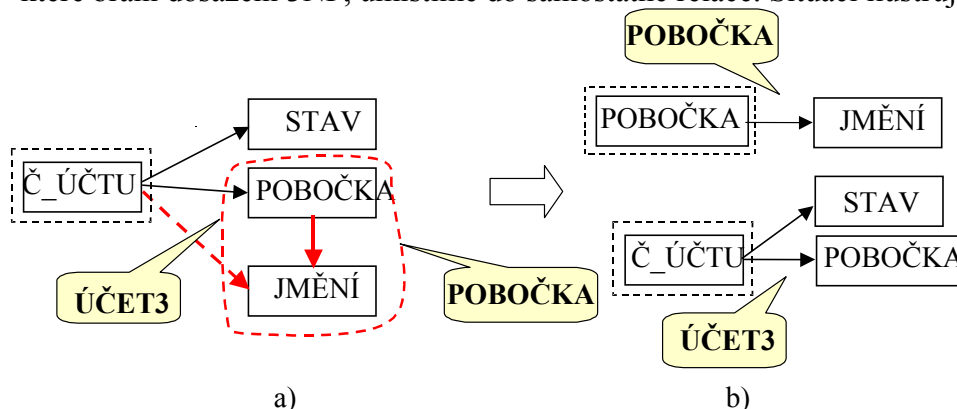
DISPONUJE	
R_ČÍSLO	Č_ÚČTU
600528/0275	100
581015/9327	100
600528/0275	130
450205/3419	150

Je vidět, že přestože je relace ÚČET2 ve 2NF, opakuje se v ní stále informace o jmění poboček. My už víme, že zdrojem problémů je funkční závislost POBOČKA → JMĚNÍ, která způsobuje tranzitivní závislost JMĚNÍ na kandidátním klíči Č_ÚČTU. Tento zdroj problémů odstraňuje *třetí normální forma*.

DEF**Definice 3.9** Třetí normální forma (3NF)

Schéma relace **R** je ve *třetí normální formě*, právě když je ve 2NF a neexistuje žádný neklíčový atribut, který je tranzitivně závislý na některém kandidátním klíči relace **R**.

Relace DISPONUJE splňuje i 3NF, ale relace ÚČET2 ne. Provedeme dekompozici na základě stejné úvahy jako při převodu do 2NF – atributy s funkčními závislostmi, které brání dosažení 3NF, umístíme do samostatné relace. Situaci ilustruje Obr. 3.56.

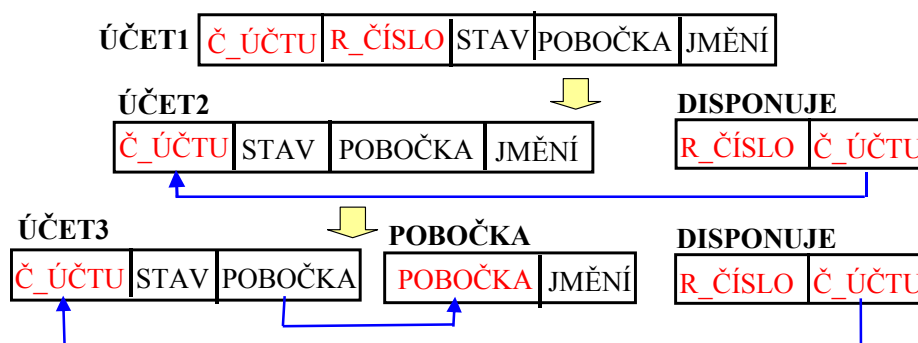


Obr. 3.56 a) Diagram funkčních závislostí relace ÚČET2
b) Diagramy funkčních závislostí relací POBOČKA a ÚČET3, které vzniknou dekompozicí při převodu do 3NF

Z diagramů na Obr. 3.56 b) je zřejmé, že obě relace již 3NF splňují.

ÚČET3			POBOČKA	
Č_ÚČTU	STAV	POBOČKA	POBOČKA	JMĚNÍ
100	100000	Jánská	Jánská	10000000
130	50000	Palackého	Palackého	5000000
150	150000	Palackého		

Postupná normalizace schématu relace ÚČET1 je ještě souhrnně znázorněna na Obr. 3.57.



Obr. 3.57 Postup normalizace relace ÚČET1

Již bylo uvedeno, že první až třetí normální formu zavedl E.F.Codd začátkem 70. let minulého století. Brzy se ukázalo, že 3NF, jak byla definována nemusí dostačovat k odstranění redundance. Ukážeme si to na následujícím příkladě.

Příklad 3.19

Uvažujme relaci DISPONUJE (R_ČÍSLO, Č_KLIENTA, Č_ÚČTU), která uchovává informaci o tom, kteří klienti banky mohou disponovat s kterými účty. Atribut

 $x+y$

$R_ČÍSLO$ je rodné číslo klienta, $Č_KLIENTA$ je identifikační číslo klienta, které každému klientovi banka přiřazuje, a $Č_ÚČTU$ je číslo účtu de kterým může daný klient disponovat. Předpokládáme, že jeden klient může disponovat s několika účty a s jedním účtem může disponovat více klientů. Za těchto předpokladů jsou v relaci dva složené kandidátní klíče - ($R_ČÍSLO$, $Č_ÚČTU$) a ($Č_KLIENTA$, $Č_ÚČTU$). Není tam tedy žádný neklíčový atribut a jedinou zajímavou funkční závislostí je vzájemná závislost atributů $R_ČÍSLO$ a $Č_KLIENTA$. Za předpokladu atomických hodnot atributů splňuje schéma relace 3NF. Přesto se ale v ní může opakovat informace. Pokud klient disponuje několika účty, bude se na každém řádku, který nese informaci o tom, se kterým účtem může disponovat, informace o rodném čísle, které přísluší jeho číslu klienta (a naopak).

Tato redundance vzniká proto, že determinant závislosti $R_ČÍSLO \rightarrow Č_KLIENTA$, resp. $Č_KLIENTA \rightarrow R_ČÍSLO$ není kandidátním klíčem, nýbrž jenom složkou kandidátního klíče. Jde tedy o závislost, která není plná. Oproti námí dříve diskutovaným problémům vyplývajícím z funkčních závislostí, které nejsou plné, je rozdíl pouze v tom, že závislým atributem je atribut klíčový. A to je právě případ, který 2NF ani 3NF nepostihují. Ty si všímají pouze závislostí neklíčových atributů na kandidátních klíčích. Vlivem existence několika kandidátních klíčů v relaci, které navíc mohou být složené a mohou se překrývat může docházet k podobným situacím jako v našem příkladě. To je důvod, proč byla později zavedena Boyce-Coddova normální forma, která zpřísňuje podmínky 3NF.

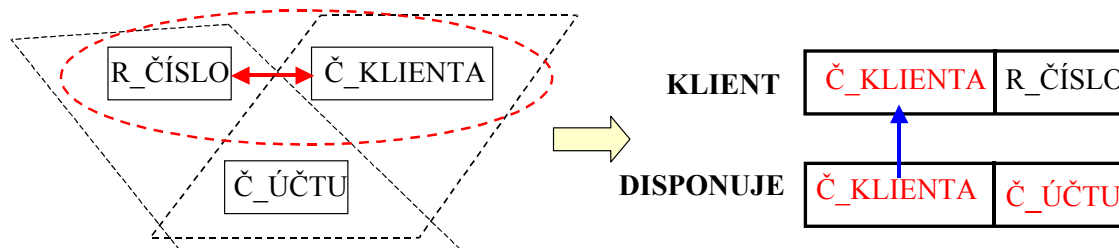
DEF

Definice 3.10 Boyce-Coddova normální forma (BCNF), superklíč

Schéma relace **R** je v *Boyce-Codově normální formě*, právě když pro každou netriviální funkční závislost $X \rightarrow Y$ je X superklíčem. *Superklíčem* rozumíme každou nadmnožinu kandidátního klíče relace **R**.

I když se definice BCNF může zdát složitá, není tomu tak. Zdánlivá komplikovanost vyplývá pouze z toho, že kromě funkčních závislostí, které jsme se domluvili, že budeme kreslit do diagramů funkčních závislostí, protože jsou důležité pro normalizaci, existuje ve striktně matematickém pojetí v relaci řada dalších – triviálních a vyplývajících z námi uvažovaných. Ve skutečnosti definice pouze říká, že všechny závislosti, které nejsou triviální, musí být dány závislostmi na celých kandidátních klíčích.

Pokud při normalizaci zjistíme, že relace je ve 3NF, ale nesplňuje BCNF, postupujeme analogicky jako dosud. Vytvoříme relaci, do které přesuneme atributy, jejichž závislost porušuje BCNF, a doplníme druhou relací, aby obě tvořily bezeztrátovou dekompozici. Postup pro Příklad 3.19 je naznačen na Obr. 3.58.



Obr. 3.58 Normalizace schématu relace DISPONUJE z příkladu Příklad 3.19

Protože atributy $R_ČÍSLO$ a $Č_KLIENTA$ jsou oba kandidátními klíči v relaci **KLIENT**, může být v relaci **DISPONUJE** s atributem $Č_ÚČTU$ kterýkoli z nich.



Zkuste nakreslit diagram funkčních závislostí relace ZKOUŠKY (STUDENT, PŘEDMĚT, POZICE), kde STUDENT je jednoznačné přihlašovací jméno studenta, PŘEDMĚT je jednoznačná zkratka předmětu a pozice je pozice daného studenta v daném předmětu z hlediska získaného počtu bodů a abecedního pořadí. Předpokládejte, že student navštěvuje několik předmětů, v předmětu je řada studentů, předmět si student zapisuje jenom jedenkrát a že nemohou být v daném předmětu dva studenti na stejné pozici.

Jakmile diagram nakreslíte, určete, ve které normální formě schéma relace ZKOUŠKY je, a v případě potřeby znormalizujte tak, aby splňovalo BCNF. Pokud jste postupovali správně, měli byste být schopni zodpovědět otázku, zda překrývající se kandidátní klíče vždy způsobují, že relace splňuje 3NF, ale ne BCNF.

Zatím jsme se zabývali otázkou určení normální formy jedné relace. Jak se určí normální forma návrhu tvořeného řadou tabulek? Výsledná normální forma je určena nejnižší normální formou schémat tabulek, které návrh tvoří. Jestliže například budou schémata všech tabulek vyhovovat BCNF s výjimkou jedné, která bude pouze ve 2NF, pak 2NF je výslednou normální formou celého návrhu.



Definice 3.11 Normální forma návrhu

Návrh databáze je v *n-té normální formě*, je-li schéma každé jeho relace alespoň v *n-té normální formě*.

4NF a 5NF se zabývat nebudeme. Zmíníme se pouze o tom, k jakému výsledku by vedla normalizace relace PŘEDMĚT se schématem PŘEDMĚT (NÁZEV, UČITEL, UČEBNICE) z příkladu Příklad 3.17 na str. 95. Připomínáme, že v relaci jsou uloženy informace o učitelích, kteří učí předměty a učebnicích, které se v předmětech používají. Podstatným integritním omezením bylo, že v daném předmětu se používají stejné učebnice bez ohledu na učitele. Ukázali jsme si, že za tohoto předpokladu existují v relaci vícehodnotové závislosti $NÁZEV \twoheadrightarrow UČEBNICE$ a $NÁZEV \twoheadrightarrow UČEBNICE$, jejichž zajištění je poměrně složité.

Pokud bychom si vysvětlovali 4NF, viděli bychom, že problémy relace PŘEDMĚT vyplývají z toho, že nesplňuje 4NF. Je pouze v BCNF. Řešením by byla opět dekompozice, která by v tomto případě měla podobu dvou relací UČITELÉ_PŘEDMĚTU se schématem UČITELÉ_PŘEDMĚTU (NÁZEV, UČITEL) a UČEBNICE_PŘEDMĚTU se schématem UČEBNICE_PŘEDMĚTU (NÁZEV, UČEBNICE). Tady už by bylo všechno v pořádku. Jedna relace uchovává informace o učitelích předmětů a druhá o učebnicích, které se v předmětech používají. Vzájemně nezávislé informace, které byly původně v jedné relaci se oddělily.



V této kapitole jsme se seznámili s teorií, která vznikla na podporu logického návrhu relačních databází. Postup návrhu s využitím této teorie se nazývá *normalizace* a jeho cílem je dosažení návrhu, který je v dostatečně vysoké tzv. *normální formě*. Normální formy určitým způsobem odráží kvalitu návrhu. Existuje hierarchie normálních forem, která je tvořena první až pátou normální formou a Boyce-Coddovou normální formou, která se nachází mezi třetí a čtvrtou normální formou. *První normální forma* požaduje, aby byl splněn jeden ze základních předpokladů, ze kterého vychází relační model dat – aby jednoduché atributy nabývaly atomických hodnot. Druhá normální a třetí normální forma definují požadavky na schéma relace z pohledu funkčních závislostí mezi atributy relace. Funkční závislost vyjadřuje skutečnost, že z významu atributů vyplývá, že hodnota jednoho určuje jednoznačně hodnotu druhého. *Druhá*

normální forma vyžaduje plné funkční závislosti neklíčových atributů na kandidátních klících. *Třetí normální forma* vyžaduje neexistenci tranzitivních závislostí neklíčových atributů na kandidátních klících relace, *Boyce-Coddova normální forma* zpřísňuje podmínku třetí normální formy tím, že se neomezuje jen na závislosti neklíčových atributů, ale vyžaduje odstranění I závislostí mezi klíčovými, které způsobují redundanci. Teprve dosažením BCNF je zajištěno odstranění redundance vlivem funkčních závislostí. *Čtvrtá normální forma* definuje požadavky z hlediska dalšího typu závislostí mezi atributy – vícehodnotových závislostí (multizávislostí) a *pátá normální forma* z hlediska závislostí na spojení.

Samotná normalizace probíhá tak, že se ověřuje splnění podmínek jednotlivých normálních forem počínaje 1NF, dokud se nedosáhne požadované normální formy. Pokud relace dané normální formě nevyhovuje, provede se dekompozice s cílem odstranění závislostí, které brání splnění dané normální formy. Dekompozice musí být bezetrátová a měla by zachovávat závislosti.

Výsledná normální forma celého návrhu je určena nejnížší normální formou jednotlivých schémat. Z hlediska logického návrhu relační databáze se za dobrý návrh považuje takový, který je v BCNF nebo alespoň ve 3NF.

Přestože využitím normalizace je v principu možné provést kompletní návrh, v praxi se používá spíše při kontrole a případném vylepšení návrhu, který vznikne z konceptuálního modelu v podobě ER diagramu nebo diagramu tříd.



Informace k funkčním závislostem, normálním formám a normalizaci můžete najít v [Pok98] na stranách 35 až 38 a 201 až 204. V učebnicích [Sil05] a [Dat03] je teorie závislostí, normální formy a normalizace diskutovány mnohem podrobněji a exaktněji, než jsme to udělali my. V [Sil05] najdete příslušnou část na stranách 263 až 310.



Cvičení:

- C3.33** Vysvětlíte pojmy funkční závislost, plná funkční závislost a tranzitivní funkční závislost.
- C3.34** Vysvětlíte, proč vzniká redundance při funkční závislosti neklíčového atributu na kandidátních klících, když tato závislost není plná.
- C3.35** Vysvětlíte, proč vzniká redundance při tranzitivní funkční závislosti neklíčového atributu na kandidátních klících.
- C3.36** Vysvětlíte, jak může vzniknout redundance při funkčních závislostech mezi klíčovými atributy.
- C3.37** Vysvětlíte, co jsou to normální formy schématu relace a které to jsou.
- C3.38** Vysvětlíte, co znamená, že normální formy tvoří hierarchii.
- C3.39** Vysvětlíte vztah normálních forem a závislostí mezi atributy relace.
- C3.40** Vysvětlíte první normální formu a jaké výhody přináší její splnění.
- C3.41** Vysvětlíte druhou normální formu a jaké výhody přináší její splnění.
- C3.42** Vysvětlíte třetí normální formu a jaké výhody přináší její splnění.
- C3.43** Vysvětlíte Boyce-Coddovu normální formu a jaké výhody přináší její splnění.
- C3.44** Vysvětlíte pojem normalizace schématu relace a jak probíhá.
- C3.45** Vysvětlíte, co rozumíme dekompozicí schématu relace.
- C3.46** Vysvětlíte, jaké vlastnosti musí mít dekompozice, kterou používáme při

normalizaci, a co tyto vlastnosti znamenají.

- C3.47** Vysvětlete, jaké normální formy bychom se při logickém návrhu měli snažit dosáhnout a proč.
- C3.48** Vysvětlete tvrzení: Je-li A množina atributů relace R a pro atribut $A_1 \in A$ platí funkční závislost $A_1 \rightarrow A$, pak A_1 je kandidátním klíčem v R .
- C3.49** Vysvětlete tvrzení: Pokud relace splňuje 1NF a všechny její kandidátní klíče jsou jednoduché, pak je minimálně ve 2NF.
- C3.50** Vysvětlete tvrzení: Pokud relace splňuje 1NF a obsahuje jen klíčové atributy, pak je minimálně ve 3NF.
- C3.51** Uvažujte relaci TITUL v prostředí systému řízení báze dat, který podporuje zanořené relace, se schématem TITUL(ISBN, název, setof(autor), vydavatel, rok, setof(kl_slovo)), kde
ISBN - jednoznačné číslo, přiřazené titulu knihy,
setof(autor) - seznam autorů, jehož prvky jsou jména všech autorů (tj. autor je samostatný atribut, se kterým musí být možné pracovat),
vydavatel - název vydavatelství,
rok - rok vydání,
setof(kl_slovo) je seznam klíčových slov, jehož prvky jsou klíčová slova, tj. slova, charakterizující zaměření knihy (tj. kl_slovo je samostatný atribut, se kterým musí být možné pracovat).
- Ve které normální formě relace TITUL je? Zdůvodněte.
 - Jak by vypadal výchozí tvar relace TITUL pro návrh relační databáze? Nakreslete záhlaví. Necht' je to relace TITUL1.
 - Určete množinu kandidátních klíčů této relace.
 - Nakreslete diagram funkčních závislostí atributů relace TITUL1. V diagramu znázorněte všechny závislosti, důležité pro normalizaci. Určete, ve které normální formě relace je. Zdůvodněte.
 - Na základě diagramu proveďte postupnou dekompozici (tj. musí být zřejmý postup přechodu k vyšší normální formě) na schéma, které bude alespoň v BCNF.
- C3.52** Při analýze systému, který dosud používal zákazník pro účely registrace řešených projektů, jste dostali formulář, který obsahoval základní údaje o projektu a zodpovědném řešiteli a seznam dalších řešitelů. Jeho analýzou jste zjistili, že v databázi bude potřeba ukládat hodnoty následujících atributů:
 projekt# (číslo projektu), název (název projektu), částka (částka rozpočtu), z_resitel# (os. číslo zodp. řešitele), z_resitel (jméno zodp. řešitele), z_odd# (číslo oddělení zodp. řešitele), z_odd (název oddělení zodp. řešitele), resitel# (os. číslo řešitele), resitel (jméno řešitele), odd# (číslo oddělení řešitele), odd (název oddělení řešitele).
 Dále jste zjistili, že jedna osoba může být zodpovědným řešitelem, případně řešitelem více projektů. Každý projekt má jednoho zodpovědného řešitele.
- Určete kandidátní klíče univerzální relace, tvořené všemi výše uvedenými atributy.
 - Nakreslete diagram funkčních závislostí atributů. V diagramu znázorněte všechny závislosti, důležité pro dekompozici. V případě potřeby můžete pro přehlednost nakreslit závislosti do několika dílčích diagramů.
 - Ve které normální formě univerzální relace je? Zdůvodněte.
 - Na základě b) proveďte dekompozici na schéma, které bude alespoň v

BCNF.

C3.53 Při analýze systému pro přihlašování studentů do kursů jste zjistili nutnost uchovávat hodnoty těchto atributů: *os_číslo_u* (jednoznačné číslo garanta kursu), *jméno_u* (jméno garanta), *ústav* (jednoznačná zkratka ústavu garanta), *os_číslo_s* (jednoznačné číslo studenta), *jméno_s* (jméno studenta), *adresa* (adresa bydliště studenta), *zkratka* (jednoznačná zkratka předmětu), *název* (název předmětu), *typ* (typ předmětu - např. povinný), *datum* (datum přihlášení do předmětu). Každý předmět má jednoho garanta, jedna osoba může být garantem více předmětů.

- Určete počet kandidátních klíčů univerzální relace a vyjmenujte je.
- Nakreslete diagram funkčních závislostí, ve kterém budou uvedeny všechny funkční závislosti, podstatné z hlediska normalizace.
- Určete normální formu univerzální relace a zdůvodněte.
- Proveďte normalizaci tak, aby výsledné schéma bylo alespoň v BCNF. Musí být zřejmý postup.

C3.54 Uvažujte relaci *R* se schématem *R*(*A,B,C,D,E,F*). Všechny atributy jsou jednoduché a nabývají atomických hodnot. Na množině atributů platí následující funkční závislosti:

$$(A,B) \rightarrow (C,D,E,F), C \rightarrow D, D \rightarrow E, B \rightarrow F$$

- Určete počet kandidátních klíčů univerzální relace a vyjmenujte je.
- Nakreslete diagram funkčních závislostí, ve kterém budou uvedeny všechny funkční závislosti, podstatné z hlediska normalizace.
- Určete normální formu univerzální relace a zdůvodněte.
- Proveďte normalizaci tak, aby výsledné schéma bylo alespoň v BCNF. Musí být zřejmý postup.

C3.55 Uvažujte databázi, tvořenou relacemi *R1* a *R2* se schématy *R1*(*A, B, C, D*) a *R2*(*U, V, X, Y*). Kandidátními klíči relace *R1* jsou atributy (*A, B*) a (*A, C*), relace *R2* atribut *X*. Všechny jednoduché atributy jsou atomické. V relaci *R1* jsou definovány tyto funkční závislosti:

$$A \rightarrow D \text{ a } U \rightarrow V.$$

- Nakreslete diagramy funkčních závislostí relací *R1* a *R2*.
- Ve které normální relace *R1* a *R2* jsou? Zdůvodněte.
- Proveďte dekompozici na schéma, které bude alespoň v BCNF

Test

T1. Plná funkční závislost znamená, že:

- každý neklíčový atribut je funkčně závislý na každém kandidátním klíči relace
- daný atribut je funkčně závislý na jiném atributu jako celku a ne jen na některé jeho složce

T2. Cílem normalizace schématu databáze je především:

- úspora místa pro uložení dat v databázi
- zrychlení přístupu k datům v databázi

T3. Pro bezztrátovou dekompozici relace *R* s množinou atributů *A* musí platit, že:

- přirozeným spojením jejích relací vznikne relace s tělem obsahujícím všechny n-tice relace *R*, případně některé navíc, ale žádná se nesmí ztratit
- přirozeným spojením jejích relací vznikne relace se stejným tělem jako má *R*

T4. První normální forma odstraňuje:

- triviální funkční závislosti

b) složené a vícehodnotové atributy

T5. Jestliže relace nemá žádný složený kandidátní klíč a hodnoty atributů jsou atomické, pak je určitě

a) nejméně ve druhé normální formě

b) nejméně v Boyce-Coddově formě

4. Organizace dat v databázi na fyzické úrovni



Cíl: V této kapitole se seznámíte se základy uspořádání dat v databázi na fyzické (interní úrovni), včetně nejpoužívanějších přístupových metod k datům v databázi.

Anotace: Databázový soubor, záznam proměnné délky, záznam pevné délky, primární index, sekundární index, hustý index, řídký index, víceúrovňový index, B⁺-strom, bitmapový index, statické hašování, dynamické hašování, shlukování, fyzický návrh.

Prerekvizitní znalosti: V této kapitole se předpokládá znalost základů systému ovládání souborů jako součástí operačního systému z předmětu Operační systémy (IOS) a znalost základních principů vyhledávacích algoritmů z předmětu Algoritmy (IAL). Předpokládají se rovněž základní znalosti vlastností paměťových médií a příslušných vstupně výstupních zařízení. Ty byste měli mít ze střední školy a rozšířené v předmětu Návrh počítačových systémů (INP). Ze znalostí získaných v předmětu Databázové systémy navážeme hlavně na logický návrh databáze a jazyk SQL.



Odhad doby studia: 10 hodin.

4.1. Úvod

Přestože systém řízení báze umožňuje vysokou úroveň abstrakce pohledu na data, konečnou podobou dat, jak jsou uložena v databázi, je posloupnost bitů. Na úvodní přednášce jsme si uvedli tzv. tříúrovňovou ANSI/SPARC architekturu pohledu na data a řekli jsme si, že nejnižší úrovní je *úroveň fyzická*, označovaná také jako interní. Na této úrovni jsou vidět data tak, jak jsou skutečně uložena. Podobně jako definujeme organizaci dat v databázi na logické úrovni databázovým (logickým) schématem, popisujeme organizaci dat na fyzické úrovni *fyzickým schématem*.

Z úvodní přednášky také víme, že databázová data jsou data perzistentní. Znamená to, že musí být dostupná i po ukončení aplikace, která je vytvořila a dokonce i po případném restartu počítače, na kterém jsou uložena. Z tohoto charakteru dat plyne, že budou ukládána na nějakém energeticky nezávislém médiu (ve smyslu uchování informace i při přerušení napájení). V dnešní době je typickým paměťovým médiem, kde jsou databázová data uložena, magnetický disk. Nejsou ale médiem jediným. Jak uvidíme, využívá SŘBD při své činnosti vyrovnávací paměti, která se nachází ve vnitřní paměti (polovodičové, energeticky závislé) počítače. Toto není samozřejmě žádná specialita databázových systémů, protože i samotné operační systémy využívají pro přístup k datům na disku vyrovnávacích pamětí. Uvidíme ale, že SŘBD má některé speciální požadavky na správu vyrovnávací paměti, které si vynucují, aby pracoval se svou vlastní vyrovnávací pamětí.

Kromě dat v na disku a ve vyrovnávací paměti je typicky obsah databáze zálohován a u rozsáhlých databází také archivován. V obou případech jde o uložení obsahu databáze, nebo její části pro účely zálohování či archivace. Zpravidla jsou zálohování a archivace chápány jako totéž. Pokud bychom chtěli nějak odlišit případ zálohování, kdy současně uvolňujeme prostor v databázi, tj. zálohovaná data jsou z databáze smazána, pak bychom ho mohli nazvat archivací. Příkladem by mohla být archivace údajů vztahujících se k výuce na fakultě před deseti a více lety. Zálohovacím či archivačním médiem může být opět magnetický disk (jiný fyzický svazek), magnetická páska, optický disk apod.

Dříve než se začneme podrobněji věnovat organizaci dat na fyzické úrovni, podívejme se na to, jakým způsobem z hlediska přístupu k datům probíhá zpracování dotazu.

Příklad 4.1

x+y

Uvažujme náš ilustrační příklad databáze informačního systému spořitelny a předpokládejme potřebu zodpovědět dotaz „Najdi klienta s rodným číslem 561021/3117“. Pro naše schéma databáze s tabulkou Klient (r_cislo, jmeno, ulice, mesto) by to byl dotaz nad touto tabulkou, který by měl v SQL tvar:

```
SELECT *  
FROM Klient  
WHERE r_cislo = '561021/3117'
```

Postup získání výsledku tohoto dotazu je uveden na **Obr. 4.1**.

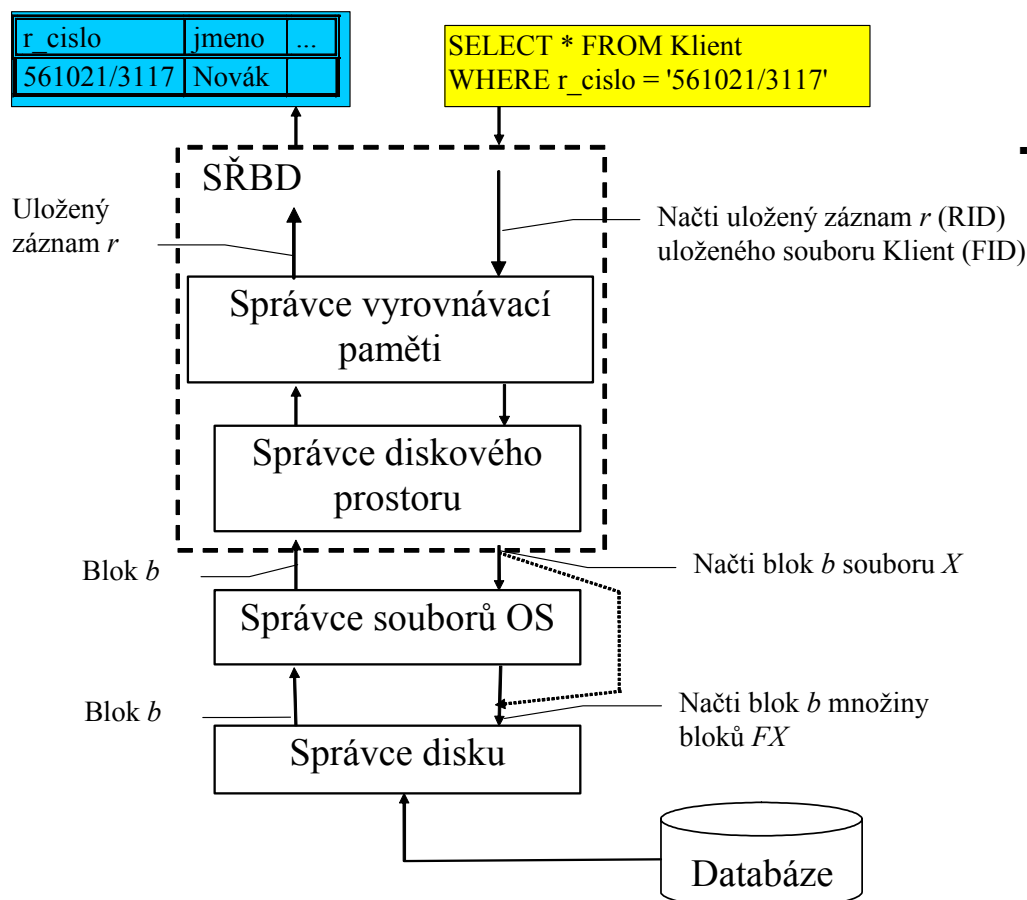


Pro jeho pochopení je důležité si uvědomit následující tři skutečnosti, které byste již měli znát nebo byly uvedeny výše:

4. Fyzický prostor na disku je rozdělen do tzv. *diskových bloků* (označovaných také jako stránky).
5. Jednou vstupně výstupní operací se načte nejméně jeden diskový blok, tj. nikdy se nenačte například jenom jeden z několika záznamů uložených v bloku.
6. SŘBD využívá při přístupu k disku vyrovnávací paměť. Pokud jsou požadovaná data ve vyrovnávací paměti, není třeba je číst z disku.

Nejprve SŘBD zkontroluje správnost syntaxe a sémantiky příkazu a provede řadu dalších kontrol. Následně převede příkaz do vnitřní reprezentace a jedna jeho komponenta, zvaná *optimalizátor*, rozhodne, jakou posloupností elementárních operací, které zhruba odpovídají operacím relační algebry, bude nejvýhodnější požadovaný výsledek získat. Optimalizátor také rozhodne o přístupové metodě, kterou se bude k uloženým záznamům přistupovat. Nejjednodušší, ale u rozsáhlých tabulek neefektivní je sekvenční prohledání, při kterém se postupně čtou všechny záznamy odpovídající řádkům uložené tabulky. Pro rozsáhlé tabulky využíváme efektivnější *přístupové metody*, mezi které patří zejména *indexování* a *hašování*. Podrobněji se s nimi seznámíme v kapitole 4.4. Budeme předpokládat, že pokud existuje v tabulce Klient nejvýše jeden klient s uvedeným rodným číslem, což předpokládáme, vrátí příslušná přístupová metoda identifikátor záznamu, který odpovídá příslušnému řádku tabulky Klient. Tento identifikátor se označuje jako *identifikátor řádku* (RID nebo také ROWID – Row Identifier). SŘBD ví, ve kterém souboru (kolekci diskových bloků) jsou záznamy tabulky Klient uloženy. Opět můžeme předpokládat, že každý takový soubor má nějaký identifikátor, označený v obrázku jako FID (File Identifier). Jako první je požadavek na zpřístupnění záznamu postoupen *správci vyrovnávací paměti*.

Prostor vyrovnávací paměti je rozdělen na stránky. Jedna stránka obsahuje jeden diskový blok s daty načtenými z disku. Pokud správce vyrovnávací paměti zjistí, že požadovaný záznam je k dispozici ve vyrovnávací paměti, zpřístupní ho komponentám SŘBD, které provedou operace, které je nutné provést před vrácením výsledku dotazu.



Obr. 4.1 Postup získání dotazu z příkladu Příklad 4.1

Jestliže však požadovaný záznam ve vyrovnávací paměti není, je potřeba načíst příslušný blok z disku. SŘBD musí vědět nejen to, ve které kolekci diskových bloků jsou uloženy záznamy dané tabulky, ale musí také vědět, ve kterém souboru operačního systému se tyto bloky nachází. Uvidíme, že současné SŘBD zpravidla již nepoužívají koncepci běžnou u databázových systémů s architekturou PC file server, tj. záznamy tabulky vždy v samostatném souboru operačního systému. Komponenta SŘBD, která „se vyzná“ v tom, co v kterém souboru na disku je, je v obrázku označena jako *správce diskového prostoru*.

Ten buď již sám provede načtení požadovaného bloku využitím nízkourovňových služeb operačního systému pro práci s diskem nebo využije služeb správy souborů operačního systému. V obou případech se načte požadovaný blok do vyrovnávací paměti spravované SŘBD. A správce jej zpřístupní komponentám SŘBD, které zajistí zbývající operace před vrácením výsledku dotazu.



Zmíněný identifikátor řádku (RID, resp. ROWID) je něco jiného, než primární klíč tabulky. Zatímco primární klíč tabulky identifikuje řádek logicky, je jedním ze sloupců tabulky, identifikátor řádku identifikuje řádek fyzicky. Jeho hodnota určitým způsobem kóduje adresu záznamu v databázi na disku. Můžeme říci, že identifikátor řádku identifikuje soubor (kolekci bloků), blok v něm a řádek v bloku. Není to žádný

další sloupec tabulky. Například u databázového serveru Oracle má každá tabulka tzv. pseudosloupec ROWID, jehož hodnota udává adresu řádku. Sestává z několika částí, které identifikují tzv. segment, soubor v rámci tzv. tabulkového prostoru, blok a řádek v něm. Terminologii používanou pro organizaci na fyzické úrovni u databázového serveru Oracle uvedeme jako konkrétní příklad organizace později.



Zamyslete se nad tím, odkud získá SŘBD informace o tom, kde je která tabulka uložena, jaké jsou dostupné přístupové metody a další informace týkající se fyzického schématu databáze.

Již jsme si uvedli, že paměťová média, na nichž se mohou nacházet databázová data, tvoří hierarchii. Tvoří ji vnitřní paměť počítače, ve které je vyrovnávací paměť, sekundární paměť - magnetický disk, na kterém je vlastní databáze, a terciální paměť – např. magnetická páska, která slouží k zálohování a archivaci. Uvedené typy pamětí se vyznačují řádově odlišnými přístupovými dobami. Zálohování není z tohoto pohledu kritické, protože probíhá v době, kdy je databázový systém mimo provoz (v režimu off-line) nebo za provozu (v režimu on-line), ale na pozadí probíhajících procesů. Z hlediska výkonnosti databázového systému a rychlosti provádění databázových operací je úzkým místem přístup k disku. Proto je důležité navrhnout fyzické schéma databáze a využívat možností SŘBD takovým způsobem, aby byl nutný počet diskových operací minimalizován.



Hlavním cílem, který bychom měli sledovat při fyzickém návrhu databáze, je minimalizace potřebných diskových operací. Toto je i jedno ze základních kritérií, které používá optimalizátor při optimalizaci provádění databázových operací.

V následujících podkapitolách se nejprve podíváme, jak jsou uložena data na disku a to jednak v blocích, jednak bloky v souborech. Největší část této kapitoly bude věnována již zmíněným přístupovým metodám – indexování a hašování. V závěru potom shrneme kritéria a doporučení, která bychom měli respektovat při fyzickém návrhu databáze.

4.2. Databázové soubory

Víme, že v relační databázi jsou na logické úrovni data organizovaná do tabulek. Budeme se teď tedy zabývat tím, jak jsou tato data fyzicky uložena v souborech operačního systému na disku.

Databázovým souborem budeme rozumět soubor, ve kterém jsou uložena na disku databázová data. Z předmětu operační systémy víte, že soubor je logicky organizován jako posloupnost záznamů. Fyzicky jsou tyto záznamy uloženy do bloků na disku. Velikost diskových bloků je pevná, v řádu kilobytů, typicky 1 až 4kB. Je dána fyzickými vlastnostmi disku a zpravidla i operačním systémem.



Je na místě poznamenat, že pojem soubor, jak ho teď budeme chápat, nemusí nutně znamenat jeden soubor operačního systému. Už v předchozí podkapitole jsme naznačili, že současné SŘBD ukládají třeba celou databázi obsahující schémata s řadou tabulek řady uživatelů v jednom souboru operačního systému. Souborem budeme proto rozumět kolekci diskových bloků, které tvoří logický celek, ale fyzicky mohou tvořit pouze část jednoho souboru operačního systému nebo se naopak mohou rozprostírat napříč několika soubory operačního systému. Pokud v dalším budeme

mluvit o souboru a nebude explicitně uvedeno, že myslíme soubor operačního systému, budeme mít na mysli databázový soubor.

V této kapitole se nejprve podíváme, jak jsou reprezentovány záznamy v souborech. Jde o to, že na jedné straně diskové bloky mají pevnou velikost, na straně druhé tabulky mohou obsahovat různý počet sloupců různých datových typů, což znamená, že velikost záznamů nesoucích data různých tabulek bude obecně různá. Navíc víme, že existují datové typy pro hodnoty proměnné délky, např. VARCHAR, které mohou způsobit, že i záznamy pro řádky jedné tabulky mohou mít různou délku. Z toho vyplývá, že potřebujeme ukládat záznamy proměnné délky do bloků pevné délky. Lze použít dva přístupy. Buď lze ukládat vždy do jednoho souboru záznamy stejné délky nebo uzpůsobit strukturu souboru tak, aby bylo možné do bloku ukládat záznamy různé délky. Naznačíme si možná řešení pro oba přístupy.

Příklad 4.2

Uvažujme tabulku Transakce z našeho příkladu databáze jednoduchého informačního systému banky. Předpokládejme, že byla vytvořena SQL příkazem:

```
CREATE TABLE Transakce (
    c_uctu    INTEGER NOT NULL,
    c_transakce INTEGER NOT NULL,
    datum    DATE,
    castka    DECIMAL(7,2),
    PRIMARY KEY (c_uctu, c_transakce),
    FOREIGN KEY (c_uctu) REFERENCES Ucet ON DELETE CASCADE
)
```

Záznam nesoucí hodnoty jednoho řádku by mohl být definován v jazyce C takto:

```
typedef struct ttransakce
{
    int c_uctu;
    int c_transakce;
    TDatum datum;
    char castka[10];
} TTransakce;
```

Pokud budeme předpokládat, že typ TDatum pro uložení data má pevnou délku, budou mít i záznamy typu TTransakce pevnou délku. Obrázek Obr. 4.2 a) ukazuje soubor se záznamy o šesti transakcích.

T ₁
T ₂
T ₃
T ₄
T ₅
T ₆

a)

Záhlaví	
	T ₁
	T ₃
	T ₄
	T ₆

b)

Obr. 4.2 Soubor se záznamy pevné délky: a) Soubor se záznamy o 6 transakcích, b) Soubor se záhlavím po zrušení záznamů o transakcích T₂ a T₅

Předpokládáme, že záznamy jsou uloženy bezprostředně po sobě. Při této organizaci existují dva problémy:

1. Při zrušení záznamu je třeba soubor reorganizovat nebo označit záznam za „zrušený“ a umožnit vložit na takto uvolněné místo záznam nový.
2. Pokud velikost bloku není celočíselným násobkem velikosti záznamu, mohou být některé záznamy uloženy ve dvou blocích.

Prostor uvolněný zrušeným záznamem by sice bylo možné zaplnit posuvem všech ostatních záznamů nebo vložení posledního z nich. V obou případech by taková reorganizace vyžadovala další přístupy k disku. Protože vkládání nových záznamů obecně převyšuje nad rušením (databáze má tendenci časem se zvětšovat), je přijatelné ponechat uvolněný prostor neobsazený a obsadit ho až při vkládání nového záznamu. Pro tyto účely je třeba mít k dispozici informaci o volném prostoru, např. formou seznamu, jehož počáteční adresa je uložena v pomocné struktuře, kterou označíme jako záhlaví souboru (file header). Situaci ilustruje Obr. 4.2 b).

Nyní se podívejme, jak může vypadat organizace v případě záznamů proměnné délky. Potřeba ukládat do souboru záznamy proměnné délky vzniká několika způsoby:

1. Do jednoho souboru ukládáme záznamy několika typů. To je případ tzv. *shlukování (clustering)*, kterému se budeme podrobněji věnovat později. Jeho cílem je ukládat do bloku záznamy, které spolu logicky souvisí a jsou proto často zpracovávány společně. V případě naší databáze banky bychom například mohli shlukovat záznamy účtu a transakcí s daným účtem. Jednou diskovou operací by se potom dal zajistit přístup jak k záznamu o účtu tak k záznamům všech nebo alespoň části jeho transakcí.
2. Do souboru ukládáme záznamy, jejichž některé pole může mít proměnnou délku. Příkladem mohou být datové typy SQL VARCHAR, BIT VARYING, BLOB nebo CLOB.
3. Do souboru ukládáme záznamy, jejichž některé pole nese hodnotu vícehodnotového atributu. Víme, že v relačních systémech takový stav nastat nemůže. Může ale nastat například u systémů objektově relačních, které připouští nenormalizované tabulky. Záznam potom může obsahovat např. pole hodnot.

Příklad 4.3

x+y

Uvažujme tabulku Ucet z našeho příkladu databáze jednoduchého informačního systému banky. Předpokládejme, že byla vytvořena SQL příkazem:

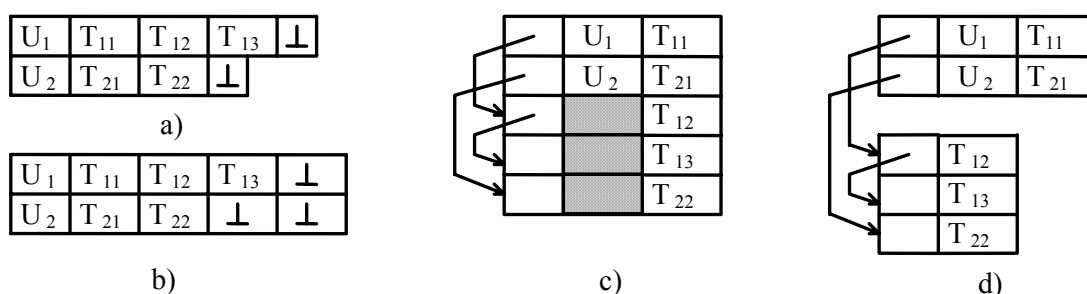
```
CREATE TABLE UCET (
    c_uctu INTEGER NOT NULL,
    stav DECIMAL(10,2),
    r_cislo CHAR(11) NOT NULL,
    pobočka VARCHAR(20),
    PRIMARY KEY (c_uctu),
    FOREIGN KEY (r_cislo) REFERENCES Klient ON DELETE
        CASCADE,
    FOREIGN KEY (pobočka) REFERENCES Pobočka
)
```

Záznam nesoucí hodnoty jednoho řádku by mohl být definován v jazyce C takto:

```
typedef struct tucet
{
    int c_uctu;
    char stav[13];
    char r_cislo[12];
    char pobočka[21];
} TUcet;
```

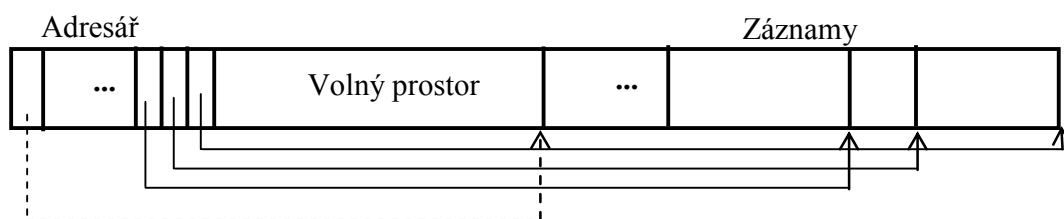
Předpokládejme, že budeme shlukovat záznamy účtů a jejich transakcí. Do jednoho souboru budeme tedy ukládat záznamy dvojího typu – TUcet a TTransakce z příkladu Příklad 4.1 a to takovým způsobem, že vždy bude uložen záznam typu TUcet a za ním budou následovat záznamy typu TTransakce.

Jedním ze způsobů, jak ukládat záznamy proměnné délky, je *reprezentace řetězcem bytů*. Jednoduchým způsobem implementace záznamů proměnné délky je potom ukončení řetězce speciálním znakem „konec záznamu“ – viz Obr. 4.3 a), podobně jako jsou ukončeny znakové řetězce v jazyce C. Nevýhodou tohoto způsobu implementace je problematické znovupoužití uvolněného prostoru (tzv. fragmentace) a žádný prostor pro zvětšování záznamů. Například pokud by v Obr. 4.3 a) byla provedena další transakce s účtem U_1 , musel by se alokovat prostor větší, do něho přesunout původní hodnotu rozšířenou o novou transakci a původní prostor uvolnit. Takový přesun by byl obecně drahý, zejména proto, že by se změnilo umístění záznamu a tedy by bylo potřeba změnit hodnoty všech ukazatelů na tento záznam (RID), které se v databázi vyskytují na různých místech, například v indexech.



Obr. 4.3 Implementace záznamů proměnné délky: a) jako řetězec bytů
b) záznamy pevné délky s rezervovaným prostorem,
c) záznamy pevné délky tvořící seznam, d) pomocí základního bloku a
přetokových bloků

V praxi se proto tato základní reprezentace nepoužívá, ale používá se modifikace nazývaná *struktura s adresářem bloku (slotted-page)*. Zde každý diskový blok obsahuje záhlaví (adresář), ve kterém je uložena informace o počtu záznamů v bloku, konec volného prostoru a pole prvků, jehož každý prvek udává pozici jednoho záznamu v bloku (relativně ke konci bloku) a délku záznamu – viz Obr. 4.4



Obr. 4.4 Struktura s adresářem bloku

Aktuální záznamy jsou uloženy spojitě od konce bloku. Prostor mezi adresářem a koncem posledního záznamu představuje volný prostor. Při vložení nového záznamu je alokován potřebný prostor na konci volného prostoru a do pole adresáře je přidán nový prvek udávající pozici záznamu a délku. Při zrušení záznamu je prostor, který zabíral, uvolněn, odpovídající prvek pole v adresáři je označen jako zrušený (např. nastavením délky na zápornou hodnotu) a všechny záznamy nacházející se před takto uvolněným prostorem jsou posunuty tak, aby volný prostor bloku opět tvořil souvislou oblast. Nakonec se aktualizuje pozice konce volného prostoru v adresáři. Zvětšování a zmenšování záznamů pak probíhá jako kombinace vložení a zrušení. Cena přesunu záznamů v bloku není vysoká, protože velikost přesouvaných dat není velká, je omezená velikostí bloku. Ta je typicky 2 až 4kB.

Struktura s adresářem bloku vyžaduje, aby ukazatele v databázi neukazovaly přímo na záznam, ale na odpovídající prvek pole adresáře. Teprve jeho hodnota definitivně určuje polohu záznamu. Jde tedy o určitou formu nepřímé adresace. Ta zajišťuje, že hodnota ukazatele zůstává konstantní při přesunech záznamů v rámci bloku.

Další možnosti implementace záznamů proměnné délky převádí tyto záznamy na záznamy pevné délky. Jednou z možností je použití rezervovaného prostoru. Pokud je známa maximální možná velikost záznamu, lze alokovat maximální potřebný prostor a využívat jenom tolik, kolik je aktuálně třeba – viz Obr. 4.3 b). Tento způsob lze například využít u záznamů obsahujících položky datových typů proměnné délky, u kterých se udává maximální možná délka. V SQL jsou to například typy VARCHAR, NVARCHAR a BIT VARYING.

Jinou možností využívající záznamů pevné délky je *struktura využívající seznamu* – viz Obr. 4.3 c). Myšlenka spočívá v tom, že se vytvoří typ záznamu pevné délky, u nichž ne všechna pole musí být vždy využita. Například v Obr. 4.3 c) by takový záznam obsahoval pole pro uložení informace o účtu a pole pro uložení informace o jedné transakci. Pro každý účet by potom existoval vždycky jeden základní záznam s informací o účtu a jedné (nebo několika, ale pevném počtu) transakci. Při zaplnění prostoru pro informaci o transakci by se při potřebě vložit informaci o další transakci vytvořil nový záznam stejného typu, u něhož by ale bylo využito jen pole pro informaci o transakci. Tyto záznamy můžeme chápat jako „přetokovou oblast“, kdy nestačí prostor pro informace o transakcích v základním záznamu. Záznamy týkající se jednoho účtu by tvořily seznam, jak je vidět z obrázku.

Nevýhodou této varianty je nevyužitý prostor záznamů, kde není informace o účtu. Ten by mohl být významný, když uvážíme, že k jednomu účtu typicky existuje řada transakcí. Odstranit tuto nevýhodu se snaží implementace, která využívá dvou typů záznamů pevné délky – základního a přetokového s tím, že do jednoho bloku souboru jsou ukládány vždy záznamy téhož typu – viz Obr. 4.3 d). V našem příkladě by základní záznam obsahoval informaci o účtu a jedné transakci a přetokový pouze informaci o transakci. Záznamy byly ukládány do bloků, které označujeme podle typu záznamu jako *základní (anchor)* a *přetokový (overflow)*. Výsledkem je, že záznamy v jednom bloku mají vždy stejnou délku, i když ne všechny záznamy daného souboru mají stejnou délku.

Dosud jsme se zabývali tím, jak jsou záznamy reprezentovány v souborech. Nyní se podíváme, jak jsou záznamy v souborech organizovány. Existuje několik možných způsobů organizace záznamů v souborech, mezi které patří:

- *Neuspořádaný soubor (heap file)* – každý záznam může být uložen v souboru na libovolné místo, kde je pro něj prostor. Záznamy nejsou nijak uspořádány. Při této organizaci jsou typicky ukládány do souboru záznamy jedné tabulky.
- *Sekvenční soubor (sequential file)* – záznamy jsou v souboru uloženy podle hodnoty tzv. *vyhledávacího klíče (search key)*. Záznamy tvoří uspořádaný seznam vytvořený použitím ukazatelů. Sekvenční soubor umožňuje zpřístupňovat záznamy v pořadí vyhledávacího klíče, což může být užitečné pro zobrazování nebo při některých operacích při zpracování dotazů (například při ORDER BY, spojování tabulek metodou sort-merge). Pokud by se však nacházely záznamy, které jdou sekvenčně za sebou, v různých blocích na disku, byla by efektivnost přístupu výrazně snížena. Proto je žádoucí, aby záznamy jdoucí sekvenčně za sebou byly uloženy i fyzicky blízko. To umožní jednou diskovou operací získat řadu uspořádaných záznamů. Zachovat fyzické uspořádání tak, aby odpovídalo uspořádání sekvenčnímu je však díky operacím vkládání a rušení obtížné. Používá se tzv. *přetokových bloků* podobně, jak jsme to viděli v Obr. 4.3 d). Při vkládání nového záznamu, který by měl být zařazen mezi záznamy v bloku se již nacházející, jej uložíme na volné místo v bloku, pokud existuje, jinak do přetokového bloku. V obou případech je záznam zařazen na příslušné místo seznamu pomocí ukazatelů. Přetokové bloky snižují efektivnost přístupu. Pokud je množství bloků v přetokových blocích velké, může být nutné provést *reorganizaci* souboru, která spočívá v novém fyzickém uspořádání tak aby odpovídalo uspořádání sekvenčnímu.
- *Hašovaný soubor (hashed file)* – každý záznam souboru je umístěn do bloku, který je určen hašovací funkcí pro hodnotu určitého pole záznamu, tzv. *hašovacího klíče (hashing key)*. Podrobněji se s hašováním seznámíme v kapitole 4.4.2.

Zpravidla jsou do jednoho souboru ukládány záznamy s daty jedné tabulky. Podobně jako jsme si vysvětlili u sekvenčního souboru, že je důležité, aby se fyzické uspořádání záznamů co nejvíce blížilo uspořádání sekvenčnímu, může být užitečné umístit do jednoho diskového bloku záznamy více než jedné tabulky, které spolu tak souvisí, že jsou často zpracovávány pohromadě. Taková organizace, která se nazývá *shlukovaný (clustered file)*, pak umožní jednou diskovou operací načíst všechny potřebné záznamy. Příkladem by mohlo být shlukování záznamu o účtu a jeho transakcích jak jsme předpokládali v příkladu Příklad 4.3.

Shlukování se provádí na základě hodnoty tzv. *shlukovacího/shlukovaného klíče (clustering/clustered key)*. O shlukování se ještě krátce zmíníme v kapitole 4.4.3. V tomto smyslu můžeme hašovaný soubor považovat za formu shlukování na základě hodnoty hašovacího klíče.



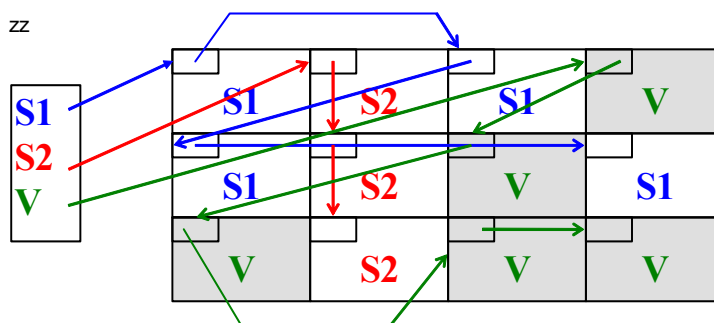
V předchozích odstavcích, které se týkaly organizace souborů, jsme si zavedli pojmy vyhledávací, hašovací a shlukovací klíč, podle jejichž hodnot jsou záznamy v souboru organizovány. Je třeba upozornit, že tyto pojmy nijak nesouvisí s pojmy kandidátní, primární a cizí klíč, které jsme si zavedli u relačního modelu dat. Vyhledávacím, hašovacím a shlukovacím klíčem může být kterýkoliv sloupec (případně složený) tabulky. Například pokud bude velice častou operací nad tabulkou Klient vyhledání podle jména klienta, může být užitečné použít na fyzické úrovni přístupovou metodu, která použije jméno klienta jako vyhledávací klíč. Samozřejmě i sloupce, které tvoří primární, cizí, případně kandidátní klíč, mohou být a velice často jsou použity jako

klíče pro sekvenční uspořádání, hašování nebo shlukování.

Už v úvodu této kapitoly jsme si řekli, že databázový soubor může ale nemusí odpovídat souboru operačního systému. Můžeme se setkat se dvěma přístupy:

- Záznamy každé tabulky jsou uloženy v samostatném souboru operačního systému. Tato organizace byla typická pro systémy s architekturou PC file-server, například dBase. Používá ji ale například i MySQL, kde je v závislosti na typu tabulky vytvořeno několik souborů. Například pro tabulky typu MyISAM využívající index-sekvenční přístupovou metodu ISAM jsou v souboru s příponou *.frm* metadata, v souboru s příponou *.MYD* data a v souboru s příponou *.MYI* index.
- Celá databáze obsahující obecně řadu schémat pro jednotlivé uživatele je uložena v jednom nebo několika souborech operačního systému. Jeden databázový soubor může být tvořen bloky několika souborů operačního systému. Tato organizace je typická pro systémy s architekturou klient-server (např. Oracle, SQLBase).

Ve druhém případě musí mít SŘBD, resp. správce diskového prostoru, přehled, kde se který databázový soubor nachází. Může být použita organizace známá ze správy souborů operačního systému založená na nějakém adresáři, ve kterém je informace o souborech tvořených bloky organizovanými jako seznam, a dále informace o volném prostoru, tj. volných blocích – viz Obr. 4.5.



Obr. 4.5 Adresář databázových souborů



V této kapitole jsme si vysvětlili, jak jsou reprezentovány záznamy s databázovými daty v databázových souborech, jak jsou databázové soubory organizovány a jaký je vztah databázových souborů a souborů operačního systému.

Ukázali jsme si, jak lze ukládat záznamy pevné délky, a také jsme si ukázali několik možných implementací ukládání záznamů proměnné délky. Viděli jsme, že se v těchto strukturách často používají ukazatele (RID, ROWID) a ještě se s nimi setkáme, až se budeme zabývat indexováním. V té souvislosti jsme zdůraznili požadavek na neměnnost jeho hodnoty a ukázali jsme si, jak lze tento požadavek řešit adresářem bloku.

Uvedli jsme si tři často používané organizace záznamů v souborech – neuspořádaný, sekvenční a hašovaný soubor. Dále jsme si vysvětlili podstatu shlukování, které umožňuje minimalizovat počet přístupů k disku při přístupu ke skupině záznamů, které jsou často zpracovávány společně.

V závěru jsme si potom uvedli, že v současné době je u systémů s architekturou klient-server nejčastěji celá databáze, která může obsahovat řadu schémat, která mohou

zahrnovat řadu tabulek a jiných databázových objektů v jednom nebo několika souborech operačního systému. Naproti tomu typické mapování na soubory operačního systému u databázových systémů s architekturou PC file-server bylo takové, že záznamy každé tabulky byly uloženy v samostatném souboru.

4.3. Správa vyrovnávací paměti

V úvodu této kapitoly jsme si ukázali, že SŘBD nepřistupuje přímo k datům na disku, nýbrž že využívá vyrovnávací paměti, ve které jsou dočasně uloženy některé diskové bloky. Již jsme také zmínili, že i když i operační systém, v jehož prostředí SŘBD běží, využívá při diskových vstupně výstupních operacích vyrovnávací paměti, svou vyrovnávací paměť si SŘBD spravuje sám. V této krátké podkapitole si vysvětlíme, proč tomu tak je.

Důvody plynou ze specifčnosti některých požadavků, které SŘBD na svou vyrovnávací paměť klade. Mezi tyto požadavky patří zejména:

- *Co nejefektivnější algoritmus uvolňování prostoru ve vyrovnávací paměti.*
Kapacita vyrovnávací paměti je omezená, a proto v ní nebudou v naprosté většině případů uloženy všechny diskové bloky, se kterými SŘBD pracuje. Dříve nebo později dojde k zaplnění vyrovnávací paměti a nastane potřeba uvolnit prostor pro načtení dalšího bloku z disku, se kterým SŘBD potřebuje pracovat. V operačních systémech se nejčastěji používá strategie LRU (Least Recently Used), která znamená, že prostor uvolňuje blok, který se v poslední době nejméně používal. Prakticky to znamená, že blok, od jehož posledního použití uplynula nejdelší doba se uloží na disk, pokud byl modifikován, nebo pouze uvolní prostor, pokud modifikován nebyl. Přestože v případě vyrovnávací paměti SŘBD by v řadě případů bylo možné lépe předvídat potřebu použití bloků umístěných ve vyrovnávací paměti, strategie LRU je díky jednoduchosti implementace nejčastěji používaná.
- *Omezení času, kdy lze zapsat blok z vyrovnávací paměti na disk.*
Běžné strategie výměny bloků ve vyrovnávací paměti, včetně strategie LRU, předpokládají, že blok, který byl modifikován, může být zapsán kdykoli na disk. Až se budeme věnovat v kapitole 5 transakčnímu zpracování, uvidíme, že z důvodu zajištění zotavení po poruchách a chybách, kdy například z důvodu poruchy a následného restartu počítače dojde ke ztrátě dat ve vyrovnávací paměti, existují pravidla, která říkají, v jakém pořadí se mají zapisovat jaké informace na disk. Z těchto pravidel vyplývá, že mohou nastat situace, kdy nelze zapsat na disk nějaký blok vybraný na základě strategie LRU dříve, než budou zapsány z vyrovnávací paměti nějaké jiné bloky. Jinými slovy, musí existovat možnost po určitou dobu fixovat blok (tzv. pinned block) ve vyrovnávací paměti. Tento požadavek zpravidla vyrovnávací paměti operačního systému nejsou schopny zaručit, a je to proto jeden z důvodů, proč si SŘBD spravuje vyrovnávací paměť sám.
- *Možnost vynuceného zápisu všech modifikovaných bloků na disk.*
Na druhé straně, jak si vysvětlíme v kapitole 5 v souvislosti se zotavením po poruchách a chybách, potřebuje SŘBD v jistých okamžicích provést zápis všech modifikovaných bloků na disk. Opět ale musí být dodrženo určité pořadí zápisu bloků. I toto je natolik specifický požadavek, že nemůže být splněn

vyrovnávací paměti ve správě běžných operačních systémů.

- *Možnost označit některé bloky, které by pokud možno měly být trvale ve vyrovnávací paměti.*

Jsou bloky, se kterými SŘBD pracuje velmi intenzivně. Jako příklad můžeme uvést bloky systémového katalogu a bloky indexů. Tento požadavek vlastně znamená využití možnosti lépe předvídat potřebu budoucího použití bloků, kterou SŘBD má, jak už jsme se o tom zmínili v souvislosti se strategií výměny bloků ve vyrovnávací paměti. I tento požadavek může být nejlépe uspokojen, pokud bude vyrovnávací paměť spravována přímo SŘBD.



SŘBD používá pro zefektivnění vstupně výstupních operací s diskem vyrovnávací paměť. Ta omezuje počet nutných čtení a zápisů na disk. Některé požadavky na správu této vyrovnávací paměti jsou natolik specifické, že nemohou být uspokojeny vyrovnávací paměti spravovanou operačním systémem. Jde zejména o požadavky plynoucí z jedné důležité funkce SŘBD – zajištění zotavení databáze po poruchách a chybách. Z tohoto důvodu je používaná vyrovnávací paměť spravována přímo SŘBD, konkrétně jeho komponentou, kterou nazýváme *správce vyrovnávací paměti (buffer manager)*.

4.4. Indexování a hašování

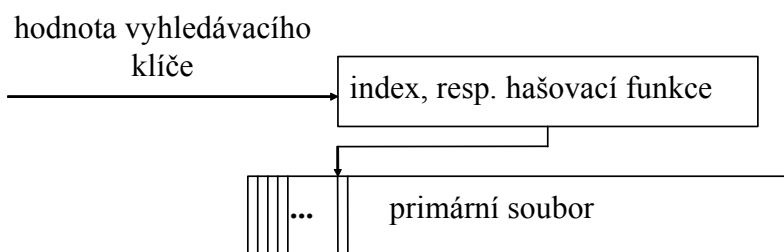
Výsledek operací s daty v databázi velice často zahrnuje jenom malou část dat tabulek, nad nimiž operace probíhá. Promítneme-li si důsledek tohoto faktu na fyzickou úroveň databáze, můžeme říci, že tyto operace se dotýkají jenom malé části záznamů databázových souborů. Uvažujme dotaz nad naší databází spořitelny „Jaká částka je na účtu s číslem účtu 100?“ Provedení tohoto dotazu vyžaduje nalezení záznamu s hodnotou 100 v poli *c_uctu* databázového souboru, ve kterém jsou záznamy tabulky Ucet. Protože číslo účtu jednoznačně určuje účet, bude nalezen nejvýše jeden záznam. Hodnota v poli *stav* by potom byla výsledkem dotazu. Pokud připustíme, že ve spořitelně je vedeno mnoho účtů, budou záznamy o nich uloženy v řadě diskových bloků. Triviální přístup založený na přečtení všech záznamů souboru s ověřováním, zda jde o záznam o účtu s číslem 100 by byl neefektivní. Proto se v databázích používají efektivnější přístupové metody založené na indexování a hašování, kterým se budeme věnovat v této podkapitole.

Nejprve si vysvětlíme základní pojmy a potom se podrobněji zaměříme na dvě přístupové metody, které se používají nejčastěji – indexování využitím uspořádaného indexu a hašování využitím hašovaného indexu. Tyto dvě přístupové metody úzce souvisí se dvěma organizacemi záznamů v databázových souborech, se kterými jsme se seznámili v kapitole 4.2, - sekvenčním a hašovaným souborem.

Při výkladu podstaty *indexu* souboru se zpravidla používá přirovnání k indexu knihy. Index v knize se používá pro rychlé nalezení místa, na kterém se vyskytuje nějaký pojem. Chceme-li tedy takové místo najít, podíváme se do indexu, najdeme v něm hledaný pojem a pokud tam je, jsou u něho uvedeny stránky, na kterých se tento pojem vyskytuje. Aby naše hledání v indexu bylo jednoduché, jsou pojmy v indexu abecedně uspořádány. Velice podobně pracuje index souboru, který označujeme jako uspořádaný, protože záznamy indexu jsou uspořádány podobně jako pojmy v indexu knihy.

Druhou možností, jak určit místo, kde se nějaká hodnota nebo pojem vyskytuje, je pomocí funkce, která když dostane jako argument hledanou hodnotu, vrátí místo, kde se tato hodnota nachází. To je podstata *hašování*. Hašování v této podobě je samozřejmě nepoužitelné u knihy, protože tam je index odvozen z obsahu, zatímco u hašování naopak určuje hašovací funkce, kde se musí nacházet daná hodnota.

I přes tyto rozdíly je použití uspořádaného indexu a hašování stejné. Předpokládejme, že existuje soubor záznamů tabulky databáze a my potřebujeme rychle přistupovat k záznamům tohoto souboru podle hodnot nějakého jejího sloupce (případně složeného). Soubor se záznamy tabulky budeme označovat jako *primární soubor* a sloupec, podle kterého záznamy zpřístupňujeme, budeme nazývat *vyhledávací klíč*. Potom v obou případech na základě hodnoty vyhledávacího získáme kolekci ukazatelů do souboru, kde se nachází záznamy se zadanou hodnotou klíče. Pokud je hodnota vyhledávacího klíče v souboru unikátní, bude výsledkem nejvýše jeden ukazatel. Situaci ukazuje Obr. 4.6.



Obr. 4.6 Podstata indexování a hašování

Zavedeme si nyní několik dalších pojmů, se kterými budeme dále pracovat.

Prvním z nich je *primární vyhledávací klíč*. Bude to takový vyhledávací klíč, podle jehož hodnot jsou uspořádány záznamy primárního souboru. Všechny ostatní vyhledávací klíče pro danou tabulku budeme nazývat *sekundární vyhledávací klíč*.

Uspořádaným indexem budeme rozumět datovou strukturu tvořenou záznamy obsahujícími hodnoty vyhledávacího klíče. Záznamy jsou uspořádány podle hodnot vyhledávacího klíče.

Hašovaným indexem budeme rozumět datovou strukturu, uspořádání jejichž záznamů je dáno hodnotou *hašovací funkce* pro hodnotu vyhledávacího klíče obsaženou v daném záznamu. Vyhledávací klíč se v tomto případě často označuje jako *hašovací klíč*.



Znovu připomínáme, že pojem vyhledávací, resp. primární vyhledávací klíč nesouvisí s pojmem primární klíč tabulky. Na druhé straně musíme připustit, že primární klíč bude často současně vyhledávacím klíčem, protože podle hodnot primárního klíče k záznamům tabulky často přistupujeme, a proto požadujeme efektivní přístupovou metodu.

V dalším budeme indexováním rozumět přístupové metody používající uspořádaný index a hašováním přístupové metody používající hašovaný index.

Jak u indexování, tak hašování mohou být záznamy primárního souboru přímo součástí datové struktury využívané danou přístupovou metodou. V případě hašování je výsledkem hašování primární soubor.

Nejprve se budeme věnovat indexování.

4.4.1. Indexování

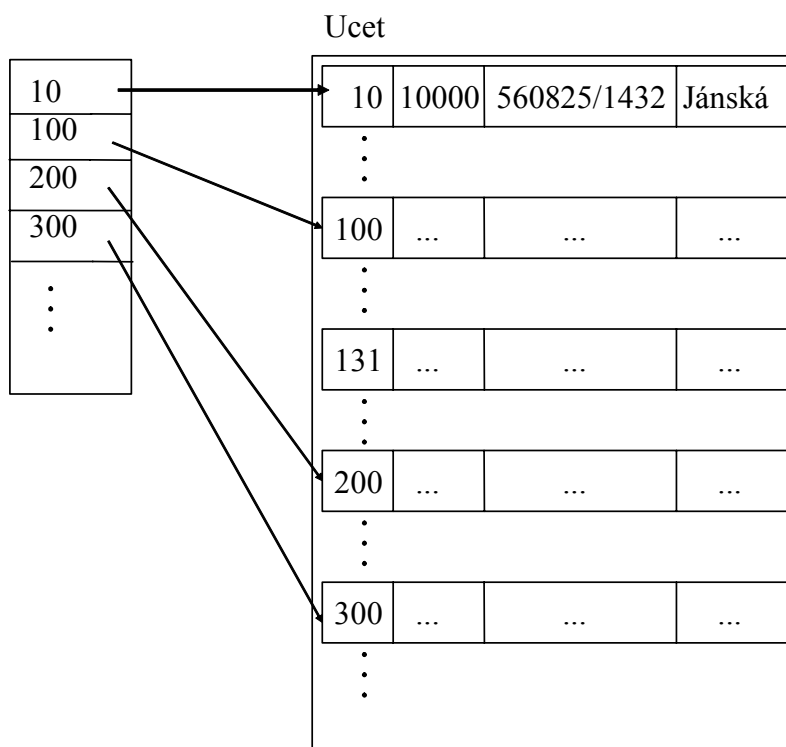
Již víme, že pro urychlení přístupu k záznamům tabulky používá datovou strukturu zvanou index, podobně jako používáme index v knize. Každý index je vytvořen pro konkrétní vyhledávací klíč. Index pro primární vyhledávací klíč se nazývá *primární index* (také označovaný jako *hlavní index*), index pro sekundární vyhledávací klíče se nazývá *sekundární index*. Primární indexy se také nazývají *shlukující (clustering) indexy*, protože vedou na shlukované soubory. Index pro vyhledávací klíč s unikátními hodnotami se nazývá *unikátní index (unique index)*.

Sekvenční soubor s primárním indexem se nazývá *indexsekvenční soubor*. Jde o jednu z nejstarších indexačních metod používaných v databázových systémech, která je vhodná pro aplikace, které vyžadují jak sekvenční zpracování záznamů souboru, tak náhodný přístup k jednotlivým záznamům.



Termín primární index se někdy nesprávně používá pro označení indexu pro primární klíč tabulky. To jestli je vyhledávacím klíčem primární klíč nebo jiný sloupec tabulky není rozhodující. Podstatnou vlastností primárního indexu je, že hodnoty vyhledávacího klíče definují uspořádání záznamů tabulky v primárním souboru. Opět ale platí i to, co jsme uvedli o pár řádků výše, že primárním vyhledávacím klíčem je právě primární klíč tabulky.

Uvažujme tabulku Ucet naší ukázkové databáze a předpokládejme, že potřebujeme často k záznamům přistupovat podle hodnot čísla účtu. Předpokládejme, že sloupec *c_uctu* je nejen primárním klíčem tabulky, ale i primárním vyhledávacím klíčem. Primární index se sekvenčním souborem by mohl vypadat, jak je naznačeno na Obr. 4.7.



Obr. 4.7 Primární index pro *c_uctu* a soubor tabulky Ucet

Index je tvořen záznamy, které nazýváme *položky indexu* nebo také *záznamy indexu*. Každý obsahuje hodnotu vyhledávacího klíče a v závislosti na tom, jde-li o index primární či sekundární jeden nebo několik ukazatelů na záznamy primárního souboru s danou hodnotou vyhledávacího klíče. Rozlišujeme dva typy uspořádaných indexů:

- *Hustý index* je index, který obsahuje položku pro každou hodnotu vyhledávacího klíče, která se v primárním souboru (a tedy i v tabulce) vyskytuje. V případě hustého primárního indexu obsahuje položka indexu hodnotu vyhledávacího klíče a ukazatel na první záznam s danou hodnotou vyhledávacího klíče. Pokud má tabulka více řádků se stejnou hodnotou vyhledávacího klíče, jsou v případě primárního indexu v primárním souboru bezprostředně za sebou. Stačí ukazatel na první z nich. V případě sekundárního indexu musí položka indexu obsahovat ukazatele na všechny záznamy s toutéž hodnotou, protože se mohou nacházet v primárním souboru kdekoliv.
- *Řídký index* je index, který obsahuje jen některé hodnoty vyhledávacího klíče. Takovým indexem může být pouze index primární, protože lze využít uspořádání záznamů v primárním souboru. Pokud například chceme najít záznam tabulky Ucet z Obr. 4.7 s hodnotou čísla účtu 131, najdou se v indexu položky vymezující interval, ve kterém hledaný záznam leží. V našem případě jsou to záznamy s hodnotami 100 a 200. Je zřejmé, že pokud účet číslo 131 existuje, pak musí ležet v primárním souboru mezi záznamy, na které ukazují odpovídající ukazatele. Stačí tedy prohledat tuto část primárního souboru.

Hodnoty vyhledávacího klíče, které se nacházejí v položkách indexu, jsou typicky hodnoty prvních záznamů v diskových blocích, takže ukazatel pak ukazuje na blok, který se má prohledat (včetně případné přetokové oblasti).

Je vidět, že u řídkého indexu se kombinuje přímý přístup se sekvenčním prohledáním části sekvenčního souboru (odtud také název indexsekvenční soubor). Proto je tento typ indexu použitelný pouze u primárního indexu. Sekundární index musí být vždycky hustý, protože primární soubor může být uspořádán pouze jedním způsobem a to podle hodnot primárního vyhledávacího klíče.

K lepšímu pochopení operací údržby indexu při vkládání nového řádku do tabulky, resp. rušení řádku může přispět následující zápis operací v pseudokódu.



Algoritmus 4.1 Vložení hodnoty do sekundárního (hustého) indexu

Najdi v indexu položku s vkládanou hodnotou vyhledávacího klíče;

IF položka existuje THEN

Vlož ukazatel na příslušný záznam primárního souboru do položky indexu

ELSE

Vytvoř novou položku s vkládanou hodnotou vyhledávacího klíče a ukazatelem na příslušný záznam primárního souboru;



Algoritmus 4.2 Vložení hodnoty do řídkého (primárního) indexu.

Položka indexu ukazuje na blok.

IF je vytvořen nový blok primárního souboru THEN

Vytvoř novou položku indexu s vkládanou hodnotou

vyhledávacího klíče a ukazatelem na příslušný blok
 ELSE

IF nově vložený záznam má nejmenší hodnotu
 vyhledávacího klíče v bloku THEN
 Aktualizuj hodnotu vyhledávacího klíče
 v položce indexu ukazující na daný blok;



Algoritmus 4.3 Zrušení hodnot v sekundárním (hustém) indexu

Najdi v indexu položku s rušenou hodnotou vyhledávacího klíče;
 IF položka obsahuje pouze ukazatel na rušený záznam THEN
 Zruš položku indexu
 ELSE
 Zruš ukazatel položky na rušený záznam;



Algoritmus 4.4 Zrušení hodnot v řídkém (primárním) indexu.

Položka indexu ukazuje na blok.

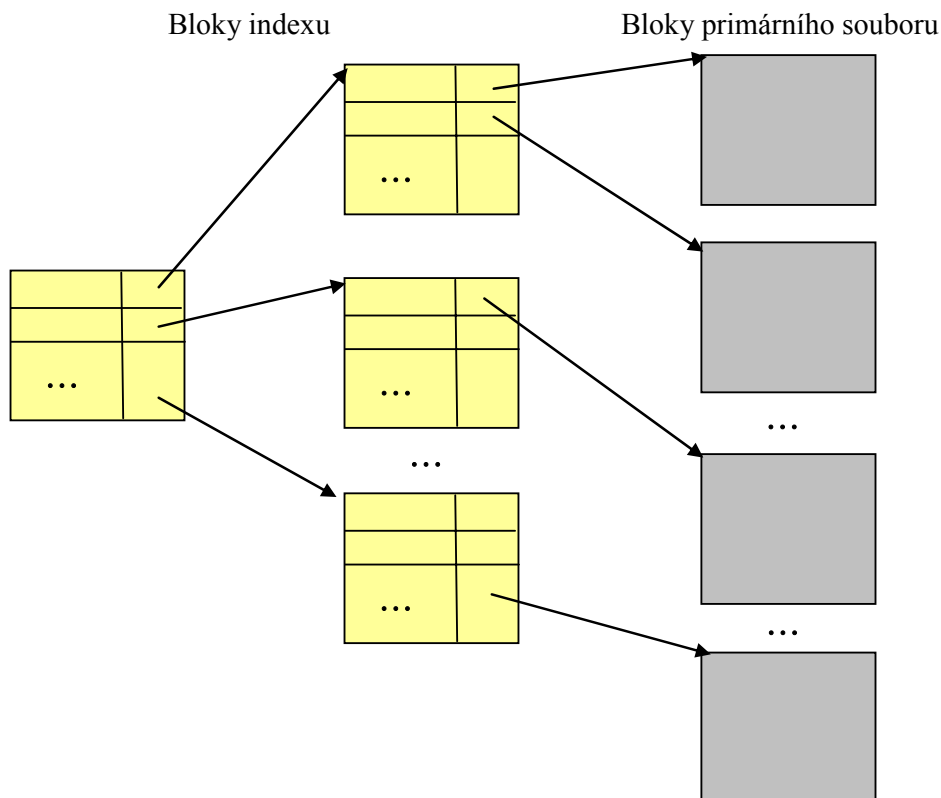
Najdi v indexu položku s rušenou hodnotou vyhledávacího klíče;
 IF položka existuje THEN
 IF položka indexu ukazuje na rušený záznam THEN
 IF rušený záznam není jediným záznamem v daném bloku primárního souboru THEN
 Aktualizuj položku indexu pro následující záznam bloku primárního souboru
 ELSE
 Zruš položku indexu;

Je důležité si už teď uvědomit, a ještě se k tomu vrátíme, až se budeme věnovat fyzickému návrhu databáze, že vložení nového nebo zrušení existujícího řádku tabulky vyvolá některou z výše uvedených operací pro každý index vytvořený pro danou tabulku.

Záznamy indexu jsou uspořádané podle hodnoty vyhledávacího klíče, proto lze využít efektivních metod hledání, například binární hledání. Zpravidla také zabírají ve srovnání s primárním souborem méně místa, budou pro rozsáhlé tabulky rozsáhlé. Na disku budou obsazovat řadu bloků. Je proto žádoucí zefektivnit i přístup k samotným položkám indexu. Můžeme použít úplně stejný přístup jako v případě primárního souboru a vytvořit index indexu. Dostáváme se tak k pojmu *víceúrovňový index*.

Víceúrovňovým indexem budeme rozumět index, který má stromovou strukturu, přičemž každá k -tá úroveň ($1 \leq k \leq N - 1$, kde N je počet úrovní indexu) je jednoúrovňovým indexem úrovně $k + 1$. Příklad dvouúrovňového indexu je uveden na Obr. 4.8.

Z obrázku je vidět, že při přístupu k záznamu primárního bloku je třeba načíst dva bloky indexu a jeden blok primárního souboru. Uvidíme, že obecně to bude vždy výška víceúrovňového indexu, blok primárního souboru, případně ještě tzv. blok sektoru ukazatelů. Přitom výška indexu ani pro velmi rozsáhlé tabulky nemusí být velká.



Obr. 4.8 Dvouúrovňový index

Z předmětu Algoritmy ale víte, že nevýhodou vyhledávacích algoritmů, které používají stromové struktury může být nevyváženost, která vzniká vkládáním a rušením hodnot. Je žádoucí, aby údržba indexu v důsledku vkládání, rušení a modifikace řádků tabulky s sebou nesla co nejmenší režii. Tyto požadavky do značné míry řeší tzv. B^+ strom, který je nejpoužívanější databázovou indexovou strukturou. Příkladem jiné v praxi rovněž používané datové struktury v podobě víceúrovňového indexu je tzv. *ISAM (Index Sequential Access Method) index*.

B^+ strom je víceúrovňový index ve tvaru n -nárního vyváženého stromu, který lze charakterizovat následujícími důležitými vlastnostmi:

- Každý uzel s výjimkou kořene a listu má $\lceil n/2 \rceil$ až n následníků.
- Kořen, není-li současně listem, má nejméně 2 následníky.
- List obsahuje $\lceil (n-1)/2 \rceil$ až $n-1$ hodnot vyhledávacího klíče.
- Listové uzly tvoří jednosměrný uspořádaný seznam.
- Uzel stromu má typicky velikost jednoho diskového bloku.
- Pro K hodnot vyhledávacího klíče není cesta od kořene k listu delší než $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$



Výraz $\lceil x \rceil$ značí nejmenší celé číslo větší nebo rovno x , tj. zaokrouhlení nahoru na celé číslo, a $\lfloor x \rfloor$ značí největší celé číslo menší nebo rovno x , tj. zaokrouhlení dolů na celé číslo.

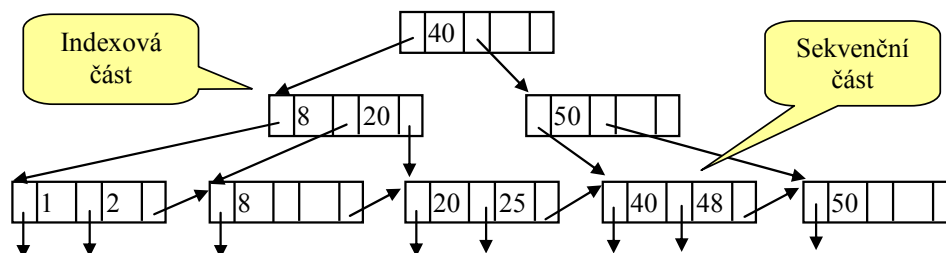
B^+ strom je dynamickou strukturou v tom smyslu, v uzlech může být ponechán prostor pro další růst s tím, že každý uzel s výjimkou vrcholu je zaplněn minimálně z 50%. V uzlu jsou uloženy jednak hodnoty vyhledávacího klíče, jednak ukazatele na blok obsahující uzel následníka. Hodnoty vyhledávacího klíče jsou v indexu seřazeny vzestupně nebo sestupně. Typický tvar uzlu je na Obr. 4.9.

P_1	V_1	P_2	V_2	...	P_{n-1}	V_{n-1}	P_n
-------	-------	-------	-------	-----	-----------	-----------	-------

Obr. 4.9 Typický tvar uzlu B^+ stromu

Předpokládejme vzestupné uspořádání hodnot vyhledávacího klíče v uzlu a předpokládejme, že nejde o listový uzel. Při úplném zaplnění je v uzlu n ukazatelů na následníky (značeny symbolem P) a $n - 1$ hodnot vyhledávacího klíče. Pokud není uzel listem stromu, ukazuje ukazatel P_1 na vrchol podstromu, ve kterém jsou uloženy hodnoty vyhledávacího klíče $h < V_1$. Je-li v uzlu k použitých ukazatelů ($k \leq n$), pak ukazatel P_k ukazuje na vrchol podstromu, ve kterém jsou uloženy hodnoty vyhledávacího klíče $h \geq V_k$. Pro ostatní ukazatele P_x ($2 \leq x \leq k-1$) v uzlu platí, že ukazují na vrchol podstromu, ve kterém jsou hodnoty vyhledávacího klíče h takové, že $V_{x-1} \leq h < V_x$. V případě listového uzlu ukazují ukazatelé na záznam primárního souboru nebo na tzv. sektor ukazatelů v případě, že index není unikátní. Navíc je jeden ukazatel použit k vytvoření jednosměrně vázaného seznamu, jak uvidíme za chvíli.

Příklad B^+ stromu pro $n = 3$ je uveden na Obr. 4.10.



Obr. 4.10 Příklad B^+ stromu pro $n = 3$

Strom obecně sestává ze dvou částí – sekvenční a indexové. *Sekvenční část* je tvořena položkami indexu, které obsahují hodnotu vyhledávacího klíče a ukazatel do primárního souboru. Má-li být B^+ strom hustým indexem, je sekvenční část hustým indexem pro daný vyhledávací klíč. Uzly sekvenční části tvoří jednosměrně vázaný seznam, který se využívá při sekvenčním zpracování záznamů, např. tzv. intervalových dotazech – viz dále. Využívá se k tomu posledních ukazatelů bloků. Každý další použitý ukazatel P_x ($1 \leq x \leq k-1$), kde $k \leq n-1$ je počet hodnot vyhledávacího klíče v uzlu, ukazuje na záznam v primárním souboru v případě, že index je unikátní, resp. na tzv. *sektor ukazatelů*, pokud může být v tabulce více řádků se stejnou hodnotou vyhledávacího klíče. Sektor ukazatelů je v takovém případě tvořen záznamy, které teprve obsahují ukazatele na záznamy v primárním souboru s danou hodnotou vyhledávacího klíče. Sektory ukazatelů se nacházejí v dalších blocích, které nejsou v obrázku uvedeny.

Minimální zaplnění listových bloků je $\lceil (n-1)/2 \rceil$ hodnot vyhledávacího klíče a tedy i ukazatelů na záznamy tabulky, resp. sektory ukazatelů. Maximální zaplnění je $(n-1)$ hodnot vyhledávacího klíče.

Indexová část B^+ stromu tvoří řídký víceúrovňový index části sekvenční. Minimální zaplnění uzlů s výjimkou kořene je $\lceil n/2 \rceil$ ukazatelů a tedy $\lceil n/2 \rceil - 1$ hodnot vyhledávacího klíče. Maximální zaplnění je n ukazatelů a $n - 1$ hodnot vyhledávacího klíče. Pokud je indexová část tvořena pouze kořenem, pak musí mít nejméně dva následníky. Význam hodnot vyhledávacího klíče a ukazatelů, které se nachází v uzlech indexové části, byl vysvětlen výše v souvislosti s Obr. 4.9.



Zkuste si překreslit B^+ strom z Obr. 4.10 na variantu se sestupným uspořádáním hodnot v uzlech.

Dosud jsme se nezabývali otázkou, čím je dána hodnota n , tedy maximální počet ukazatelů v uzlu. Protože velikost uzlu odpovídá diskovému bloku, je tato hodnota závislá jednak na prostoru potřebném pro uložení hodnoty ukazatele, jednak na datovém typu vyhledávacího klíče. V každém případě je to hodnota, kterou neurčuje uživatel při zadávání příkazu pro vytvoření indexu, nýbrž je určena SŘBD.

Nyní se blíže podíváme na operace nad B^+ stromem. Nejprve si připomeneme příkaz SQL pro vytvoření indexu. Přestože tento příkaz není součástí standardu SQL-92, jeho základní podoba je zpravidla stejná v různých dialektech a má tvar:

```
CREATE [UNIQUE] INDEX jméno_indexu ON jméno_bázové_tabulky
(jméno_sloupce [ASC|DESC], ... )
```

Říká, pro jakou tabulku má být index vytvořen, jaké sloupce tvoří vyhledávací klíč, zda mají být hodnoty sloupce uspořádány vzestupně nebo sestupně a zda jde o index unikátní. Implicitně nejde o unikátní index a hodnoty sloupce se uspořádají vzestupně (ASC). Index lze vytvořit kdykoliv, tedy nejen pro prázdnou tabulku.



Příklad 4.4

Předpokládejme, že příkazem

```
CREATE UNIQUE INDEX Iucet ON Ucet (c_uctu)
```

je vytvořen unikátní index v podobě B^+ stromu pro prázdnou tabulku Ucet naší ukázkové databáze. Vyhledávacím klíčem je sloupec c_uctu , který je současně primárním klíčem v tabulce.

Protože předpokládáme, že tabulka Ucet je v okamžiku provádění příkazu prázdná bude index obsahovat jediný uzel, který bude kořenem a zároveň listem. Splývá tedy také sekvenční a indexová část stromu. Kořen bude prázdný.

Nyní předpokládejme, že jsou vytvářeny jednotlivé účty a tedy také vkládány odpovídající řádky do tabulky Ucet. Vložení každého nového řádku vyžaduje také vložení informace o novém záznamu primárního souboru do indexu. Do kořene se vždy vloží položka indexu tvořená ukazatelem na záznam primárního souboru a hodnota čísla účtu. Tyto dvojice se v uzlu udržují uspořádané vzestupně podle hodnoty čísla účtu. Zároveň se kontroluje, že v uzlu již taková hodnota neexistuje. Protože jde o index unikátní, představuje opakovaná hodnota čísla účtu chybu. Vidíme, že zatím je vkládání jednoduché a výpočetně nenáročné.

Nyní předpokládejme, že je potřeba vložit do již zaplněného bloku kořene stromu další položku. V tomto okamžiku je potřeba provést již náročnější operaci spojenou s vkládáním položky indexu do B^+ stromu, která se nazývá *štěpení uzlu*. V našem případě vznikne nový uzel, do kterého se přesune polovina záznamů zaplněného

původního kořene. Vzniknou tedy dva uzly sekvenční části indexu, které se propojí do jednosměrně vázaného seznamu. Dále je ale nutné přidat další úroveň – nový kořen, který bude tvořit indexovou část B^+ stromu. Bude obsahovat ukazatele na oba uzly sekvenční části a nejnižší hodnotu vyhledávacího klíče druhého z bloků. Takovým způsobem poroste B^+ strom společně s růstem tabulky Ucet.

Podívejme se nyní obecněji na tři základní *operace nad B^+ stromem* – vyhledání záznamu tabulky s danou hodnotou vyhledávacího klíče, vložení informace o novém řádku tabulky a zrušení informace při zrušení řádku tabulky. Budeme uvažovat vzestupné uspořádání hodnot vyhledávacího klíče, hustý index a pro jednoduchost i unikátní. Operace pro sestupné uspořádání a u indexu pro vyhledávací klíč, jehož hodnoty v tabulce nemusí být unikátní, by vyžadovaly pouze mírné úpravy.

Operace *vyhledání* má vstupní parametr hodnotu vyhledávacího klíče hledaného záznamu. Výstupem je ukazatel na záznam, pokud takový řádek v tabulce existuje. Vyhledávání začíná od kořene a rekurzivně pokračuje po cestě stromu k uzlu sekvenční úrovně, ve kterém by měla být uložena daná hodnota vyhledávacího klíče a ukazatel na záznam, pokud existuje. Prohledání uzlu indexové části vždy spočívá v nalezení nejmenší hodnoty vyhledávacího klíče, jehož hodnota je větší než zadaná, resp. dosažení konce zaplněné části uzlu. V prvním případě je následníkem uzel na který ukazuje ukazatel bezprostředně předcházející nalezené hodnotě, ve druhém případě poslední ukazatel uzlu. Protože záznamy uzlu jsou uspořádané, lze použít efektivní vyhledávací metody, například binární hledání. Tímto způsobem se pokračuje tak dlouho, až se dosáhne bloku sekvenční části stromu. Tam se analogicky prohledá uzel, ale už na rovnost s hledanou hodnotou. Pokud je nalezena, bezprostředně předcházející ukazatel ukazuje na hledaný záznam primárního souboru.

x+y

Příklad 4.5

Předpokládejme že B^+ strom z Obr. 4.10 je indexem z příkladu Příklad 4.4 pro stav tabulky Ucet, kdy má tabulka 8 řádků s hodnotami čísla účtu 1, 2, 8, 20, 25, 40, 48 a 50. Necht' je zadán příkaz SQL

```
SELECT *
FROM Ucet
WHERE c_uctu = 25
```

a komponenta SŘBD zvaná optimalizátor zpracování dotazu rozhodne použít pro vyhledání index pro vyhledávací klíč *c_uctu*. Nejprve je načten blok kořene. Jeho prohledáním funkce implementující operaci vyhledání v B^+ stromu najde nejbližší vyšší hodnotu 40. Následuje proto načtení dalšího bloku s hodnotami 8 a 20. Jeho prohledáním se nejbližší vyšší hodnota nenajde, načte se proto blok určený posledním ukazatelem. To už je blok sekvenční části. Jeho prohledáním se najde hodnota 25 a ukazatel na odpovídající záznam primárního souboru.

Uvedeme si postupně opět i formálnější zápis všech tří operací v pseudokódu. Algoritmy nebudeme popisovat podrobně, půjde spíše o zachycení podstaty způsobu provedení příslušné operace.



Algoritmus 4.5 Vyhledání hodnoty v unikátním hustém indexu v podobě B^+ stromu

```
Nastav kořen jako aktuální uzel;
WHILE aktuální uzel není listem DO
BEGIN
```

```

    Najdi v uzlu nejmenší hodnotu vyhledávacího klíče
     $V_i$ , která je větší než hledaná hodnota;
    IF existuje THEN
        Nastav jako aktuální uzel, na který ukazuje  $P_i$ 
    ELSE
        Nastav jako aktuální uzel, na který ukazuje
        poslední ukazatel v aktuálním uzlu
    END;
    IF existuje v aktuálním uzlu hodnota  $V_i$  rovná hledané
    hodnotě THEN
        Ukazatel  $P_i$  ukazuje na hledaný záznam
    ELSE
        Záznam s hledanou hodnotou neexistuje;

```

Jednou z výhod použití B^+ stromu jako přístupové metody je, že umožňuje efektivní přístup k záznamům tabulky i u tzv. *rozsahových dotazů* (*range query*). Nemusí jít pouze o dotazy, ale i prohledávací varianty příkazů DELETE a UPDATE, které mají v klauzuli WHERE uvedenou podmínku, která pro operaci vybírá záznamy z určitého intervalu hodnot. S výhodou se využívá uspořádání uzlů sekvenční části B^+ stromu do uspořádaného jednosměrného seznamu. Stačí pomocí operace vyhledání najít v sekvenční části první záznam vyhovující dolní či horní hranici intervalu (podle uspořádání B^+ stromu). Následně jsou zpřístupněny pouhým sekvenčním procházením sekvenční části stromu, dokud se nenarazí na první hodnotu vyhledávacího klíče, která již není v daném intervalu, nebo se nenarazí na konec seznamu.

Příklad 4.6

x+y

Uvažujme B^+ strom z Obr. 4.10 a předpokládejme, že se provádí příkaz SQL

```

SELECT *
FROM Ucet
WHERE c_uctu BETWEEN 30 and 100

```

Prvním krokem je vyhledání záznamu s hodnotou čísla účtu 30 nebo nejbližší vyšší, která v tabulce Ucet existuje. Algoritmus vyhledání najde uzel sekvenční části, ve kterém by se hodnota 30 měla nacházet. Je to uzel s hodnotami 20 a 25. Protože tam taková hodnota není, pokračuje hledání nejbližší vyšší hodnoty v seznamu položek sekvenční části B^+ stromu. Takto nalezne položku s hodnotou čísla účtu 40. Počínaje záznamem primárního souboru, na který ukazuje ukazatel této položky, jsou do výsledku zařazeny i všechny další záznamy, na které ukazují následující položky sekvenční části až do konce seznamu, neboť nebude nalezena hodnota větší než 100. Vybrány jsou tedy záznamy s číslem účtu 40, 48 a 50.

Operace *vložení* informace o novém řádku tabulky má jako vstupní parametr vkládanou položku indexu, která je tvořena ukazatelem na záznam v primárním souboru a hodnotou vyhledávacího klíče ve vkládaném řádku tabulky. Nová položka indexu se vždy vkládá do některého uzlu sekvenční části. Operace je proto provedena ve dvou krocích. V prvním se postupem analogickým s operací vyhledání najde uzel sekvenční části, do které je potřeba vložit novou položku indexu. Zároveň se kontroluje u unikátního indexu, že taková hodnota ještě v tabulce neexistuje. Pokud existuje, dochází k chybě „Porušení integritního omezení unikátnosti hodnoty“.

Ve druhém kroku probíhá samotné vložení položky indexu do nalezeného bloku. Je-li

v bloku místo pro uložení položky, je tam vložena a operace končí. V opačném případě proběhne již zmíněná operace *štěpení uzlu* (*splitting*). Spočívá v tom, že se vytvoří nový uzel, polovina záznamů zaplněného uzlu se přesune do uzlu nového a nový uzel se zapojí do seznamu uzlů sekvenční části. Je tak zajištěno, že oba uzly budou zaplněny minimálně z 50%. Nově vkládaná položka se potom vloží do toho uzlu, kam z hlediska uspořádání podle hodnot vyhledávacího klíče patří.

Tím ale operace nekončí. Informace o vzniku nového uzlu v sekvenční části se musí vložit do příslušného bloku nejnižší úrovně indexové části stromu. Je tam opět potřeba vložit indexovou položku tvořenou nejmenší hodnotou vyhledávacího klíče v novém uzlu a ukazatelem na uzel. Při vkládání může nastat opět potřeba štěpení uzlu. Takovým způsobem se může operace štěpení uzlu šířit od listů ke kořeni stromu a může vést až ke zvýšení počtu úrovní stromu o jednu, pokud dojde ke štěpení kořene. V každém případě zůstává B^+ strom vyvážený a všechny uzly, případně s výjimkou kořene, jsou zaplněny minimálně z 50%.

Je zřejmé, že pokud nedojde ke štěpení uzlu, je náročnost operace vložení prakticky stejná jako operace vyhledání. Pouze při štěpení uzlu se náročnost zvyšuje. To je důvod, proč je v uzlech ponecháván prostor na růst. Tabulky databáze a tedy i indexy nad nimi totiž mají tendenci se s časem zvětšovat.

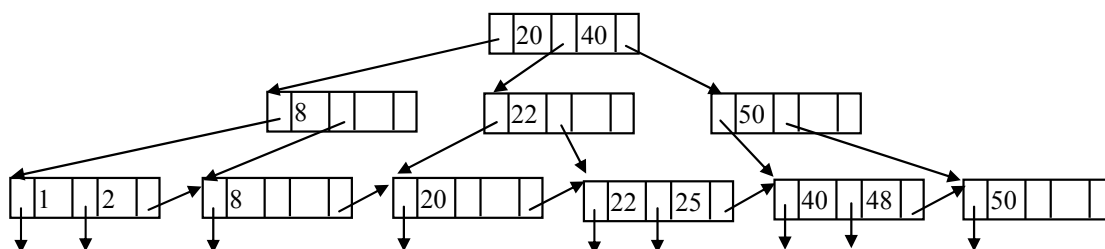
Příklad 4.7

x+y

Uvažujme B^+ strom z Obr. 4.10 a předpokládejme, že se do tabulky Ucet vkládá příkazem SQL

```
INSERT INTO Ucet
VALUES (22, 0, '440726/0672', 'Jánská')
```

nový řádek s číslem účtu 22 (pomiňme fakt, že v praxi by pravděpodobně měl nový účet vždy nejvyšší číslo v posloupnosti existujících čísel účtu banky). Informace o záznamu vkládaném do primárního souboru se musí promítnout do všech indexů vytvořených pro tabulku Ucet, tedy i našeho indexu Iucet. V prvním kroku se jako uzel sekvenční části indexu pro vložení nové položky vybere uzel s hodnotami čísla účtu 20 a 25. Protože je uzel zaplněn, musí se rozštěpit. Alokuje se diskový blok pro uložení nového uzlu a provede se rozdělení indexových položek do těchto dvou uzlů. Je třeba rozdělit tři položky s hodnotami čísla účtu 20, 22 a 25. Rozdělení se provede tak, že $\lceil (n-1)/2 \rceil$ položek bude v původním uzlu a zbytek v uzlu novém. V našem případě zůstane v původním uzlu položka s hodnotou 20 a do nového uzlu se přesune položka s hodnotou 25 vloží se tam nová položka s hodnotou 22. Tím však operace nekončí. Informace o novém uzlu a nejmenší hodnotě čísla účtu v něm se musí vložit do uzlu předchůdce v B^+ stromu, kde je ukazatel na uzel, který se rozštěpil. Položka s hodnotou čísla účtu 22 a ukazatelem na nově alokovaný uzel se musí vložit do uzlu s hodnotami čísla účtu 8 a 20. Protože je tento uzel již zaplněn, musí se také rozštěpit analogicky jako v předchozím případě. V původním uzlu zůstane položka s hodnotou čísla účtu 8 a do nově vytvořeného uzlu se přesune ukazatel z položky s hodnotou 20 a vloží nově vkládaná s hodnotou 22. Opět je v důsledku rozštěpení uzlu potřeba vložit informaci do uzlu předchůdce. Tím je v našem příkladu již kořen. Je v něm místo na vložení nové položky, takže vložení položky celá operace končí. Výsledný B^+ strom je uveden na Obr. 4.11.



Obr. 4.11 B+ strom z Obr. 4.10 po vložení položky s hodnotou 22

Zápis operace vložení položky indexu do B^+ stromu by mohl vypadat v pseudokódu takto:



Algoritmus 4.6 Vložení položky do unikátního hustého indexu v podobě B^+ stromu

Najdi listový uzel U , který by měl obsahovat novou položku indexu;

IF je v uzlu U místo pro vložení položky THEN

Vlož položku indexu do uzlu U

ELSE BEGIN /* Rozštěpení uzlu */

Alokuj nový uzel U' ;

Rozděl hodnoty štěpeného uzlu U včetně vkládané hodnoty na dvě „stejně velké“ skupiny. Jedna zůstane v uzlu U a druhá se přesune do uzlu U' ;

REPEAT

Rekurzivně pokračuj s vkládáním nové položky s první hodnotou vyhledávacího klíče uzlu U' a ukazatelem na tento uzel do uzlu předchůdce ve stromu

UNTIL nedošlo k dalšímu štěpení nebo byl rozštěpen kořen

END;

IF byl rozštěpen kořen THEN

Vytvoř nový kořen s ukazateli na dva uzly vzniklé rozštěpením původního kořene;

Operace *rušení* položky indexu, která odpovídá rušení řádku tabulky má jako vstupní parametr hodnotu vyhledávacího klíče v rušeném řádku tabulky. Operace opět probíhá ve dvou krocích. V prvním se najde položka indexu v sekvenční části s danou hodnotou vyhledávacího klíče. Pokud neexistuje, dochází k chybě „Rušený řádek neexistuje“.

Pokud byla položka indexu nalezena, je zrušena. Pokud po zrušení položky nepoklesne zaplnění uzlu pod minimální přípustnou mez, operace končí. V opačném případě dojde buď ke slévání uzlů nebo redistribuci hodnot. Uvažuje se jak pravý, tak levý soused málo zaplněného uzlu. Při *slévání uzlů* (*coalescence*) se zjišťuje, zda lze přesunout položky málo zaplněného uzlu do uzlu sousedního. Pokud ano, pak se přesunou, prázdný uzel se zruší a je nutné zrušit položku předchůdce v B^+ stromu, která ukazovala na zrušený uzel. Opět může dojít ke slévání nebo redistribuci a takovým způsobem se může operace slévání uzlů šířit od listů ke kořeni a může vést až ke snížení výšky stromu, pokud by kořen měl jenom jednoho následníka.

V případě, že v sousedním uzlu není dostatek místa pro slévání, provede se

redistribuce (redistribution) položek indexu takovým způsobem, aby oba sousední uzly splňovaly podmínku minimálního zaplnění. Tentokrát sice žádný uzel nezaniká ani nevzniká, ale změní se nejmenší hodnota vyhledávacího klíče v jednom z uzlů, mezi kterými k redistribuci dochází. Tato změna se musí promítnout do uzlu předchůdce v B^+ stromu a opět se tímto způsobem může šířit až ke kořenu.

Podobně jako při vkládání položky do B^+ stromu je i při rušení režie malá, pokud nedojde ke slévání uzlů nebo k redistribuci.

Za zmínku ještě stojí upozornit, že po operaci rušení mohou v indexové části B^+ stromu mohou zůstat hodnoty vyhledávacího klíče, které se nevyskytují v sekvenční části. Dojde k tomu tehdy, když se ruší položka, která obsahuje nejmenší hodnotu vyhledávacího klíče v nějakém uzlu sekvenční části stromu a po zrušení zůstane zaplnění uzlu dostatečné. Uvedli jsme si totiž, že vliv zrušení položky se do indexové části promítá pouze při slévání uzlů či redistribuci. Skutečnost, že nejmenší hodnota v uzlu uvedená v indexové části je ve skutečnosti menší než ta, která je skutečně v příslušném uzlu sekvenční části, nemá na algoritmus vyhledávání vliv a v důsledku vkládání a rušení položek indexu časem s velkou pravděpodobností zmizí.

Opět platí, že po provedení operace rušení zůstává B^+ strom vyvážený a má všechny uzly, případně s výjimkou kořene, zaplněny minimálně z 50%.

Příklad 4.8

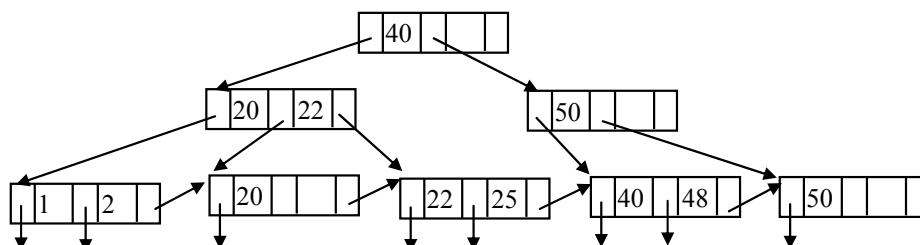
$x+y$

Uvažujme B^+ strom z Obr. 4.11 a předpokládejme, že se provádí příkaz SQL

```
DELETE FROM Ucet
WHERE c_uctu = 8.
```

Zrušení řádku tabulky, pokud existuje, se opět musí promítnout do všech indexů, které pro tabulku Ucet existují, tedy i do našeho indexu Iucet.

Nejprve se zjistí, že v sekvenční části položka s číslem účtu 8 existuje a provede se zrušení této položky. Protože to ale byla jediná položka v uzlu, klesne zaplnění pod přípustné minimum. Následuje slévání uzlů. To je v našem případě díky hodnotě $n = 3$ triviální a spočívá v pouhém zrušení prázdného uzlu. Následek zrušení uzlu je nutné promítnout do rodičovského uzlu. Zrušením odpovídající položky tam zůstane pouze ukazatel na nejlevější uzel sekvenční části a zaplnění uzlu tedy kleslo pod přípustné minimum (v našem příkladu 2 ukazatele, tj. jedna hodnota). Levého souseda uzel nemá, ale lze provést slévání s pravým sousedem. Protože i na této úrovni stromu zanikl uzel, šíří se operace zrušení položky indexu dál. Další úroveň je již kořen, který je zaplněn natolik, že i po zrušení položky zůstanou dva následníci, a operace končí.



Obr. 4.12 B^+ strom z Obr. 4.11 po zrušení položky s hodnotou 8

Zápis algoritmu operace zrušení hodnoty v unikátním hustém v pseudokódu by mohl vypadat následovně:


Algoritmus 4.7 Vložení položky do unikátního hustého indexu v podobě B^+ stromu

```

Najdi listový uzel  $U$ , který obsahuje rušenou položku
indexu;
Zruš položku indexu v uzlu  $U$ ;
IF kleslo zaplnění uzlu  $U$  pod  $\lceil (n-1)/2 \rceil$  THEN
  IF lze přesunout obsah do levého nebo pravého souseda
    uzlu  $U$  THEN
    /* slévání uzlů */
    BEGIN
      Přesuň obsah uzlu  $U$  a zruš prázdný uzel  $U$ ;
      REPEAT
        Rekurzivně pokračuj s rušením položky indexu
        s ukazatelem na uzel  $U$  v uzlu předchůdce ve
        stromu
      UNTIL nedošlo k dalšímu slévání nebo bylo dosaženo
        kořene;
      IF kořen má jen jednoho následníka THEN
        Zruš kořen, novým kořenem se stane následník
      END
    ELSE
      BEGIN /* redistribuce hodnot */
        Redistribuuj hodnoty a ukazatele sousedních uzlů
        tak, aby oba splňovaly podmínku minimálního
        zaplnění;
        Rekurzivně aktualizuj hodnotu vyhledávacího klíče
        v uzlu
        předchůdce ve stromu
      END;

```



B^+ strom může sloužit nejen jako indexová struktura, která na základě hodnoty vyhledávacího klíče dodá ukazatel na záznam, resp. na sektor ukazatelů na záznamy s danou hodnotou vyhledávacího klíče. Může být také způsobem organizace souborů. V takovém případě sekvenční část neobsahuje hodnoty vyhledávacího klíče a ukazatele, nýbrž přímo záznamy reprezentující řádky tabulky. Ve srovnání s indexsekvenčními soubory zde nedochází v důsledku vkládání a rušení záznamů k degradaci výkonnosti vlivem rostoucího narušení souladu fyzického a logického uspořádání záznamů primárního souboru. Protože však úplné záznamy zpravidla zabírají více místa než pouhá hodnota vyhledávacího klíče a ukazatele, jak je tomu u B^+ stromu jako indexu, je snaha zajistit vyššího využití bloků. Používají se proto někdy složitější varianty štěpení a slévání uzlů a redistribuce hodnot, než jsme si uváděli. Více se lze dozvědět např. v [Sil05].

Nyní se podíváme na vztah mezi počtem hodnot vyhledávacího klíče, resp. počtem řádků tabulky a velikostí B^+ stromu. Již víme, že pro K hodnot vyhledávacího klíče není výška stromu nikdy větší než $\lceil \log_{\lceil n/2 \rceil} (K) \rceil$. Zkusíme si vymežit velikost indexu přesněji. Při výpočtu si současně ověříte, zda jste strukturu indexu v podobě B^+ stromu správně pochopili.

Protože zaplnění uzlů je mezi 50 a 100%, nelze pro daný počet hodnot vyhledávacího klíče, resp. počet řádků tabulky stanovit počet úrovní a počet uzlů B^+ stromu přesně.

Lze stanovit pouze meze, mezi kterými se skutečné hodnoty nachází. Analogicky pokud známe počet úrovní B^+ stromu, nelze přesně určit počet hodnot vyhledávacího klíče, resp. počet řádků příslušné tabulky. Opět lze vymezit pouze rozsah hodnot. Hranice intervalu v obou případech vychází z předpokladu minimálního a maximálního zaplnění uzlů stromu. Situaci budeme ilustrovat dvěma příklady.

x+y

Příklad 4.9

Uvažujme index ve tvaru hustého B^+ stromu pro primární klíč tabulky, která má 15000 řádků. Blok primárního souboru obsahuje až 20 záznamů tabulky. Jaký bude počet bloků a úrovní indexu, obsahuje-li blok indexu maximálně 50 hodnot primárního klíče a jaká bude cena přístupu k záznamu vyjádřená počtem načítaných bloků?

Pro zodpovězení otázky jsou ze zadání důležité čtyři údaje:

- a. počet řádků tabulky,
- b. jde o index pro primární klíč, tedy unikátní index,
- c. jde o hustý index.
- d. maximální počet položek indexu vyjádřený počtem hodnot vyhledávacího klíče.

Informace o počtu záznamů v bloku primárního souboru je irelevantní.

První tři údaje určují, že ze sekvenční části B^+ stromu musí do primárního souboru ukazovat 15000 ukazatelů a uzly sekvenční části musí obsahovat rovněž 15000 hodnot vyhledávacího klíče. Na základě čtvrtého údaje můžeme určit počet položek indexu maximálně a minimálně zaplněného uzlu stromu. Obsahuje-li blok maximálně 50 hodnot vyhledávacího klíče, pak obsahuje maximálně 51 ukazatelů, tj. parametr B^+ stromu $n = 51$. Při maximálním zaplnění obsahuje uzel sekvenční části $n-1 = 50$ hodnot vyhledávacího klíče (jeden ukazatel je využit na vytvoření jednosměrného seznamu). Naopak při minimálním zaplnění obsahuje uzel sekvenční části $\lceil (n-1)/2 \rceil = 25$ hodnot vyhledávacího klíče. Podobnou úvahu provedeme pro uzly indexové části kromě kořene. Při maximálním zaplnění uzel obsahuje $n = 51$ ukazatelů na následníky a při minimálním $\lceil n/2 \rceil = 26$ ukazatelů. U kořene je maximální zaplnění stejné, minimálně musí mít dva následníky.

Nyní již můžeme začít určovat počty uzlů na jednotlivých úrovních počínaje sekvenční částí směrem ke kořenu a to pro varianty maximálního a minimálního zaplnění. Začneme variantou s maximálním zaplněním.

Víme, že do primárního souboru musí ukazovat 15000 ukazatelů a že v jednom uzlu sekvenční části jich může být nejvýše 50. Pro uložení 15000 ukazatelů tedy budeme potřebovat $\lceil 15000/50 \rceil = 300$ uzlů. Na těchto 300 uzlů musí ukazovat 300 ukazatelů z indexové části stromu. K jejich uložení potřebujeme $\lceil 300/51 \rceil = 6$ uzlů. Na těchto 6 uzlů musí ukazovat 6 ukazatelů z další úrovně indexové části. Je zřejmé, že pro jejich uložení už stačí jeden uzel ($\lceil 6/51 \rceil = 1$), který bude tvořit kořen stromu. Při maximálním zaplnění bude tedy mít B^+ strom 3 úrovně a bude mít celkem $300 + 6 + 1 = 307$ uzlů.

Pro určení podoby B^+ stromu pro minimální zaplnění uzlů budeme uvažovat analogicky. Pro uložení 15000 ukazatelů do primárního souboru je potřeba při 25 ukazatelích v bloku $\lceil 15000/25 \rceil = 600$ uzlů. Tentokrát zaokrouhlujeme dolů, protože

zaplnění nesmí klesnout pod stanovené minimum, což by se při zaokrouhlení nahoru mohlo stát. Na těchto 600 uzlů musí ukazovat 600 ukazatelů z indexové části stromu. K jejich uložení potřebujeme $\lfloor 600/26 \rfloor = 23$. Na těchto 23 uzlů musí ukazovat 23 ukazatelů z další úrovně indexové části. Je zřejmé, že pro jejich uložení už stačí jeden uzel, který bude tvořit kořen stromu. Při minimálním zaplnění bude tedy mít B^+ strom rovněž 3 úrovně a bude mít celkem $600 + 23 + 1 = 624$ uzlů.

Souhrnně je způsob výpočtu a získané hodnoty ještě uveden v následující tabulce:

Úroveň	Minimální zaplnění (maximum bloků)	Maximální zaplnění (minimum bloků)
sekvenční	$\lfloor 15000/25 \rfloor = 600$ uzlů	$\lceil 15000/50 \rceil = 300$ uzlů
indexová k	$\lfloor 600/26 \rfloor = 23$ uzlů	$\lceil 300/51 \rceil = 6$ uzlů
indexová $k-1$	1	1

Odpověď na první část otázky v zadání tedy bude: B^+ strom bude mít 3 úrovně, sekvenční část bude mít 300 až 600 bloků a indexová část kořen a 6 až 23 uzlů další úrovně.

Nyní již můžeme odpovědět i na druhou část otázky. Cena přístupu vyjádřená počtem načítaných bloků je dána výškou B^+ stromu a počtem načítaných bloků primárního souboru. Je tedy potřeba načíst 3 bloky indexu a jeden blok primárního souboru s hledaným záznamem.

Příklad 4.10

$x+y$

Uvažujme index ve tvaru hustého B^+ stromu pro primární klíč tabulky. Kolik řádků má tabulka, jestliže index má 3 úrovně a blok indexu obsahuje maximálně 50 hodnot primárního klíče?

Tentokrát řešíme opačnou úlohu než v předchozím příkladu. Známe výšku B^+ stromu a naším úkolem je stanovit počet řádků tabulky. Opět půjde o vymezení intervalu, ve kterém se skutečný počet řádků bude nacházet. Meze budou dány počtem řádků pro minimální a maximální zaplnění uzlů. Pro zodpovězení otázky jsou ze zadání důležité čtyři údaje:

- jde o index pro primární klíč, tedy unikátní index,
- jde o hustý index.
- maximální počet položek indexu vyjádřený počtem hodnot vyhledávacího klíče,
- počet úrovní indexu.

Z prvních dvou plyne, že počet ukazatelů ze sekvenční části indexu odpovídá počtu řádků tabulky. Třetí údaj určuje maximální a minimální zaplnění uzlů. Hodnoty jsou stejné jako v předchozím příkladu, tj. maximální zaplnění uzlu sekvenční části 50 ukazatelů do primárního souboru a u uzlů indexové části B^+ stromu 51 ukazatelů na následníky. Minimální zaplnění pro sekvenční část je 25 ukazatelů, pro uzly indexové části kromě kořene 26 a pro kořen 2 ukazatelé. Poslední údaj říká, kolik kroků při výpočtu ukazatelů budeme muset provést.

Narozdíl od předchozího příkladu, při kterém jsme postupovali od sekvenční části, protože jsme znali počet ukazatelů do primárního souboru a museli jsme určit

potřebnou výšku stromu, tentokrát vyjdeme od kořene a budeme počítat počty ukazatelů na následníky. Skončíme výpočtem počtu ukazatelů ze sekvenční části indexu a to pro variantu maximálního a minimálního zaplnění. Začneme variantou s maximálním zaplněním.

Při maximálním zaplnění bude mít kořen 51 následníků, každý z nich opět 51 následníků. Těmi už budou uzly sekvenční části (výška stromu je 3) a každý z nich bude obsahovat při maximálním zaplnění 50 ukazatelů na záznamy primárního souboru. Celkem tedy bude maximální počet ukazatelů do primárního souboru a tedy i maximální počet řádků tabulky roven $51 \cdot 51 \cdot 50 = 130050$.

Při minimálním zaplnění bude mít kořen 2 následníky, každý z nich 26 uzlů sekvenční části a z každého z nich bude ukazovat do primárního souboru 25 ukazatelů. Celkem tedy bude minimální počet ukazatelů do primárního souboru a tedy i minimální počet řádků tabulky roven $2 \cdot 26 \cdot 25 = 1300$.

Odpověď na zadanou otázku bude, že tabulka bude mít 1300 až 130050 řádků. Vidíme, že 1500 řádků, což byl vstupní údaj příkladu **Příklad 4.9**, při němž vyšla výška stromu 3, padne do námi vypočteného intervalu.



Na základě řešení těchto dvou příkladů byste si měli uvědomit, že počet ukazatelů ze sekvenční části a tedy i možný počet hodnot vyhledávacího klíče roste rychle (exponenciálně) s výškou B^+ stromu a naopak že výška stromu roste pomalu (logaritmicky) s počtem hodnot vyhledávacího klíče. Protože cena dotazu je dána výškou stromu a počtem bloků sektoru ukazatelů a primárního souboru, které je potřeba načíst, je v praxi i pro velice rozsáhlé tabulky velmi nízká. Lze ukázat, že potřebný počet bloků pro nejhorší případ při vkládání i rušení je také úměrný výšce stromu, proto lze říci, že všechny operace B^+ stromu jsou velice rychlé. Proto jsou indexy v podobě B^+ stromu v praxi používány velice často. Uvidíme, že ve srovnání s hašováním, kterému se budeme věnovat v následující podkapitole, mají B^+ stromy ještě jednu obrovskou výhodu v tom, že jsou univerzálnější. Umožňují efektivní přístup jak pro zadanou hodnotu vyhledávacího klíče, tak pro interval hodnot.



Dosud jsme mlčky předpokládali, že vyhledávacím klíčem je jednoduchý atribut, tedy sloupec tabulky. Vyhledávání v tabulkách je ale často řízeno ne jednou jednoduchou podmínkou, nýbrž několika takovými podmínkami. Uvažujme příkaz:

```
SELECT c_uctu, stav
FROM Ucet
WHERE r_cislo='530610/4532' AND pobočka='Jánská'
```

Pokud bychom chtěli poskytnout možnost využití indexování jak podle rodného čísla, tak podle pobočky, můžeme vytvořit jeden index pro rodné číslo a druhý pro pobočku. Druhou možností je vytvořit index pro složený vyhledávací klíč (r_cislo , $pobočka$), který by se mohl při vyhledávání využít.

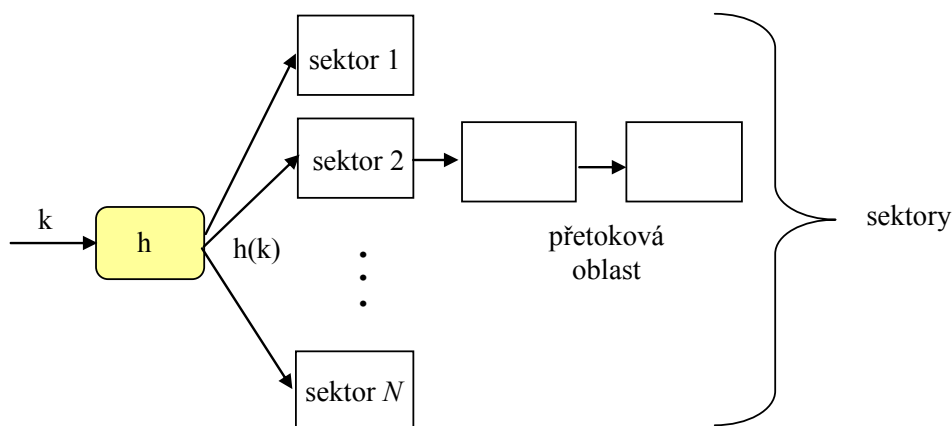
Když si vzpomenete na tvar příkazu CREATE INDEX, uvědomíte si, že vyhledávací klíč může být i složený, tvořený několika sloupci tabulky. Navíc hodnoty jednotlivých složek mohou být v indexu uspořádány nezávisle na sobě vzestupně či sestupně. Princip indexování a používané struktury, jak jsme si je ukázali v této kapitole, nijak nebrání použití složeného vyhledávacího klíče. V uvedeném příkladu bychom ho ale asi nepoužili, ledaže by toto byl natolik častý a kritický dotaz, že by se to vyplatilo. Indexy pro složené vyhledávací klíče vytváříme nebo je automaticky vytváří SŘBD v případech, kdy máme v tabulce složený primární nebo cizí klíč. Podrobněji se

problematikou rozhodnutí, kdy index vytvořit budeme zabývat v kapitole 4.5. o fyzickém návrhu databáze.

4.4.2. Hašování

Již víme, že podstata hašování spočívá v tom, že existuje tzv. *hašovací funkce*, která na základě hodnoty vyhledávacího klíče určí buď přímo nebo nepřímo adresu, kde se záznam, resp. záznamy s touto hodnotou vyhledávacího klíče nacházejí. V každém případě hašovací funkce převádí hodnotu vyhledávacího klíče na adresu. Pokud je touto adresou už blok, kde se nachází hledaný záznam, hovoříme o *přímém hašování*, pokud je to adresa místa, kde teprve najdeme informaci o adrese bloku, kde se záznam nachází, hovoříme o *nepřímém hašování*. Navíc paměťový prostor, na který hašovací funkce mapuje hodnotu vyhledávacího klíče může mít konstantní velikost nebo se jeho velikost může v průběhu existence příslušného hašování měnit. První případ nazýváme *statickým hašováním* a druhý *dynamickým hašováním*. Podíváme se nejprve na statické hašování a potom si jako příklad dynamického hašování uvedeme tzv. rozšiřitelné hašování.

Při statickém hašování jsou položky indexu, kterými jsou v případě přímého hašování záznamy s daty řádků tabulky (jde tedy o *hašovaný soubor*) nebo záznamy s ukazatelem na záznam primárního souboruho v případě nepřímého hašování (hovoříme o *hašovaném indexu*), rozděleny do disjunktních množin, označovaných jako *sektory* (*bucket*). Sektor, do kterého přísluší záznam s hodnotou vyhledávacího klíče k , je určen aplikací hašovací funkce h . Tedy $h(k)$ je adresou (resp. pořadovým číslem) sektoru. Podobně jako u B^+ stromu uzel i zde sektor je typicky uložen v jednom diskovém bloku. Hašovací funkce by měla přidělovat položky do sektorů rovnoměrně. V praxi však často rozdělení zcela rovnoměrné není například proto, že ani rozdělení hodnot vyhledávacího klíče v tabulce není rovnoměrné. Výsledkem je, že u některých sektorů se nedostává místa pro uložení položek, zatímco jiné jsou obsazeny jen z části. Tento problém se řeší pomocí přetokových oblastí, což je řetěz bloků rozšiřujících kapacitu sektoru. Přetokové oblasti se využívají i k pokrytí zvýšených požadavků na úložný prostor při kolizích způsobených stejnou hodnotou hašovací funkce pro různé hodnoty vyhledávacího klíče. Podstata statického hašování je znázorněna na Obr. 4.13.



Obr. 4.13 Statické hašování

Vyhledání záznamu s danou hodnotou vyhledávacího klíče při použití hašování probíhá tak, že se pro tuto hodnotu spočítá výsledek hašovací funkce, který určuje sektor, ve kterém se v případě hašovaného souboru již nachází hledaný záznam či záznamy, v případě hašovaného indexu určuje sektor, ve kterém se nachází položka či položky s ukazatelem na záznam v primárním souboru. Z Obr. 4.13 je zřejmé, že v případě ideální hašovací funkce bude pro přístup k datům potřeba načíst u hašovaného souboru pouze blok sektoru a v případě hašovaného indexu blok sektoru a obecně tolik bloků primárního souboru, kolik záznamů s danou hodnotou vyhledávacího klíče existuje (každý může obecně ležet v jiném bloku). Není-li zaplnění sektorů rovnoměrné a vznikají přetokové oblasti, nebude přístupová doba konstantní, protože u některých sektorů bude potřeba načíst i bloky přetokové oblasti.

Počet sektorů N je určen staticky a je parametrem definice hašování. Jeho velikost musí zohledňovat počet záznamů, které lze uložit do sektoru za předpokladu rovnoměrného rozložení hodnot vyhledávacího klíče, předpokládaný počet různých hodnot vyhledávacího klíče a řádků tabulky. Hodnota bude vždy kompromisem mezi využitím celkového paměťového prostoru sektorů a degradací rychlosti vlivem nutnosti načítání bloků přetokových oblastí. Čím větší bude využití paměťového prostoru sektorů, tím větší bude i pravděpodobnost potřeby přetokových oblastí v důsledku nerovnoměrného rozdělení hodnot vyhledávacího klíče a kolizí. Označíme-li si n_z celkový počet záznamů, které bude potřeba uložit (odpovídá maximálnímu očekávanému počtu řádků tabulky) a z_s počet záznamů, které lze uložit do jednoho sektoru, bude potřeba minimálně $N = n_z/z_s$ sektorů. Při tomto počtu by ale byla vysoká pravděpodobnost potřeby přetokových oblastí. Proto se tento počet zvyšuje, typicky o asi 20% v závislosti na tom, zda dáváme přednost výkonnosti nebo využití paměťového prostoru.

Klíčová z hlediska kvality hašování je volba hašovací funkce. Ideální hašovací funkce by měla rozdělovat položky rovnoměrně do všech sektorů. Protože v době návrhu není přesně známo, jaké hodnoty vyhledávacího klíče budou v tabulce a jaké bude jejich rozložení, požadujeme, aby hašovací funkce přiřazovala položky do sektorů takovým způsobem, že rozložení bude mít tyto vlastnosti:

- Bude *rovnoměrné*, tj. hašovací funkce přiřadí do každého sektoru stejný počet hodnot vyhledávacího klíče z množiny všech možných hodnot vyhledávacího klíče.
- Rozdělení bude *náhodné*, tj. nebude korelovat s nějakým externě viditelným uspořádáním hodnot vyhledávacího klíče, například podle abecedy.

Příklad 4.11

x+y

Uvažujme, že jsme se rozhodli organizovat soubor se záznamy tabulky Klient naší databáze spořitelny jako hašovaný podle hodnoty ve sloupci *jmeno* a že jsme jako hašovací funkci zvolili funkci, která bude rozdělovat záznamy do sektorů podle počátečního písmene jména klienta. Taková hašovací funkce bude velice jednoduchá, ale rozdělení záznamů nebude rovnoměrné, protože určitě nebude stejný počet jmen začínající jednotlivými písmeny abecedy.

Nyní předpokládejme, že navíc vytvoříme hašovaný index pro vyhledávací klíč *r_cislo*. Pro jednoduchost předpokládejme klienty narozené v letech 1900 až 1999, tedy rodná čísla začínající dvojicí 00 až 99 a rozdělení možných čísel do desíti intervalů: 00 až 09, 10 až 19, ..., 90 až 99. Rozdělení hodnot vyhledávacího klíče je

rovnoměrné, ale není náhodné, protože vychází z uspořádání na množině hodnot vyhledávacího klíče. I přes rovnoměrné rozdělení hodnot vyhledávacího klíče lze očekávat, že zaplnění sektorů rovnoměrné nebude, protože počty klientů ve věkových kategoriích vymezených intervaly dat narození budou různé.

Typické hašovací funkce používají zbytek po celočíselném dělení počtem sektorů. Součástí příkazu CREATE INDEX je potom buď přímo zadání počtu sektorů nebo předpokládaného počtu řádků tabulky a SŘBD počet sektorů z této hodnoty vypočítá. V obou případech se z důvodu lepšího rozdělení zaokrouhluje hodnota na nejbližší větší prvočíslo. Takovou funkci lze potom použít pro vyhledávací klíče numerických datových typů přímo, znakové řetězce se nejprve převedou na číslo tak, že se sečtou binární reprezentace znaků tvořících řetězec.

Příklad 4.12

x+y

V dialektu SQL databázového serveru SQLBase lze například vytvořit **hašovaný soubor příkazem**:

```
CREATE CLUSTERED HASHED INDEX index  
ON tabulka(sloupec (ASC|DESC), ...)  
SIZE pocet ROWS|BUCKETS
```

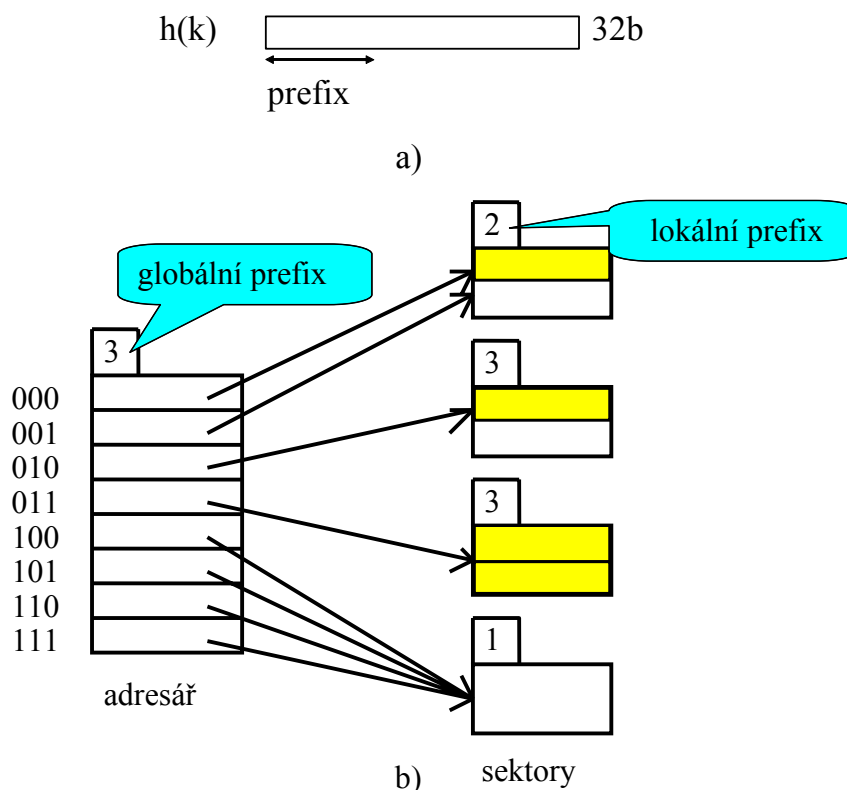
U databázového serveru Oracle se hašovaný soubor vytváří jako tzv. hašovaný shluk (cluster) a řekneme si o něm něco v kapitole 4.4.3.

Hlavní nevýhodou statického hašování je, že se alokuje potřebný počet bloků pro zadaný nebo vypočítaný počet sektorů už v okamžiku vytvoření přístupové metody a až na případné přetokové bloky zůstává neměnný. Znamená to, že již při návrhu musíme být schopni dobře odhadnout počet řádků tabulky a hodnot vyhledávacího klíče. Pokud bude počet sektorů malý, budou vznikat přetokové oblasti a dojde k poklesu výkonnosti, pokud bude velký, plýtváme diskovým prostorem.

Tuto nevýhodu se snaží odstranit *dynamické hašování*. Jeho podstata spočívá v tom, že počet sektorů se může po dobu existence přístupové metody podle potřeby měnit. Když dojde k zaplnění sektoru, neměla by se vytvářet přetoková oblast, ale mělo by být možné přidat další sektory. Podobně při uvolnění paměťového prostoru sektorů by mělo docházet k redukci jejich počtu. Jako příklad konkrétní techniky dynamického hašování si uvedeme tzv. *rozšiřitelné hašování*. Jeho podstata spočívá v tom, že hodnota vyhledávacího klíče se hašuje na binární vektor určité délky, který definuje velikost adresového prostoru, tj. maximální počet sektorů, který lze použít. Z tohoto vektoru se ale v daném okamžiku používá pouze taková část (prefix), která zajišťuje dostatečný počet sektorů.

Ukázka rozšiřitelného hašování je na Obr. 4.14. Obrázek a) ukazuje 32 bitový vektor, na který se hašuje hodnota vyhledávacího klíče. To umožňuje adresovat až 2^{32} sektorů. V obrázku b) je potom ukázána struktura, která s rozšiřitelným hašováním souvisí. Jedná se o hašování, u kterého existuje pomocná struktura zvaná *adresář*. Je to tabulka, která má tolik řádků, jaký je momentálně používaný adresový prostor určený aktuálním prefixem. Hodnota prefixu použitá v obrázku je 3. Tomu odpovídá adresář o osmi položkách adresovaných 000, 001, ..., 111. Každá položka obsahuje ukazatel na sektor se záznamy. V obrázku předpokládáme, že sektor může obsahovat maximálně dva záznamy. Jsou zde uvedeny také dva prefixy – tzv. *globální prefix* a *lokální prefix*. Globální prefix je aktuálně použitý prefix bitového vektoru, lokální prefix je prefix, který se týká daného sektoru. Lokální prefix může být menší než

prefix globální. V takovém případě na daný sektor ukazují ukazatelé ze všech položek adresáře, které mají stejnou hodnotu části adresy odpovídající lokálnímu prefixu. Například poslední sektor v obrázku má hodnotu lokálního prefixu 1. Při hodnotě globálního prefixu 3 to znamená, že na tento sektor ukazují 4 ukazatelé a budou se sem ukládat záznamy s hodnotou vyhledávacího klíče, která se hašuje na vektor začínající v našem případě jedničkou. Je-li g hodnota globálního prefixu a l hodnota lokálního prefixu sektoru, pak na tento sektor ukazuje z adresáře 2^{g-l} ukazatelů.



Obr. 4.14 Příklad rozšiřitelného hašování:

- a) bitový vektor jako výsledek hašování,
- b) adresář a význam globálního a lokálního prefixu

Podívejme se, jak může postupně tato struktura vzniknout. Na počátku má adresář jednu položku a ta obsahuje ukazatel na jediný sektor, do kterého se budou ukládat všechny záznamy bez ohledu na hodnotu bitového vektoru, dokud se sektor nezaplní. Hodnota globálního a lokálního prefixu jsou v tomto okamžiku rovny nule. Po vložení dvou řádků do tabulky se jediný sektor zaplní a při pokusu o vložení třetího musí dojít ke štěpení sektoru a tedy i zvýšení prefixu. Alokujeme blok pro nový sektor, zvýší se hodnota globálního prefixu na 1, adresář bude mít dvě položky a každá z nich ukazuje na jeden sektor. Současně dojde k redistribuci záznamů podle hodnoty hašovací funkce. V jednom budou ty, pro něž bitový vektor začíná nulou a ve druhém ty, pro které začíná jedničkou. Lokální prefix obou sektorů bude mít hodnotu 1.

Analogicky bude docházet ke štěpení sektoru vždy, když se v něm nedostává místa pro uložení záznamu. Vždy také dojde ke zvýšení lokálního prefixu, a tato hodnota se nastaví pro oba sektory vzniklé rozštěpením. Ne vždy ale musí dojít i ke zvýšení hodnoty globálního prefixu. K tomu dojde pouze v případě, že byla hodnota lokálního prefixu před štěpením rovna hodnotě prefixu globálního, tj. že na štěpený sektor ukazoval pouze jeden ukazatel. Uvažujme situaci znázorněnou na Obr. 4.14, kde jsou

barevně znázorněny obsazené záznamy v sektorech. Předpokládejme, že příkaz INSERT vloží do tabulky tři řádky, pro něž hodnoty hašovací funkce začínají jedničkou, konkrétně prefixy 100, 110 a 100. Prvé dva řádky se vloží do posledního ze sektorů uvedeného v obrázku. Pro vložení třetího ale již v sektoru není místo a musí dojít k rozštěpení. Alokuje se blok pro nový sektor, hodnota lokálního prefixu pro oba bude nastavena na hodnotu 2, na první budou ukazovat ukazatele z adresáře z adres 100 a 101 a na druhý z adres 110 a 111. Druhý z vložených záznamů se přesune do nového sektoru a v původním, který se rozštěpil, bude první a třetí.

Nyní uvažujme, že se do tabulky vkládá řádek, jehož hodnota hašovací funkce začíná prefixem 0111. Odpovídající záznam by měl být zařazen do sektoru pro prefix 011. Tento sektor je již ale zaplněn a musí dojít k jeho rozštěpení. Protože však hodnota lokálního prefixu je stejná jako hodnota prefixu globálního, musí dojít i ke zvětšení globálního prefixu a tedy i adresovacího prostoru hašovací funkce. Globální prefix se zvětší na 4 a počet řádků tabulky adresáře se zdvojnásobí, Zdvojnásobí se také počet ukazatelů ukazujících na jednotlivé sektory. Poté již může proběhnout rozštěpení zaplněného uzlu výše popsaným postupem.



Zkuste si překreslit Obr. 4.14 do stavu po popsaném rozštěpení sektoru pro prefix 001.

Podobně jako u B^+ stromu může u dynamického hašování při rušení řádků tabulky naopak docházet ke slévání sektorů a redukci adresového prostoru. Protože však operace vkládání bývá v praxi častější než rušení (tabulky mají tendenci se zvětšovat), volí se s ohledem na tento fakt často strategie, která se snaží vyhnout zbytečnému slévání.



Pokuste se určit podmínku, která je nutná pro slévání sektorů, a popište, jak by slévání proběhlo.

Závěrem zdůrazňme jednu podstatnou vlastnost hašování, která, jak uvidíme později, do určité míry omezuje jeho použití. Je jí schopnost zpřístupňovat data pouze pro konkrétní hodnoty vyhledávacího klíče. Nejsme schopni je použít obecně pro zpřístupnění záznamů s hodnotou vyhledávacího klíče ležící v určitém intervalu.

4.4.3. Shlukování

Podstatu shlukování záznamů jsme si již vysvětlili – jde o umístění záznamů se stejnými vlastnostmi, konkrétně stejnou hodnotou vyhledávacího klíče, fyzicky blízko, tj. do stejného diskového bloku. Pokud to jsou záznamy, které se často zpracovávají současně, ušetří se vstupně výstupní operace a přístup k záznamům se zrychlí.

Ve skutečnosti jsme se už se shlukováním setkali při výkladu indexování a hašování. Bylo to při takové organizaci souborů, kdy jsou datové záznamy nesoucí hodnoty řádků tabulky umístěny přímo v listech B^+ stromu nebo v sektorech u hašování. To je případ shlukování záznamů jedné tabulky. Podstata shlukování ale nevyklučuje shlukování záznamů několika tabulek. Obecně můžeme říci, že shluk (cluster) je množina tabulek, které sdílejí diskové bloky pro uložení datových záznamů, protože sdílejí nějaký společný sloupec (případně spojený), který se označuje jako *shlukující/shlukovaný klíč* (*clustering/clustered key*) a jsou použity často společně.

Příkladem by mohly být tabulky ODDĚLENÍ a ZAMĚSTNANCI z příkladu Příklad 3.10. V této podkapitole se zaměříme právě na shlukování záznamů několika tabulek. Ukážeme si, jak se takové shlukování provádí v prostředí databázového serveru Oracle. Pro tyto účely využijeme příklady a převezmeme některé obrázky z dokumentace dostupné na stránkách Oracle Documentation Library (<http://www.oracle.com/pls/db102/homepage>), konkrétně z dokumentace Oracle® Database Concepts. 10g Release 2 (10.2) a Administrator's Guide.. 10g Release 2 (10.2).

x+y

Příklad 4.13

Uvažujme tabulky Employees (employee_id, last_name, department_id, ...) a Departments(department_id, department_name, location_id). Sloupec *department_id* je primárním klíčem tabulky Departments a cizím klíčem v tabulce Employees, kde udává, ve kterém oddělení daný zaměstnanec pracuje. Předpokládejme, že častou operací je zpracování řádku oddělení a jeho zaměstnanců a že chceme využít možnost shlukování, kterou nám nabízí databázový server Oracle 10g.

Shlukujícím klíčem bude sloupec *department_id*. V případě shlukování bude v jednom bloku (doplněném případnými přetokovými bloky) uložen jeden záznam s hodnotami oddělení a záznamy všech zaměstnanců, kteří v tomto oddělení pracují, tj. záznamy obou tabulek se stejnou hodnotou shlukujícího klíče. V případě neshlukovaného souboru by tyto záznamy byly v oddělených souborech, jak ukazuje Obr. 4.15.

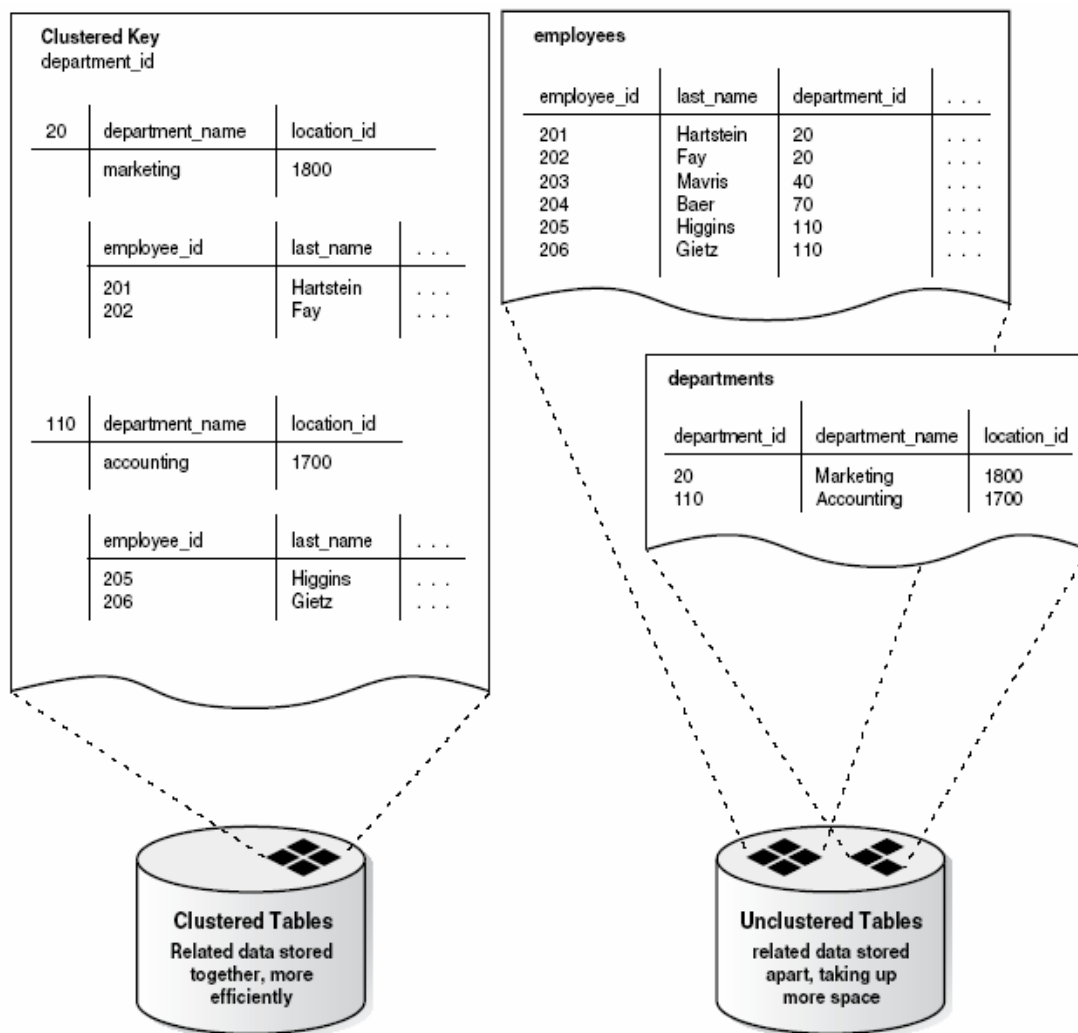
Vytvoření shlukovaného souboru sestává z následujících kroků:

1. Vytvoření prázdného shlukovaného souboru příkazem CREATE CLUSTER.
2. Vložení prázdných shlukovaných tabulek příkazem CREATE TABLE.
3. Vytvoření indexu pro přístup ke shlukovanému souboru, není-li použito shlukování.

Pro Příklad 4.13 by mohl být vytvořen shlukovaný soubor příkazem:

```
CREATE CLUSTER emp_dept (department_id NUMBER(3))
  SIZE 600
  TABLESPACE users
  STORAGE (INITIAL 200K
    NEXT 300K
    MINEXTENTS 2
    MAXEXTENTS 20
    PCTINCREASE 33);
```

Kromě prvního řádku příkazu jsou všechny zbývající nepovinné. Týkají se parametrů fyzického návrhu a dávají návrháři či administrátorovi možnost využít znalosti předpokládané velikosti tabulek, růstu tabulek apod. Stručně se o některých z nich zmíníme později.



Obr. 4.15 Shlukovaná data tabulek Employees a Departments z příkladu Příklad 4.13

Poté, co je prázdný shlukovaný soubor vytvořen, lze do něho vložit tabulky:

```
CREATE TABLE Employees (
    employee_id NUMBER(5) PRIMARY KEY,
    last_name VARCHAR2(15) NOT NULL,
    ...
    department_id NUMBER(3) REFERENCES Departments)
    CLUSTER emp_dept (department_id);

CREATE TABLE Deapartments (
    department_id NUMBER(3) PRIMARY KEY, . . . )
    CLUSTER emp_dept (department_id);
```

Je vidět, že na rozdíl od příkazu CREATE TABLE, jak ho známe, obsahuje navíc klauzuli CLUSTER, která říká, do kterého shlukovaného souboru se má tabulka vložit a který sloupec je shlukujícím klíčem.

Přístup k záznamům shlukovaného souboru může být buď prostřednictvím indexu (implicitně) nebo využitím hašování. Ve druhém případě by se v příkazu CREATE CLUSTER objevily některé informace specifikující počet hodnot vyhledávacího klíče, případně hašovací funkci, podobně jako jsme viděli u hašování. My si ukážeme použití indexu. V takovém případě je potřeba vytvořit index, například příkazem:

```
CREATE INDEX emp_dept_index
ON CLUSTER emp_dept
...);
```

Je vidět, že se již nespecifikuje vyhledávací klíč, neboť je jím shlukující klíč.

Závěrem můžeme říci, že shlukování přináší následující dvě výhody:

- Snížení počtu diskových vstupně výstupních operací při spojování shlukovaných tabulek a tím zrychlení přístupu k datům při spojování
- Každá hodnota shlukovacího klíče je uložena na disku pouze jedenkrát, tj. neopakuje se již v záznamech tabulek, jak je vidět na Obr. 4.15.

Na druhé straně je pomalejší operace vkládání nových záznamů a shlukovat bychom neměli tabulky, jejichž záznamy jsou často zpřístupňovány individuálně.



Hašování se u Oracle nedefinuje pomocí příkazu CREATE INDEX (narozdíl například od SQL Base – viz Příklad 4.12), nýbrž právě příkazem CREATE CLUSTER s hašováním pro přístup k záznamům shlukovaného souboru. Takový shlukovaný soubor, který obsahuje pouze jednu tabulku, odpovídá v námi zavedené terminologii hašovanému souboru.

4.4.4. Bitmapový index

Když jsme si vysvětlovali podstatu uspořádaných indexů, viděli jsme, že položka indexu obsahuje hodnotu vyhledávacího klíče a ukazatele na záznamy, ve kterých se tato hodnota nachází. V případě bitmapových indexů je informace o tom, ve kterých záznamech se nachází jaká hodnota vyhledávacího klíče reprezentována jiným způsobem. Má podobu bitové mapy (vektoru) pro každou hodnotu vyhledávacího klíče, která se v indexované tabulce nachází. Každá pozice v mapě reprezentuje jeden řádek tabulky s konkrétní hodnotou identifikátoru řádku ROWID. Hodnota 1 na dané pozici značí, že v příslušném řádku tabulky je daná hodnota vyhledávacího klíče. Transformační funkce převádí pozice nastavených bitů na skutečné ROWID a poskytuje tak stejnou funkcionalitu jako uspořádaný index.

Příklad 4.14

$x+y$

Uvažujme tabulku Klient (r_cislo, jmeno, prijimova_skupina, mesto)

r_cislo	jmeno	prijimova_skupina	mesto
440726/0672	Jan Novák	S2	Brno
530610/4532	Petr Veselý	S3	Brno
601001/2218	Ivan Zeman	S1	Brno
510230/048	Pavel Tomek	S2	Brno
580807/9638	Josef Mádr	S5	Brno
625622/6249	Jana Malá	S2	Vyškov

a předpokládejme vytvoření bitmapového indexu pro sloupec *prijimova_skupina*. Bude tvořen čtyřmi bitovými mapami:

prijmová skupina

S1	001000
S2	100101
S3	010000
S5	000010

V prostředí databázového serveru Oracle by tento index mohl být vytvořen příkazem:

```
CREATE BITMAP INDEX IKlient_bm
ON Klient(prijmová_skupina);
```

Dále předpokládáme bitový index pro sloupec *mesto*:

mesto

Brno	111110
Vyškov	000001

Pokud je počet různých hodnot vyhledávacího klíče v tabulce malý, jsou bitmapové indexy velmi efektivní z hlediska paměťových nároků. Navíc umožňují efektivně využívat několik bitmapových indexů pro zodpovězení dotazů s několika podmínkami spojenými logickými spojkami AND nebo OR nebo obsahující negaci NOT. Efektivně lze rovněž provést výpočet agregační funkce COUNT(*) pro tabulku, která je reprezentovaná bitovou mapou vzniklou operacemi nad bitmapovými indexy.

Příklad 4.15

x+y

Uvažujme dotaz „Najdi klienty z Brna z příjmové skupiny S2“. Odpovídající příkaz SQL by mohl být:

```
SELECT *
FROM Klient
WHERE mesto='Brno' AND přijmová_skupina='S2'
```

Při použití bitmapových indexů z příkladu Příklad 4.14 by bitová mapa reprezentující výběr řádků jako výsledek dotazu vznikla bitovým logickým součinem příslušných bitmap, tedy $111110 \wedge 100101 = 100100$. Pozice s jedničkami by určily ROWID prvního a čtvrtého řádku tabulky, které by byly vybrány jako výsledek dotazu.

Nyní uvažujme dotaz „Kolik klientů z Brna patří do příjmové skupiny S2?“. Odpovídající příkaz SQL by mohl být:

```
SELECT COUNT(*)
FROM Klient
WHERE mesto='Brno' AND přijmová_skupina='S2'
```

Při použití bitmapových indexů z příkladu Příklad 4.14 by výsledek vznikl spočítáním počtu jedniček v bitové mapě reprezentující výběr definovaný zanořenou tabulkou, tedy $\text{počet_jedniček}(111110 \wedge 100101) = \text{počet_jedniček}(100100) = 2$.

Bitmapové indexy byly navrženy především pro použití v datových skladech, kde obsahují tabulky velké množství dat, počet různých hodnot ale bývá díky pohledu na některá data na vyšší úrovni hierarchie (například adresa-město-kraj-stát) relativně malý a počet souběžně pracujících uživatelů nebývá velký. Navíc protože datové

sklady slouží na podporu rozhodování, je u nich častá potřeba zodpovězení tzv. ad hoc dotazů, tj. dotazů, které vznikají neplánovaně na základě aktuální potřeby a jejich zodpovězení proto není naprogramováno předem. Za těchto podmínek přináší bitmapové indexy především tyto výhody:

- Urychlují zodpovězení řady ad hoc dotazů.
- Výrazně redukuje potřebný paměťový prostor ve srovnání s jinými indexačními technikami.
- Přináší výrazné zvýšení výkonnosti i na nepříliš výkonném hardware.

Na druhé straně bitmapové nejsou bitmapové indexy vhodné pro běžné databáze, ke kterým přistupuje souběžně řada uživatelů, resp. probíhá řada transakcí. Nejsou také, podobně jako hašování, vhodné pro sloupce, které dotazovány zejména na základě porovnávání použitím operátorů menší nebo větší.



V databázových systémech používáme pro zefektivnění přístupu k záznamům tabulek přístupové metody využívající uspořádaných indexů a hašování. V obou případech přístupová metoda na základě hodnoty vyhledávacího klíče vrátí ukazatel, resp. ukazatele na záznam, resp. záznamy tabulky se zadanou hodnotou vyhledávacího klíče. V této kapitole jsme se seznámili se základy indexování i hašování, ukázali jsme si nejpoužívanější indexační strukturu, která se v oblasti databází používá – B⁺strom, a vysvětlili si základní operace nad ním. Vysvětlili jsme si také podstatu shlukování záznamů s cílem snížení počtu nezbytných vstupně výstupních operací pro přístup k datům. Se shlukováním se můžeme setkat u datových souborů organizovaných jako index i u hašovaných souborů. Ukázali jsme si, že lze také shlukovat záznamy z více než jedné tabulky. Seznámili jsme se také s bitmapovým indexem, který se používá především při indexování dat v datových skladech.

4.5. Fyzický návrh databáze

V předchozích kapitolách jsme se seznámili se základy organizace dat v databázi na fyzické úrovni. Zaměřili jsme se zejména na metody, které nám systémy řízení báze dat nabízejí pro optimalizaci výkonnosti aplikací pracujících s daty v databázi. Cílem této kapitoly je ukázat, jaké faktory bychom měli vzít do úvahy při výběru nabízených metod a jak jich při rozhodování použít. Tuto úlohu řeší především návrhář ve fázi fyzického návrhu databáze, i když prvotní rozhodnutí, ze kterého již fyzický návrh databáze vychází, totiž volba SŘBD (pokud není zadavatelem předepsán), musí proběhnout podstatně dříve.

Ve fázi fyzického návrhu musí návrhář vycházet ze dvou zdrojů informací. Jednak jsou to nefunkční požadavky, zejména výkonnostní, kladené na vyvíjenou aplikaci, jednak je to zátěž představovaná databázovými příkazy, jejíž zvládnutí aplikace od SŘBD vyžaduje. *Výkonnostní požadavky*, které nás v této souvislosti zajímají, budou například maximální doba odezvy, počet současně pracujících uživatelů, vymezení operací, které považuje zadavatel za kritické z hlediska rychlosti provedení, četnost provádění různých operací apod. Je dobré, aby byly alespoň přibližně odhadnuty počty entit množin, které jsme identifikovali při konceptuálním návrhu. Ty nám umožní odhadnout velikost tabulek, počty různých hodnot v některých sloupcích apod.

Při *analýze zátěže*, kterou pro databázi bude představovat vyvíjená aplikace, musíme v

prvé řadě vzít do úvahy dotazy, přesněji příkazy, které tyto dotazy realizují. Dotazování je u databázových aplikací podstatně častější, než modifikace obsahu tabulek. V případě použití jazyka SQL se tedy zaměřujeme na příkazy SELECT. Budou nás zajímat tabulky, ze kterých se data vybírají, a hodnoty kterých sloupců se vybírají. Podstatné pro návrh přístupových metod budou podmínky pro výběr řádků tabulky, tedy podmínky v klauzuli WHERE a podmínky ve výrazech spojení JOIN. Důležité pro naše rozhodování bude nejen to, které sloupce se v podmínkách vyskytují, ale také s jakými operátory porovnání, a tzv. *selektivita dotazu*. Rozumíme jí relativní počet řádků, který se z tabulky vybere při dotazu na rovnost hodnoty. Selektivitu spočítáme z odhadu velikosti tabulky a počtu různých hodnot ve sloupci vyhledávacího klíče jako průměrný počet řádků připadající na jednu hodnotu vyhledávacího klíče. Je-li n_R počet řádků tabulky a $V(X, R)$ je počet různých hodnot ve sloupci X tabulky R , pak selektivitu spočítáme jako $n_R/V(X, R)$ (případně násobeno 100 v procentech). Uplatnit se mohou i sloupce v klauzulích GROUP BY a ORDER BY.

Podobně postupujeme i u operací, které modifikují obsah tabulek. Zde nás bude opět zajímat, kterých tabulek se operace týkají, jak vypadají podmínky klauzule WHERE a jak selektivní tyto podmínky jsou. Dále nás bude zajímat typ modifikace, tj. zda jde o příkaz INSERT, DELETE či UPDATE. Jak v případě dotazů, tak aktualizací zohledňujeme důležitost pro zákazníka, tj. zejména které operace považuje za kritické.

Po provedené analýze výkonnostních požadavků a zátěže můžeme přistoupit k rozhodování o použití některé z dostupných přístupových metod, která by měla zajistit splnění výkonnostních požadavků. Jednou z otázek, na které budeme hledat odpověď, je, pro které tabulky a které sloupce použít jakou přístupovou metodu. Nejčastěji zvažujeme, kdy a zda vytvořit index. Zásady pro volbu indexu by se daly shrnout do následujících bodů:

1. *Nevytvářej index, pokud nepřispěje ke zrychlení dotazů, resp. přístupu při aktualizaci.* Důležitými kritérii jsou zejména velikost tabulky a selektivita dotazu. Uspořádaný index v podobě B^+ stromu se vyplatí vytvářet jen pro rozsáhlejší tabulky a současně musí platit, že je selektivita malá, tj. pokud se dotazem vybere relativně malý počet řádků, neboli pokud počet různých hodnot ve sloupci vyhledávacího klíče je relativně velký. Malá tabulka, u níž lze očekávat, že záznamy budou uloženy v několika málo diskových blocích se indexovat nevyplatí. Podobně pokud bude selektivita vysoká, bude připadat na jednu hodnotu vyhledávacího klíče značná část primárního souboru a indexování se nevyplatí. Zpravidla se vyplatí indexovat, je-li selektivita v jednotkách procent (2 až 4%), přičemž tato hodnota může být vyšší, když se budou vybrat použitím indexu všechna data (například se využije uspořádání) nebo se index bude využívat při spojování s jinými tabulkami. V případě bitmapového indexu je tomu naopak, ten se vyplatí při vysoké selektivitě, tedy malém počtu různých hodnot ve sloupci.
2. *Kandidáty na vyhledávací klíče jsou sloupce v podmínkách klauzulí WHERE a podmínkách výrazu spojení JOIN.*
3. *Zvažuj uvážlivě použití složených vyhledávacích klíčů.* Složené vyhledávací klíče bychom měli zvažovat v situaci, kdy klauzule WHERE nebo výraz JOIN obsahuje podmínky pro více než jeden sloupec tabulky, jde o často prováděnou nebo kritickou operaci a sloupce, které by tvořily složený vyhledávací klíč se nepoužívají často samostatně.

4. *Shlukování může přispět ke zvýšení výkonnosti při častém spojování shlukovaných tabulek.* Musíme ale zvažovat případné přetečení bloků apod.
5. *Hašování je efektivnější než B⁺ strom pro porovnání na rovnost, není ale vhodné pro rozsahové dotazy.* Pokud se v podmínkách klauzule WHERE a výrazu JOIN vyskytuje porovnání na rovnost, je efektivnější hašování. Pro rozsahové dotazy je nutné použít B⁺ strom. Uvažujme následující dva dotazy:

```
SELECT *  
FROM Klient  
WHERE r_cislo = '440726/0672'  
a  
SELECT *  
FROM Transakce  
WHERE datum BETWEEN '1998-10-01' AND '1998-10-31'
```

První dotaz je dotaz s podmínkou na rovnost. Ten je řešitelný použitím jak indexu ve tvaru B⁺ stromu, tak hašování. Za předpokladu rozsáhlé tabulky a dobře nastavených parametrů (zejména počet různých hodnot vyhledávacího klíče, případně vhodná hašovací funkce) bude statické hašování efektivnější. V ideálním případě bude vyžadovat pouze načtení bloku se sektorem určeným hodnotou hašovací funkce. Při použití B⁺ stromu bude potřeba načíst tolik bloků indexu, jaká je výška stromu a jeden blok primárního souboru.

Ve druhém případě jde o rozsahový dotaz. Hašování je nevhodné, i když teoreticky by použit šlo také. Musel by se brát jeden den po druhém v uvedeném intervalu a zjišťovat, zda existuje v tabulce záznam s odpovídající hodnotou. Tento postup by byl obecně neefektivní a v případě, že by byl sloupec vyhledávacího klíče například typu FLOAT, i nerealizovatelný. Naproti tomu rozsahové dotazy nepředstavují pro B⁺ strom žádný problém, naopak uspořádaná sekvenční část umožňuje jejich efektivní provádění. Můžeme tedy říci, že hašování je výhodnější v těchto situacích:

- a. Pro podporu přirozeného spojení a spojení na rovnost.
- b. Existuje-li velmi významný dotaz na rovnost a nejsou rozsahové dotazy se sloupci vyhledávacího klíče.
- c. Lze-li určit potřebný prostor pro statické hašování.

V ostatních případech bychom měli dát přednost B⁺ strom, který je z hlediska použitelnosti obecnější.

6. *Po vytvoření seznamu žádoucích indexů zvažuj dopad na aktualizace.* Vysvětlili jsme si, že vkládání řádků do tabulky, rušení a aktualizace má za následek nutnost promítnutí změn v obsahu tabulky do indexů, které jsou pro danou tabulku vytvořeny i upravit případně seskupení záznamů při shlukování. Tato činnost se označuje jako údržba. Pokud údržba indexu zpomalí významné aktualizací operace, neměli bychom ho vytvářet. Musíme ale vzít v úvahu i fakt, že index urychluje prohledávací varianty příkazů UPDATE a DELETE. Musíme tedy vážit, jestli přínos indexu či shlukování pro urychlení přístupu k datům bude výrazně větší než ztráta způsobená režii údržby.
7. *Použij v případě potřeby dostupné prostředky pro zjištění způsobu provedení*

dotazu. Již několikrát jsme v této studijní opoře zmínili komponentu SŘBD zvanou optimalizátor zpracování dotazu, která se snaží najít co nejefektivnější způsob provedení databázové operace. Součástí dialektu SQL SŘBD bývá příkaz nebo existuje nějaký nástroj, který umožňuje zobrazit tzv. prováděcí plán, tj. způsob provedení dotazu, který optimalizátor vybral. Z něho například můžeme poznat, zda byl vybrán index, který jsme pro účely efektivního provedení vytvořili. Některé současné dialekty SQL také umožňují jako součást příkazů SELECT, INSERT, DELETE a UPDATE předat optimalizátoru pokyn (hint) např. pro použití indexu. O optimalizaci si řekneme na přednáškách něco později.

8. *Zvaž, zda by se případně vyplatilo modifikovat pro zvýšení výkonnosti logický návrh, tj. návrh tabulek*. V úvahu připadá například spojení některých tabulek, které by z hlediska kvality logického návrhu být samostatné. Tím se vyvarujeme nutnosti spojovat tabulky při dotazování, ale aplikace musí zajistit, že vzniklá redundance nezpůsobí při žádných operacích nekonzistenci.

Jiným příkladem může být opačný postup, kdy může být užitečné nějakou tabulku s velkým počtem sloupců vertikálně rozčlenit tak, aby v jedné tabulce byl primární klíč se sloupci, se kterými se pracuje při častých nebo kritických operacích, a ve druhé by byl primární klíč se sloupci zbývajících.

Další variantou může být přepis příkazů kritických operací nebo těch, které způsobují výkonnostní problémy, tak, aby byly rychlejší. Tato úprava však vyžaduje hlubší znalost a zkušenost konkrétního databázového prostředí, včetně použitých přístupů a strategií k optimalizaci zpracování dotazů.

Poměrně významný vliv na výkonnost databázového systému může mít používání deklarativních integritních omezení typu PRIMARY KEY, UNIQUE a FOREIGN KEY, neboť je s nimi vždy spojena nějaká režie při provádění aktualizací operací. Návrhář by proto měl vždy vážit, zda je vhodné ponechat tuto kontrolu na SŘBD a provádět ji při každé operaci, která může potenciálně integritu porušit, nebo zda zajistit, že nedojde k porušení integrity, případně provádět příslušné kontroly na aplikační úrovni.



V některých případech SŘBD vytváří indexy sám nebo index musí explicitně vytvořit uživatel a to bez ohledu na velikost tabulky a další kritéria, která jsme si uvedli. Typickým je unikátní index při zadání deklarativního integritního omezení PRIMARY KEY nebo UNIQUE.



V této části jsme se zabývali zajištěním výkonnostních ve fázi fyzického návrhu databáze. Před podobným problémem ale může stát návrhář nebo správce databáze i později po nasazení do provozu v situaci, kdy dochází k poklesu výkonnosti aplikace. I pro řešení těchto situací platí řada z výše uvedených zásad s tím, že mnohem významnější roli než ve fázi fyzického návrhu hraje využívání nástrojů pro ladění výkonnosti, jak je naznačeno v bodu 7.



V této kapitole jsme si vysvětlili, jaké zdroje informací využívá návrhář při fyzickém návrhu databáze. Jsou to jednak výkonnostní požadavky, které musí vyvíjený systém splňovat, jednak informace o zátěži, kterou představují databázové operace zajišťující požadovanou funkcionalitu. Uvedli jsme si také některá doporučení, která bychom měli vzít do úvahy při fyzickém návrhu databáze.

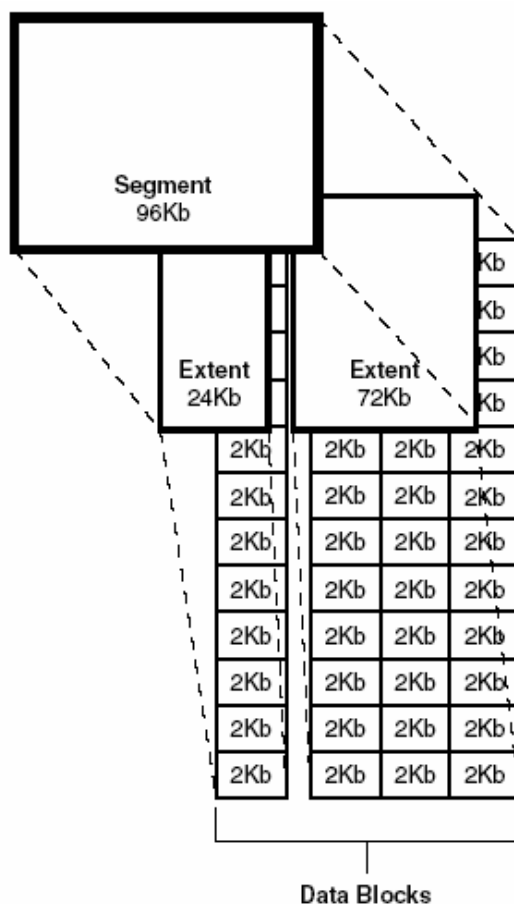
4.6. Organizace dat na fyzické úrovni u serveru Oracle 10g



V této závěrečné podkapitole části studijní opory zabývající se organizací dat v databázi na fyzické úrovni se velice stručně zmíníme o tom, jak vypadá organizace u jednoho z nejrozšířenějších SŘBD, databázového serveru Oracle, konkrétně její poslední verze Oracle 10g. Přestože organizace u jiných SŘBD bude více či méně odlišná, setkáme se i u nich s řadou pojmů, které si v následujících odstavcích vysvětlíme. Zpracování této kapitoly vychází z dokumentace na stránkách Oracle Documentation Library (<http://www.oracle.com/pls/db102/homepage>), zejména z dokumentu Oracle® Database Concepts. 10g Release 2 (10.2). Z tohoto pramene také byly převzaty některé obrázky uvedené v této podkapitole.

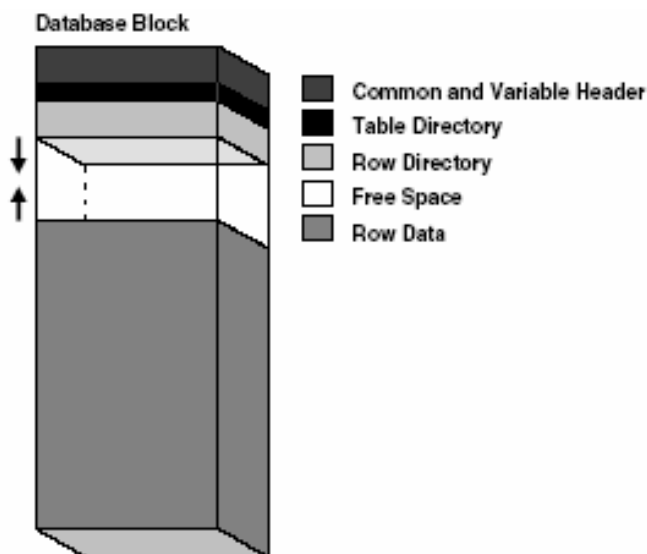
Architektura databáze Oracle se opírá o několik pojmů, které označují logické nebo fyzické paměťové struktury. Jsou to pojmy datový blok, extent, segment, tabulkový prostor, datový soubor a řídicí soubor. Datový blok, extent a segment jsou pojmy související s alokací databázového prostoru.

Datový blok je nejmenší jednotka paměťového prostoru v databázi. Velikost datového bloku se nastavuje inicializačním parametrem při vytváření databáze. Jeho velikost by měla být celočíselným násobkem diskového bloku hostitelského operačního systému. Prostor databáze je alokován v jednotkách nazývaných *extent*. Jsou to souvislé paměťové oblasti tvořené několika bloky. Množina extentů, které tvoří paměťový prostor pro uložení dat konkrétního databázového objektu, například konkrétní tabulky, se nazývá *segment*. Vztah těchto tří pojmů ilustruje Obr. 4.16.



Obr. 4.16 Vztah pojmů datový blok, extent a segment

Každý datový blok obsahuje určitý prostor pro uložení některých systémových informací a prostor pro uložení dat. Formát bloku se může lišit podle typu segmentu (datový, indexový apod.), ale v zásadě vypadá, jak je uvedeno na Obr. 4.17.



Obr. 4.17 Formát datového bloku

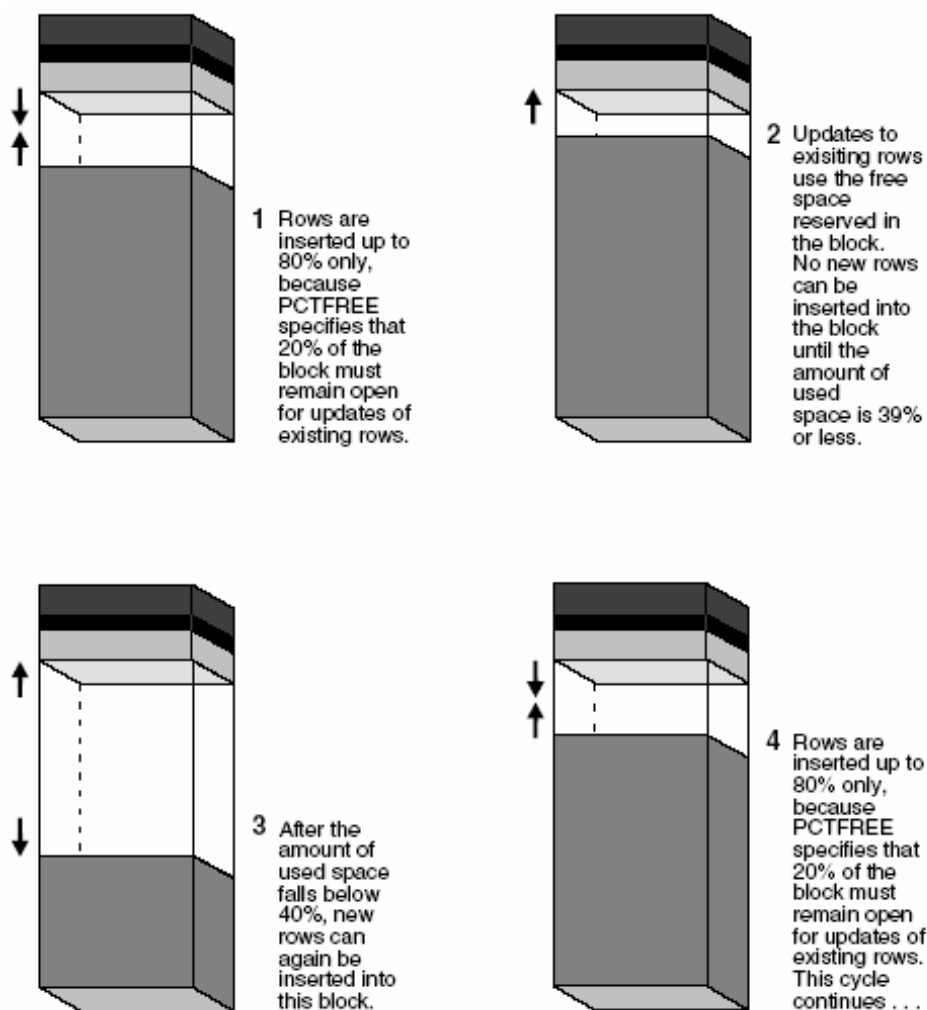
Záhlaví (header) obsahuje základní informaci o bloku, jako je jeho adresa a typ segmentu, jehož je součástí, *Adresář tabulky* (table directory) obsahuje informace o tabulce (v případě shlukování případně o několika tabulkách), data jejichž řádků jsou v bloku uložena. *Adresář řádků* (row directory) obsahuje informaci o datech řádků v bloku včetně adresy, jak jsme si vysvětlovali v kapitole 4.2. Zbytek prostoru datového bloku slouží k uložení dat databázového objektu ukládaného do daného segmentu.

Využívání prostoru pro data je řízeno dvěma parametry. Parametr PCTFREE udává minimální prostor (v procentech), který musí zůstat po vkládání nových záznamů odpovídajících provedení příkazu SQL INSERT, na aktualizace záznamů v bloku již uložených příkazem UPDATE. Je třeba si uvědomit, že záznamy mohou obsahovat datové typy proměnné délky (např. VARCHAR) a prostor pro uložení konkrétní hodnoty se při aktualizaci může změnit. Implicitní hodnota je 10%. Druhým parametrem je parametr PCTUSED, který definuje úroveň zaplnění bloku (opět v procentech), kdy je možné do bloku opět začít vkládat nové záznamy. Implicitní hodnota je 40%. Význam obou parametru ilustruje Obr. 4.18.

Hodnoty parametrů v obrázku jsou PCTFREE = 20 a PCTUSED = 40. Po alokaci nového bloku je prostor pro vkládání nových záznamů 80% z celkového prostoru pro ukládání dat databázového objektu. Jakmile je překročena tato hranice, není již blok pro vkládání nových záznamů dostupný a zbývající prostor je ponechán na pokrytí případného růstu vlivem aktualizací. Pokud rušením záznamů, případně aktualizacemi klesne zaplnění prostoru pro data pod hodnotu PCTUSED, tj. 40% v tomto případě, mohou být opět vkládány do bloku nové záznamy.

Databázový server ukládá data logicky do *tabulkových prostorů* a fyzicky do *datových souborů*, které jsou asociovány s odpovídajícím tabulkovým prostorem. Databáze Oracle je vždy tvořena jedním nebo několika tabulkovými prostory. Vztah pojmů tabulkový prostor a datový soubor ukazuje Obr. 4.19. Je zřejmé, že tabulkový prostor se může rozprostírat přes několik souborů operačního systému, ale v souboru jsou vždy segmenty pouze jednoho tabulkového prostoru. Zároveň je vidět, že segment,

tedy například data nějaké tabulky mohou být uložena ve více než jednom datovém souboru. V obrázku je také naznačena existence různých typů segmentů, především datových pro uložení dat tabulek a indexových pro uložení indexů.

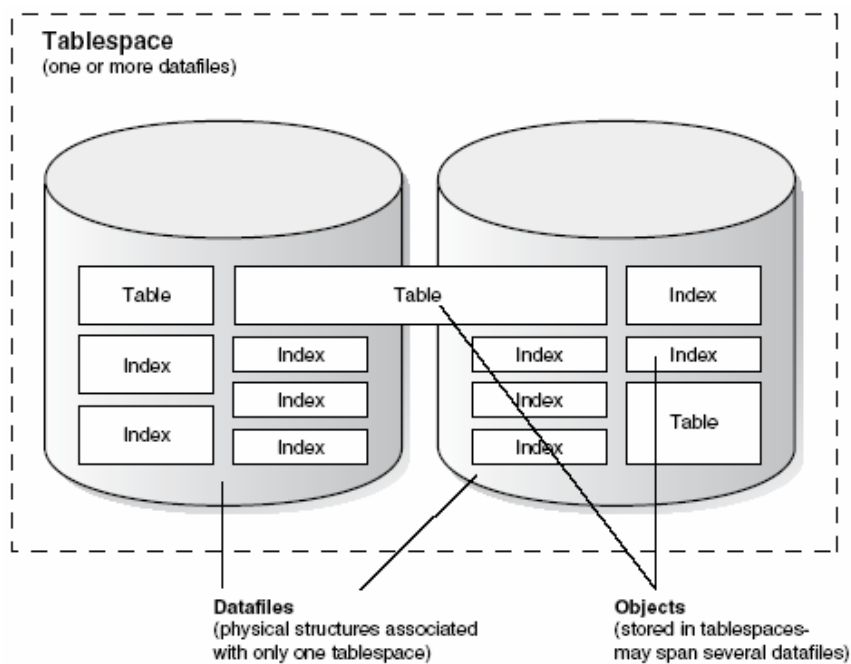


Obr. 4.18 Řízení volného prostoru bloku parametry PCTFREE (20) a PCTUSED (40)

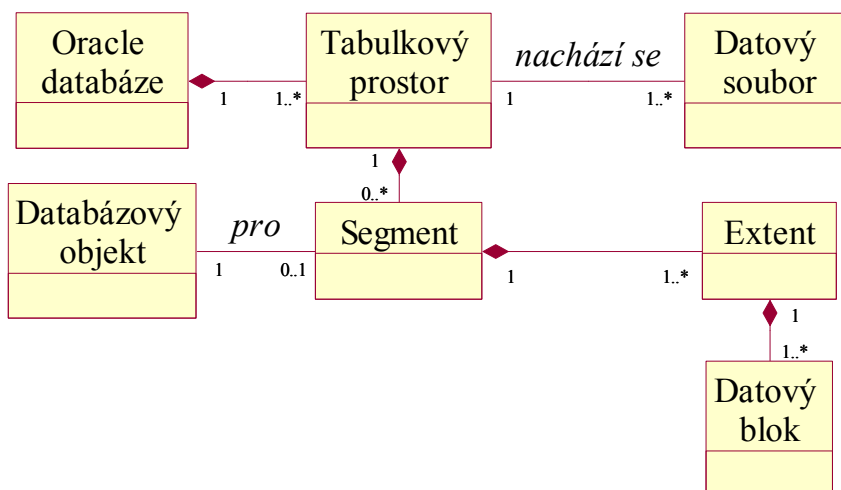
Každá databáze obsahuje tabulkový prostor SYSTEM, který obsahuje tabulky systémového katalogu. Kromě něho existují některé další pomocné tabulkové prostory využívané systémem a lze vytvořit tabulkové prostory další, např. pojmenovaný TEMP, který se dá k dispozici systému jako pracovní prostor pro přechodné ukládání dat, tabulkový prostor, USER pro ukládání uživatelských dat apod.

Pokud je potřeba zvětšit celkový prostor databáze, lze to provést jedním ze tří způsobů. Buď se zvětší prostor tabulkového prostoru přidáním dalšího souboru nebo se přidá další paměťový prostor nebo se zvětší velikost některého z používaných datových souborů.

Vztahy všech zatím vysvětlených pojmů ještě souhrnně znázorňuje Obr. 4.20.



Obr. 4.19 Vztah tabulkového prostoru a datových souborů



Obr. 4.20 Vztahy pojmů fyzické úrovně databáze Oracle

x+y

Příklad 4.16

Uvažujme následující příklad pro vytvoření shlukovaného souboru:

```
CREATE CLUSTER emp_dept (department_id NUMBER(3))
  SIZE 600
  TABLESPACE users
  STORAGE (INITIAL 200K
    NEXT 300K
    MINEXTENTS 2
    MAXEXTENTS 20
    PCTINCREASE 33);
```

Shlukovaný soubor jako databázový objekt se bude vytvářet v tabulkovém prostoru USERS, počáteční extent bude mít velikost 200KB (INITIAL), na začátku budou vytvořeny dva takové extenty (MINEXTENTS), třetí extent, který bude alokovaný po zaplnění prvních dvou, bude mít velikost 300KB (NEXT) a každý další bude mít o 33% (PCTINCREASE) větší velikost než předchozí, tj. čtvrtý 400KB atd. Maximální počet alokovaných extentů je 20 (MAXEXTENTS).

Posledním pojmem, který jsme si ještě nevysvětlili, jsou *řídící soubory (control file)*. Obsahují informace důležité pro činnost SŘBD ve vztahu k dané databázi, například jméno, datum a čas vytvoření databáze, jména používaných datových souborů, informace o tabulkových prostorech, informace důležité pro zotavení, zálohování apod.

V závěru této kapitoly ještě zmíníme metody přístupu k datům, které server Oracle 10g nabízí. Kromě vždy existujícího úplného procházení souboru (full scan) jsou to především tyto:

- *B⁺ strom*
- *Indexovaný shlukovaný soubor (B-tree cluster index)* - B⁺ strom se záznamy tabulek v uzlech sekvenční části. V kapitole 4.4.3 jsme si ukázali, že shlukovat lze i záznamy několika tabulek na základě společného shlukujícího klíče.
- *Hašovaný soubor (hash cluster index)*
- *Bitmapový index*

Index v podobě B⁺ stromu a bitmapový index lze navíc vytvořit nejen pro vyhledávací klíč tvořený jedním nebo několika sloupci tabulky, ale i pro hodnoty, které vzniknou vyčíslením nějakého výrazu. Takové indexy se zde nazývají *indexy založené na funkci (function-based index)*. Například index vytvořený příkazem:

```
CREATE INDEX uppercase_idx ON Klient (UPPER(jmeno));
```

kde UPPER je SQL funkce převodu na velká písmena, může urychlit vyhledávání bez uvažování malých/velkých písmen (case-insensitive search), např. v dotazu

```
SELECT * FROM Klient WHERE UPPER(jmeno) = 'Jan Novák';
```



V této kapitole jsme si vysvětlili některé základní pojmy související s organizací dat na fyzické úrovni u databázového serveru Oracle 10g. Jde sice o kapitolu rozšiřující, jejíž obsah není předmětem zkoušky, přesto může být užitečným doplňkem obecnějšího pohledu na problematiku organizace dat v databázi na fyzické úrovni, jak byla prezentována v předchozích kapitolách. Navíc se s podobnou terminologií můžete setkat i u jiných systémů řízení báze dat.



Informace k organizaci dat v databázi na fyzické úrovni můžete najít v základní literatuře [Pok98] na stranách 27 až 34 a 81 až 88. Je zde popsáno indexování a hašování, B-stromy a bitmapové indexy. V učebnici [Sil05] je problematice organizace datových souborů věnována část kap. 11 na stranách 464 až 475 a indexování a hašování v kap. 12 na stranách 481 až 525. K problematice fyzického návrhu databáze lze najít poznámky a některá doporučení v učebnici Ramakrishnan, R.: Database Management Systems. WCB/McGraw-Hill, 1998 na stranách 436 až 468. Řadu užitečných informací a praktických doporučení lze nalézt i v dokumentaci k databázovému serveru Oracle [Oracle], především v dokumentu Oracle® Database Concepts.

**Cvičení:**

- C4.1** Vysvětlete, jak probíhá v databázovém systému zpracování dotazu z hlediska přístupu k datům, tj. vysvětlete základní kroky od příjmu dotazu SŘBD po předání výsledku.
- C4.2** Vysvětlete možné způsoby uložení záznamů proměnné délky v databázových souborech.
- C4.3** Vysvětlete rozdíl mezi neuspořádaným, uspořádaným a hašovaným souborem.
- C4.4** Vysvětlete, jak se u uspořádaného souboru zajišťuje uspořádání při vkládání nových záznamů, rušení a modifikaci existujících.
- C4.5** Vysvětlete význam identifikátoru řádku (RID, ROWID) při organizaci dat na fyzické úrovni.
- C4.6** Vysvětlete, k čemu slouží SŘBD vyrovnávací paměť a jaké požadavky jsou na ni kladeny.
- C4.7** Vysvětlete, proč SŘBD vracuje s vlastní vyrovnávací pamětí a nevyužívá vyrovnávací paměť spravovanou operačním systémem.
- C4.8** Vysvětlete podstatu uspořádaného indexu.
- C4.9** Vysvětlete rozdíl mezi hustým a řídkým uspořádaným indexem.
- C4.10** Vysvětlete kolik hustých a kolik řídkých indexů může existovat pro jednu databázovou tabulku.
- C4.11** Charakterizujte B⁺ strom jako indexovou strukturu a uveďte jeho podstatné vlastnosti.
- C4.12** Uvažujte B⁺ strom pro množinu {2, 3, 5, 7, 11, 17, 19 23, 29, 31} hodnot vyhledávacího klíče. Nakreslete B⁺ strom, vejdou-li se do uzlu maximálně 4 ukazatelé a v indexu mají být hodnoty uspořádány vzestupně. Zachyťte strom v takovém stavu, kdy není žádný uzel plně obsazen.
- C4.13** Uvažujte B⁺ strom pro množinu {2, 3, 5, 7, 11, 17, 19 23, 29, 31} hodnot vyhledávacího klíče. Nakreslete B⁺ strom, vejdou-li se do uzlu maximálně 4 ukazatelé a v indexu mají být hodnoty uspořádány sestupně. Zachyťte strom v takovém stavu, kdy není žádný uzel plně obsazen.
- C4.14** Uvažujte B⁺ strom pro množinu {2, 3, 5, 7, 11, 17, 19 23, 29, 31} hodnot vyhledávacího klíče. Zkonstruuje B⁺ strom, vejdou-li se do uzlu maximálně 4 ukazatelé a v indexu mají být hodnoty uspořádány vzestupně. Předpokládejte, že na začátku byl strom prázdný a postupně do něho vkládejte náhodně vybrané hodnoty z uvedené množiny.
- C4.15** Na B⁺ stromu z příkladu C4.12 vysvětlete vyhledání záznamu tabulky s hodnotou vyhledávacího klíče 19.
- C4.16** Na B⁺ stromu z příkladu C4.12 vysvětlete vyhledání záznamu tabulky s hodnotou vyhledávacího klíče z intervalu <5, 15>.
- C4.17** Vysvětlete operaci štěpení uzlu B⁺ stromu, kdy k ní dochází a jak probíhá.
- C4.18** Vysvětlete operace slévání uzlů B⁺ stromu a redistribuce, kdy k nim dochází a jak probíhají.
- C4.19** Uvažujte, že existuje index ve tvaru B⁺ stromu pro sloupec r_cislo tabulky Klient z příkladu na přednáškách. Uzel indexu obsahuje maximálně 100 ukazatelů. Výška stromu je 3. Kolik bloků souboru se záznamy tabulky Klient je potřeba načíst při postupném prohlédnutí celého souboru (full scan), např. pro vyhledávání podle města, obsahuje-li blok souboru průměrně 25 záznamů. Váš výsledek musí být co nejpřesnější s ohledem na zaplnění uzlů B⁺ stromu.
- C4.20** Vysvětlete podstatu hašování. Jaké požadavky jsou kladený na hašovací funkci.
- C4.21** Vysvětlete rozdíl mezi přímým a nepřímým hašováním.

- C4.22** Vysvětlete rozdíl mezi statickým a dynamickým hašováním.
- C4.23** Vysvětlete podstatu rozšiřitelného hašování.
- C4.24** Vysvětlete, co se rozumí bitmapovým indexem a kdy je vhodné takový index použít.
- C4.25** Vysvětlete podstatu shlukování záznamů a uveďte, jak souvisí s indexováním a hašováním.
- C4.26** Uveďte, jak lze popsat zátěž, kterou analyzujeme v rámci fyzického návrhu databáze.
- C4.27** Vysvětlete, jaké faktory je potřeba vzít do úvahy při rozhodování o vytvoření indexu a použití hašování.
- C4.28** Uveďte, kdy je výhodnější použít hašování než indexování.
- C4.29** Vysvětlete pojem selektivita dotazu. Uveďte, jak ji při rozhodování o vytvoření indexu určíte, jaká by měla být pro efektivní použití B+ stromu a jaká pro použití bitmapového indexu. Svoje tvrzení zdůvodněte.



Test

- T1.** Uvažujte tabulku *Studenti*, ve které je primárním klíčem sloupec *c_studenta*. Pro tento sloupec tabulky existuje index *ICSTUD* ve tvaru B⁺ stromu (parametr *n* tohoto *n*-nárního stromu je 11). Jeden blok primárního souboru obsahuje maximálně 5 záznamů pro 5 řádků tabulky. V daném okamžiku má tabulka 1000 řádků. B+ strom bude mít při této velikosti tabulky:
- a) 3 úrovně
 - b) buď 3 nebo 4 úrovně podle zaplněnosti uzlů
- T2.** Mezi vlastnosti B+stromu patří, že:
- a) každý uzel, včetně kořene, je zaplněn minimálně z 50%
 - b) pouze každý listový uzel je zaplněn minimálně z 50%
- T3.** Pro jednu tabulku lze vytvořit:
- a) jen jeden řídký a libovolné množství hustých indexů
 - b) libovolné množství řídkých indexů a jen jeden hustý
- T4.** Režie související s údržbou indexů tabulky se týká:
- a) jen příkazů UPDATE a DELETE
 - b) příkazů UPDATE, DELETE a INSERT
- T5.** Bitová mapa bitmapového indexu udává
- a) které hodnoty vyhledávacích klíčů se nachází v daném řádku tabulky
 - b) ve kterých rádcích tabulky jsou neprázdné hodnoty

5. Transakční zpracování



Cíl: V této kapitole se seznámíte se základy transakčního zpracování v databázích a to z hlediska vlastností transakce a způsobů jejich zajištění. Získané poznatky uplatníte jak na pozici vývojáře, tak správce databáze.

Anotace: Databázová transakce, vlastnosti a stavy transakce, zotavení po poruše, žurnál (denník transakce), zotavení s okamžitou modifikací, kontrolní bod, obnova stavu a zotavení po poruše disku, řízení souběžného přístupu k datům, plán souběžných transakcí, sériový plán, uspořádatelný plán, uzamykací protokoly, příkazy pro transakční zpracování v SQL.

Prerekvizitní znalosti: V této kapitole se předpokládá znalost hierarchie paměti používaných v počítačích a jejich vlastností. Dále je předpokládá znalost základů binárních relací a teorie grafů. Navážeme také na znalosti získané dosud v tomto předmětu, například využijeme našich znalostí o vyrovnávací paměti, kterou používá SRBD při přístupu k disku.



Odhad doby studia: 10 hodin.

5.1. Úvod

Dosud jsme předpokládali, že aplikační programátor vystačí při realizaci databázového systému se čtyřmi základními databázovými operacemi – výběrem, vložením, zrušením a modifikací informace. V praxi se ale setkáváme se situacemi, kdy potřebujeme seskupit několik takových elementárních operací do většího celku, který by se měl chovat jako jedna operace. Typickým příkladem může být převod částky z jednoho účtu na jiný v bankovním informačním systému. Operace převodu bude sestávat v získání záznamu řádku zdrojového účtu, snížení hodnoty stavu na účtu o převáděnou částku, uložení modifikovaného záznamu, načtení záznamu řádku cílového účtu, zvýšení hodnoty stavu na účtu o převáděnou částku a uložení modifikovaného záznamu cílového účtu. Přitom celá tato posloupnost dílčích operací musí proběhnout jako celek, jako nedělitelná operace. Takovou posloupnost elementárních databázových operací označujeme jako *transakci*.

K tomu, aby byla zajištěna integrita dat v databázi, musí mít transakce určité vlastnosti. My jsme si uvedli jen jednu z nich – nedělitelnost neboli *atomičnost*. Od SRBD očekáváme, že tyto vlastnosti transakce zajistí.

Zatím jsme v tomto předmětu neuvažovali, že v průběhu provádění databázových operací může docházet k chybám a poruchám, jejichž následkem může dojít k porušení konzistence dat v databázi. Podobně jsme se nezabývali tím, jestli mohou vznikat nějaké problémy, pokud bude k datům v databázi přistupovat současně několik uživatelů nebo aplikací. Uvidíme, že toto jsou dvě situace, pro které musí SRBD zajistit, že nedojde při provádění posloupnosti elementárních operací vlivem jejího nedokončení nebo vlivem ovlivňování jinou probíhající transakcí k porušení konzistence, tedy k narušení správnosti dat. SRBD musí být schopen provést *zotavení po poruchách* a musí zajistit *řízení souběžného přístupu* transakcí k datům.

Nejprve se blíže podíváme na vlastnosti databázové transakce a na stavy, ve kterých se transakce od svého spuštění do úplného dokončení může nacházet. Poté si ukážeme, jakým způsobem provádí SŘBD zotavení po chybách a poruchách. V této fázi budeme předpokládat, že neprobíhá více souběžných transakcí, tj. například bude probíhat pouze jedna bankovní transakce. V další části této kapitoly si naopak ukážeme, jak SŘBD zajišťuje řízení souběžně probíhajících transakcí k datům a budeme předpokládat, že nedochází k poruchám. Teprve potom obě tyto situace zkombinujeme a ukážeme si, jak proběhne zotavení po poruše, pokud v okamžiku poruchy probíhalo několik souběžných transakcí.

Přestože v této kapitole budeme věnovat velkou pozornost vnitřnímu fungování SŘBD, v závěru si ukážeme, jaké příkazy poskytuje programátorovi SQL pro ohraničení příkazů, které tvoří transakci, a vysvětlíme si, jaké další prostředky má programátor k dispozici pro ovlivnění průběhu provádění transakcí.

5.2. Transakce

Databázovou transakcí budeme rozumět logickou jednotku provádění programu operujícího s daty v databázi. Je tvořena posloupností databázových operací, které čtou a modifikují data v databázi. Její základní vlastností, jak už víme, je atomičnost.

Aby byla zajištěna integrita dat v databázi, požadujeme od transakce ještě některé další důležité vlastnosti. Nejčastěji se používá model transakce, u kterého je skupina základních vlastností označena zkratkou ACID a hovoříme o *ACID transakci*. Zkratka vznikla z prvních písmen anglických názvů čtveřice vlastností, které u transakce očekáváme a které tedy musí SŘBD zajistit. Jsou to tyto vlastnosti:

- atomičnost (Atomicity)
- konzistence (Consistency)
- izolace (Isolation)
- trvalost (Durability)

Atomičnost znamená, že transakce představuje z hlediska provedení nedělitelný celek. Buď jsou provedeny všechny dílčí databázové operace, které ji tvoří, nebo není provedena žádná z nich. Pokud budeme například uvažovat transakci, která převádí nějakou částku z jednoho účtu na druhý, není možné, aby například vlivem poruchy, jejímž následkem bylo nutné restartovat počítač nebo databázový server byla provedena pouze část operací. Z pohledu změn realizovaných v databázi touto transakcí musí platit, že po jejím provedení musí být součet stavů na obou účtech stejný jako před jejím provedením. Každý jiný stav znamená, že databáze se dostala do nekonzistentního stavu. Pokud tedy dojde k tomu, že vlivem již zmíněné poruchy byla provedena jen část transakce, například pouze snížení stavu zdrojového účtu, musí to SŘBD poznat a v rámci procesu zotavení po restartu transakci zrušit, tj. nastavit původní hodnotu stavu zdrojového účtu a uvést tím databázi do stavu před zahájením transakce.

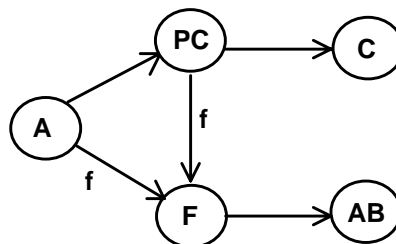
Konzistence znamená, že izolovaná transakce, tedy taková, která není ovlivňována žádnými jinými souběžně probíhajícími transakcemi, neporušuje konzistenci databáze. Musí tedy platit, že pokud byla databáze v konzistentním stavu před zahájením transakce, bude v konzistentním stavu i po skončení transakce. Zodpovědnost za zajištění této vlastnosti transakce nese v první řadě aplikační programátor, který

transakci programuje, neboť on definuje v tomto smyslu chování transakce. Pokud by programátor při programování transakce pro převod částky mezi účty na cílový účet převedl jinou částku, než odečetl z účtu zdrojového, bude z pohledu SŘBD všechno v pořádku. Přesto by data v databázi nebyla správná. Vlivem poruch a vzájemným ovlivňováním se souběžných transakcí by ale mohl být stav databáze po provedení i dobře naprogramované transakce nekonzistentní. Proto musí SŘBD zajišťovat i tuto vlastnost.

Izolace transakce znamená, že pokud bude probíhat několik transakcí souběžně, SŘBD zajistí, že transakce budou z hlediska provádění operací s daty v databázi izolované. Tedy efekt jejich operací bude takový, jako kdyby neprobíhaly souběžně, ale jedna za druhou. Formálněji bychom mohli říci, že pro každou dvojici souběžných transakcí T_i a T_j se T_i jeví, že T_j skončila dříve, než T_i zahájila provádění nebo T_j zahájila provádění až poté, co T_i skončila. Uvidíme, že pokud by souběžné transakce mohly libovolně přistupovat k datům v databázi, číst a modifikovat jejich hodnoty, mohla by se databáze dostat do nekonzistentního stavu. Proto musí SŘBD přístup souběžných transakcí k datům řídit tak, aby si žádná transakce nebyla vědoma, že probíhá současně s ní nějaká jiná.

Trvalost transakce znamená, že poté, co transakce úspěšně skončí, budou mít všechny změny v databázi, které transakce provedla, trvalý charakter a to i při výpadku systému. Zajištění této vlastnosti opět souvisí s nebezpečím ztráty některých dat až po skončení transakce vlivem poruchy vedoucí k restartu počítače či databázového serveru. SŘBD musí v procesu zotavení trvalost zajistit.

Transakce může proběhnout úspěšně nebo může být zrušena. Zrušena může být buď způsobem naprogramovaným programátorem v rámci ošetření nějaké chyby, která se může v průběhu provádění transakce vyskytnout, nebo ji může zrušit SŘBD v průběhu zotavení nebo při výskytu nějaké chyby, kterou neošetřuje aplikace, nýbrž systém. Od okamžiku zahájení provádění proto transakce postupně prochází některými ze stavů znázorněných na Obr. 5.1.



Obr. 5.1 Stavy transakce

Stav *aktivní* (A - *active*), představuje počáteční stav, který reprezentuje zahájení provádění transakce. Setrvává v něm do okamžiku provedení poslední operace, která transakci tvoří, nebo do okamžiku, kdy došlo k poruše nebo chybě, která vyžaduje zrušení transakce, tedy uvedení databáze do stavu před zahájením transakce. V prvním případě se transakce dostane do stavu *částečného potvrzení* (PC - *partial commit*). Tento stav znamená, že sice byly úspěšně provedeny všechny operace tvořící transakci, ale SŘBD ještě neprovedl všechny činnosti potřebné k zajištění trvalosti změn. Transakce je proto z pohledu SŘBD stále nedokončená a pokud dojde k poruše, bude při zotavení transakce zrušena. Pokud k poruše nedojde, dostane se poté, co SŘBD provede všechny potřebné operace k zajištění trvalosti změn provedených v databázi, do stavu *potvrzená* (C - *commit*). Transakce úspěšně proběhla.

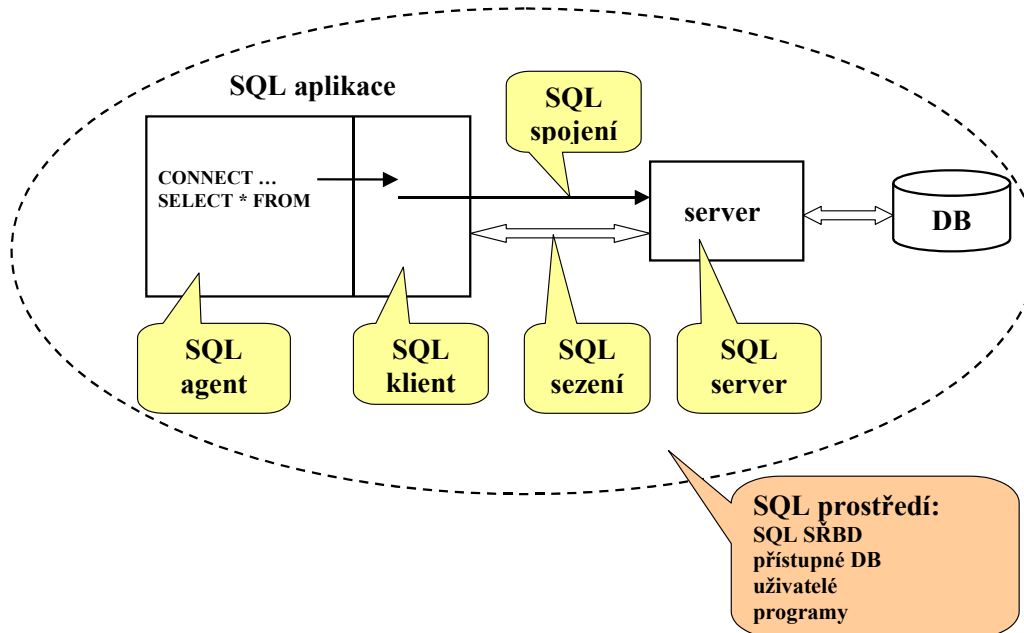
Zmíněné situace, kdy nastává přechod vlivem chyby či poruchy ze stavu A nebo PC,

jsou v obrázku označeny písmenem *f* (fault). Řekli jsme si, že za této situace musí SŘBD transakci zrušit. Než proběhnou operace anulující všechny změny provedené v databázi, setrvává transakce ve stavu *porucha* (F - *fault*). Proces anulování změn v databázi se označuje jako *rollback*. Po jeho ukončení se transakce dostává do druhého možného koncového stavu - *zrušená* (AB – *abort*). Transakce byla zrušena.

5.3. Transakce v jazyce SQL

Dříve, než si začneme vysvětlovat, jakým způsobem zajišťuje SŘBD požadované vlastnosti transakce, podívejme se, jakou podporu pro používání transakcí poskytuje jazyk SQL.

Začneme poněkud širším pohledem na to, jakým způsobem databázová aplikace v pojetí standardu komunikuje se SŘBD. Standard SQL-92 zavádí pojem *SQL prostředí*. Rozumí se jím v první řadě SŘBD, který je schopen provádět příkazy jazyka SQL, dále přístupné databáze spravované takovým SŘBD, aplikace, které služeb SŘBD využívají, a uživatelé, kteří používají tyto aplikace. Situaci ilustruje Obr. 5.2, který ukazuje některé další pojmy, které standard zavádí. Především je to *SQL server*, kterým rozumí již zmíněný SŘBD schopný provádět příkazy SQL. Dále je to *SQL aplikace*. Tou se rozumí každá aplikace, která komunikuje s SQL serverem prostřednictvím příkazů jazyka SQL. SQL aplikace je tvořena dvěma částmi. Část, která obsahuje příkazy SQL, se nazývá *SQL agent* a část, která je zodpovědná za zprostředkování komunikace s SQL serverem *SQL klient*. Komunikace klienta se serverem se uskutečňuje prostřednictvím tzv. *SQL spojení* (*SQL connection*). Každé spojení zahajuje tzv. *SQL sezení* (*SQL session*).



Obr. 5.2 SQL prostředí

Spojení se v jazyce SQL vytváří příkazem CONNECT, který má tvar:

```
CONNECT TO {DEFAULT|string1[AS string2][USER string3]}
```

Řetězec *string1* identifikuje SQL server, řetězec *string2* pojmenovává spojení a řetězec *string3* identifikuje uživatele, tj. obsahuje přihlašovací jméno a heslo. Klíčové

slovo `DEFAULT` slouží k vytvoření spojení, které je někde (například v konfiguračních parametrech SQL klienta) nastaveno jako implicitní.

Provedením příkazu `CONNECT` se stává vytvořené spojení aktivní, tj. všechny následující SQL příkazy jsou k provedení předávány připojenému serveru. V rámci SQL sezení lze vytvořit několik spojení, ale pouze jedno z nich může být v daném okamžiku aktivní. Nastavit jiné dříve vytvořené spojení jako aktivní lze příkazem `SET CONNECTION` tvaru:

```
SET CONNECTION TO {DEFAULT | string}
```

Provedením příkazu se aktivním spojením stane spojení implicitní nebo spojení uvedeného jména.

Ukončit existující spojení a tedy i sezení na tomto spojení lze buď explicitně nebo implicitně. V prvním případě se použije příkaz `DISCONNECT` tvaru:

```
DISCONNECT {DEFAULT | CURRENT | ALL | string}
```

kterým se ukončí implicitní, aktuální aktivní, všechna nebo jen pojmenované spojení. Implicitně dojde k ukončení všech vytvořených spojení ukončením aplikace.



Způsob připojování se k databázovému serveru se u různých systémů a dialektů SQL liší. Proto není potřeba, abyste si pamatovali přesný tvar příkazů podle standardu. Podstatné je především to, abyste si uvědomili, že dříve, než aplikace může vytvářet databázové objekty či manipulovat s daty v databázi, musí se k databázi nejprve připojit a to prostřednictvím databázového serveru, který databázi spravuje.

Nyní se podívejme, co říká standard SQL o transakcích. V první řadě říká, že každý SQL příkaz sám o sobě je atomický. Pokud tedy například probíhá rušení řádků nějaké tabulky použitím prohledávací varianty příkazu `DELETE`, pak musí SŘBD zajistit, že v případě vzniku poruchy v průběhu provádění tohoto příkazu nebude po restartu databáze v nekonzistentním stavu proto, že nebyly zrušeny všechny řádky, které měly být.

Víme ale, že transakce je posloupnost databázových operací, a proto může být tvořena několika příkazy jazyka SQL. V našem příkladě převodu částky mezi účty by takovou transakci mohly tvořit dva příkazy `UPDATE`. Z tohoto důvodu musí existovat způsob, jak říci, kde transakce začíná a kde končí. Jazyk SQL nezavádí žádný speciální příkaz pro zahájení transakce. Zahájení je implicitní a to takovým způsobem, že SQL agent začne provádět příkaz SQL, který má schopnost inicializovat transakci, a nemá žádnou transakci zahájenou. Schopnost inicializovat transakci má většina příkazů, se kterými jsme se v tomto předmětu seznámili. Výjimkou z těchto příkazů, které známe, jsou například příkazy `CONNECT`, `SET CONNECTION`, `DECLARE CURSOR`. V každém případě může být transakce inicializovaná jakýmkoliv příkazem DDL a příkazy `SELECT`, `INSERT`, `UPDATE` a `DELETE`.

Důležitá je ale druhá část výše uvedené věty „a nemá žádnou transakci zahájenou“. Z této podmínky vyplývá, že transakce nelze zanořovat. Používá se tedy pouze jednoúrovňový (v angličtině *flat*) model transakce.

Ukončit transakci lze jedním ze dvou následujících příkazů:

```
COMMIT  
ROLLBACK
```

Příkaz COMMIT ukončuje transakci a označuje ji jako úspěšně provedenou, tj. v diagramu stavů transakce na Obr. 5.1 vyvolá přechod do stavu částečného potvrzení PC a následně, pokud nedojde k poruše, do stavu potvrzená C. Naopak příkaz ROLLBACK ukončuje transakci s tím, že požaduje anulovat všechny případné změny, které transakce v databázi až po tento příkaz provedla. V diagramu stavů tedy vyvolá přechod do stavu porucha F a následně po provedení operace rollback do stavu zrušená AB.

Implicitně transakce také končí (implicitní příkaz COMMIT) ukončením SQL sezení SQL agentem, tedy při ukončení spojení.

V praxi příkazy COMMIT a ROLLBACK používáme takovým způsobem, že pokud chceme prohlásit transakci za úspěšně provedenou a změny provedené v databázi za trvalé, použijeme příkaz COMMIT, zatímco pokud vznikne nějaká logická chyba, která brání úspěšnému pokračování transakce, nebo z nějakého jiného důvodu chceme nastavit stav databáze, který byl před zahájením transakce, použijeme příkaz ROLLBACK.



Pokud programujete databázovou transakci v prostředí, které podporuje transakce, musíte si uvědomit, že jestli nepoužijete ani jednu příkaz COMMIT nebo ROLLBACK, pak celý běh aplikace tvoří jednu transakci a uživatel může v případě poruchy přijít o veškeré výsledky své práce. Zároveň je ale potřeba říci, že v současné době zpravidla vyvíjíme takové aplikace použitím vývojových prostředí, která umožňují snadno vytvářet obrazovkové formuláře, tiskové sestavy apod. V takových případech bývá nastaveno potvrzení transakce (příkaz COMMIT) automaticky při uzavření formuláře. Jiným příkladem může být potvrzení transakce při vytvoření nového spojení příkazem CONNECT. Takové automatické potvrzení se často označuje jako *autocommit* a často se také dá nastavit nebo zrušit příkazem SET s parametrem AUTOCOMMIT. Takovým způsobem mohou být některé situace ošetřeny.

Na druhé straně je ale potřeba dávat na takovéto automatické potvrzení pozor v situaci, kdy sami chceme řídit začátek a konec transakce. V takovém případě by například implicitně nastavené potvrzování po každém příkazu způsobilo něco, co nechceme. Rovněž při používání kurzorů si je potřeba uvědomit, že příkaz COMMIT či ROLLBACK může uzavřít všechny otevřené kurzory apod.

Vraťme se ještě krátce k nemožnosti zanořovat transakce v SQL. Jedinou možnou úroveň zanoření představují samotné příkazy SQL, o kterých jsme si řekli, že implicitně tvoří transakci. Protože jednoúrovňový model transakce představuje značné omezení, poskytovaly již dříve a později byla tato možnost zakotvena ve verzi standardu z roku 1999 tzv. *částečné zrušení transakce (partial rollback)*. Pro tyto účely lze použít příkazy:

```
SAVEPOINT p  
ROLLBACK p
```

Tyto příkazy se používají jako párové. Příkaz SAVEPOINT p umožňuje SŘBD „zapamatovat“ si místo *p* v posloupnosti příkazů tvořících transakci a příkazem ROLLBACK p provést nikoliv úplné zrušení transakce, tedy návrat do stavu před zahájením transakce, nýbrž pouze částečné – do stavu, ve kterém transakce byla v místě *p*.

x+y

Příklad 5.1

Uvažujme následující posloupnost příkazů nějaké transakce T:

```
příkaz1_transakce T
SAVEPOINT p1
příkaz2_transakce T
SAVEPOINT p2
příkaz3_transakce T
ROLLBACK p2
příkaz4_transakce T
ROLLBACK p1
```

Provedením příkazu ROLLBACK p2 se vrátí stav databáze do stavu po provedení příkazu *příkaz2* a provádění pokračuje následujícím příkazem, tj. příkazem *příkaz4*. Nejde tedy o skok do bodu P2!

5.4. Zotavení po chybách a poruchách

Když jsme si vysvětlovali vlastnosti transakce, uvedli jsme si dvě situace, které musí SŘBD ošetřit, aby byly zachovány vlastnosti ACID transakce. Jde o zotavení po chybách a poruchách a řízení souběžného přístupu. V této kapitole se budeme věnovat první z nich. Abychom si situaci zjednodušili, budeme předpokládat, že v daném okamžiku probíhá nejvíce jedna transakce. Jinými slovy nebudeme předpokládat souběžné transakce. Jakým způsobem ovlivní proces zotavení prostředí, typické prostředí provozu databázových aplikací, tj. řada současně probíhajících transakcí, si ukážeme, až si vysvětlíme způsob řízení souběžného přístupu.

Zotavením budeme rozumět proces prováděný SŘBD, při kterém obnovuje konzistentní stav databáze po nějaké chybě, poruše či výpadku systému. Způsob, jakým SŘBD zajistí obnovu konzistentního stavu databáze, který platil v okamžiku poruchy, se nazývá *schéma zotavení*.

Při provozu databázových systémů může docházet z různých důvodů k chybám a poruchám. Můžeme rozlišit tři kategorie chybových stavů:

- chyby transakce
- zhroucení systému
- porucha disku se ztrátou dat

V *chybě transakce* může jít o logické chyby nebo o systémové chyby. *Logické chyby* by měl být v řadě případů aplikační programátor schopen předpokládat a při programování by je měl ošetřit. V případě, že chce v rámci ošetření chyby zrušit transakci, použije příkaz ROLLBACK. V případě, že některou logickou chybu neošetří, může to vést k implicitnímu ošetření systémem řízení báze dat, typicky výpisem nějakého hlášení a často i zrušením transakce či ukončením aplikace. Příkladem *systémové chyby* může být tzv. uváznutí (deadlock) vlivem vzájemného zablokování transakcí při řízení souběžného přístupu, jak uvidíme v kapitole 5.5. V takovém případě SŘBD rozhodne o ošetření, v případě uváznutí například zrušením jedné z transakcí. Pokud jsou transakce napsány správně, včetně ošetření logických chyb, nemělo by dojít v případě chyb transakcí ke ztrátě dat. Buď transakce pokračuje

v činnosti, nebo je zrušena, tedy provedena operace rollback.

Ke *zhroucení systému* může dojít vlivem nějaké vnitřní chyby SŘBD nebo jako důsledek zhroucení operačního systému či výpadku počítače. Ve všech těchto případech dochází ke ztrátě dat v pracovních oblastech využívaných SŘBD a transakcemi. Data v databázi zůstávají zachována, ale mohou být v nekonzistentním stavu.

Pokud má být SŘBD schopen zotavit se z kterékoli z výše uvedených chyb a poruch, musí provádět dvě skupiny akcí:

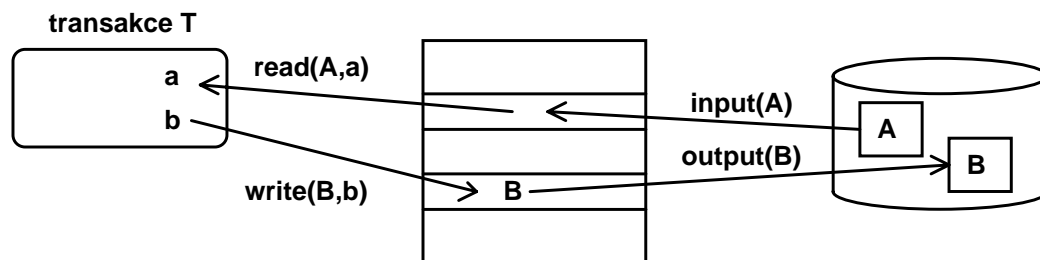
1. Akce, které během normálního provádění transakcí zajistí dostatek informací pro úspěšné zotavení.
2. Akce, které na základě těchto informací provedou samotné zotavení.

Abychom byli schopni určit, jak by se měl systém zotavit z chyb a poruch, musíme rozumět důsledkům poruch jednotlivých typů pamětí, které používáme v počítačích k uložení dočasných a perzistentních dat.

Když si uvědomíme, jaké typy pamětí se u současných počítačů používají, můžeme je rozdělit z hlediska závislosti uchování obsahu na dodávce elektrické energie do dvou skupin – na *energeticky závislé (volatile)* a *energeticky nezávislé (nonvolatile)*. V prvním případě se obsah paměti při přerušení dodávky energie ztrácí, zatímco ve druhém případě zůstává zachován. Příkladem energeticky závislé paměti jsou polovodičové paměti RAM, příkladem energeticky nezávislé paměti jsou paměti magnetické nebo optické. Z pohledu místa, kde se nachází zpracovávaná databázová data pro nás bude představitelem energeticky závislé paměti vnitřní paměť počítače a představitelem energeticky nezávislé paměti magnetický disk.

Přestože data na disku zůstanou zachována i při výpadku dodávky energie, můžeme o ně přijít jiným způsobem, například poškozením povrchu média. Zavedeme si proto ještě jeden typ paměti, který pro nás bude představovat paměť ideální. Budeme ji označovat jako *stabilní* (v angličtině *stable*) a budeme u ní předpokládat, že o data nepřijde za žádných okolností. Taková paměť samozřejmě reálně neexistuje. V praxi používáme pro uložení těch nejcennějších, „životně důležitých“ dat různé způsoby, kterými se snažíme takové ideální paměti co nejvíce přiblížit. Tato řešení se opírají zpravidla o vícenásobné ukládání dat, například zrcadlením souborů (file mirroring), tj. současným ukládáním dat na různé disky, disková pole apod.

Nyní už se podívejme, jakým způsobem může dojít v důsledku zhroucení systému k porušení konzistence dat v databázi. Budeme předpokládat model přístupu transakce k datům



Obr. 5.3 Model přístupu transakce k datům

Budeme rozlišovat data nacházející se ve třech oblastech. Jsou to databázové soubory na disku, vyrovnávací paměť systému řízení báze dat a lokální pracovní prostor

s lokálními proměnnými transakce. Dále budeme předpokládat, existenci následujících čtyř operací, které souvisejí se čtením a zápisem hodnot do/z lokálních proměnných transakce: $read(A, a)$, $write(A, a)$, $input(A)$, $output(A)$. Operace $read(A, a)$ uloží hodnotu databázového záznamu A z vyrovnávací paměti do lokální proměnné transakce a . Operace $write(A, a)$ zapíše hodnotu lokální proměnné transakce a do databázového záznamu A ve vyrovnávací paměti. Operace $input(A)$ přečte blok se záznamem A z disku do vyrovnávací paměti a operace $output(A)$ zapíše blok se záznamem A z vyrovnávací paměti na disk.

Předpokládejme, že transakce T z obrázku potřebuje načíst záznam A z databáze do své lokální proměnné a . Požaduje provedení operace $read(A, a)$. Jestliže se příslušný diskový blok nachází ve vyrovnávací paměti, je hodnota požadovaného záznamu uložena do proměnné a . V opačném případě nejprve správce vyrovnávací paměti provede operaci $input(A)$, kterou blok do vyrovnávací paměti načte a teprve potom operaci $read(A, a)$ dokončí.

Nyní předpokládejme, že transakce T vytvořila nějakou novou hodnotu uloženou v lokální proměnné b , kterou chce zapsat do databázového záznamu B . Volá operaci $write(B, b)$. Pokud je odpovídající diskový blok se záznamem B ve vyrovnávací paměti, zapíše se do něho hodnota a operace končí. V opačném případě nejprve správce vyrovnávací paměti provede operaci $input(B)$, kterou blok do vyrovnávací paměti načte a teprve potom operaci $write(B, b)$ dokončí.



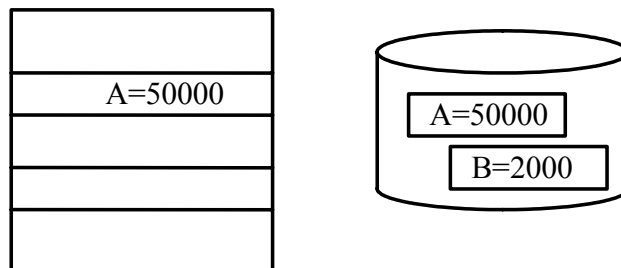
Pro pochopení dalších odstavců je podstatné si uvědomit několik skutečností:

1. Čtení hodnoty nějakého databázového záznamu transakcí (operací $read$) může vyvolat čtení z disku (operaci $input$), případně ještě před ní zápis na disk (operaci $output$), pokud je potřeba uvolnit prostor ve vyrovnávací paměti.
2. Zápis hodnoty nějakého databázového záznamu transakcí (operací $write$) neznamena zápis na disk, ale jen do vyrovnávací paměti. Na druhé straně může vyvolat čtení z disku (operaci $input$), případně ještě před ní zápis na disk (operaci $output$), pokud je potřeba uvolnit prostor ve vyrovnávací paměti.
3. Některá aktuální databázová data se nacházejí při činnosti databázového systému ve vyrovnávací paměti, zatímco v databázovém souboru na disku je jejich neaktuální verze. Databázi proto budeme chápat jak data na disku, tak ve vyrovnávací paměti.



Příklad 5.2

Uvažujme transakci T , která převádí částku 10000 Kč z účtu A na účet B a předpokládejme, že stav účtu A je 50000 Kč a účtu B 2000 Kč. Záznam účtu A nechť se nachází jak ve vyrovnávací paměti, tak na disku, záznam účtu B je jen na disku = viz Obr. 5.4.



Obr. 5.4 Hodnoty na účtech A a B před zahájením transakce

Transakce T budou tvořit následující operace:

```
read(A, a)
a = a - 10000
write(A, a)
read(B, b)
b = b + 10000
write(B, b)
```

Předpokládejme že nová hodnota stavu účtu A , tj. 40000 Kč, se operací $write(A, a)$ zapíše do vyrovnávací paměti. Dále předpokládejme, že při operaci $read(B, b)$ je potřeba uvolnit místo pro načtení bloku se záznamem účtu B do vyrovnávací paměti. Necht' je pro uvolnění místa vybrán blok se záznamem účtu A , tedy provede se operace $output(A)$. Hodnota účtu B , tj. 12000 Kč, skončí pouze ve vyrovnávací paměti.

Po ukončení transakce bude aktuální hodnota záznamu účtu A na disku a aktuální hodnota záznamu účtu B ve vyrovnávací paměti. Pro konzistentní stav databáze musí platit, že součet hodnot na obou účtech musí být konstantní. To určitě platí, protože před zahájením transakce byl součet $50000 + 2000 = 52000$ Kč a po skončení $40000 + 12000 = 52000$ Kč.

Uvažujme, že v tomto okamžiku dojde k výpadku systému, jehož následkem se provede restart SŘBD. V rámci tohoto procesu se alokuje nová vyrovnávací paměť. Data, která byla ve vyrovnávací paměti před výpadkem, jsou ztracena. V databázovém souboru na disku je nový stav účtu A , ale starý stav účtu B , databáze je v nekonzistentním stavu. Součet na účtech je $40000 + 2000 = 42000$ Kč. Přestože transakce proběhla celá, z pohledu dat v databázi nebylo provedeno všechno a SŘBD musí zajistit atomičnost transakce. Měl by v rámci zotavení provést transakci znovu nebo naopak neprovádět nic? Ani jedna z těchto variant by nevedla ke konzistentnímu stavu. SŘBD potřebuje pro úspěšné zotavení transakce mít informace o tom, jaká data transakce modifikovala. Pokud si SŘBD vždy před modifikací databázových dat, tj. před zápisem operací $write$ uloží na bezpečné místo informaci o provedené modifikaci, bude se později v případě potřeby moci vrátit k původním hodnotám nebo bude moci znovu zapsat do databáze hodnoty nové.

Toto je způsob, jak může SŘBD zajistit tři z ACID vlastností, které jsou ohroženy v případě chyb a poruch.

Které vlastnosti to jsou?



Přestože se u komerčních i nekomerčních SŘBD používají různé způsoby zotavení, naprostá většina z nich je založena na použití tzv. žurnálu.

Žurnál (v češtině někdy také *deník transakcí*, v angličtině *log file*) je soubor tvořený *záznamy žurnálu* (log record).

Předpokládejme, že může obsahovat následující typy záznamů:

- $\langle T_i, \text{start} \rangle$ - znamená, že transakce T_i zahájila provádění, tj. dostala se do stavu *aktivní* v diagramu stavů transakce.
- $\langle T_i, X_i, H_1, H_2 \rangle$ - znamená, že transakce T_i provedla zápis datové položky X_i . H_1 značí původní a H_2 novou hodnotu položky X_i .

- $\langle T_i, \text{commit} \rangle$ - znamená, že transakce T_i potvrdila změny (skončila úspěšně). Tento záznam v žurnálu znamená, že se transakce dostala do stavu částečného potvrzení, resp. do stavu potvrzení, pokud se tento záznam již nachází ve stabilní paměti.
- $\langle T_i, \text{abort} \rangle$ - znamená, že transakce T_i byla zrušena. Tento záznam v žurnálu znamená, že se transakce dostala do stavu porucha, resp. zrušená po uložení do stabilní paměti.



V souvislosti s žurnálem je potřeba zdůraznit dvě věci:

1. Protože informace pro zotavení jsou pro úspěšnou činnost systému kritické, bude se žurnál nacházet ve stabilní paměti.
2. Záznam o provedené modifikaci bude vždy zapsán do žurnálu ještě před tím než je samotná modifikace provedena. Musíme totiž předpokládat, že může v kterémkoliv okamžiku činnosti databázového systému dojít k poruše. Pokud by byla provedena modifikace, ale informace o ní se nestihla uložit do žurnálu, nebylo by zotavení možné. Při opačném pořadí je vše v pořádku, protože pokud se v důsledku poruchy nestihne provést modifikace, ale informace o ní bude v žurnálu, je možné ji provést v rámci zotavení. Toto pravidlo se označuje jako WAL (write-ahead logging) a ještě se k němu později vrátíme.

Ukážeme si techniku použití žurnálu s okamžitou modifikací databáze. Její podstata spočívá v tom, že se do žurnálu záznamy prováděné transakcí a ty se bezprostředně provádějí. V případě poruchy systému a následného restartu je potřeba v rámci zotavení anulovat změny provedené nedokončenými transakcemi a naopak znovu provést změny transakcemi úspěšně dokončenými. Schéma zotavení proto zahrnuje odpovídající dvě zotavovací procedury, které $\text{undo}(T_i)$ a $\text{redo}(T_i)$.

Příklad 5.3

Uvažujme transakci T_0 převádějící částku 10000 Kč z účtu A na účet B (viz z příkladu Příklad 5.2), po které bude probíhat transakce T_1 , která sníží stav na účtu C o 2000 Kč. Předpokládejme počáteční stav na účtu C 20000 Kč.

Průběh provádění transakcí, zápisu záznamů do žurnálu a hodnot do databáze je zachycen v tabulce na Obr. 5.5 na následující straně. Budeme postupně diskutovat, jak proběhne zotavení ve třech okamžicích poruchy a následného restartu, které jsou uvedeny v posledním sloupci tabulky.

Okamžik 1

K poruše došlo po poslední operaci transakce, která modifikovala data, přesněji po zápisu záznamu o této modifikaci do žurnálu, ale před provedením vlastní modifikace. Po restartu SRBD je zahájen proces zotavení. Při něm komponenta SRBD zvaná *správce zotavení* (*recovery manager*) prochází žurnál a zjišťuje, které transakce probíhaly a zda byly dokončeny. Zjistí, že transakce T_0 byla zahájena, modifikovala záznamy účtů A a B (záznam účtu B ve skutečnosti zmodifikovat nestihla), ale záznam o úspěšném ukončení transakce v žurnálu nenajde. Přestože transakce z hlediska modifikací provedla, co chtěla, pro účely zotavení ji musí správce zotavení považovat za nedokončenou. Zotavení proto znamená uvedení databáze do stavu, který platil před zahájením transakce T_0 . To zajistí zotavovací procedura $\text{undo}(T_0)$, která prochází od konce záznamy žurnálu a když narazí na záznam o modifikaci, provede období

operace *write*, kterou zapíše původní hodnotu. Jakmile v žurnálu narazí na záznam $\langle T_0, \text{start} \rangle$, zotavení transakce T_0 končí.

transakce	žurnál	databáze	okamžik poruchy
T_0 : read(A,a)	$\langle T_0, \text{start} \rangle$		
a = a - 10000			
write(A, a)	$\langle T_0, A, 50000, 40000 \rangle$		
		A=40000	
read (B, b)			
b = b + 10000			
write (B, b)	$\langle T_0, B, 2000, 12000 \rangle$		
			← 1
		B=12000	
	$\langle T_0, \text{commit} \rangle$		
T_1 : read (C, c)	$\langle T_1, \text{start} \rangle$		
c = c - 2000			
write(C, c)			
	$\langle T_1, C, 20000, 18000 \rangle$		
		C=18000	
			← 2
	$\langle T_1, \text{commit} \rangle$		
			← 3

Obr. 5.5 Průběh transakcí z příkladu Příklad 5.3

Okamžik 2

K poruše došlo po modifikaci provedené transakcí T_1 , ale před vložením záznamu žurnálu, který říká, že transakce skončila. V rámci zotavení prochází správce zotavení žurnál a zjistí, že byla zahájena transakce T_0 , modifikovala záznamy účtů A a B a úspěšně skončila. Dále zjistí, že byla zahájena transakce T_1 , modifikovala záznam účtu C a narazí na konec žurnálu. Transakce T_1 tedy neproběhla celá. Výsledkem bude, že u transakce T_1 je potřeba anulovat provedené změny, zatímco u transakce T_0 je potřeba provedené změny zopakovat. Důvodem pro nutnost zopakování změn je skutečnost, že tyto změny mohly být provedeny pouze ve vyrovnávací paměti, nikoliv na disku. A data vyrovnávací paměti byla v důsledku poruchy ztracena. Jako první provede správce zotavení proceduru $\text{undo}(T_1)$, která prochází od žurnál od konce a zapíše původní hodnotu 20000 stavu do záznamu účtu C . Poté narazí na záznam $\langle T_1, \text{start} \rangle$ a končí. Následně aplikuje zotavovací proceduru $\text{redo}(T_0)$. Při té prochází žurnál od začátku a když narazí na nějaký záznam říkající, že transakce T_0 modifikovala data, znovu zapíše novou hodnotu. V našem případě znovu zapíše hodnotu 40000 do záznamu účtu A a 12000 do záznamu účtu B . Jakmile narazí na záznam $\langle T_0, \text{commit} \rangle$, končí zotavení transakce T_0 a končí i celý zotavovací proces.

Okamžik 3

Tentokrát k poruše došlo po úspěšném ukončení obou transakcí. Správce zotavení zjistí při průchodu žurnálem, že byla zahájena transakce T_0 a také úspěšně skončila a totéž zjistí i o transakci T_1 . Na obě proto aplikuje výše popsáním způsobem zotavovací proceduru redo .

Na základě předchozího příkladu byste už měli být schopni popsat schéma zotavení

techniky žurnálu s okamžitou modifikací. Mohli bychom ho charakterizovat tak, že na každou transakci T_i , jejíž záznam nalezne správce transakce v žurnálu, aplikuje následující zotavovací proceduru:

- $undo(T_i)$, jestliže žurnál obsahuje záznam $\langle T_i, start \rangle$, ale neobsahuje záznam $\langle T_i, commit \rangle$
- $redo(T_i)$, jestliže žurnál obsahuje $\langle T_i, start \rangle$ i $\langle T_i, commit \rangle$.

Prakticky zotavení proběhne ve dvou krocích:

1. Správce zotavení prochází žurnál a klasifikuje transakce na ty, které úspěšně skončily a ty, které nebyly dokončeny.
2. Na nedokončené transakce aplikuje zotavovací proceduru *undo* a na dokončené zotavovací proceduru *redo*.



Možná vás napadlo, že nedokončená může zůstat nejvýše jedna transakce. To je pravda pouze pro náš zjednodušující předpoklad, který jsme si pro výklad zotavení zavedli, a sice, že v daném okamžiku neprobíhají souběžné transakce. Při souběžných transakcích může samozřejmě zůstat nedokončených více.

Vidíme, že u techniky použití žurnálu s okamžitou modifikací se v databázi objevují nepotvrzené hodnoty, tj. hodnoty, které se de facto stanou platnými až po potvrzení změn příkazem COMMIT. Až se budeme bavit o řízení souběžného přístupu uvidíme, že takové hodnoty mohou způsobovat problémy.



Zmíníme se ještě velice stručně o dvou dalších technikách zotavení. První z nich také používá žurnál, ale jiným způsobem. Jde techniku použití *žurnálu s odloženou modifikací*. Opět dochází k zápisu záznamů o zahájení, modifikacích a úspěšném ukončení transakce do žurnálu. Modifikace databáze je ale odložena až do okamžiku, kdy se transakce dostane do stavu potvrzení. Pokud dojde k poruše v průběhu provádění transakce, nejsou data v databázi modifikovaná, a proto je zotavení velice jednoduché – není potřeba dělat nic. Pokud správce zotavení zjistí, že transakce skončila, aplikuje ze stejných důvodů jako u okamžité modifikace zotavovací proceduru *redo*, která na základě záznamu v žurnálu znovu zapíše nové hodnoty.

Druhou technikou je tzv. *stínové stránkování* (*shadow paging*). Využívá myšlenku stránkování, jak ji možná znáte z operačních systémů. Její podstata spočívá v tom, že nejsou adresovány přímo diskové bloky, ale logické stránky a existuje mapování logických stránek na diskové bloky. Toto mapování zajišťuje tzv. *tabulka stránek*, která udává, v kterém bloku na disku se nachází která logická stránka. Existuje tzv. *stínová tabulka stránek*, která mapuje logické stránky na bloky s potvrzenými hodnotami. Je to tedy tabulka stránek platná pro databázi. Nemění se při provádění transakcí. Je-li zahájena nějaká transakce, vytvoří si kopii této globální tabulky stránek, tzv. *aktuální tabulku stránek*, jejíž obsah se ale při provádění transakce může měnit. Pokud transakce modifikuje nějaký záznam, vytvoří se kopie bloku se záznamem a na ten se pro danou transakci mapuje příslušná logická stránka. Při ukončení transakce se zapíše modifikované bloky na disk a aktuální tabulka stránek se prohlásí za globální stínovou tabulku stránek. Zotavení je v tomto případě ještě jednodušší, než u žurnálu s odloženou modifikací. Není totiž potřeba dělat vůbec nic. U nedokončených transakcí platí stále původní stínová tabulka stránek a u dokončených již byly modifikované bloky zapsány na disk. Situace se ale komplikuje,

pokud probíhají transakce souběžně, což je v praxi typická situace v řadě aplikací.

Vraťme se k technikám založeným na použití žurnálu. Zatím jsme předpokládali, že správce zotavení prohlédne celý žurnál, aby rozhodl, jak kterou transakci, které probíhaly, ošetřit. Vzniká otázka, ke kterému okamžiku se vztahuje počátek žurnálu. Pokud by nebyla přijata žádná opatření, byl by to okamžik zahájení první transakce po nainstalování příslušného SŘBD. A od toho okamžiku mohlo uplynout i několik let, žurnál by byl velice rozsáhlý a zotavení neúměrně časově náročné. Proto SŘBD provádí v určitých intervalech akce, které definují v žurnálu okamžik, ke kterému je potřeba vztáhnout počátek zotavování. Tyto akce se označují *kontrolní bod* (*checkpoint*).

Podstata kontrolního bodu spočívá v tom, že správce vyrovnávací paměti uloží potřebné informace z vyrovnávací paměti do energeticky nezávislé paměti, tedy na disk. Dosud jsme uvažovali, že se ve vyrovnávací paměti nacházejí pouze bloky databáze a záznamy žurnálu že se zapisují do žurnálu ve stabilní paměti. To by ale bylo časově značně náročné. Proto používá SŘBD vyrovnávací paměť i pro zápis záznamů žurnálu. Při kontrolním bodu se uloží jednak na disk modifikované diskové bloky, jednak do stabilní paměti záznamy žurnálu z vyrovnávací paměti.

Důležité je pořadí, v jakém se ukládání provádí. Mělo by vám být jasné, že priorita ukládání je dána důležitostí pro zotavení. Poznali jsme to už u použití žurnálu, kdy jsme si řekli, že nejprve se musí v žurnálu vytvořit záznam o modifikace databáze a teprve potom lze modifikaci provést. Opačné pořadí by mohlo způsobit nemožnost zotavení. Kontrolní bod proto sestává ze tří kroků, jejichž pořadí je podstatné:

1. Do žurnálu ve stabilní paměti jsou uloženy všechny záznamy z vyrovnávací paměti žurnálu.
2. Na disk jsou z vyrovnávací paměti uloženy všechny modifikované bloky databáze.
3. Do žurnálu ve stabilní paměti se vloží záznam $\langle \text{checkpoint}, T_1, T_2, \dots, T_k \rangle$.

První krok zajistí, že v žurnálu ve stabilní paměti budou všechny informace potřebné pro zotavení. Druhý krok zajistí, že se bude možné při zotavení spolehnout na to, že všechny modifikace provedené před kontrolním bodem, byly promítnuty do databáze na disku. Třetí krok potvrzuje, že kontrolní bod úspěšně proběhl. Teprve po jeho vložení do žurnálu se efekt kontrolního bodu projeví při zotavení. Záznam žurnálu $\langle \text{checkpoint}, T_1, T_2, \dots, T_k \rangle$ říká, že v okamžiku provedení kontrolního bodu probíhaly transakce T_1, T_2 až T_k . My zatím předpokládáme nejvýše jednu transakci běžící v daném okamžiku, ale víme, že je to pouze naše zjednodušení pro výklad problematiky zotavení.

Tím, že kontrolní bod zajistí uložení všech modifikací databáze na disk, stačí, aby správce zotavení analyzoval pouze část žurnálu od posledního kontrolního bodu. Můžeme si tedy modifikovat zotavení při použití žurnálu s okamžitou modifikací takto:

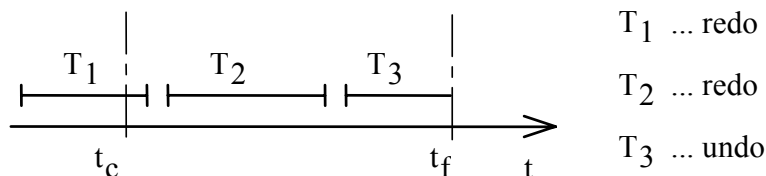
1. Správce zotavení prochází žurnál v úseku od konce do posledního kontrolního bodu a klasifikuje transakce na ty, které úspěšně skončily a ty, které nebyly dokončeny.
2. Na nedokončené transakce aplikuje zotavovací proceduru podle použité

techniky. V případě žurnálu s okamžitou modifikací *undo* na nedokončené a *redo* na dokončené transakce.

x+y

Příklad 5.4

Uvažujme transakce T_1 , T_2 a T_3 probíhající za sebou v čase, jak ukazuje Obr. 5.6.



Obr. 5.6 Příklad zotavení transakcí z příkladu Příklad 5.4

Správce zotavení prochází žurnál od konce. Zjistí, že probíhala transakce T_3 , ale nenarazil u ní na záznam $\langle T_3, \text{commit} \rangle$, tedy nebyla dokončena. Dále narazí na záznam $\langle T_2, \text{commit} \rangle$, který říká, že probíhala transakce T_2 a úspěšně skončila, najde pravděpodobně i nějaké záznamy o tom, že modifikovala databázi, a v každém případě záznam $\langle T_2, \text{start} \rangle$. Dále v žurnálu najde záznam $\langle T_1, \text{commit} \rangle$, na základě kterého identifikuje T_2 jako další úspěšně dokončenou transakci. Možná bude v žurnálu nějaký záznam, že modifikovala databázi, v každém případě tam ale bude záznam $\langle \text{checkpoint}, T_1 \rangle$. Tím fáze analýzy obsahu žurnálu a klasifikace transakcí na dokončené a nedokončené končí. Ve druhém kroku aplikuje správce zotavovací proceduru *undo* na transakci T_3 . Ta ve směru od konce žurnálu zapíše do databáze původní hodnoty modifikovaných objektů. Zotavovací proceduru *redo* poté aplikuje u transakce T_1 a T_2 . Ta postupně znovu zapíše nové hodnoty vytvořené transakcí T_1 od kontrolního bodu po $\langle T_1, \text{commit} \rangle$ a znovu provede všechny zápisy transakce T_2 .

Shrňme si ještě některé zásady správy vyrovnávací paměti, které souvisí se zajištěním zotavení po poruše. Všechny tyto zásady vyplývají z již uvedeného základního pravidla, že informace se ukládají z vyrovnávací paměti v pořadí daném důležitostí pro zotavení. Pokud jde o bloky žurnálu, ty se ukládají z vyrovnávací paměti do stabilní paměti. Platí následující zásady:

1. Transakce T_i se dostane do stavu potvrzení (C v diagramu na Obr. 5.1) teprve tehdy, až je uložen do stabilní paměti záznam žurnálu $\langle T_i, \text{commit} \rangle$.

Tato zásada je zřejmá, když si uvědomíte, že správce zotavení právě podle tohoto záznamu pozná, jestli byla transakce dokončena nebo ne.

2. Před záznamem $\langle T_i, \text{commit} \rangle$ musí být do stabilní paměti uloženy všechny záznamy žurnálu týkající se transakce T_i .

Tato zásada zajišťuje, že u potvrzené transakce budou k dispozici všechny záznamy žurnálu, které se transakce týkají, a bude tak možné v případě poruchy transakci správně zotavit jako potvrzenou.

3. Před uložením bloku dat do databáze musí být uloženy všechny záznamy žurnálu, týkající se daného bloku.

Tato zásada se v anglické literatuře označuje zkratkou *WAL* (*Write-Ahead Logging*) a my už jsme se o ní zmínili, když jsme si vysvětlovali podstatu žurnálu. Opět jde o to, že pokud byla data v bloku modifikována, je potřeba dříve, než se

modifikovaná data uloží do databáze, uložit do žurnálu ve stabilní paměti záznamy popisující tyto modifikace. Pokud by se nejprve uložil modifikovaný blok a dříve, než by proběhlo uložení záznamů o modifikacích do žurnálu ve stabilní paměti, by došlo k poruše, správce zotavení by neměl o těchto modifikacích informace a nemohla by například zotavovací procedura *undo* nastavit původní hodnoty.



Protože jsme se při výkladu zotavení po poruchách a chybách zaměřili výhradně na to, jak probíhá zotavení po poruše, je potřeba konstatovat, že zotavení po chybě transakce, která vede na zrušení transakce, probíhá analogicky. Připomínáme, že v tomto případě je zotavení iniciované příkazem ROLLBACK zadaným programátorem, nebo vyvolané odpovídající operací iniciovanou SŘBD. Nedochází zde ke ztrátě dat z vyrovnávací paměti, pouze je potřeba nastavit stav databáze platný před zahájením rušené transakce. To se opět provede v závislosti na použité technice žurnálu. V případě žurnálu s okamžitou modifikací se aplikuje zotavovací procedura *undo* a do žurnálu se vloží záznam $\langle T_i, \text{abort} \rangle$, kde T_i je rušená transakce.

Dosud jsme se zabývali pouze zajištěním požadovaných vlastností transakce v případě chyby transakce nebo poruchy energeticky závislé paměti. Předpokládali jsme, že data na disku zůstanou zachována. Nyní se podíváme na to, jaké prostředky poskytuje SŘBD pro to, aby mohly být zajištěny požadované vlastnosti transakce i v případě takové poruchy disku, tedy energeticky nezávislé paměti, při které dojde k porušení, resp. ztrátě databázových souborů.

Z hlediska dostupných prostředků a jejich použití je situace podobná, jako v případě dat ukládaných do souborů operačního systému. Pokud se chcete zajistit proti ztrátě dat v případě poruchy disku, na kterém máte data uložena, provádíte jejich zálohování. To znamená, že vytvoříte kopii souborů na jiném paměťovém médiu (jiném disku, optickém disku, magnetické pásce apod.). V případě poruchy disku potom provedete obnovu souboru ze záložní kopie.

Rovněž u databázových systémů je ochrana dat postavena na zálohování a obnově v případě poruchy. *Zálohováním* označovaným také jako *archivace (backup)* budeme rozumět ukládání obsahu databáze do stabilní paměti, typicky v pravidelných intervalech. *Obnova (restore)* potom znamená uvedení databáze do stavu před posledním zálohováním. SŘBD zpravidla poskytují speciální prostředky pro zálohování a obnovu, které dávají větší možnosti, než pouhé zkopírování databázových souborů.

Při zálohování musí být opět dodrženo určité pořadí prováděných akcí, které zajistí možnost následné obnovy a zotavení. Probíhá v následujících čtyřech krocích:

1. Uložení záznamů žurnálu do stabilní paměti.
2. Uložení modifikovaných bloků databáze z vyrovnávací paměti na disk.
3. Uložení databáze z disku do stabilní paměti.
4. Vložení záznamu $\langle \text{dump} \rangle$ do žurnálu ve stabilní paměti.

První dva kroky představují stejné akce, které probíhají při provádění kontrolního bodu. Třetí krok realizuje vytvoření vlastní zálohy. Čtvrtý krok potvrzuje v žurnálu úspěšné dokončení zálohování, analogicky jako tomu bylo u kontrolního bodu.

Pokud dojde k poruše disku, která má za následek porušení databáze, proběhne

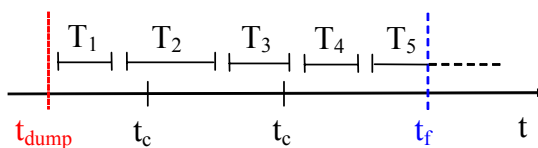
zotavení ve dvou krocích:

1. Obnovení databáze z poslední záložní kopie.
2. Zotavení provedené ne od posledního kontrolního bodu, nýbrž od okamžiku posledního zálohování.

První krok zajistí obnovení stavu disku existujícího při posledním zálohování. Víme, že součástí zálohování jsou i akce odpovídající kontrolnímu bodu. Pro to stačí provést následné zotavení od tohoto okamžiku.

Příklad 5.5

Uvažujme průběh transakcí zachycený na Obr. 5.7. Na časové ose t_{dump} zde značí okamžik zálohování, t_c kontrolní bod a t_f okamžik poruchy.



Obr. 5.7 Obnova a zotavení po poruše energeticky nezávislé paměti

Zotavení proběhne tak, že se nejprve ze zálohy obnoví stav databázových souborů na disku z okamžiku t_{dump} . Následně správce již dříve popsaným postupem prochází žurnál a klasifikuje transakce na potvrzené a nedokončené. Při použití žurnálu s okamžitou modifikací by následně provedl *undo* pro transakci T_5 a *redo* pro transakce T_1 , T_2 , T_3 a T_4 .



Jak byste vysvětlili, že zotavení v příkladu Příklad 5.5 musí proběhnout od okamžiku t_{dump} a ne od okamžiku posledního kontrolního bodu.



Je potřeba zdůraznit, že pokud jde o zajištění bezpečnosti dat z hlediska potenciální ztráty při poruše disku, leží zodpovědnost na správci databáze, protože on zajistit strategii zálohování stanovenou bezpečnostní politikou dané instituce. SŘBD často poskytuje různé možnosti a režimy zálohování. Zálohování například může probíhat při pozastavené činnosti nebo za chodu databázového systému, zálohovat lze celou nebo jen část databáze apod.

5.5. Řízení souběžného přístupu

V této kapitole se budeme věnovat druhému problému, který komplikuje transakční zpracování, - souběžnému běhu transakcí. Budeme předpokládat, že v daném okamžiku může současně probíhat několik transakcí a že tyto transakce mohou pracovat se stejnými daty v databázi. Vysvětlíme si podrobněji, v čem spočívá problém a ukážeme si některé základní techniky, kterými SŘBD zajišťuje dodržení požadovaných vlastností transakce. Zjednodušíme si ale situaci pokud jde o poruchy. Budeme předpokládat, že se poruchy nevyskytují.

Řízením souběžného přístupu budeme rozumět řízení přístupu souběžných transakcí ke sdíleným datům v databázi zajišťované SŘBD. Způsob, jakým to SŘBD provádí, tj. soustava pravidel a omezení použitých k zajištění souběžného přístupu, se nazývá *schéma řízení*.

Nejprve si vysvětlíme důležitý pojem, se kterým budeme po celou tuto kapitolu pracovat – plán souběžných transakcí. Víme, že transakce je tvořena posloupností

databázových operací, resp. instrukcí pro práci s daty v databázi. Pokud probíhá souběžně několik transakcí, může v daném okamžiku ke sdíleným datům v databázi přistupovat pouze jedna z nich.

DEF

Definice 5.1 Plán souběžných transakcí.

Plánem souběžných transakcí (schedule) nazýváme chronologické pořadí provádění databázových operací *read* a *write* souběžně probíhajících transakcí. Označuje se také jako *rozvrh souběžných transakcí*.

Obecně jsou v plánu souběžných transakcí operace jednotlivých transakcí promíchány tak, jak jsou prováděny v čase. Speciálním případem plánu je *sériový plán*. Je to takový plán, ve kterém jsou vždy všechny operace jednotlivých transakcí pohromadě, tj. bezprostředně za sebou. Jinými slovy to znamená, že transakce neprobíhají souběžně, nýbrž jsou provedeny postupně jedna za druhou, tedy sériově.

Z hlediska výkonnosti databázového systému pro nás sériové plány nejsou atraktivní. U souběžných transakcí se přece snažíme využít toho, že některé činnosti transakcí mohou probíhat skutečně paralelně, například prezentace a zadávání dat realizovaná klientskou částí aplikace v architektuře klient/server. To, čím jsou sériové plány zajímavé, je skutečnost, že neporušují konzistenci databáze. Uvažujme například transakce T_1 , T_2 a T_3 . Za předpokladu, že jsou správně naprogramovány, jak jsme si uváděli, žádná z nich neporuší konzistenci databáze. Je-li databáze konzistentní před zahájením transakce, bude konzistentní i po jejím dokončení. Pokud budou uvedené transakce provedeny například v pořadí T_2 T_1 T_3 , bude databáze konzistentní po provedení T_2 následně T_1 a konečně i T_3 . To bude platit pro jakýkoliv sériový plán těchto transakcí, ale nemusí to platit pro plán, který sériový není.

?

Zkuste se zamyslet nad tím, kolik různých sériových plánů existuje pro n souběžných transakcí.

x+y

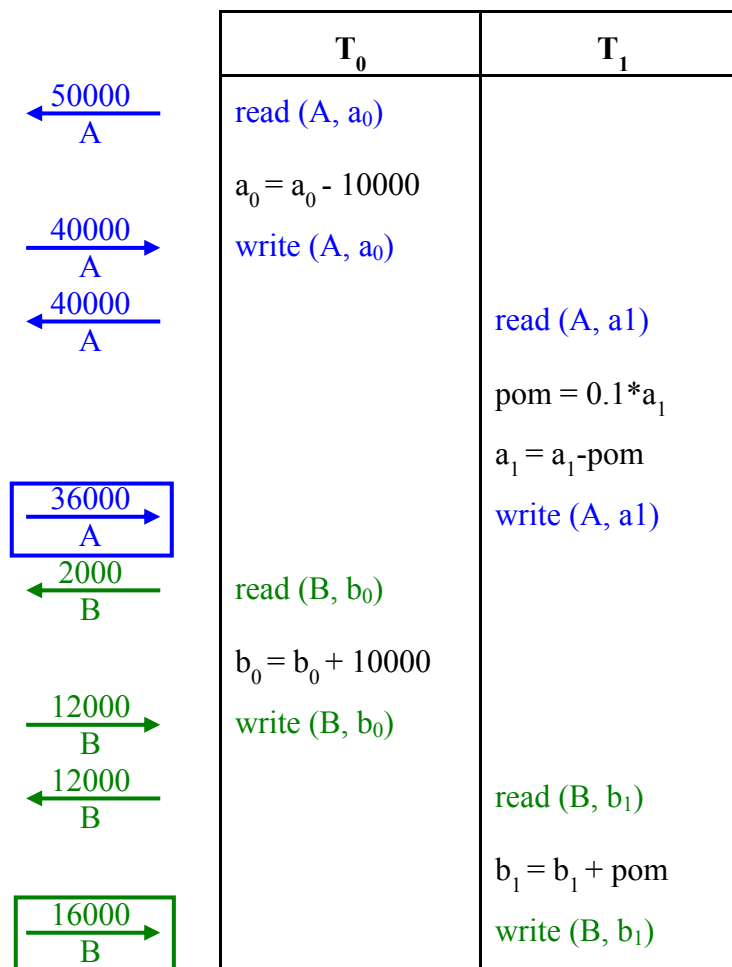
Příklad 5.6

Uvažujme souběžné transakce T_0 a T_1 . Transakce T_0 převádí částku 10000 Kč z účtu A na účet B a transakce T_1 převede na účet B částku rovnající se 10% stavu účtu A . Operace, které tvoří transakce T_0 a T_1 by mohly vypadat podle Obr. 5.8. V obrázku jsou ukázány nejen samotné databázové operace *read* a *write*, nýbrž i operace, které vykonávají transakce se svými lokálními proměnnými ve svém paměťovém prostoru. Tyto transakce nepředstavují z hlediska řízení souběžného přístupu žádný problém, protože mohou být prováděny i paralelně s jinými operacemi, pokud to prostředí SŘBD dovoluje.

T_0	T_1
read (A, a_0)	read (A, a_1)
$a_0 = a_0 - 10000$	$pom = 0.1 * a_1$
write (A, a_0)	$a_1 = a_1 - pom$
read (B, b_0)	write (A, a_1)
$b_0 = b_0 + 10000$	read (B, b_1)
write (B, b_0)	$b_1 = b_1 + pom$
	write (B, b_1)

Obr. 5.8 Operace transakcí T_0 a T_1

Předpokládejme stav účtu A před provedením transakcí 50000 Kč a účtu B 2000 Kč a uvažujme jeden z možných plánů těchto dvou transakcí uvedený na Obr. 5.9.

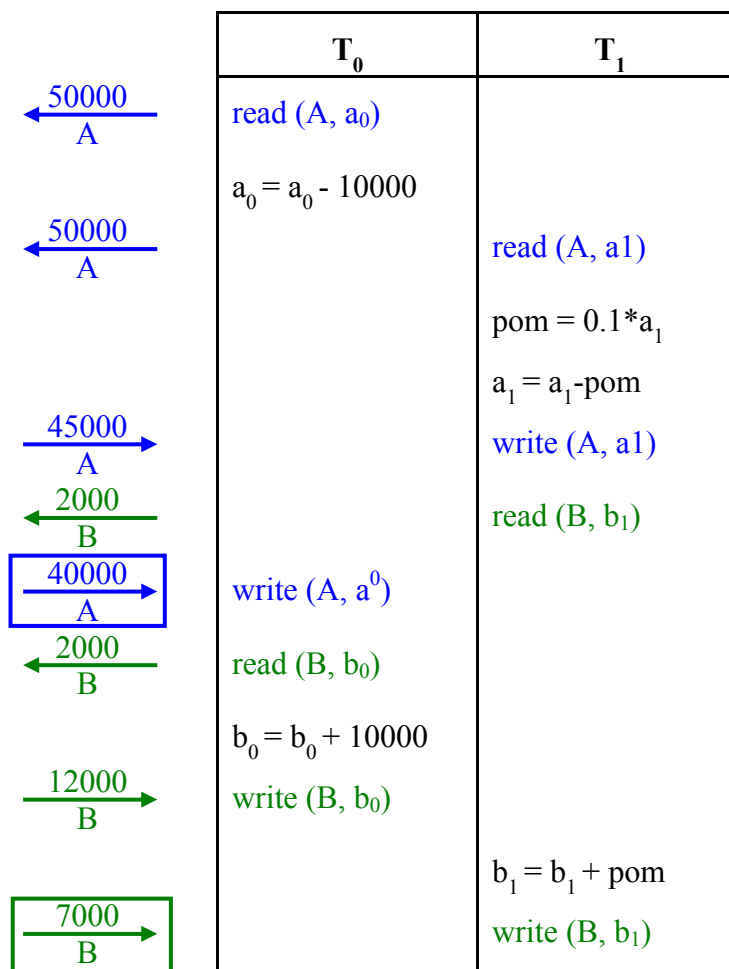


Obr. 5.9 Plán transakcí T_0 a T_1 , který není sériový a neporušuje konzistenci

Šipky po levé straně znázorňují čtení a zápisy do databáze. Barevně jsou odlišeny operace se záznamy obou účtů. Je zřejmé, že plán není sériový. Podívejme se, jestli jeho provedením dojde k porušení konzistence databáze. Z pohledu dat, se kterými transakce T_0 a T_1 pracují, je konzistentní stav charakterizován konstantní hodnotou součtu stavů obou účtů. Před zahájením provádění transakcí byl součet $50000 + 2000 = 52000$ Kč a po provedení je $36000 + 16000 = 52000$ Kč. Vidíme, že k porušení konzistence nedošlo. Ke stejnému závěru bychom došli pro libovolné počáteční stavy účtů A a B . Proto můžeme konstatovat, že uvedený plán zachovává konzistenci databáze.

Když se na plán podíváte podrobněji, možná vás napadne, proč tomu tak je. V plánu jsou vždy pohromadě všechny operace transakce, které provádějí něco se sdíleným databázovým záznamem. Nejprve transakce T_0 přečte i запиše novou hodnotu do záznamu účtu A a teprve potom s tímto záznamem pracuje transakce T_1 . Totéž se opakuje s účtem B . Transakce T_1 vždy pracuje s novými hodnotami vytvořenými transakcí T_0 . Výsledek je tedy stejný, jako kdyby nejprve proběhla celá transakce T_0 a teprve potom transakce T_1 . Uvidíme, že tato podstatná vlastnost plánu se nazývá *uspořadatelnost*.

Nyní uvažujme jiný plán transakcí T_0 a T_1 , který je uveden na Obr. 5.10.



Obr. 5.10 Plán transakcí T_0 a T_1 , který není sériový a porušuje konzistenci

Jde opět o plán, který není sériový, Za předpokladu stejných počátečních stavů účtů A a B , tj. 50000 a 2000 bude tentokrát po provedení transakcí stav na účtu A 40000 Kč a na účtu B 7000 Kč, což je dohromady 47000 Kč. Došlo k porušení konzistence databáze. Přestože transakce jsou naprogramovány tak, aby konzistenci neporušovaly, došlo vlivem souběžného provádění k jejímu porušení. 5000 Kč se někam „ztratilo“.

Podívejme se opět na plán podrobněji. Oproti předchozímu nejsou tentokrát operace obou transakcí v čase tak pěkně uspořádány. Především obě transakce přečtou hodnotu počátečního stavu účtu A , se kterou potom pracují. Protože obě hodnotu stavu účtu A také modifikují, už tento samotný fakt vylučuje možnost, aby byl výsledek stejný jako při sériovém provádění, tj. T_0T_1 nebo T_1T_0 , které konzistenci zaručuje vždy.

Z předchozího příkladu vyplývá důležitý závěr, že plán transakcí, který není sériový může porušit konzistenci databáze. Systém řízení báze dat proto nemůže ponechat transakcím při přístupu k datům úplnou volnost, nýbrž ho musí řídit takovým způsobem, aby mohly nastat (říkáme také „aby byly přípustné“ pouze takové plány souběžných transakcí, které konzistenci neporuší.

Dříve, než si ukážeme, jaké techniky SŘBD k tomu používá, podíváme se na několik typických situací při souběžném přístupu, které mohou vést na porušení

provádějí transakce se svými lokálními proměnnými, nejsou z hlediska řízení souběžného přístupu důležité, proto se v dalším zaměříme výhradně na databázové operace *read* a *write*, kterými transakce čte, resp. zapisuje data do databáze (přesně vyrovnávací paměti).

Uvažujme nějaký databázový objekt, např. záznam Q , se kterým pracují souběžné transakce T_1 a T_2 . První situací, kterou si uvedeme, je tzv. *ztráta aktualizace*. Nastává tehdy, když obě transakce modifikují objekt Q a jedna z transakcí přepíše novou hodnotu objektu Q vytvořenou transakcí druhou, přičemž obě transakce při výpočtu nové hodnoty vychází z téže hodnoty původní – viz Obr. 5.11.

T_1	T_2
...	
read (Q, q)	...
...	read (Q, q)
write (Q, q)	...
...	write (Q, q)
...	...

Obr. 5.11 Porušení konzistence vlivem ztráty aktualizace

Je to přesně ta situace, která nastala v plánu na Obr. 5.10 v předchozím příkladu.

Druhou situací je tzv. *čtení nepotvrzené hodnoty* vytvořené jinou transakcí. Tato situace nastane tehdy, když má transakce možnost přečíst nepotvrzenou hodnotu a pracovat s ní. Pokud je transakce, která hodnotu vytvořila, zrušena a to ať už příkazem ROLLBACK nebo při zotavení, nepotvrzená hodnota, se kterou druhá transakce pracovala, nebyla platná – viz Obr. 5.12.

T_1	T_2
...	...
	write (Q, q)
...	...
read (Q, q)	
...	...
...	ROLLBACK

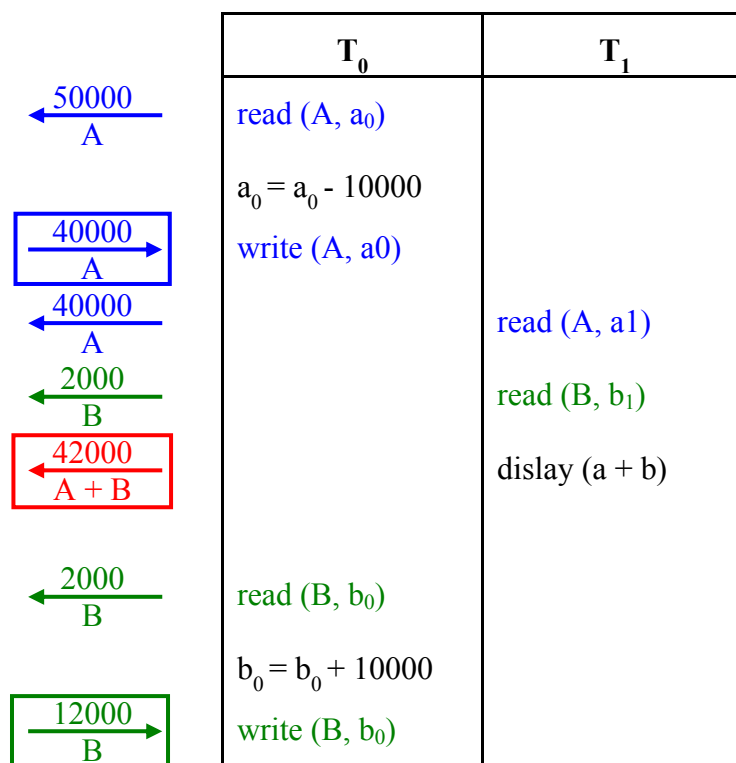
Obr. 5.12 Porušení konzistence vlivem přečtení nepotvrzené hodnoty

Podobná situace nastává, když transakce přepíše nepotvrzenou hodnotu – viz Obr. 5.13. Pokud je transakce T_2 , která nepotvrzenou hodnotu vytvořila, zrušena, nastaví se původní hodnota objektu Q , tj. hodnota vytvořená transakcí T_1 se ztratí.

T_1	T_2
...	...
	write (Q, q)
...	...
write (Q, q)	
...	...
...	ROLLBACK

Obr. 5.13 Porušení konzistence vlivem přepisem nepotvrzené hodnoty

Poslední z typických situací, se kterou se můžete v literatuře setkat, je tzv. *nekonzistentní analýza*. Je to případ, kdy transakce pracuje s nekonzistentními daty. Uvažujme například jako jednu ze dvou souběžných transakcí transakci T_0 z příkladu Příklad 5.6, která realizuje převod částky 10000 Kč. Z účtu A na účet B. Nechť souběžně s ní probíhá nějaká transakce T_1 , která přečte hodnoty stavů na účtech A a B a zobrazí jejich součet. Pokud bychom předpokládali stav na účtech před zahájením transakcí 50000 a 2000 Kč a takový plán transakcí, u kterého transakce T_1 přečte již sníženou hodnotu stavu účtu A, tj. 40000, ale také původní hodnotu stavu účtu B, tj. 2000 Kč, zobrazí nekonzistentní stav 42000 Kč – viz Obr. 5.14.



Obr. 5.14 Příklad nekonzistentní analýzy

V příkladu Příklad 5.6 jsme viděli, že pokud nějaký plán souběžných transakcí, který není sériový, má tu vlastnost, že dává stejný výsledek, jako kdyby sériový byl, pak neporušuje konzistenci databáze. Z této myšlenky vychází pojem *uspořadatelnosti plánu souběžných transakcí*. Nejprve si zavedeme binární relaci „je v konfliktu“ z množiny databázových operací jedné do množiny databázových operací druhé

transakce z dvojice transakcí.

DEF

Definice 5.2 Binární relace $je_v_konfliktu$

Nechť $O_1 = \{o_{11}, o_{12}, \dots, o_{1k}\}$ je množina operací *read* a *write* transakce T_1 a $O_2 = \{o_{21}, o_{22}, \dots, o_{2l}\}$ je množina operací *read* a *write* transakce T_2 , takové, že $T_2 \neq T_1$. Operace $\in O_1$ je v relaci $je_v_konfliktu$ s operací $o_y \in O_2$, právě když obě pracují se stejným databázovým objektem a alespoň jednou z nich je operace *write*.

Tabulka pro definici relace $je_v_konfliktu$, jak vyplývá z definice, je uvedena na Obr. 5.15.

o_x	o_y	
	read (Q, q)	write (Q, q)
read (Q, q)	N	A
write (Q, q)	A	A

Obr. 5.15 Definice binární relace $je_v_konfliktu$.

Neformálně můžeme říci, že binární relace $je_v_konfliktu$ znamená, že pořadí provedení operací je podstatné pro výsledek. Budou-li obě operace *read*, bude výsledek vždy stejný bez ohledu na pořadí. Bude-li jednou operací čtení a druhou operace zápisu, pak přečtená hodnota obecně závisí na tom, jestli předchází operaci zápisu nebo ne. V prvním případě transakce přečte původní hodnotu objektu, zatímco ve druhém novou. Analogicky pokud budou obě operace zapisovat, bude výsledek provedení obou závislý na pořadí.

Dalším pojmem, který si zavedeme, je ekvivalence plánů vzhledem ke konfliktům.

DEF

Definice 5.3 Ekvivalence plánů vzhledem ke konfliktům.

Nechť S_1 a S_2 jsou dva plány provádění operací transakcí. Řekneme, že plán S_1 je *ekvivalentní vzhledem ke konfliktům* s plánem S_2 , jestliže lze S_1 převést na S_2 přehozením nekonzistentních operací, tj. operací, které nejsou v relaci $je_v_konfliktu$.

Příklad 5.7

Uvažujme transakce z příkladu Příklad 5.6 a plán z Obr. 5.9 (nyní už jen pouze operace čtení a zápisu). Ten je v Obr. 5.16 označen jako S_1 . Plán S_2 v obrázku je s ním

Plán S_1		Plán S_2	
T_0	T_1	T_0	T_1
read (A, a ₀) write (A, a ₀)	read (A, a ₁) write (A, a ₁)	read (A, a ₀) write (A, a ₀) read (B, b ₀) write (B, b ₀)	read (A, a ₁) write (A, a ₁) read (B, b ₁) write (B, b ₁)
read (B, b ₀) write (B, b ₀)	read (B, b ₁) write (B, b ₁)		

Obr. 5.16 Plány ekvivalentní vzhledem ke konfliktům

ekvivalentní vzhledem ke konfliktům, protože plán S_1 lze převést na S_2 tak, že zaměníme pořadí bloku operací čtení a zápisu záznamu účtu B v transakci T_1 a bloku obsahujícího čtení a zápis záznamu účtu A v transakci T_2 . Tyto operace nejsou v konfliktu, protože přistupují k různým databázovým objektům.

Nyní už můžeme definovat klíčový pojem řízení přístupu souběžných transakcí.



Definice 5.4 Plán. uspořadatelný vzhledem ke konfliktům

Plán souběžných transakcí S se nazývá *uspořadatelný vzhledem ke konfliktům*, jestliže existuje sériový plán ekvivalentní s plánem S vzhledem ke konfliktům.



Protože uspořadatelnost plánu vzhledem ke konfliktům znamená, že stav databáze po provedení plánu je stejný jako kdyby transakce plánu probíhaly po sobě, neporušuje plán uspořadatelný vzhledem ke konfliktům konzistenci databáze. Pokud SŘBD zajistí, že souběžné transakce budou moci probíhat pouze podle takových plánů, nebude moci dojít k porušení konzistence. Toto je nejběžnější strategie řízení souběžného přístupu v databázových systémech.

Zjistit, zda je daný plán uspořadatelný vzhledem ke konfliktům, lze na základě definice uspořadatelnosti a ekvivalence vzhledem ke konfliktům. Pokud po změně pořadí provádění nekonfliktních operací v plánu získáme plán sériový, je původní plán uspořadatelný. Na základě takových úprav můžeme například prohlásit, že plán S_1 v Obr. 5.16 je uspořadatelný vzhledem ke konfliktům, protože plán S_2 , který je s ním ekvivalentní, je sériový. Potvrzuje to naše zjištění v Obr. 5.9, že tento plán konzistenci neporušuje.



Příklad 5.8

Uvažujme opět transakce z příkladu Příklad 5.6 a plán z Obr. 5.10, který je pouze s operacemi čtení a zápisu znázorněn na Obr. 5.17. Pro snadnější odkazy jsme jednotlivé operace plánu očíslovali.

	T_0	T_1
1	read (A, a_0)	
2		read (A, a_1)
3		write (A, a_1)
4		read (B, b_1)
5	write (A, a_0)	
6	read (B, b_0)	
7	write (B, b_0)	
8		write (B, b_1)

Obr. 5.17 Plán, který není uspořadatelný

Abychom dostali sériový plán, potřebovali bychom přesunout operaci 1 v plánu za všechny operace transakce T_2 nebo naopak blok operací 2 až 4 přesunout za všechny operace transakce T_1 . Ani jedna varianta není ale možná. V prvním případě například zjistíme, že operace číslo 1 (read (A, a_0)) je v konfliktu s operací 3 (write (A, a_1)), a proto není možné jejich pořadí přehodit. Ve druhém případě podobně zjistíme, že

například operace číslo 2 (read (A, a_1)) je v konfliktu s operací 5 (write (A, a_0)). Plán, který by byl ekvivalentní vzhledem ke konfliktům nalézt nelze. Už jsme viděli, že tento plán porušuje konzistenci databáze.

Už umíme rozhodnout, zda je daný plán transakcí uspořadatelný. Ukážeme si ale ještě elegantnější způsob, jak toto zjistit. Je založen na využití teorie grafů, přesněji vyjádření důsledku relace je_v_konfliktu grafem.

DEF

Definice 5.5 Graf relace precedence plánu souběžných transakcí

Grafem relace precedence plánu souběžných transakcí rozumíme orientovaný graf, jehož každý vrchol reprezentuje jednu transakci vyskytující se v plánu a každá orientovaná hrana vyjadřuje pořadí dvojice konfliktních operací v plánu.

DEF

Věta 5.1

Plán je uspořadatelný vzhledem ke konfliktům, právě když je odpovídající graf precedence acyklický.

Protože hrany grafu precedence odráží pořadí konfliktních operací v plánu, vyjadřuje graf také pořadí odpovídajících transakcí v ekvivalentním sériovém plánu, pokud existuje. Jestliže graf obsahuje cyklus, nelze transakce uspořádat, a proto ekvivalentní sériový plán neexistuje.

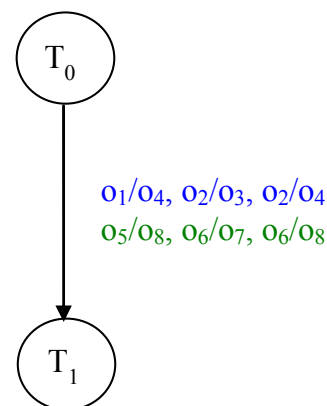
 $x+y$

Příklad 5.9

Vytvoříme graf precedence pro plán S_1 z příkladu Příklad 5.7. Abychom se mohli lépe odkazovat na jednotlivé operace plánu, očíslovujeme si je – viz Obr. 5.18 a).

	T_0	T_1
1	read (A, a_0)	
2	write (A, a_0)	
3		read (A, a_1)
4		write (A, a_1)
5	read (B, b_0)	
6	write (B, b_0)	
7		read (B, b_1)
8		write (B, b_1)

a)



b)

Obr. 5.18 Zjištění uspořadatelnosti grafem precedence:

- a) Plán S_1 s očíslovanými operacemi
b) Graf precedence

Pro ohodnocení hran v grafu si zavedeme konvenci, že ohodnocení o_j/o_k značí, že o_j je j -tá operace a o_k k -tá operace v plánu, operace jsou konfliktní a o_j předchází v plánu o_k . Dále se domluvíme, že stejně orientované hrany mezi stejnými vrcholy budeme kreslit pouze jednou s ohodnocením zahrnujícím ohodnocení všech hran, které nahrazuje.

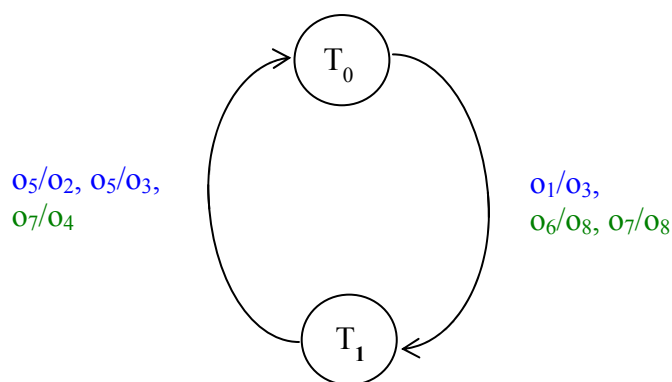
V plánu S_1 je operace číslo 1 transakce T_0 v konfliktu s operací číslo 4 transakce T_1 a v plánu ji předchází, operace číslo 2 je v konfliktu s operací číslo 3 a 4 a předchází

jím, operace číslo 5 předchází konfliktní operaci číslo 8 a podobně operace číslo 6 je v konfliktu s operacemi 7 a 8 a předchází jim. Výsledný graf precedence je nakreslen na Obr. 5.18 b). Je z něho zřejmé, že všechny operace transakce T_0 , které jsou v konfliktu s nějakou operací transakce T_1 , tuto operaci předcházejí. Bylo by tedy možné plán upravit tak, aby vznikl sériový plán ekvivalentní vzhledem ke konfliktům. Plán S_I je proto uspořádatelný a ekvivalentní sériový plán je T_0T_1 . Dospěli jsme ke stejnému závěru jako v příkladu Příklad 5.7.

Příklad 5.10

x+y

Rozhodněme použitím grafu precedence o uspořádatelnosti plánu z Obr. 5.17 příkladu Příklad 5.8. Operace číslo 1 je operace transakce T_0 , je v konfliktu s operací číslo 3 transakce T_1 a v plánu ji předchází. V grafu je jejich vztah reprezentován hranou vedoucí z uzlu T_0 do c ohodnocenou o_1/o_3 . Zbývající dvojice konfliktních operací a jim odpovídajících hran v grafu si doplňte sami. Výsledný graf precedence je na Obr. 5.19.



Obr. 5.19 Graf precedence plánu z Obr. 5.17

Protože graf obsahuje cyklus, není daný plán uspořádatelný. Nelze totiž najít ekvivalentní sériový plán, ve kterém by transakce T_0 předcházela transakci T_1 a současně transakce T_1 předcházela transakci T_0 . Dospěli jsme tedy ke stejnému závěru jako při použití úprav v příkladu Příklad 5.8.

Nyní už rozumíme významu uspořádatelnosti plánu souběžných transakcí a víme, že SRBD by měl zajistit, že mohou nastat, tedy budou dosažitelné, pouze uspořádatelné plány. V následujících odstavcích se zaměříme na techniky, kterými toho SRBD dosahuje.

Techniky plánování (rozvrhování) transakcí lze rozdělit do dvou skupin. *Pesimistické techniky* jsou takové, které preventivně zabrání vzniku situací, které by mohly (ale také nemusely) vést na porušení konzistence databáze. Naproti tomu *optimistické techniky* dávají transakcím volnost v provádění databázových operací a teprve v okamžiku, kdy by se měl výsledek transakce zpřístupnit ostatním souběžným transakcím, zjišťuje SRBD, jestli nenastala situace, která by mohla způsobit nekonzistenci databáze. Pesimistické techniky jsou poměrně jednoduché a jsou převládajícím způsobem používaným pro řízení souběžného přístupu v prostředích typických OLTP (OnLine Transaction Processing) systémů, tj. běžných systémů na podporu procesů probíhajících ve firmách a institucích, které jsou charakteristické velkým množstvím poměrně krátkých souběžných transakcí s relativně vysokou pravděpodobností možných konfliktů. Výhodou optimistických technik je, že neomezuji souběžné transakce, ale pokud se při vyhodnocování v závěru transakce

zjistí, že došlo ke konfliktu s jinou souběžnou transakcí, je transakce zrušena. Proto jsou tyto techniky vhodné pro prostředí, kdy se na jedné straně požaduje možnost souběžného přístupu, ale na druhé straně pravděpodobnost konfliktů je relativně malá, a kde mohou probíhat relativně dlouhé transakce. Příkladem takového prostředí mohou být návrhové (CAD) systémy. Nejpoužívanější pesimistické techniky jsou založeny na protokolech využívajících mechanismu *uzamykání* (*locking*). V praxi však často SRBD používá k řízení souběžného přístupu více než jednu techniku. Kromě uzamykání se potom zpravidla používá různých technik založených na *mechanismu časových razítek* (*timestamping*). My se v dalším zaměříme výhradně základní pesimistickou techniku založenou na uzamykání.

Podstata *uzamykání* spočívá v tom, že transakce přistupující k nějakému databázovému objektu, například záznamu řádku tabulky, dříve než hodnotu přečte operací *read* nebo zapíše operací *write*, požaduje uzamčení objektu zámkem, resp. přidělení zámku pro daný objekt. V praxi se používají různě složité typy zámků a uzamykacích režimů. My se zde omezíme na velice jednoduchý případ. Budeme předpokládat dva typy zámků, resp. uzamykacího režimu:

- sdílený zámek (shared lock)
- výlučný zámek (exclusive lock)

Sdílený zámek je zámek, který může současně vlastnit (tj. uzamkat tímto zámkem) několik transakcí, které s daným databázovým objektem pracují. Budeme předpokládat, že požadavek na přidělení sdíleného zámků pro objekt Q realizuje transakce *uzamykací operací* $lockS(Q)$.

Výlučný zámek je zámek, který může v každém okamžiku vlastnit nejvýše jedna transakce, které tak má k danému databázovému objektu výlučný přístup. Budeme předpokládat, že požadavek na přidělení výlučného zámků pro objekt Q realizuje transakce *uzamykací operací* $lockX(Q)$.

Na Obr. 5.20 je uvedena *matice kompatibility*, která udává závislost výsledku operace $lockS(Q)$, resp. $lockX(Q)$ na tom, jestli je již objekt Q nějakým zámkem uzamčen nebo ne. Je vidět, že pokud objekt uzamčen není, bude uzamykací operace vždy úspěšná, tj. zámek bude transakci přidělen neboli transakce objekt Q požadovaným zámkem uzamkne. Je-li již objekt uzamčen jinou transakcí, bude operace úspěšná pouze v případě, že je uzamčen zámkem sdíleným a sdílený zámek je rovněž požadován. Odpovídá to výše uvedené sémantice obou typů zámků.

	stav objektu		
	neuzamčený	uzamčený sdíleným	uzamčený výlučným
požadovaný zámek			
sdílený	ANO	ANO	NE
výlučný	ANO	NE	NE

Obr. 5.20 Matice kompatibility uzamykání sdíleným a výlučným zámkem.

V případě, že uzamykací operace není úspěšná (v tabulce kompatibility vyjádřeno hodnotou NE), transakce *čeká*, dokud jí nebude požadovaný zámek přidělen, tj. dokud transakce, které objekt uzamkají, své zámků neuvolní. Budeme předpokládat, že transakce odemkne objekt Q (vrátí či uvolní zámků, které pro daný objekt vlastní) *odemkací operací* $unlock(Q)$. Odemykací operace je jedna, nezávisí na typu zámků,

který transakce vlastní.

Vzniká otázka, kdy a jakým zámekem by měla transakce objekt, se kterým potřebuje pracovat, uzamykat, a kdy zámek uvolňovat. Musíme si uvědomit, že uzamykáním chceme dosáhnout toho, aby byly dosažitelné pouze plány uspořádatelné vzhledem ke konfliktům. Operace čtení nejsou konfliktní. Číst hodnotu objektu proto může v libovolném pořadí libovolný počet transakcí. Všechny přečtou totéž. Mohlo by se zdát, že pro čtení není potřeba uzamykat. Čtení je ale v konfliktu se zápisem. Proto je potřeba zabránit zápisu, pokud nějaká transakce hodnotu přečetla a ještě bude k objektu přistupovat. Budeme proto předpokládat následující jednoduché schéma uzamykání:

1. Transakce bude před provedením operace čtení požadovat přidělení sdíleného, resp. výlučného zámku, pokud bude hodnotu objektu pouze číst, resp. později bude do něho i zapisovat.
2. Transakce bude před provedením operace zápisu požadovat přidělení výlučného zámku, pokud daný objekt už takovým zámekem neuzamyká.
3. Transakce odemkne objekt, když už nebude potřebovat k němu přistupovat.

Příklad 5.11

$x+y$

Uvažujme dvě souběžné transakce přistupující k záznamům účtů A a B . Transakce T_0 převede částku 10000 Kč z účtu A na účet B a transakce T_1 zobrazí součet stavů obou účtů. Jeden z plánů dosažitelných při výše popsaném schématu uzamykání je uveden na Obr. 5.21 na další straně. Z důvodu větší srozumitelnosti jsou tentokrát v plánu uvedeny i operace transakcí s lokálními proměnnými. Pokud se na plán podíváte pozorně a některým z nám již známých způsobů rozhodnete o jeho uspořádatelnosti, měli byste konstatovat, že uspořádatelný není. Navíc vede na jeden z typických problémů souběžného přístupu, které jsme si uváděli – nekonzistentní analýzu. Transakce T_1 totiž zobrazí součet již nové hodnoty stavu účtu A po snížení o 10000 Kč a původní hodnoty stavu účtu B . Bude tedy zobrazovat hodnotu o 10000 Kč menší, než je hodnota v konzistentním stavu databáze.

Z uvedeného příkladu můžeme vyvodit jeden důležitý závěr. Samotné uzamykání ještě uspořádatelnost nezajišťuje. Problém našeho schématu uzamykání asi spočívá v tom, že transakce T_0 odemkla záznam účtu A příliš brzy, takže transakce T_1 mohla přečíst novou hodnotu dříve, než transakce T_0 zapsala novou hodnotu stavu účtu B . Zkusíme si proto modifikovat naše pravidlo pro odemykání tak, aby podstatně přísněji omezovalo přístup k objektu, se kterým nějaká transakce pracuje. Mohlo by znít takto:

3. Transakce odemkne všechny objekty, které uzamyká teprve po úspěšném dokončení, tj. až se dostane do stavu potvrzení, nebo když je zrušena.

Příklad 5.12

$x+y$

Uvažujme příklad Příklad 5.11 s výše uvedeno modifikací odemykání. Začátek plán dosažitelný za těchto podmínek je uveden na Obr. 5.22. Plán ukazuje stav, kdy transakce T_0 uzamčela záznam účtu A výlučným zámekem a nyní by potřebovala získat výlučný zámek na záznam účtu B . Ten ale již uzamyká transakce T_1 , která naopak čeká na uvolnění zámku pro záznam účtu A , drženého transakcí T_0 . Nastal stav, kdy ani jedna z obou transakcí nemůže pokračovat, protože se vzájemně blokuji. Takový stav se označuje jako *uváznutí (deadlock)*.

T_0	T_1
$\text{lock_X}(A)$ $\text{read}(A, a_0)$ $a_0 = a_0 - 10000$ $\text{write}(A, a_0)$ $\text{unlock}(A)$	
	$\text{lock_S}(B)$ $\text{read}(B, b_1)$ $\text{unlock}(B)$ $\text{lock_S}(A)$ $\text{read}(A, a_1)$ $\text{unlock}(A)$ $\text{display}(a_1 + b_1)$
$\text{lock_X}(B)$ $\text{read}(B, b_0)$ $b_0 = b_0 + 10000$ $\text{write}(B, b_0)$ $\text{unlock}(B)$	

Obr. 5.21 Plán dosažitelný při uzamykání, který není uspořádatelný

T_0	T_1
$\text{lock_X}(A)$ $\text{read}(A, a_0)$ $a_0 = a_0 - 10000$ $\text{write}(A, a_0)$	
	$\text{lock_S}(B)$ $\text{read}(B, b_1)$ $\text{lock_S}(A)$
$\text{lock_X}(B)$	

Obr. 5.22 Uváznutí transakcí vlivem uzamykání



Z předchozích dvou příkladů můžeme vyvodit dva důležité závěry:

1. Samotné uzamykání nezajišťuje uspořádatelnost dosažitelných plánů.
2. Uzamykání může způsobit uváznutí transakcí.

K řešení prvního problému je potřeba najít takové, tzv. *uzamykací protokoly*, které

zajistí uspořádatelnost všech dosažitelných plánů. Druhý problém lze řešit použitím protokolů, které uváznutí vylučují nebo ho připustit a případné ošetření ponechat na programátorovi nebo na SŘBD, který ho ošetří jako systémovou chybu, včetně zotavení.

Uzamykacím protokolem rozumíme soustavu pravidel, která stanoví, kdy může transakce objekty v databázi uzamykat a odemykat. Existují protokoly, které zajišťují uspořádatelnost a případně i zabráňují vzniku uváznutí. Mezi nejpoužívanější patří *dvoufázový uzamykací protokol*.

Dvoufázový uzamykací protokol (Two-Phase Locking Protocol, často se používá zkratka 2PL) je protokol, jehož podstatou je striktní oddělení dvou základních fází:

1. *Fáze růstu (growing)*. Během této fáze může transakce databázové objekty pouze uzamykat, ale žádný nesmí odemknout.
2. *Fáze zmenšování (shrinking)*. Během této fáze již může transakce pouze odemykat a žádný objekt již nesmí uzamčít.

Je zřejmé a plyne to i z názvu, že je striktně odděleno uzamykání a odemykání. První odemykací operace zahajuje druhou fázi protokolu.

Lze ukázat, že dvoufázový uzamykací protokol zajišťuje uspořádatelnost vzhledem ke konfliktům, ale nevylučuje uváznutí.

Podívejme se na plány na Obr. 5.21 a Obr. 5.22 z pohledu pravidel dvoufázového uzamykacího protokolu. Plán z Obr. 5.21 tomuto protokolu neodpovídá, protože transakce poté co odemkly záznam, požadují uzamčení jiného. V případě plánu na Obr. 5.22 může jít o dvoufázový uzamykací protokol, protože do okamžiku uváznutí transakce pouze uzamykaly.

Příklad 5.13

x+y

Uvažujme plán souběžných transakcí T_1 a T_2 uvedený na Obr. 5.23. Naším úkolem je rozhodnout, zda je dosažitelný dvoufázovým uzamykacím protokolem.

	T_1	T_2	T_3
1	read (A,a)		
2		read (B,b)	
3			read (C,c)
4	read (C,c)		
5			read (B,b)
6		write (B,b)	
7			write (A,a)
8	write (A,a)		
9	write (C,c)		

Obr. 5.23 Plán souběžných transakcí z příkladu Příklad 5.13

Pro vyřešení tohoto úkolu je nutné si uvědomit, že uspořádatelnost plánu vzhledem ke konfliktům je nutnou podmínkou pro to, aby plán mohl být dosažitelný dvoufázovým uzamykacím protokolem, protože ten uspořádatelnost zajišťuje. Zjistíme-li, že plán

není uspořádatelný vzhledem ke konfliktům, nemůže ani být dosažitelný dvoufázovým uzamykacím protokolem. Na druhé straně, je-li uspořádatelný, neznámá to nutně, že musí být dosažitelný dvoufázovým uzamykacím protokolem. Mohou existovat uspořádatelné plány, které nejsou dvoufázovým uzamykacím protokolem dosažitelné. Matematik by to jednoduše zdůvodnil tak, že uspořádatelnost je podmínka nutná, ale ne postačující.



Rozhodnout o uspořádatelnosti již umíte, takže nebude pro vás probléme tento příklad dořešit.

Uvažujme transakci T_0 , která převádí částku z účtu A na účet B a transakci T_1 , která zobrazí součet stavů na obou účtech (ukázány jsou jen operace čtení a zápisu):

T_0	T_1
read (A, a_0)	read (A, a_0)
write (A, a_0)	read (B, b_0)
read (B, b_0)	
write (B, b_0)	

Při použití dvoufázového uzamykacího protokolu: musí transakce T_0 uzamknout záznam účtu A výlučným zámek (bude do něho hodnotu zapisovat) už před jeho čtením. To zabraňuje jakémukoliv přístupu k záznamu jiným souběžným transakcím. Ve skutečnosti ale potřebuje transakce T_0 výlučný přístup k záznamu až při zápisu. Pokud by mohla zpočátku uzamknout záznam ve sdíleném režimu a teprve před zápisem přísnost uzamčení zvýšit na výlučné, mohly by transakce T_0 i T_1 číst záznam účtu A současně. Zavedeme si proto další dvě operace pro konverzi režimu uzamykání a zpřesníme si s ohledem na to dvoufázový uzamykací protokol. Pro konverzi zámku drženého transakcí pro objekt Q ze sdíleného na výlučný zavedeme operaci *upgrade* (Q) a pro konverzi opačnou *downgrade* (Q). Potom operace *upgrade* (Q) se při použití dvoufázového uzamykacího protokolu může vyskytovat pouze ve fázi růstu a operace *downgrade* (Q) pouze ve fázi zmenšování.

Začátek plánu transakcí T_0 a T_1 dosažitelný takto zpřesněným dvoufázovým uzamykacím protokolem je uveden na

T_0	T_1
lock_S(A)	
read (A, a_0)	
	lock_S (A)
	read (A, a_1)
upgrade (A)	
write (A, a_0)	
...	...

Obr. 5.24 Úsek plánu transakcí s operací *upgrade*

Nyní můžeme popsat jednoduché schéma, které může být použito ke generování vhodných uzamykacích operací pro transakci. Jestliže transakce požaduje číst hodnotu databázového objektu Q , vygeneruje se uzamykací operace $lockS(Q)$ následovaná operací čtení $read(Q, q)$. Jestliže transakce požaduje zapsat hodnotu do objektu Q , systém kontroluje, zda transakce již tento objekt uzamyká sdíleným zámkem. Pokud ano, generuje uzamykací operaci $upgrade(Q)$ následovanou operací zápisu $write(Q, q)$. V opačném případě generuje uzamykací operaci $lockX(Q)$ a operaci zápisu $write(Q, q)$.

Dvoufázový uzamykací protokol může být implementován různým způsobem. Jednoduchá implementace je taková, kdy transakce postupně objekty uzamyká a odemkne je, až se dostane do stavu potvrzená nebo zrušená. V této souvislosti se můžete v literatuře setkat s variantami *striktní* a *rigorózní dvoufázový uzamykací protokol*. V prvním případě jde o variantu, kdy transakce uvolňuje výlučné zámky až ve stavu potvrzená, resp. zrušená, sdílené může uvolnit dříve. Ve druhém případě transakce uvolňuje všechny zámky až ve stavu potvrzená nebo zrušená.

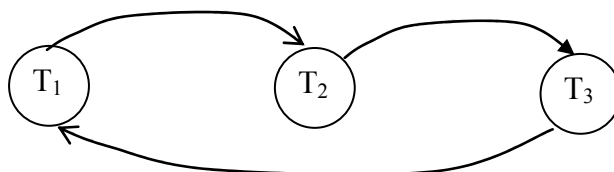
Uzamykání a odemykání databázových objektů má u SŘBD na starosti komponenta zvaná *správce uzamykání (lock manager)*. Ten spravuje informaci o uzamčených objektech a čekajících transakcích. Může být uložena například v hašované tabulce zámků.

Doposud jsme mlčky předpokládali, že transakce uzamyká záznam nesoucí hodnoty jednoho řádku tabulky. To jednak nemusí být vždy pravda, jednak jsou často k dispozici prostředky, kterými může programátor ovlivnit, jak velká část databáze podléhá uzamykací operaci. Hovoříme potom o tzv. *granularitě uzamykání*. Typickými úrovněmi granularity jsou řádek tabulky, fyzický blok s řádkem tabulky, celá tabulka a celá databáze. Vždy existuje úroveň, na které provádí SŘBD implicitně a tuto úroveň lze nějak změnit. Například databázový server Oracle provádí uzamykání na úrovni řádků tabulky, ale poskytuje příkaz SQL LOCK TABLE, který umožňuje uzamknout celou tabulku až do okamžiku potvrzení či zrušení transakce. Server SQL Base firmy Gupta uzamyká celé fyzické bloky a poskytuje příkazy LOCK DATABASE, resp. UNLOCK DATABASE pro uzamknutí, resp. odemknutí celé databáze.

Obecně platí, že čím jemnější je granularita uzamykání, tím větší je možná míra souběžnosti. Může ale znamenat vyšší režii při intenzivní práci s tabulkou či databází. Jestliže budeme například jedním prohledávacím příkazem UPDATE modifikovat všechny nebo značnou část řádků tabulky, představovalo by uzamykání jednotlivých řádků značnou režii, která by u rozsáhlé tabulky mohla znamenat neúměrné prodloužení operace. V takovém případě může být užitečné uzamknout na dobu provedení operace celou tabulku výlučným zámkem. Tím sice po tuto dobu nebude moci pracovat s tabulkou žádná jiná transakce, ale operace proběhne rychleji a kompletní data tabulky budou dříve k dispozici ostatním transakcím. Analogická situace může být při čtení, například při rozsáhlém tisku, archivaci apod.

Když jsme si vysvětlovali podstatu uzamykání a ukazovali použití uzamykacích operací, narazili jsme na problém možného *uváznutí transakcí (deadlock)*. Tento jev může nastat tehdy, kdy transakce musí čekat, až se uvolní nějaké systémové prostředky (v případě uzamykání zámek), které vlastní jiná transakce, ale ta je také nemůže uvolnit. Podívejme se nyní, jaké jsou možnosti řešení.

První možností je použití takového protokolu, který uzáznutí vyloučí. Viděli jsme ale, že nejčastěji používaný dvoufázový uzamykací protokol uzáznutí nezabraňuje. V praxi proto SŘBD spíše uzáznutí detekuje, až nastane nebo mohl-li nastat. V prvním případě SŘBD dat může využít k detekci informací o čekajících transakcích a analyzovat tzv. *graf čekání* (*wait-for graph*). Můžeme si ho představit jako orientovaný graf, jehož uzly představují čekající transakce a orientovaná hrana (T_i, T_j) vyjadřuje fakt, že transakce T_i čeká, až se uvolní nějaký zámek držený transakcí T_j . Cyklus v grafu indikuje uzáznutí. Na Obr. 5.25 je uveden příklad grafu čekání, ve kterém transakce T_1 čeká na uvolnění zámku drženého transakcí T_2 a ta naopak čeká, až jí uvolní nějaký zámek transakce T_3 . Transakce se vzájemně blokují.



Obr. 5.25 Příklad grafu čekání indikujícího uzáznutí

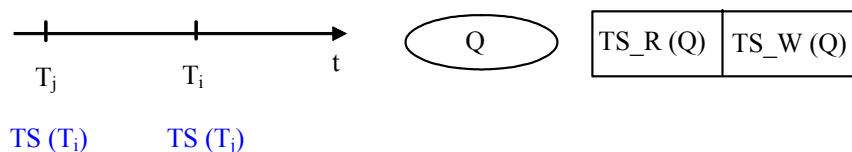
Předpokládejme, že transakce T_3 z obrázku teprve požaduje uzamčení nějakého objektu, jehož zámek drží transakce T_1 a správce uzamykání zjistí, že přidělením zámku by vznikla situace znázorněná na obrázku, tedy uzamčení by vedlo k uzáznutí, správce zámek transakci T nepřidělí a naopak signalizuje chybu, kterou by měl ošetřit programátor zrušením transakce nebo novým pokusem o provedení operace po nějaké prodlevě.

Jinou možností, která zabrání uzáznutí, je nastavení maximální doby čekání na přidělení zámku. I v tomto případě by měl programátor ošetřit příslušnou chybu.



Pro ilustraci si uvedeme i jeden *protokol založený na časových razítkách* (*timestamp-based*). Časovým razítkem budeme rozumět časový údaj vztažený k nějaké události v databázi, v tomto případě související s transakčním zpracováním. Podstata protokolů založených na časových razítkách spočívá v tom, že transakcím a databázovým objektům jsou přiřazena časová razítka, která nesou informaci o čase určitých operací. Používají se potom při zajištění uspořadatelnosti. Uvedeme si příklad *protokolu s uspořádáním časových razítek* (*timestamp-ordering*).

Předpokládejme, že každé transakci T_i je při jejím zahájení přiřazeno časové razítko $TS(T_i)$, jehož hodnota je unikátní (nemusí nutně vyjadřovat explicitně čas, ale například pořadové číslo). Každému databázovému objektu Q , např. řádku tabulky, je přiřazena dvojice razítek $TR_S(Q)$, resp. $TW_S(Q)$. Hodnota každého z nich bude odpovídat nejvyššímu časovému razítku transakce, která hodnotu objektu Q četla, resp. zapisovala – viz Obr. 5.26. Protokol zajistí, že budou dosažitelné pouze takové plány, které jsou ekvivalentní vzhledem ke konfliktům se sériovým plánem s transakcemi uspořádanými vzestupně podle hodnoty jejich časových razítek, tedy tak, jak byly zahajovány.



Obr. 5.26 Časová razítka protokolu s uspořádáním časových razítek

Dosažitelné budou proto pouze takové plány, u nichž budou konfliktní operace uspořádány podle časových razítek odpovídajících transakcí. Jestliže například transakce T_i bude chtít číst hodnotu nějakého Q , nesmí existovat transakce, která začala později, ale už zapsala hodnotu objektu Q . Jestli taková situace nastala, se pozná podle hodnoty razítka $TW_S(Q)$. Jestli bude chtít transakce T_i hodnotu do objektu Q zapsat, nesměla žádná transakce, která byla zahájena později, před transakcí T_i hodnotu objektu Q ani číst ani zapsat. Plyne to ze skutečnosti, že operace zápisu je v konfliktu jak se čtením, tak zápisem. Nyní již můžeme zformulovat pravidla protokolu.

Pro operaci *read* (Q, q) požadovanou transakcí T_i :

```

if TS( $T_i$ ) < TS_W( $Q$ ) then
    /* hodnota již přepsána pozdější transakcí */
    rollback( $T_i$ )
else
begin
    read( $Q, q$ ) ;
    TS_R( $Q$ ) = max {TS_R( $Q$ ) , TS( $T_i$ ) }
end

```

Pro operaci *write* (Q, q) požadovanou transakcí T_i :

```

if TS( $T_i$ ) < TS_R( $Q$ ) then
    /* hodnota již přečtena pozdější transakcí */
    rollback( $T_i$ )
else
    if TS( $T_i$ ) < TS_W( $Q$ ) then
        /* hodnota je zastaralá */
        rollback( $T_i$ )
    else
begin
    write( $Q, q$ ) ;
    TS_W( $Q$ ) = TS( $T_i$ )
end

```

Pokud jste podmínky protokolu dobře pochopili, tak vám neuniklo, že transakce nikdy nečeká. Buď je operace čtení či zápisu provedena nebo je transakce zrušena. Z toho by vám mělo vyplynout, že nikdy nemůže nastat uváznutí transakcí. Můžeme tedy konstatovat, že uvedený protokol zajišťuje uspořadatelnost vzhledem ke konfliktům a zabráňuje uváznutí. Za tuto výhodu ale platíme tím, že jsou znevýhodněny delší transakce, u kterých s větší pravděpodobností dochází ke zrušení vlivem konfliktu.



Závěrem uvedme tři poznámky.

1. Reálné SRBD často používají větší počet režimů uzamykání, resp. typů zámků. Například databázový server Oracle rozlišuje zámky úrovně řádku a zámky úrovně tabulky. V prvním případě o obdobu zámků, jak jsme je uvažovali my



s tou výjimkou, že při čtení se záznamy neuzamýkají, takže se používá pouze výlučný zámek a to při zápisu. Často ale dochází před provedením operací k uzamčení celé tabulky některým z následujících režimů uzamčení tabulky:

- *Sdílení řádků (Row Share Table Lock – RS)* – používá se v případě příkazu SELECT s klauzulí FOR UPDATE, tj. při použití kurzoru tohoto typu. Jde o nejméně přísný režim uzamčení tabulky, který pouze říká, že transakce bude uzamykat některé řádky výlučným zámkem. Umožňuje jiným transakcím číst a modifikovat data tabulky. Neumožní jiné transakci uzamknout celou tabulku příkazem LOCK TABLE tabulka IN EXCLUSIVE MODE, tj. výlučným zámkem typu X.
 - *Výlučný přístup k řádkům (Row Exclusive Table Lock – RX)* – tento režim indikuje, že transakce jeden nebo více řádků tabulky. Umožňuje jiným transakcím číst a modifikovat data tabulky. Neumožní jiné transakci uzamknout celou tabulku příkazem LOCK TABLE tabulka IN SHARE | EXCLUSIVE | SHARE EXCLUSIVE MODE, tj. zámků typu S, .X a SRX.
 - *Sdílená tabulka (Share Table Lock – S)* – v tomto režimu se uzamkne tabulka provedením příkazu LOCK TABLE tabulka IN SHARE MODE. Umožňuje jiným transakcím pouze číst data, modifikovat řádky vybrané příkazem SELECT s klauzulí FOR UPDATE a uzamknout celou tabulku ve sdíleném režimu, tj. stejným typem zámků – S. Neumožní jiné transakci uzamknout celou tabulku příkazem LOCK TABLE tabulka IN EXCLUSIVE | SHARE EXCLUSIVE MODE, tj. zámků typu X a SRX.
 - *Sdílená tabulka s výlučným přístupem k řádkům (Share Row Exclusive Table Lock – SRX)* – v tomto režimu se uzamkne tabulka provedením příkazu LOCK TABLE tabulka IN SHARE ROW EXCLUSIVE MODE. Je to přísnější režim než sdílená tabulka. Pouze jedna transakce může v daném okamžiku uzamykat tabulku tímto typem zámků. Jiné transakce mohou pouze číst nebo modifikovat konkrétní řádky vybrané příkazem SELECT s klauzulí FOR UPDATE. Žádná jiná transakce nemůže uzamknout tabulku příkazem LOCK TABLE.
 - *Výlučný přístup k tabulce (Table Exclusive Lock – X)* – v tomto režimu se uzamkne tabulka provedením příkazu LOCK TABLE tabulka IN EXCLUSIVE MODE. Je to nejpřísnější režim uzamknutí tabulky, ve kterém ostatní transakce mohou z tabulky data pouze číst příkazem SELECT.
2. Reálné SŘBD někdy kombinují uzamykání ještě s další metodou řízení souběžného přístupu s cílem zajistit vyšší úroveň souběžnosti. Server Oracle takto kombinuje uzamykání s *verzováním (Multiversion Concurrency Control)*. Podstata verzování spočívá v tom, že jsou k dispozici nejenom aktuální hodnoty databázových objektů, nýbrž i hodnoty dřívější, které byly později přepsané. Víme, že tyto hodnoty jsou k dispozici v žurnálu, v případě Oracle v tzv. *rollback segmentech*. To umožňuje, aby žádné uzamčení řádku či celé tabulky nebránilo dotazování nad tabulkou. Proto také nejsou při čtení příkazem SELECT řádky uzamýkány žádným řádkovým zámkem. Oracle poskytuje dva režimy pro čtení konzistentních dat – *konzistentní čtení úrovně*

příkazu (*statement-level read consistency*) a *konzistentní čtení úrovně transakce (transaction-level read consistency)*. V obou případech je zajištěno, že jsou zpřístupněna konzistentní data, platná v jistém časovém okamžiku. V prvním případě je tím okamžikem zahájení transakce, ve druhém začátek provádění příkazu. Příkaz SELECT tak nepřečte data, která by nebyla ještě potvrzena nebo byla potvrzena nějakou transakcí až v době provádění tohoto příkazu. Asi tušíte, že při práci s verzemi a konzistencí vztaženou k začátku transakce nebo začátku provádění příkazu, používá SŘBD časových razítek. Ta zde mají podobu unikátních *systémových čísel změny (system change number - SCN)*.

3. Předpokládali jsme pouze operace čtení a zápisu do databáze a nezabývali jsme se operacemi vkládání nových záznamů a rušení záznamů existujících. Mohli bychom si zavést odpovídající dvě operace a diskutovat, které z nyní již čtyř operací jsou v konfliktu. Zjistili bychom, že operace rušení i vkládání je v konfliktu se všemi operacemi, podobně jako zápis. Při zamykání proto bude vyžadovat uzamknutí výlučným zámekem.

5.6. Zotavení souběžných transakcí

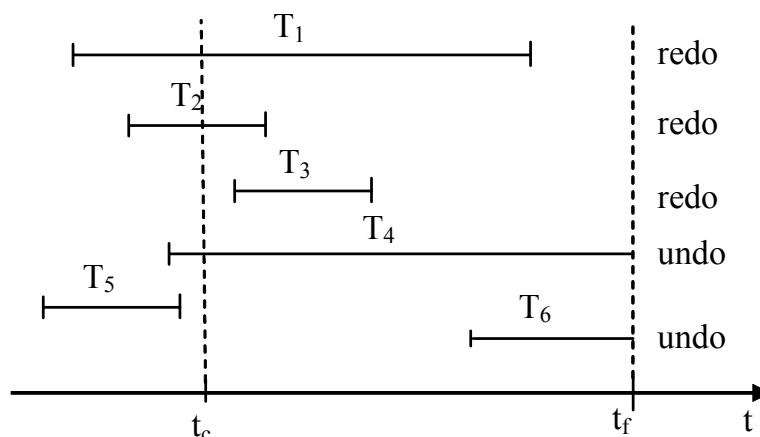
V předchozích dvou kapitolách jsme se odděleně zabývali zotavením po chybách a poruchách a řízením souběžného přístupu. Nyní oba okruhy problémů spojíme a ukážeme si, jak proběhne zotavení v případě, že připustíme existenci souběžných transakcí.

Ve skutečnosti jsme na zotavení souběžných transakcí již dostatečně připraveni. Když si vzpomenete, co jsme si řekli o záznamu žurnálu, který potvrzuje provedení kontrolního bodu, pak víte, že tento záznam obsahuje seznam všech transakcí probíhajících v okamžiku kontrolního bodu. Připomeňme jeho tvar $\langle \text{checkpoint}, T_1, T_2, \dots, T_k \rangle$. Víme také, že zotavení při chybě či poruše vyžadující restart SŘBD probíhá od posledního kontrolního bodu. Můžeme tedy bez problému rozšířit naši znalost zotavení o souběžné transakce.

Příklad 5.14

Předpokládejme průběh transakcí podle Obr. 5.27.

$x+y$



Obr. 5.27 Zotavení souběžných transakcí

Zotavení proběhne v následujících třech krocích:

1. Analýza žurnálu. V tomto kroku správce prochází žurnál od konce a klasifikuje transakce, o nichž je v žurnálu záznam na dokončené a nedokončené. Dokončená je taková transakce T_i , u které najde v žurnálu záznam $\langle T_i, \text{commit} \rangle$. Pokud správce zotavení narazí na záznam $\langle T_i, \text{start} \rangle$ a transakce T_i není v seznamu dokončených transakcí, zařadí ji do seznamu nedokončených transakcí. Jakmile narazí na záznam o kontrolním bodu $\langle \text{checkpoint}, T_1, T_2, \dots, T_k \rangle$, doplní do seznamu nedokončených transakcí ty transakce uvedené v záznamu, které nejsou v seznamu dokončených transakcí. Jsou to ty, které začaly před kontrolním bodem a neskončily do okamžiku poruchy.
2. Postupně je na všechny transakce klasifikované jako nedokončené aplikována zotavovací procedura *undo*, tj. jsou zapsány původní hodnoty. Zápisy se provádí ve směru od konce žurnálu.
3. Postupně je na všechny transakce klasifikované jako dokončené aplikována zotavovací procedura *redo*, tj. jsou znovu zapsány nové hodnoty. Zápisy se provádí ve směru od posledního kontrolního bodu.

5.7. Zotavení a souběžný přístup v SQL

Standard SQL-92 požaduje, aby byla zajištěna uspořádatelnost dosažitelných plánů souběžných transakcí. Standard neříká, jakým způsobem toho má být dosaženo a neobsahuje ani žádný příkaz pro explicitní uzamykání. Existuje pouze příkaz pro nastavení některých parametrů transakce:

SET TRANSACTION volby

kde *volby* umožňují specifikovat, zda transakce bude pouze číst (READ ONLY) nebo i zapisovat (READ WRITE), umožňuje nastavit velikost určité diagnostické oblasti a umožňuje nastavit izolační úroveň transakce. První parametr dává SRBD možnost efektivního řízení souběžnosti, když ví, že transakce nebude modifikovat žádná data.

Podívejme se trochu blíže na poslední parametr příkazu – *izolační úroveň*. Ta určuje, do jaké míry bude transakce izolována od ostatních souběžných transakcí, tj. jak přísná pravidla na izolovanost budou kladena.

Standard definuje tři způsoby porušení izolovanosti transakce a pomocí nich potom čtyři izolační úrovně – viz Obr. 5.28.

	čtení nepotvrzené hodnoty (dirty read)	neopakovatelné čtení (nonrepeatable read)	fantomy (phantoms)
READ UNCOMMITTED	ano	ano	ano
READ COMMITTED	ne	ano	ano
REPEATABLE READ	ne	ne	ano
SERIALIZABLE	ne	ne	ne

Obr. 5.28 Izolační úrovně v SQL

Definovaná porušení uspořádání jsou:

- *Čtení nepotvrzené hodnoty (dirty read)* značí, že transakce může přečíst dosud nepotvrzenou hodnotu, tj. připouští se porušení uspořádatelnosti typu závislosti na potvrzení, jak jsme si je ukazovali.
- *Neopakovatelné čtení (nonrepeatable read)* značí, že transakce při opakovaném čtení nemusí načíst vždy stejnou hodnotu. Jinými slovy transakce má možnost vidět hodnoty potvrzené mezi dvěma po sobě jdoucími čteními.
- *Fantomy (phantoms)* značí, že při opakovaném načtení skupiny řádků tabulky splňujících jistou podmínku se u druhého čtení mohou objevit řádky, které u prvního čtení nebyly nebo naopak některé zmizí. Jinými slovy transakce uvidí vložené nebo naopak neuvidí zrušené řádky potvrzené v průběhu provádění transakce.

Izolační úroveň READ UNCOMMITTED potom umožňuje všechna tři porušení, READ COMMITTED zabraňuje čtení nepotvrzené hodnoty, REPEATABLE READ navíc zajišťuje opakované načtení téže hodnoty a SERIALIZABLE zabraňuje všem třem způsobům porušení uspořádatelnosti.



SŘBD často poskytují jen některé z uvedených izolačních úrovní, případně některé jiné (např. CURSOR STABILITY u SQL Base od Gupty). Databázový server Oracle poskytuje implicitně izolační úroveň REPEATABLE READ. Druhou úroveň, kterou lze nastavit příkazem SET TRANSACTION, je SERIALIZABLE. Pokud jste si přečetli předchozí poznámku o uzamykání u Oracle, pak možná tušíte, že nastavení izolační úrovně SERIALIZABLE mimo jiné vyvolá režim konzistentního čtení úrovně transakce, které právě zabraňuje vzniku fantomů.



V této kapitole jsme si vysvětlili jeden velmi důležitý pojem, se kterým se setkáváme u databázových systémů a to nejen relačních. Jde o pojem transakce. Řekli jsme si, že jde o posloupnost databázových operací, které tvoří z hlediska zpracování celek – buď jsou provedeny všechny operace tvořící transakci nebo žádná z nich. Tato vlastnost transakce se nazývá atomičnost. Vysvětlili jsme si ještě další tři důležité vlastnosti, které od transakce požadujeme. Jsou to konzistence, izolace a trvalost změn. Souhrnně se označují tyto čtyři základní vlastnosti transakce podle prvních písmen anglických názvů jako ACID vlastnosti. Dále jsme si ukázali, v jakých stavech se transakce může nacházet a že koncovým stavem je buď stav, kdy transakce potvrdí všechny provedené změny, a tím jsou považovány za trvalé, nebo jsou naopak všechny provedené změny anulovány a transakce je zrušena. Rovněž jsme si vysvětlili, že pro zahájení transakce není v jazyce SQL žádný speciální příkaz, nýbrž je transakce zahájena implicitně prvním příkazem, který provádí nějakou operaci s daty v databázi. Ukončit transakci lze příkazem COMMIT, který potvrzuje změny, nebo příkazem ROLLBACK, který má naopak za následek zrušení transakce.

Ve zbývajících částech kapitoly jsme se věnovali tomu, jak SŘBD zajišťuje ACID vlastnosti a to při situacích, které by mohly způsobit jejich nedodržení. Jde o chyby a poruchy a o souběžně probíhající transakce. V prvním případě jsou ohroženy vlastnosti atomičnost, konzistence a trvalost změn. Aby je SŘBD zajistil, provádí po poruchách zotavení databáze, resp. po poruše a ztrátě dat na disku nejprve obnovu databáze. Vysvětlili jsme si, že pro tyto účely si zpravidla uchovává určité informace o průběhu transakcí v datové struktuře zvané žurnál a pro případ obnovy databáze také zálohuje (archivuje) data z databáze na disk.

V případě souběžného provádění několika transakcí může dojít k nedodržení vlastností transakce konzistence a izolace. Aby je SŘBD zajistil, musí souběžný přístup k datům v databázi řídit. V této souvislosti jsme si vysvětlili důležitý pojem - uspořadatelnost vzhledem ke konfliktům a ukázali jsme si použití uzamykání k zajištění uspořadatelnosti. Víme, že používaným uzamykacím protokolem, který uspořadatelnost zajišťuje, je dvoufázový uzamykací protokol.

V závěru jsme si řekli něco i o tzv. izolačních úrovních, které umožňují programátorovi zmírnit omezení kladená na transakce z hlediska jejich izolace.



Informace k některým tématům z oblasti transakčního zpracování můžete najít v základní literatuře [Pok98] na stranách 57 – 60, 69 – 72, 187 – 190, 205 – 208 a 217 – 220. Učebnice [Sil05] vysvětluje problematiku správy transakcí v ještě větším rozsahu, než jsme se mu věnovali my, v části 5 na str. 609 až 720. Zájemce o hlubší studium problematiky transakčního zpracování lze odkázat na knihu Kifer, M: Database Systems. Application Oriented Approach, Addison Wesley, 2005.



Cvičení:

- C5.1** Vysvětlíte pojem transakce.
- C5.2** Vysvětlíte ACID vlastnosti transakce.
- C5.3** Vysvětlíte stavy transakce, kdy se transakce do daného stavu dostane a co provádí v daném stavu pro transakci SŘBD.
- C5.4** Vysvětlíte pojmy SQL prostředí, SQL agent, SQL klient a SQL server, jak jsou vymezeny ve standardu SQL-92, a vysvětlíte, jakým způsobem navazuje a ukončuje SQL klient spojení s SQL serverem.
- C5.5** Vysvětlíte, jakými příkazy je v SQL ohraničena transakce a vysvětlíte vztah ukončujících příkazů ke koncovým stavům transakce.
- C5.6** Vysvětlíte, co značí plochý (flat) model transakce v SQL a jaký význam v tomto smyslu mají příkazy SAVEPOINT a ROLLBACK.
- C5.7** Vysvětlíte, jaké typy chyb a poruch se mohou při činnosti databázového systému vyskytnout a které z požadovaných vlastností transakcí jimi mohou být porušeny. Vysvětlíte proč.
- C5.8** Vysvětlíte rozdíl mezi třemi typy paměti – energeticky závislou, energeticky nezávislou a stabilní.
- C5.9** Vysvětlíte, proč je potřeba, aby SŘBD provedl po restartu počítače zotavení databáze.
- C5.10** Vysvětlíte, co je to žurnál a k čemu a jak ho SŘBD používá.
- C5.11** Vysvětlíte, jak probíhá zotavení při použití žurnálu s okamžitou modifikací databáze. Vysvětlíte přesně význam zotavovacích procedur.
- C5.12** Vysvětlíte význam kontrolního bodu (checkpoint) a jak probíhá. Zaměřte se na správné stanovení pořadí ukládání různých typů informace.
- C5.13** Vysvětlíte, jaké jsou kladeny požadavky na správu vyrovnávací paměti z hlediska zotavení po chybách a poruchách. Zaměřte se na správné stanovení pořadí ukládání různých typů informace.
- C5.14** Vysvětlíte, jakým způsobem lze zajistit vlastnosti proběhlých transakcí při ztrátě dat na disku.
- C5.15** Vysvětlíte, jaké problémy pro zpracování transakcí přináší souběžné provádění transakcí, proč transakce probíhají souběžně a které z požadovaných vlastností transakce mohou být nedodrženy.
- C5.16** Vysvětlíte pojmy plán souběžných transakcí a sériový plán. Vysvětlíte význam a využitelnost sériových plánů.

- C5.17** Vysvětlete, co značí řízení souběžného přístupu a vysvětlete typické problémy, které je třeba při řízení řešit.
- C5.18** Vysvětlete, co značí, že databázové operace jsou v konfliktu.
- C5.19** Vysvětlete, co značí uspořadatelnost plánu souběžných transakcí vzhledem ke konfliktům.
- C5.20** Použitím grafu precedence rozhodněte, zda by mohl být níže uvedený plán transakcí T_1 , T_2 a T_3 dosažitelný dvoufázovým uzamykacím protokolem. Svůj závěr řádně zdůvodněte. V grafu precedence ohodnoťte hrany ve tvaru o_i/o_j , kde i , resp. j je číslo operací, které jsou v analyzovaném plánu v konfliktu.

Č. operace	T_1	T_2	T_3
1	read (A,a)		
2		read (B,b)	
3			read (C,c)
4	read (C,c)		
5			read (B,b)
6		write (B,b)	
7			write (A,a)
8	write (A,a)		
9	write (C,c)		

- C5.21** Vysvětlete rozdíl mezi pesimistickými a optimistickými technikami řízení souběžného přístupu.
- C5.22** Vysvětlete podstatu řízení souběžného přístupu na bázi uzamykání a uveďte, jestli zajišťuje uzamykání uspořadatelnost vzhledem ke konfliktům..
- C5.23** Vysvětlete podstatu dvoufázového uzamykacího protokolu a uveďte jeho vlastnosti.
- C5.24** Vysvětlete, co značí granularita uzamykání a uveďte, jaké jsou typické úrovně uzamykání.
- C5.25** Vysvětlete pojem uváznutí transakcí a vysvětlete, jak se mu lze vyhnout, resp. jak ho SRBD řeší.
- C5.26** Vysvětlete, jak probíhá zotavení po poruše v případě, že mohou probíhat souběžné transakce.
- C5.27** Co říká standard SQL-92 o řízení souběžného přístupu, jaké příkazy pro uzamykání a odemykání definuje.
- C5.28** Vysvětlete pojem izolační úroveň a jaké úrovně s jakým významem standard SQL-92 zavádí.

Test



T1. Vlastnost izolace (izolovanosti) u transakce znamená, že:

- a) transakce není závislá na datech, se kterými pracuje
- b) transakce je izolována od případných poruch

T2. Příkaz SQL:

- a) tvoří implicitní transakci
- b) tvoří transakci pouze, následuje-li bezprostředně za ním příkaz COMMIT

T3. Data na disku mohou být po restartu počítače v nekonzistentním stavu pouze proto, že:

- a) některé transakce do okamžiku poruchy neskončily
- b) některé transakce do okamžiku poruchy neskončily nebo některá aktuální data byla pouze ve vyrovnávací paměti

T4. Operace write (A,a) transakce:

- a) zapisuje hodnotu objektu A z lokální proměnné transakce do vyrovnávací paměti
- b) zapisuje hodnotu objektu A z lokální proměnné transakce na disk:

T5. Dvě instrukce souběžných transakcí jsou v konfliktu,:

- a) jestliže alespoň jedna z nich je operace zápisu (write), mohou pracovat s různými objekty
- b) pouze když obě jsou operace zápisu (write) a obě operace pracují se stejným objektem

T6. Standard SQL/92 pro řízení souběžného přístupu:

- a) neobsahuje žádný příkaz pro uzamykání
- b) obsahuje příkaz LOCK, odemykání je automatické na konci transakce

6. Trendy v databázových technologiích



Cíl: V této kapitole se seznámíte se dvěma z významných trendů v oblasti databázových technologií – s databázemi založenými na objektech a s přístupem k databázím z WWW.

Anotace: Objektový přístup, objektová databáze, objektově-relační databáze, uživatelem definovaný typ, databázová aplikace s webovým rozhraním, vícevrstvá architektura s webovým prohlížečem, skriptovací jazyk PHP, Internet Information Server, Active Server Pages, JDBC, SQLJ.

Prerekvizitní znalosti: Pro zvládnutí této kapitoly se předpokládá znalost základních principů objektové orientace na úrovni základních pojmů, jako je objekt, třída, operace a metoda, zapouzdření, dědičnost, polymorfismus. Dále budeme předpokládat znalost relačního modelu dat.



Odhad doby studia: 3 hodiny.

6.1. Úvod

Od doby definitivního prosazení se relačních databází na poli databázových technologií v 80. letech minulého století můžeme zaznamenat několik významných trendů v dalším rozvoji databázové teorie i samotných databázových technologií. Mezi tyto trendy patřila snaha o přímou podporu některých aspektů důležitých pro aplikace databázových technologií. Vznikla tak řada modelů i databází, které nesou speciální označení, ale velmi často vycházejí z relačního základu. Příkladem mohou být *temporální databáze*, jejichž snahou je poskytnout podporu pro práci s časem na úrovni modelu dat a dotazovacího jazyka. Jiným příkladem mohou být *deduktivní databáze*, jejichž cílem je skloubení logického programování a databázových technologií. Známým dodazovacím jazykem takových databází je jazyk *Datalog*. Zabudování prostředků pro možnost aktivní reakce databáze na různé události bylo cílem *aktivních databází*.

Kromě těchto aktivit k podpoře některých významných obecných aspektů vznikaly technologie na podporu oblastí aplikací, pro které relační model a na něm postavené klasické relační databáze nepředstavovaly nejlepší řešení. Typickým prostředím aplikací klasických relačních databází je prostředí, ve kterém pracuje současně zpravidla velký počet uživatelů s poměrně jednoduchými daty (n-ticemi skalárních hodnot). Operace, které s těmito daty provádějí, jsou také relativně jednoduché. Můžeme tedy říci, že pro takové prostředí je charakteristické poměrně velké množství souběžných, poměrně jednoduchých a relativně krátkých transakcí.

Zejména od přelomu 80. a 90. let minulého století se ale ve větší míře začaly objevovat oblasti aplikací, které byly charakteristické odlišnými požadavky na správu perzistentních dat. Uvedme některé z nich:

- *Návrhové systémy (CAD – Computer-Aided Design, CASE – Computer-Aided Software Engineering apod.).* Lze je charakterizovat především složitými objekty se kterými jsou prováděny poměrně složité operace. Objekty mají často hierarchickou strukturu, tedy jednou z důležitých vazeb je vazba „je částí“. Narozdíl od relačních tabulek, které často mají mnoho řádků, tj. existuje mnoho záznamů téhož typu, u návrhových databází se složité objekty často vyskytují v jedné nebo jen několika instancích. S těmito objekty zpravidla souběžně pracuje nejvýše několik uživatelů a odpovídající transakce jsou poměrně dlouhé. Velký důraz je také kladen na správu verzí objektů, což v relačních databázích běžné není – modifikací řádku tabulky se původní hodnota ztrácí.
- *Geografické informační systémy.* Z pohledu požadavků na správu perzistentních dat je lze charakterizovat potřebou ukládat jak běžná strukturovaná data, jak je známe z relačních databází, tak především geografická *prostorová data (spatial data)*. Ta opět představují data se zpravidla složitější strukturou, ale především vyžadují efektivní zodpovídání tzv. *prostorových dotazů*.
- *Informační systémy úřadů (Office information systems).* Pro ty je charakteristickým požadavkem efektivní správa dokumentů, podpora pro vyhledávání v nich, případně i podpora pro jejich oběh (workflow). Navíc se v poslední době dostávají do popředí kromě nestrukturovaných dokumentů také dokumenty strukturované či částečně strukturované. Z hlediska formátu zde sehrává stále významnější roli značkovací jazyk XML.
- *Multimediální databáze.* Ty vyžadují schopnost ukládat a efektivně zpřístupňovat data v podobě různých typů médií jako je obraz, video, zvuk, text apod. Klasické relační databáze neposkytovaly dostatečnou podporu ani pro ukládání těchto dat. Aplikace pracující s multimediální databází navíc očekávají od SŘBD podporu pro efektivní přístup k takovým datům, včetně *vyhledávání podle obsahu (content-based retrieval)*.
- *Datové sklady (Data warehouses).* Zmínili jsme se o nich už v souvislosti s výkladem materializovaných pohledů u jazyka SQL. Datové sklady jsou datová úložiště, v nichž jsou data organizována takovým způsobem, aby umožňovala jejich efektivní analýzu pomocí tzv. OLAP (OnLine Analytic Processing) operací. To vyžaduje organizovat data tak, aby na logické úrovni, tvořila vícerozměrnou kostku, kde dimenzemi jsou některé atributy, typicky čas (např. prodeje výrobků), místo (prodeje), typ (výrobku) apod. Dvourozměrná tabulka tady nevyhovuje, i když i vícerozměrná data lze samozřejmě uložit do relačních tabulek.

Pro tyto specifické oblasti časem vznikly specializované databáze, například prostorové pro ukládání prostorových dat, XML databáze pro ukládání XML dokumentů apod. V některých případech se jedná skutečně o zcela specializované řešení, např. nativní XML databáze, které jsou „šité na míru“ práci s XML dokumenty. Často jde ale o určitá rozšíření a podporu rozšiřující funkcionalitu původně čistě relačního databázového serveru nebo, dokonce jen o vrstvu nad databázovou vrstvou, která umožňuje data ukládat do relačních databází spravovaných různými SŘBD. Například databázový server Oracle 10g má zabudovanou podporu pro práci s prostorovými daty, pro budování datových skladů, pro dolování z dat (data mining) i

pro práci s multimediálními daty.

Významným trendem, který se nevyhnul ani databázovým technologiím, byl vliv objektově-orientovaného programování a objektově-orientovaného přístupu k vývoji programů obecně. Ten se projevil v oblasti databázových technologií ve dvou směrech. Jednak to byl vznik *objektově-orientovaných databází*, jednak rozšiřování schopností relačních databází směrem k objektové orientaci. Výsledkem druhého směru jsou *objektově-relační databáze*. O obou těchto směrech se stručně zmíníme v následující kapitole 6.2.

S rozvojem internetu a zejména webu se mění architektura databázových aplikací výrazně ve prospěch vícevrstevných architektur. Již víme, že taková architektura má minimálně tři vrstvy, kde mezi klientem, který zajišťuje hlavně prezentační služby a prezentační logiku, a databázovou vrstvou, která naopak zajišťuje databázové služby, je vrstva aplikačního serveru. V případě webových aplikací je klientem, který není databázovým, nýbrž webovým klientem, webový prohlížeč. Databázovým klientem je aplikační server. Ten může mít podobu webového serveru s modulem interpretujícím nějaký skriptovací jazyk, podobu prostředí pro běh programů v Javě apod. Jakým způsobem se přistupuje k databázím z některých typických jazyků a prostředí pro tvorbu webových aplikací si ukážeme v kapitole 6.3.

Berte ale následující dvě kapitoly skutečně jako pouhý velice stručný úvod do problematiky. Více o objektově-orientovaných databázích se dozvíte v předmětu Informační systémy. Ti z vás, kteří budou pokračovat v navazujícím magisterském programu, potom absolvují předmět pokročilé databázové systémy, kde se dozví více o objektově-relačních, multimediálních, deduktivních i prostorových databázích. S jazyky a prostředími pro tvorbu webových aplikací také se podrobněji seznámíte na v některých jiných předmětech jak bakalářského, tak magisterského studijního programu.

6.2. Databáze založené na objektech

Jak už jsme uvedli v úvodu, vliv objektové orientace (OO) na databáze se projevil ve dvou směrech. Jednak vedl ke vzniku objektově-orientovaných databází, jednak vedl k rozšiřování funkcionality relačních databází o objektově orientované rysy.

Tak jako je základem relačních databází relační model dat, je základem objektově-orientovaných (také zvaných objektových) databází objektově-orientovaný (objektový) model dat. Ten zahrnuje všechny základní koncepty, které znáte z objektově orientovaného programování, tedy především:

- abstrakci (klasifikaci),
- zapouzdření,
- dědičnost a
- polymorfismus.

Objekty jako instance třídy (též objektového typu) mají atributy, poskytují svému okolí operace, jejichž implementace má podobu metod. Objekty mohou obsahovat jiné zanořené objekty, třídy mohou dědit vlastnosti jiných tříd a mohou existovat různé formy polymorfismu. Z pohledu identifikace objektů je důležitý *identifikátor objektu* (OID – Object Identifier). Narozdíl od relačních databází, kde k identifikaci

používáme hodnot primárního klíče, tedy jednoho ze sloupců (případně složeného) tabulky, hodnota identifikátoru objektu je vytvořena objektově-orientovaným systémem řízení databáze (OOSŘBD). OID můžeme chápat jako určitou obdobu identifikátoru řádku či záznamu (RID, ROWID), jak jsme ji poznali když jsme si vysvětlovali organizaci dat v databázi na interní úrovni.

Oproti používání objektů v běžných objektově-orientovaných jazycích je nutné samozřejmě řešit i problémy souvisejícími s perzistencí objektů a funkcemi očekávanými od každého SŘBD, jako je podpora dotazování, transakční zpracování, zajištění bezpečnosti dat apod.

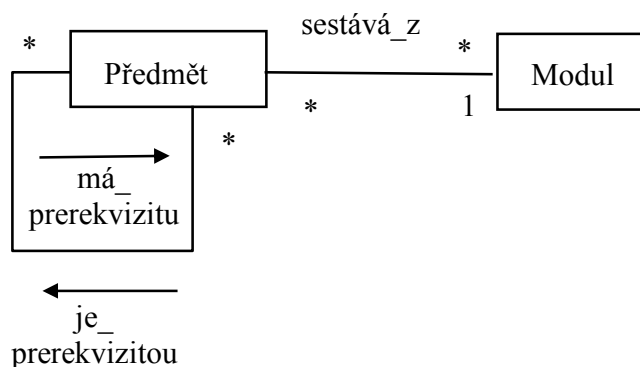
Ve druhé polovině 80. let minulého století vznikla řada OOSŘBD, např. GemStone, Jasmine, O2, ODE, Objectivity, ObjectStore. Tyto systémy zpravidla používaly jako manipulační jazyk C++ nebo Smalltalk. Používaly navigační programování po objektové struktuře podobně, jak jsme se o něm zmínili na úvodní přednášce, když jsme vzpomněli síťové a hierarchické systémy jako představitele předrelačních systémů, a jak je znáte z OO programování.

V roce 1991 vznikla skupina ODMG (Object Database Management Group) zahrnující přední společnosti dodávající či vyvíjející OO databázové produkty. Cílem byla snaha o standardizaci v oblasti OO databázového modelu a databázových jazyků pro definici tříd perzistentních objektů a pro manipulaci s objekty. Výsledkem bylo několik verzí standardu s označením ODMG standard. První verze vznikla v roce 1993 a poslední v roce 2000. Standard zahrnuje neformální definici objektového modelu dat, definici jazyka ODL (Object Definition Language) pro definici objektového schématu, definici deklarativního dotazovacího jazyka OQL (Object Query Language) a dále definoval vazbu na některé OO programovací jazyky jako je C++, Smalltalk, Java apod.



Příklad 6.1

Uvažujme jednoduchý diagram tříd z Obr. 6.1.



Obr. 6.1 Jednoduchý diagram tříd

Popis schématu v jazyce ODL by mohl mít podobu:

```

interface Predmet
// vlastnosti typu (třídy):
( extent Predmety
  keys cislo)
// vlastnosti instance (objektu):
{ attribute String nazev;

```



```

    attribute String cislo;
    relationship List <Modul> sestava_z_modulu
        inverse Modul::je_modulem{order_by Modul::cislo};
    relationship Set <Predmet> ma_prerekvizity
        inverse Predmet::je_prerekvizitou;
    relationship Set <Predmet> je_prerekvizitou
        inverse Predmet::ma_prerekvizity;

// operace instance:
    void nabidka (in Integer semestr) raises
(jiz_nabizeny);
    void zruseni (in Integer semestr) raises
(nenabizeny);
}

```

Vlastnost třídy extent umožňují vytvořit pojmenovanou množinu všech objektů dané třídy. Atribut keys umožňuje definovat atributy, jejichž hodnoty jsou unikátní. Je rovněž vidět, že narozdíl od relačních databází jsou vazby mezi objekty pojmenované a definované explicitně na úrovni schématu.

Příkaz jazyka OQL, který zodpoví dotaz „Najdi předmět s názvem Databázové systémy“ by potom mohl mít tvar:

```

define dsi as
    select x
    from x in Predmety
    where x.nazev="Databázové systémy"

```

Podobně dotaz „Jaké prerekvizity (název předmětu, číslo předmětu) má předmět s názvem Databázové systémy?“ by mohl být v jazyce OQL zapsán jako:

```

define prerekv_dsi as
    select struct(nazev: y.nazev, cislo: y.cislo)
    from x in Predmety, y in x.ma_prerekvizity
    where x.nazev = "Databázové systémy"

```

Je zřejmé, že jde o jazyk orientovaný syntakticky na SQL.

Přestože OO databáze mají ve srovnání s relačními databázemi některé výhody, neprosadily se v oblasti databází ani zdaleka tak výrazně jako OO programovací jazyky mezi programovacími jazyky. Lze říci, že jejich podíl na trhu databázových produktů je nízký. Příčin je možné vidět několik, zejména:

1. Relační databáze si postupně od začátku 80. let minulého století vytvořily na trhu velmi silnou pozici.
2. Relační databáze, včetně jazyka SQL reagovaly na úspěchy OO programovacích jazyků zavedením rozšíření směrem k objektové orientaci.

Můžeme říci, že především komerční SŘBD předních producentů a jimi spravované databáze výrazným způsobem rozšiřují možnosti relačních databází směrem k objektové orientaci, že je můžeme označovat jako *objektově relační*. Tento trend se odrazil i ve verzi standardu SQL z roku 1999. Podívejme se velmi stručně, co nového v tomto smyslu SQL:1999 přinesl.

Jednak rozšiřuje „relační“ funkcionalitu a to například zavedením nových datových

typů BLOB a CLOB pro ukládání rozsáhlých binárních a znakových objektů. Typ BLOB s výhodou využijeme pro ukládání multimediálních dat a CLOB pro ukládání rozsáhlých textových dokumentů. Dalším novým zabudovaným datovým typem je BOOLEAN. Největší změnou v oblasti zabudovaných datových typů je ale typ ARRAY. Jde o pole a jeho zavedení lze chápat jako opuštění požadavku na to, aby tabulka splňovala 1NF.



Někteří oponenti namítají, že tomu tak není, neboť pole lze chápat jako složený atribut.

Z dalších rozšíření můžeme uvést možnost definovat složené sloupce pomocí konstruktoru ROW, například:

```
CREATE TABLE Osoba (  
    id INTEGER,  
    jmeno ROW (  
        krestni VARCHAR(20),  
        prijmeni VARCHAR(30),  
    ),  
    ...  
)
```

Pro přístup ke složkám se používá tečková notace:

```
SELECT O.jmeno.krestni  
FROM Osoba O
```

Za užitečné rozšíření lze považovat i predikát SIMILAR, který je obdobou predikátu LIKE, ale umožňuje definovat vzor v podobě regulárního výrazu. Dále standard například zmírňuje podmínky pro aktualizovatelnost pohledů, takže se stávají aktualizovatelné některé pohledy, které podle standardu SQL-92 aktualizovatelné nejsou. Zavádí také konstrukci umožňující rekurzivní dotazy, standardizovány jsou databázové triggerly a možnost částečného zrušení transakce pomocí dvojic SAVEPOINT a ROLLBACK.

Nejvýraznější změnou ale je možnost vytváření *strukturovaných uživatelem definovaných typů* (UDT – User-Defined Types). Jde o obdobu tříd. Typ zahrnuje atributy, které mohou být zabudovaných typů, jiných uživatelem definovaných typů nebo kolekce. Dále mohou mít metody a je podporována jednoduchá dědičnost.

Uvažujme, že chceme vytvořit typ Zamestnanec, který dědí vlastnosti typu Osoba, je to typ, od kterého lze vytvářet instance a může být dále děděn. Protože uživatelem definovaný typ je databázovým objektem, budeme ho vytvářet příkazem CREATE:

```
CREATE TYPE zamestnanec_t UNDER t_osoba  
AS (os_cislo    INTEGER,  
    plat        REAL )  
INSTANTIABLE  
NOT FINAL  
REF (os_cislo)  
INSTANCE METHOD  
    Zvys_plat(abs_proc    BOOLEAN, castka REAL) RETURNS REAL
```

Zápis REF (os_cislo) říká, že reference na objekt se bude vytvářet pomocí hodnoty atributu os_cislo, tedy analogicky cizímu klíči.

Takto vytvořený typ lze použít jako typ sloupce v tabulce:

```
CREATE TABLE Zamestnanci_fakulty (  
    id      INTEGER PRIMARY KEY,  
    zam     zamestnanec_t,  
    ustav   CHAR(3) FOREIGN KEY REFERENCES Ustav)
```

V tomto případě instanci typu, tedy objekt lze chápat jako zanořený objekt, a proto nemá identitu (OID). Chceme-li pracovat s objekty s identitou, musíme je ukládat jako řádky do tzv. typované tabulky.

```
CREATE TABLE Zamestnanci OF zamestnanec_t
```

Každý řádek takové tabulky je potom identifikovaný hodnotou typu REF, se kterou jsme se již setkali. Lze tak vytvářet vazby mezi objekty i odkazovat se na objekt z řádku relační tabulky, např.:

```
CREATE TABLE Ustav (  
    ...zkratka CHAR(3) PRIMARY KEY  
    vedouci REF(zamestnanec_t),  
    ...)
```

Pomocí hodnot typu REF lze pak zpřístupňovat atributy i metody objekty, např. jednoduchý dotaz „Kdo je vedoucím ústavu se zkratkou ÚIFS?“ bychom zodpověděli příkazem:

```
SELECT vedouci -> jmeno  
FROM Ustav  
WHERE zkratka = 'UIFS'
```

Jako konkrétní příklad uveďme, že Oracle poskytuje objektová rozšíření již od verze Oracle 8, uvolněné v roce 1997. Lze říci, že v oblasti strukturovaných uživatelem definovaných typů dialekt SQL až na menší odchylky odpovídá standardu. Existují zde dva typy uživatelem definovaných typů:

- *objektové typy* – odpovídají strukturovaným uživatelem definovaným typům
- *kolekce* – dostupný je typ pole a vnořená tabulka (nested table). Vnořená tabulka umožňuje do řádků sloupce tohoto typu vložit několik řádků zanořené tabulky. V tomto případě dochází zcela určitě k opuštění požadavku tabulky v 1NF. Fyzicky je vnořená tabulka uložena v samostatné tabulce mimo tabulku rodičovskou.

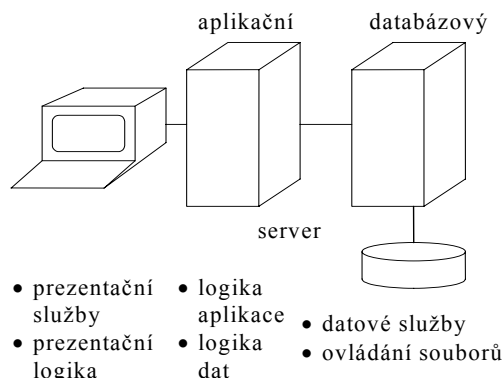
Uživatelem definované typy se používají k definici standardizovaných balíků rozšiřujících SQL, například pro práci s multimediálními daty SQL/MM a pro dolování z dat SQL/MM DM. Rovněž například rozhraní podpory pro prostorová data u databázového serveru Oracle je definováno pomocí uživatelem definovaných typů.

6.3. Přístup k databázím z WWW

V současné době, kdy dochází k rychlému rozšíření internetu a velkému rozvoji WWW technologií, se stále častěji využívá možnost přístupu k databázím pomocí aplikací na WWW. Vzhledem k tomu, že uživatel může nejen číst uložená data, ale i ukládat data do databází jedná se o tzv. transakční aplikace. Existuje mnoho různých možností přístupu k databázím z WWW, jejichž společným rysem bývá použití třívrstvé architektury. Celá aplikace je rozdělena na tři úrovně a tvoří ji databázová vrstva, aplikační vrstva a klientská vrstva.

Databázová vrstva slouží nejen jako bezpečné úložiště dat, ale zajišťuje také integritu dat, prostředky pro rychlý přístup k datům a poskytuje základní zabezpečení přístupu k datům. Prostřední vrstvu této architektury tvoří aplikační vrstva, která obsahuje aplikační logiku. Základním úkolem aplikační vrstvy je poskytovat všechny potřebné funkce pro klienty z klientské vrstvy. Poslední vrstvu tvoří klientská vrstva, která slouží k prezentaci dat uživatelům a zároveň jim poskytuje přístup k jednotlivým službám dané aplikace. Mezi hlavní výhody této architektury patří centralizace údržby aplikace, možnost využití sdílených objektů několika aplikacemi (business objects), tenký klient a snadnější rozšiřitelnost aplikace. Schéma této architektury obsahuje následující obrázek.

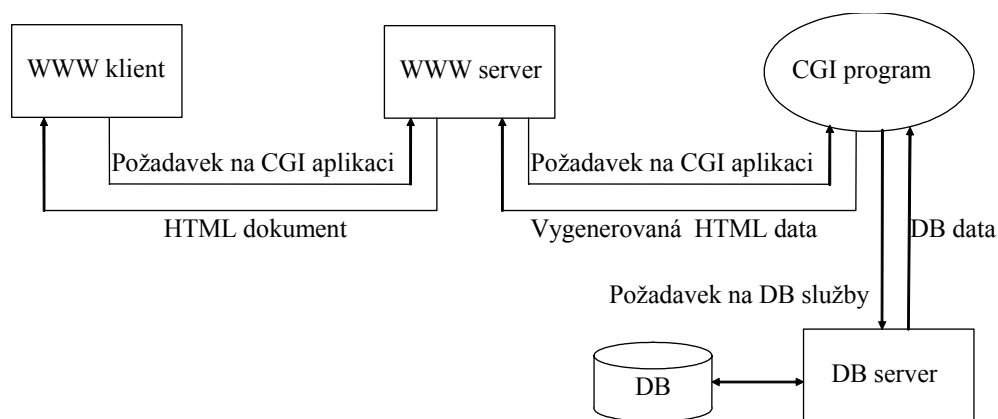
Třívrstvá architektura



Obr. 6.2 Model třívrstvé architektury

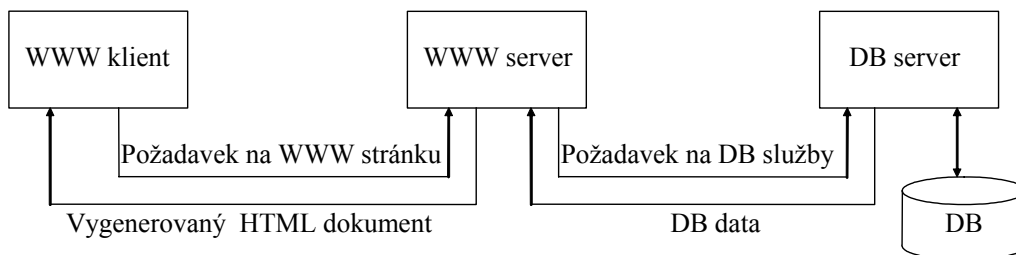
Jednotlivé vrstvy architektury komunikují vždy pouze se sousední vrstvou. Komunikace mezi klientskou a aplikační vrstvou serverem probíhá většinou výměnou HTML nebo XML dokumentů pomocí HTTP protokolu. Pro komunikaci mezi aplikačním a databázovým serverem je možné využít následující přístupy:

- *Použití CGI prostředků* – s databází komunikuje CGI program. Nevýhodou tohoto přístupu je režie spouštění nových procesů. Dochází k interpretaci skriptů a k opakovanému otevírání spojení s DB serverem. Tohoto přístupu využívají například skriptovací jazyky Perl nebo PHP, nebo programy psané v jazyce C.



Obr. 6.3 Model použití CGI prostředků

- *Použití aplikací ve tvaru DLL* – snížení režie spouštění procesů oproti klasickým CGI aplikacím. (př. ISAPI)
- *Vyhodnocení dokumentu na WWW serveru formou přidávaných direktiv*. (př. Internet Information Server + IDC, ASP)



Obr. 6.4 Model vyhodnocení dokumentu na www serveru formou přidávaných direktiv

- *Přístup k databázím z jazyka Java*

Následující podkapitoly popisují přístup k databázím pomocí PHP, ASP, JDBC a SQLJ.

6.3.1. PHP (Personal Home Page)

Jedná se o skriptovací jazyk na straně WWW serveru. Podporuje přístup k řadě SŘBD (Adabas D, dBase, Empress, FilePro, Informix, InterBase, mSQL, MySQL, Oracle, PostgreSQL, Solid, Sybase, Velocis, Unix dbm). Nevýhodou PHP je specializovaný přístup k jednotlivým druhům SŘBD. Znamená to, že pokud chceme přenést projekt například z MySQL na ORACLE, musíme přepisovat skripty. Funkce pro přístup k databázím totiž mají tento formát: *nazevdbstroje_jmenofunkce*.

Nejčastěji se PHP používá ve spojení s MySQL, proto pro popis funkcí sloužících k přístupu k databázím jsou použity funkce pro MySQL databáze. Nejprve je nutné připojit se k databázovému serveru MySQL a nastavit aktuální aktivní databázi. K tomu slouží následující funkce:

mysql_connect – otevře spojení s databázovým serverem MySQL, vrací identifikátor spojení

mysql_pconnect – otevře trvalé spojení s MySQL serverem (neukončuje se při ukončení skriptu ani funkcí *mysql_close()*), vrací identifikátor spojení

mysql_select_db - nastaví aktuální aktivní databázi pro dané spojení s databázovým serverem

mysql_close - uzavře spojení s databázovým serverem MySQL

Pomocí PHP lze také vytvářet a rušit databáze:

mysql_create_db - vytvoří databázi spravovanou serverem MySQL

mysql_drop_db - zruší databázi spravovanou serverem MySQL

Práce s databází obvykle spočívá ve vykonávání SQL příkazů. Pro provádění těchto příkazů a pro práci s kurzorem (v terminologii PHP3 výsledek SQL příkazu, tedy funkce pro zpracování výsledku) lze využít tyto funkce:

mysql_query - pošle zadaný příkaz současné aktivní databázi serveru MySQL, pro INSERT, UPDATE a DELETE vrací příznak úspěšnosti, pro SELECT číslo kurzoru

mysql_db_query - vrátí číslo kurzoru pro zadaný dotaz pro danou databázi

mysql_affected_rows - vrátí počet řádků ovlivněných posledním příkazem INSERT, UPDATE nebo DELETE

mysql_fetch_array - vrátí řádek kurzoru (řádek dat z výsledku dotazu) jako asociativní pole (výběr podle jména sloupce)

mysql_fetch_row - vrátí řádek kurzoru jako pole s prvky zpřístupňovanými pořadovým číslem

mysql_fetch_object - vrátí řádek kurzoru jako objekt - přístup přes jména (obdoba *fetch_array*)

Každým dalším voláním posledních tří funkcí dojde k přečtení dalšího řádku dat z výsledku dotazu. Pokud dojde k přečtení posledního řádku, funkce vrátí *false*.

mysql_result - vrátí hodnotu daného sloupce daného řádku daného kurzoru, neměla by být používána s jinými funkcemi zpřístupňujícími řádky kurzoru

mysql_data_seek - posune ukazatel kurzoru na řádek s daným pořadovým číslem

mysql_free_result - uvolní paměťový prostor kurzoru

mysql_num_rows - vrátí počet řádků kurzoru

mysql_num_fields - vrací počet sloupců kurzoru

mysql_fetch_lengths - vrátí délky polí posledně vybraného řádku kurzoru funkcí *mysql_fetch_row*

mysql_insert_id - vrátí identifikátor generovaný posledním příkazem INSERT pro sloupec typu AUTO_INCREMENTED

Vzhledem k tomu, že při práci k databázím může docházet k chybám, PHP poskytuje funkce pro ošetření chyb:

mysql_errno - vrátí číslo chyby poslední operace MySQL

mysql_error - vrátí text chybového hlášení poslední operace MySQL

Pomocí PHP můžeme přistupovat nejen k datům uloženým v databázi, ale i k metadatům, která popisují strukturu databáze. PHP pro tyto účely poskytuje následující funkce:

mysql_list_dbs - vrátí kurzor obsahující seznam dostupných databází, k procházení slouží funkce *mysql_tablename*

mysql_list_tables - vrátí kurzor obsahující seznam tabulek dané databáze, k procházení slouží funkce *mysql_tablename*

mysql_tablename - vrátí jméno tabulky s daným pořadovým číslem

prostřednictvím kurzoru vráceného funkcí *mysql_list_tables*

mysql_fetch_field - vrátí objekt, který nese informaci o daném sloupci daného kurzoru

mysql_field_seek - nastaví ukazatel kurzoru na daný sloupec

mysql_field_name - vrátí jméno daného sloupce kurzoru

mysql_field_table - vrátí jméno tabulky, které patří zadaný sloupec kurzoru

mysql_field_type - vrátí datový typ daného sloupce kurzoru

mysql_field_flags - vrátí příznaky ("not_null", "primary_key", ...), spojené se zadaným sloupcem výsledku

mysql_field_len - vrátí délku specifikovaného sloupce daného kurzoru

mysql_list_fields - zpřístupní metadata pro danou tabulku, vrací číslo kurzoru, které lze použít ve funkcích pro práci s metadaty (*mysql_field_flags()*,

mysql_field_len(), *mysql_field_name()* a *mysql_field_type()*)

x+y

Příklad 6.2

V tomto příkladu si ukážeme použití některých z uvedených funkcí. Následující kód v PHP popisuje připojení k databázi uivt, zjistí počet řádků a sloupců této tabulky. Dále vypíše jména všech sloupců, jejich datový typ, délku sloupce a atributy sloupce.

```
<?php
mysql_connect($host,$user,$password);
//připojení k MySQL serveru pomocí jména a hesla uložených v
//proměnných user a password
mysql_select_db("uivt"); //výběr databáze
$result = mysql_query("SELECT * FROM osoby");
//položení SQL dotazu a uložení ukazatele na výsledek dotazu
$fields = mysql_num_fields($result);
$rows = mysql_num_rows($result);
//počet sloupců a řádků výsledku SQL dotazu
$i = 0;
$table = mysql_field_table($result, $i);
//dotaz, který vrátí jméno tabulky
echo "Tabulka '$table.' má '$fields.' Sloupců a '$rows.'
řádků <BR>";
echo "Tabulka má následující sloupce: <BR>";
while ($i < $fields) { //cyklus přes všechny sloupce tabulky
    $type = mysql_field_type ($result, $i); //typ sloupce
    $name = mysql_field_name ($result, $i); //jméno sloupce
    $len = mysql_field_len ($result, $i); //délka sloupce
    $flags = mysql_field_flags ($result, $i); //atributy sloupce
    echo $name." ".$type." ".$len." ".$flags."<BR>";
    $i++;
}
mysql_close(); //uzavření spojení s databázovým serverem
?>
```

x+y

Příklad 6.3

Potřebujeme-li zpracovat jednotlivé řádky, které tvoří výsledek SQL dotazu, můžeme využít tři různé funkce. Každá tato funkce vrací jeden řádek dat v různém formátu. Následující tři ukázky PHP kódu ukazují, jak přistupovat k těmto datům.


```
<?php
mysql_connect($host,$user,$password);
$result = mysql_db_query("uivt","select * from osoby");
while($row = mysql_fetch_array($result)) {
    //řádek kurzoru - asociativní pole, výběr podle jména sloupce
    echo $row["os_cislo"];
    echo $row["jmeno"];
}
mysql_free_result($result);
?>

<?php
mysql_connect($host,$user,$password);
$result = mysql_db_query("uivt","select * from osoby");
while($row = mysql_fetch_row($result)) {
    //řádek kurzoru - pole s prvky zpřístupňovanými pořadovým číslem
    echo $row[0];
    echo $row[1];
}
mysql_free_result($result);
?>

<?php
mysql_connect($host,$user,$password);
$result = mysql_db_query("uivt","select * from osoby");
while($row = mysql_fetch_object($result)) {
    //řádek kurzoru - objekt, přístup přes jména
    echo $row->os_cislo;
    echo $row->jmeno;
}
mysql_free_result($result);
?>
```

6.3.2. Internet Information Server a ASP (Active Server Pages)

Internet Information Server (IIS) je součástí operačního systému Windows NT 4.0. Jednou ze služeb, které poskytuje, je služba WWW (Web server), umožňující zpřístupnit WWW stránky klientům pracujícím s WWW prohlížeči. Dynamické WWW stránky je možno generovat pomocí standardního CGI rozhraní nebo rozhraní ISAPI. Pro přístup k databázím lze využít kromě CGI aplikací další dvě technologie - Internet Database Connector a Active Server Pages.

Microsoft Active Server Pages je prostředí, které umožňuje vykonávat skripty na straně serveru. Active Server Pages jsou soubory s příponou „asp“. Jsou to v podstatě HTML dokumenty, které kromě textové informace a html značek obsahují nějaký programový kód (skript). ASP podporuje dva skriptovací jazyky - VBScript a JavaScript. Jako oddělovače příkazů skriptu jsou použity dvojice `<% %>`, resp. značky `<SCRIPT RUNAT=SERVER ... > </SCRIPT>`. Před odesláním stránky klientovi se musí na straně serveru vykonat tyto skripty. Kromě serverovských skriptů může stránka obsahovat i klientské skripty, jejichž vykonání server ponechá na klientovi.

V ASP aplikacích lze používat pět zabudovaných objektů, pomocí kterých je možné komunikovat s okolím, načítat parametry, vypisovat data na klienta a podobně. K dispozici jsou tyto objekty:

- *Request* - získání informací od uživatele.

- *Response* - zaslání informací uživateli.
- *Server* - práce s ActiveX komponentami.
- *Session* - uchování informací o uživatelském sezení.
- *Application* - uchování informací o uživatelské aplikaci.

Z hlediska přístupu k databázím je nejdůležitější komponenta Database Access objektu Server, která používá ActiveX Data Object (ADO) pro přístup k informacím uloženým v databázi či jiných tabulkově orientovaných datových strukturách.

Pro vytvoření instance jakéhokoliv objektu, který je v systému nainstalován je k dispozici metoda `CreateObject` (např. `ADODB.Connection` – pro přístup k databázím).

Pro přístup k datům v databázi je zde k dispozici několik tříd:

- *Connection* - spojení se zdrojem dat. Poskytuje metody pro nastavení vlastností spojení, otevření a uzavření spojení, provedení příkazu v otevřeném spojení, správu transakcí v otevřeném spojení, zpracování chyb apod.
- *Recordset* - reprezentace tabulky vrácené po vykonání příkazu. V každém okamžiku umožňuje zpřístupnit jeden řádek. K průchodu tabulkou lze použít kurzorů několika typů (statický, dynamický, keyset, forward-only).
- *Command* - definice příkazu pro zdroj dat. Pomocí metod a vlastností lze především definovat příkaz v textové podobě, přiřadit spojení k danému příkazu, provést příkaz, otestovat, zda existuje podpora pro předkompilování příkazu apod.
- *Field* - reprezentuje sloupec dat objektu třídy *Recordset*. Metody slouží především ke zpřístupnění metadat a hodnot sloupců.
- *Parameter* - reprezentuje parametr objektu třídy *Command*. Metody slouží především ke zpřístupnění metadat a hodnot parametrů.
- *Error* - informace o chybě vzniklé při vykonání příkazu.
- *Property* - reprezentuje dynamickou vlastnost ADO objektu.

x+y

Příklad 6.4

Použití objektů třídy *Connection* a *Recordset* pro načtení hodnot z tabulky by mohlo vypadat takto (JavaScript):

```
...
<%Conn = Server.CreateObject("ADODB.Connection")
Conn.Open("ADOZamestnanci")
sql="SELECT Jmeno, Plat FROM Zamestnanci WHERE plat > 10000"
RS = Conn.Execute(sql)
counter=0
while (!RS.EOF) {
    hodnota[counter].jmeno = RS("Jmeno")
    hodnota[counter].plat = RS(1)
    RS2.MoveNext()
    counter = counter + 1
} %>
...
```

6.3.3. Přístup k databázím z jazyka Java

Z prostředí jazyka Java lze k databázím přistupovat pomocí JDBC (Java Database Connectivity) nebo SQLJ (vložený SQL). Nejprve se zaměříme na JDBC.

JDBC

JDBC je Java API pro styk s relačními databázemi. Skládá se z množiny tříd a rozhraní napsaných v programovacím jazyce Java, které umožňují posílat SQL příkazy systémům řízení relačních bází dat. Z toho vyplývá, že nemusíme psát různé programy pro jednotlivé databáze, ale stačí napsat jeden program používající JDBC API, který bude schopen komunikovat s libovolnou databází. Další výhodou je nezávislost programovacího jazyka Java na platformě operačního systému. Jedná se o rozhraní nízké úrovně, protože se používá k přímému volání SQL příkazů. Zároveň je ale považováno za bázi, na níž lze budovat uživatelsky „přítulná“ rozhraní a utility vyšší úrovně. Pro přístup k databázi se typicky používají následující kroky:

- Vytvoření spojení

```
Connection con = DriverManager.getConnection  
("jdbc:odbc:db", "login", "password");
```
- Vytvoření příkazu

```
Statement stmt = con.createStatement();
```
- Vykonání příkazu a vytvoření objektu pro zpracování výsledků

```
ResultSet rs = stmt.executeQuery ("SELECT a, b, c  
FROM table1");
```
- Zpracování výsledků

```
WHILE(rs.next()) {  
    int x = rs.getInt("a");  
    String s = rs.getString(2);  
    Float f = rs.getFloat("c");  
}
```

Pro vykonání uvedených kroků poskytuje JDBC API kolekci Java tříd a rozhraní. Mezi nejdůležitější rozhraní patří:

- **java.sql.DriverManager** – rozhraní pro práci s ovladači, sleduje použití ovladačů, stanovuje spojení mezi databází a zvoleným ovladačem. Potřebné ovladače je třeba v systému nejprve registrovat. Poskytuje metodu *getConnection (URL, user, password)*, která ustaví spojení s požadovanou databází. URL se udává ve tvaru *jdbc:<subprotokol>:<jmeno>*.
- **java.sql.Connection** – reprezentace spojení s databází. Jedna aplikace může mít jedno nebo více spojení s jednou nebo několika databázemi. Spojení se vytváří metodou *getConnection* rozhraní *DriverManager*. Databáze je identifikována pomocí parametru URL, kde *<subprotokol>* je obvykle název ovladače nebo mechanismu spojení s databází a složka *<jmeno>* je vlastní identifikace databáze. Metoda *getConnection* vybere vhodný ovladač, vytvoří spojení s databází a vrátí objekt *Connection*, který ho reprezentuje. Rozhraní *Connection* poskytuje rovněž metody pro práci s transakcemi, včetně nastavení izolační úrovně.
- **java.sql.Statement** – kontejner pro vykonání SQL příkazů v daném spojení. JDBC žádným způsobem nekontroluje SQL příkazy, ve skutečnosti ani nemusí jít o příkazy SQL. Jedinou podmínkou je, aby systém řízení báze dat (SŘBD), ke kterému budou příkazy posílány, těmto příkazům rozuměl. JDBC poskytuje tři rozhraní pro posílání SQL příkazů a zároveň tři metody v rozhraní *Connection*, které vytvářejí instance těchto tříd:
Statement - slouží k posílání neparametrizovaných SQL příkazů. K provedení příkazu slouží metody *executeQuery*, *executeUpdate*,

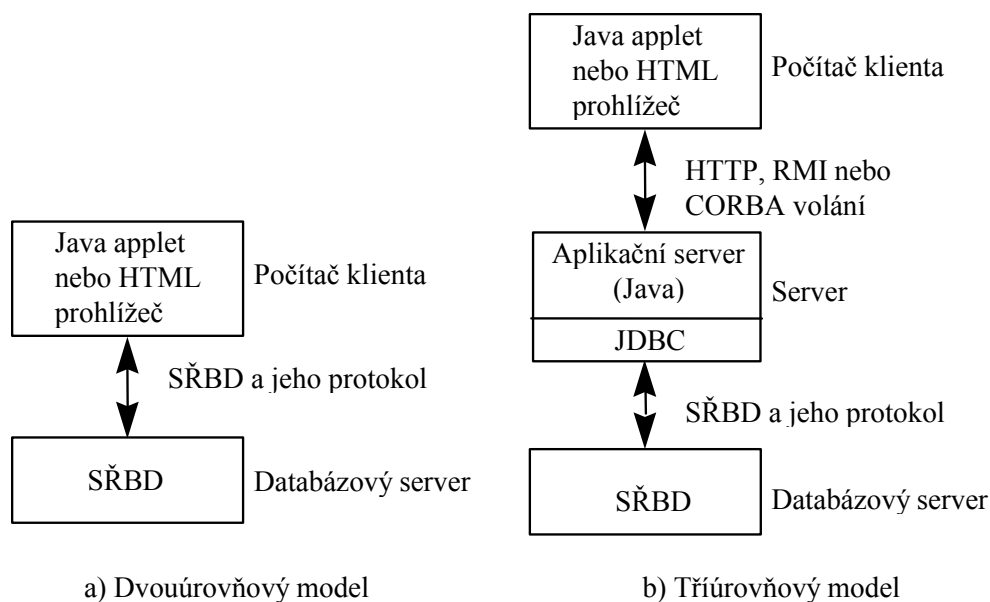
resp.*execute*.

PreparedStatement – představuje rozšíření rozhraní *Statement*. Používá se pro parametrizované SQL příkazy a může být výhodnější i pro opakovaně prováděné neparаметrizované příkazy, protože v tomto případě je příkaz předkompilován a uložen pro pozdější použití. Obsahuje skupinu metod pro nastavení všech vstupních parametrů.

CallableStatement - je rozšířením rozhraní *PreparedStatement*, objekty se používají pro volání SQL uložených procedur.

- ***java.sql.Result*** – reprezentuje výsledek dotazu a řídí přístup k jednotlivým řádkům tabulky (obdoba kurzoru z SQL). Objekty této třídy vrací metoda *executeQuery*. K dispozici jsou především metody pro posun na další řádek a zpřístupnění hodnot v jednotlivých sloupcích (*next()* a *getXXX()*, kde XXX značí konkrétní datový typ). K dispozici jsou i metody pro získání metadat (vlastností sloupců) výsledku dotazu. Při volání metody *getXXX()* se JDBC ovladač pokouší konvertovat typy dat v databázi na požadovaný datový typ programovacího jazyka Java. Protože SQL definuje speciální příznak NULL pro reprezentaci chybějící hodnoty, existuje i metoda k jeho testování.

JDBC podporuje jak dvouúrovňové tak tříúrovňové modely přístupu k databázi. Ve dvouúrovňovém modelu Java applet či aplikace komunikují se SŘBD přímo. Tato varianta vyžaduje JDBC ovladač, který umožňuje komunikaci s konkrétním SŘBD. V tříúrovňovém modelu (viz. Obr. 6.5 b) jsou SQL příkazy zaslány střední vrstvě služeb, která je zašle požadovanému SŘBD. Ten příkazy vykoná a pošle výsledky zpět střední vrstvě, která je předá uživateli. Tento model je výhodnější v tom, že umožňuje udržovat kontrolu přístupu a způsoby modifikace dat. Další výhodou je, že uživatel má možnost používat jednodušší API vyšší úrovně, které je převedeno střední vrstvou do volání nízké úrovně.



Obr. 6.5 Dvouúrovňový a tříúrovňový model přístupu k databázi

SQLJ

Druhou možností přístupu k databázím z jazyka Java je vložený SQL, označovaný jako SQLJ. Příkazy SQL jsou zde vkládány přímo do prostředí hostitelského programovacího jazyka, jsou uvozeny direktivou, která umožňuje jejich extrakci a předzpracování. Součástí předzpracování může být statická kontrola příkazů SQL a to i vůči schématu databáze, je-li přístupné. Nevýhodou SQLJ na rozdíl od JDBC je omezení na použití pouze statických příkazů SQL, tedy příkazů, jejichž syntax a sémantika může být zkontrolována už v době překladu. JDBC umožňuje sestavovat příkazy SQL až za běhu aplikace, ale zase neumožňuje provádět statickou kontrolu SQL příkazů.

Při tvorbě SQLJ aplikace píše programátor program v Javě a definovaným způsobem do něj vkládá příkazy SQL. Překladačem SQLJ jsou potom tyto příkazy transformovány na standardní program v Javě, který může být potom přeložen libovolným překladačem jazyka Java. Každý SQL příkaz se vyskytuje v SQLJ programu jako tzv. *SQLJ klauzule* ve tvaru:

```
#sql {příkaz_SQL} | deklarace_objektu_SQLJ;
```

Klauzule může být deklarační (deklarace objektu SQLJ – např. iterátor nebo spojení) nebo proveditelná (SQL příkaz). Výsledek dotazu, který vrací jediný řádek tabulky je možné uložit přímo do hostitelských proměnných. Jestliže dotaz vrací více řádků tabulky, je nutné tento výsledek zpracovat pomocí objektu zvaného *iterátor*. Ten zpřístupňuje jednotlivé řádky výsledku podobně jako kurzor. SQLJ poskytuje dva mechanismy pro mapování sloupců dotazu na sloupce iterátoru: poziční a podle jména. Typ iterátoru se rozlišuje v deklaraci iterátoru. SQLJ umožňuje také volání uložených procedur a funkcí, které mohou být implementovány v různých programovacích jazycích.

Provedení příkazu v proveditelné SQL klauzuli vyžaduje identifikaci spojení s databází. K tomuto účelu slouží v SQLJ objekt *kontextu spojení*. Kontext spojení je objekt odpovídající třídy, která je definována pomocí deklarační SQLJ klauzule pro spojení s databází. Objekt *kontext* by mohl být vytvořen takto (při použití

URL databáze):

```
#sql context DBkontext;
DBkontext kontext; //deklarace objektu kontextu spojeni
kontext = new DBkontext(url, true); //inicializace objektu,
                                     včetně otevření spojení
```

Druhý argument konstruktoru je příznak pro automatické potvrzení každého úspěšně provedeného příkazu SQL (AUTOCOMMIT).

x+y

Příklad 6.5

Použití pozičního iterátoru pro zpracování výsledku dotazu může vypadat takto:

```
#sql iterator ZamestnanciUstavu(int, String);
ZamestnanciUstavu iterP; // deklarace objektu iterP
// inicializace objektu iterP
#sql iterP = {SELECT OS_CISLO AS OSCISLO, PRIJMENI FROM UCITEL
              WHERE ZKRATKA = :zkratkaUstavu};

int c; String p;
#sql {FETCH :iterP INTO :c, :p};
while (!iterP.endFetch()){
    System.out.println(p + " (os.číslo: "c + ")")
    #sql {FETCH :iterP INTO :c, :p};
}
```

Σ

V této kapitole jsme se velice stručně seznámili s některými trendy v oblasti databázových technologií. Především jsme se zmínili o dvou typech databází založených na objektech-objektově orientovaných a objektově-relačních. Uvedli jsme si, že podpora objektové orientace je obsažena ve standardu SQL:1999 v podobě strukturovaných uživatelem definovaných typů a že je podporována u předních databázových produktů. Dále jsme si ukázali, jak lze přistupovat k databázím z některých jazyků a prostředí používaných pro tvorbu webových aplikací. Typická komunikace sestává z vytvoření spojení s databázovým serverem, sestavení příkazu a jeho provedení. Pokud je příkazem dotaz, pak typicky zpracování výsledku v cyklu, jak to známe z přístupu k tabulce prostřednictvím kurzoru.



Informace k objektovým a objektově-relačním databázím lze nalézt v učebnici [Pok98] na str. 39 – 42, 49 – 52 a 109 – 118 a v učebnici [Sil05] v kapitole 5 Object-Based Databases na str. 361 až 393. Jako další vhodné prameny lze doporučit tyto zdroje:

Eisenberg, A., Melton, J.: SQL:1999, formerly known as SQL3. SIGMOD Record, Vol. 28, No. 1, March 1999. str.131 – 138.

SQL standards. Dostupné na <http://www.wiscorp.com/SQLStandards.html> (září 2005).

Price, J.: Oracle Database 10g SQL. Oracle Press McGra-Hill/Osborne. 2004. str. 367 – 426.

K problematice webových aplikací a přístupu k databázím z nich lze použít například:

Welling, L., Thomsonová, L.: PHP a MySQL - rozvoj webových aplikací - 2. vydání, SOFTPRESS, 2003.

JDBC Technology. Sun Developer Network. Dostupné na URL <http://java.sun.com/products/jdbc/>. (září 2005).

Esposito, D.: ASP.NET a ADO.NET tvorba dynamických webových stránek. Grada.

**Cvičení:**

- C6.1** Uveďte, jaké je typické prostředí aplikací relačních databází a charakterizujte některé oblasti aplikací, které se vyznačují odlišnými požadavky.
- C6.2** Charakterizujte výstižně deduktivní, temporální, aktivní, multimediální a prostorové databáze.
- C6.3** Charakterizujte objektově-orientované databáze.
- C6.4** Charakterizujte objektově-relační databáze a uveďte, jakým způsobem zavádí objektovou orientaci standard SQL:1999.
- C6.5** Popište třívrstvou architekturu. Vysvětlete její výhody a nevýhody.
- C6.6** Jak probíhá komunikace mezi jednotlivými vrstvami třívrstvé architektury?
- C6.7** Vysvětlete specializovaný přístup PHP k jednotlivým druhům SŘBD. Jaký formát mají funkce PHP pro přístup k databázím.
- C6.8** Jakými způsoby lze přistupovat k databázím z prostředí jazyka Java? Výstižně je charakterizujte.
- C6.9** Porovnejte přístup k databázi z jazyka Java prostřednictvím JDBC a SQLJ.

**Test**

T1. JDBC je programátorské rozhraní pro

- a) přístup k relačním databázím z WWW
- b) přístup k relačním databázím z jazyka Java

T2. SQLJ je:

- a) hostitelská verze (embedded) jazyka SQL pro jazyk Java
- b) rozšíření jazyka SQL o konstrukce jazyka Java

Dodatek A Řešení testových otázek

Kapitola 2: b, - , b, a, b

Kapitola 3.2: a, b, b, a, -

Kapitola 3.3: b, -, b, -

Kapitola 3.4: b, -, b, b. a

Kapitola 4: b, -, a, b,-

Kapitola 5: -, a, b, a, -, a

Kapitola 6: b, a