Databázové a informační systémy

Verze 20070403

Michal Krátký

2.2 Konceptuální model

2.2.1 Úvod do konceptuálního modelování

Konceptuální modelování je proces vývoje sémantického popisu nějakého systému, který je uplatněn při analýze databázové aplikace. Významné je, že konceptuální model je nezávislý na SŘBD. Systém modelujeme bez ohledu na to, zda bude použit SŘBD Oracle [22] nebo MS SQL Server [19].

Konceptuální model umožňuje identifikovat *entitní typy* a vazby mezi *entitními typy*, které je nutné v systému evidovat. Entitou rozumíme objekt z reálného světa jehož vlastnosti (*hodnoty atributů*) chceme evidovat. Entitní typ je tedy popsán jménem a množinou *atributů*, entita je konkrétní výskyt (instance) entitního typu s hodnotami atributů. Mezi atributy rozlišujeme tzv. *klíč*, což je podmnožina množiny atributů, dle jejichž hodnot je entita odlišitelná od ostatních entit stejného typu.

Vazba modeluje vztah mezi *n* entitními typy, mluvíme o *n*-ární vazbě. Vazba je definována názvem a množinou entitních typů, které jsou v příslušném vztahu. Nejčastější jsou vazby binární, které klasifikujeme na:

1. *1 : M* (angl. *one–to–many*) – vazba, kde entita prvního typu má vazbu na *M* entit druhé typu a kde entita druhého typu má vazbu na jednu entitu prvního typu.
2. *M : N* (angl. *many–to–many*) – vazba, kde entita prvního typu má vazbu na *M* entit druhé typu a kde entita druhého typu má vazbu na *N* entit prvního typu.

Pro tuto klasifikaci používáme označení *násobnost* nebo *kardinalita*. Další kritériem klasifikace vazby je *povinnost v členství*:

1. Typ není povinen členstvím ve vazbě – existuje entita, která nemá vazbu na jinou entitu.
2. Typ je povinen členstvím ve vazbě – každá entita musí mít vazbu alespoň na jednu entitu.

Příklad 2.1 (Entitní typy, entity a vazby).

Entitním typem je `Student` nebo `Ucitel` s atributy `login`, `jmeno`, `prijmeni`, `rocnik` resp. `login`, `jmeno`, `prijmeni` a `uvazek`. Entitní typy zapisujeme pomocí *lineárního zápisu* při dodržení jednoduché konvence: názvy entitních typů začínají velkým písmenem, názvy atributů malým písmem. Entitní typ zapisujeme jako n-tici: *<název entitního typu>(<název atribut 1>, <název atribut 2>,..., <název atribut n>)*. Klíč je naznačen potržením názvu klíčových atributů.

V tomto případě zapisujeme entitní typy `Student(login, jmeno, email, rocnik)` a `Ucitel(login, jmeno, email, uvazek)`. Entitou typu `Student` je např. `('hon001', 'Jan Knop', 'jan.knop@vsb.cz', 3)`.

Uvažujme entitní typ `Kurz(kod, nazev, kapacita)`. Vazbu mezi entitními typy zapisuje opět jako n-tici: *<název vazby>(<entitní typ1>, < entitní typ 2>,...,< entitní typ n>)*. Název vazby je psán velkými písmeny. Vazbou mezi entitními typy `Student` a `Kurz` je `STUDUJE(Student, Kurz)`, mezi entitními typy `Učitel` a `Kurz` je vazba `UCI(Ucitel, Kurz)`.

Datový model ukazuje jakým způsobem jsou data modelována a dotazována. Při konceptuálním modelování tedy identifikujeme jednotlivé entitní typy a vazby, datový model potom určuje zda budeme data modelovat jako relace a dotazovat pomocí jazyka *SQL*

nebo jako XML dokumenty a dotazovat pomocí jazyka *XQuery*. Nejznámější datové modely jsou uvedeny v kapitole 2.3, dotazovací jazyky jsou popsány v kapitole 2.4.

2.2.2 Konceptuální model Entity Relationship (ER)

2.2.2.1 Úvod

Entity Relationship (*ER*) [10] je konceptuální model jež byl publikován Peterem P. Chenem v roce 1976. V další letech byl tento model rozšířen (např. o dědičnost), jeho použití je ovšem v dnešní době problematické. Při vývoji informačního systému jsem nuceni použít pro analýzu aplikace UML a pro konceptuální model databáze ER. Zjevně je vhodnější použít pro oba modely UML.

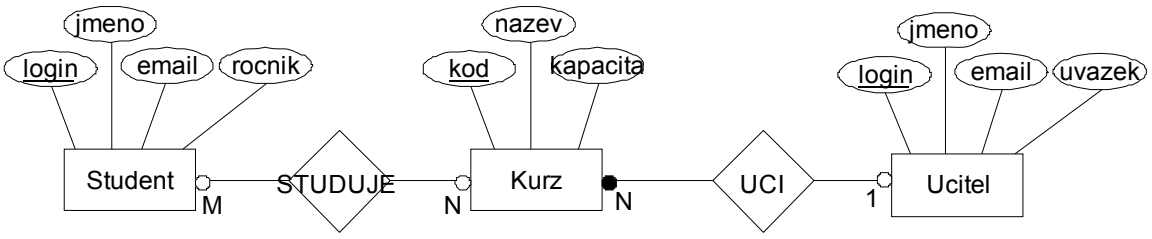
ER graficky modeluje entitní typy a vazby databáze. Diagram označujeme *ER diagram* (*ERD*). ERD používá pro entitní typy obdélníky s vepsaným názvem, jednotlivé atributy jsou pak zapsány v elipsách, názvy klíčových atributů jsou podtrženy. Elipsy jsou spojeny s příslušnými obdélníky entitních typů. *N*-ární vazba mezi typy je kreslena kosodélníkem s *n* hranami spojenými s typy.

2.2.2.2 Vazby a integritní omezení

Násobnost vazby naznačujeme znaky 1, *M* nebo *N* zakreslenými u entitního typu. Mějme vazbu mezi entitními typy *A* a *B*. *N* u entitního typu *A* značí, že *n* entit typu *A* má vazbu na jednu entitu typu *B*. *1* u entitního typu *B* značí, že *1* entita typu *B* má vazbu na jednu entitu typu *A*. Povinnost/nepovinnost v členství značíme plným resp. prázdným kolečkem na spojnici obdélníku entity a hrany vazby.

Příklad 2.2 (ER diagram).

Vezměme entitní typy `Student`, `Kurz` a `Ucitel` z příkladu 2.1. Na obrázku 2.3 vidíme ER diagram modelující tyto entitní typy a jejich vazby. Vidíme, že entitní typ `Student` má atributy `login`, `jmeno`, `email` a `rocnik` a atribut `login` je klíč. Vazba `STUDUJE` mezi entitním typem `Student` a `Kurz` má násobnost *M : N*. Oba entitní typy nemají povinnost v členství. Vazba `UCI` mezi entiními typy `Ucitel` a `Kurz` má násobnost 1 : N. Tedy, jeden učitel učí více kurzů, jeden kurz je vyučován jedním učitelem. Kurz musí být učen nějakým učitelem, učitel nemusí učit žádný předmět.



Obr. 2.3 ER diagram modelující entitní typy `Student`, `Kurz` a `Učitel` jejich vazby.

2.2.2.3 Enhanced ERD

Později bylo nutné obohatit původní ERD o dědičnost a další rysy, proto byl vyvinut *Enahanced ERD* [12]. Problémem je, že takovým obohacením získáme poměrně komplikovaný jazyk, který ovšem neumožňuje modelovat více rysů než standardizovaný jazyk UML.

2.2.3 Konceptuální model UML

2.2.3.1 Úvod

Unified Model Language (*UML*) [21] je vizuální, objektově-orientovaný modelovací jazyk zachycující strukturální a dynamické aspekty softwarového systému. UML, na rozdíl od ERD, je kolekce modelovacích technik, které jsou aplikovány na různé aspekty vývoje softwaru. Každá technika poskytuje odlišný statický nebo dynamický pohled na aplikaci. Kolekci pohledů nazýváme *model*. UML poskytuje tyto techniky:

- *Třídní diagram* (*Class Diagram*) - statický diagram struktury.
- *Objektový diagram* (*Object Diagram*) - ukazuje specifické instance tříd.
- *Diagram případu použití* (*Use-Case Diagram*) - poskytuje popis systému a jeho použití z pohledu uživatele.
- *Stavový diagram* (*State diagram*) - popisuje stavy objektů a změny ve stavech, které nastávají jako odpověď na události.
- *Sekvenční diagram* (*Sequence diagram*) - prezentuje interakce mezi objekty.
- *Diagram aktivit* (*Activity diagram*) - graf toků ukazující úlohy, které jsou vykonávány ve výpočetním procesu.
- *Diagram spolupráce* (*Collaboration diagram*) - popisuje jak spolu různé elementy spolupracují.

Jazyk obsahuje celou řadu modelů, které je možné použít i při konceptuálním modelování databází [12]. Jedná se především o třídní digram, který modeluje entitní typy jako třídy a vazby jako asociace.

2.2.3.2 Třídy, atributy a metody

Typy entit modelujeme v UML pomocí *tříd*, vlastnosti objektů pomocí *atributů*. Třídu kreslíme jako obdélník rozdělený na tři části. V první části deklarujeme *název třídy* a tzv. *stereotyp*, který nám umožňuje více upřesnit význam třídy. V konceptuálním modelu bývá zvykem používat stereotyp `persistent` pokud se jedná o třídu jejíž instance budou uloženy v SŘBD. Do druhé části obdélníku třídy zapisujeme atributy a do třetí metody (funkce, které je možné vykonávat nad třídou nebo její instancí).

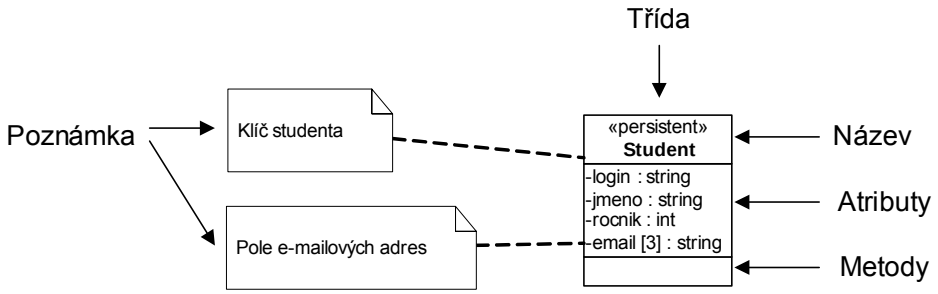
Obecná specifikace atributu je následující: `[stereotype] [visibility] [/] attributeName [multiplicity]: [type] [=initialValue]`. Nyní popišme jednotlivé části specifikace:

- `stereotype`: význam atributu.
- `visibility`: používáme zkratky + pro *public*; # pro *protected* a − pro *private*.
- / – označuje odvozený atribut.
- `attributeName`: jméno atributu.
- `multiplicity`: indikuje vícehodnotový atribut.

Příklad 2.3 (Třída).

Vezměme entitní typ `Student` z příkladu 2.2. Na obrázku 2.4 vidíme třídu reprezentující studenta. Evidujeme atributy `login`, `jmeno`, `email` a `rocnik`. Viditelnost

všech atributů je `private`, tedy na hodnotu atributu můžeme přistupovat pouze v instanci třídy `Student`. Datový typ atributů `login` a `jmeno` je řetězec, datovým typem atributu `rocnik` je číselný datový typ. Atribut `email` je pole tří řetězců – pole tří e-mailových adres. Vidíme, že pomocí *poznámek* můžeme více vysvětlit význam atributů. Jelikož UML neobsahuje označení pro *klíčový atribut*, tedy atribut, který jednoznačně identifikuje instanci třídy, použijeme pro specifikaci takového atributu poznámku. ■



Obr. 2.4 Třída reprezentující studenta.

Zdálo by se, že atributy končí seznam prvků třídy použitelných při konceptuálním modelování databáze. Ne tak docela. V klasických datových modelech jako je model relační (viz kapitola 2.3) ukládáme instance tříd do záznamů tabulek. Zde, zdá se, není pro metody místo. V poslední době je snaha o přenesení funkcionality na stranu SŘBD. Tzn. do SŘBD neukládáme pouze data, ale i metody, které vykonávají operace nad daty. V takovém případě využíváme tzv. *uložených procedur* v případě relačních SŘBD nebo *metod* v případě objektově-relačních SŘBD resp. objektových SŘBD. Vidíme tedy, že má smysl ve třídách definovat metody a ty potom realizovat v konkrétním datovém modelu.

Obecná specifikace metody je následující: `[stereotype] [visibility] methodName [parameters]: [returnType]`. Nyní popišme jednotlivé části specifikace:

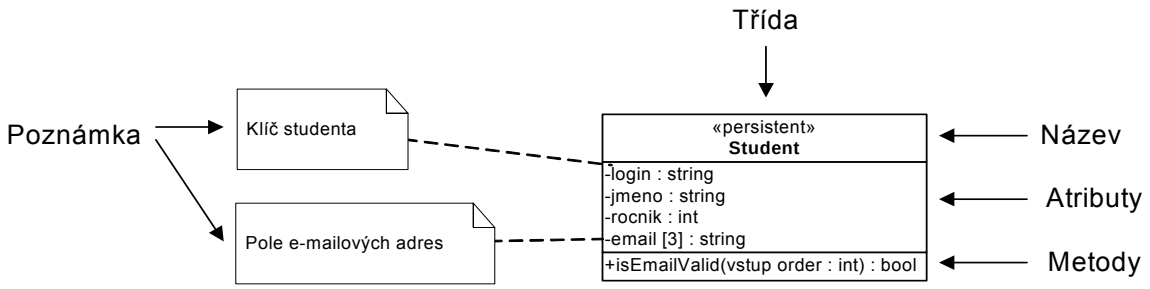
- `stereotype`: význam metody.
- `visibility`: používáme zkratky + pro *public*; # pro *protected* a − pro *private*.
- `parameters`: uspořádaný seznam parametrů oddělených čárkami.
- `methodName`: název metody.
- `returnType`: návratový typ metody.

Příklad 2.4 (Metody třídy).

V případě objektově-orientovaných jazyků definujeme tzv. *nastavovacích* a *zpřístupňujících* metody, což jsou metody, které umožňují nastavovat resp. číst data se soukromým přístupem. Například `setLogin()` resp. `GetLogin()`. V oblasti databází nemají tyto metody velký význam, protože členská data mají ze své podstaty veřejný přístup.

Můžeme ovšem deklarovat a definovat metody, kterými přeneseme část funkcionality na server. V případě třídy `Student` z obrázku 2.4 můžeme implementovat metodu `bool isEmailValid(int index)`, která bude vracet `true` v případě, že je *i*-tý email v poli platný email, `false` pokud platný není. Na obrázku 2.5 vidíme způsob zápisu této metody ve třídě.

V objektově–orientovaných technologiích rozlišujeme *členská* a *třídní* data a metody. Pro každou instanci třídy je vytvořena kopie členského atributu, zatímco všechny instance třídy sdílejí jednu kopii třídního atributu, jehož hodnotu je možné získat bez vytvoření instance. Členské metody je možné volat na instanci třídy, zatímco třídní metody jsou spjaty přímo se jménem třídy. Je zřejmé, že v třídních metodách není možné použít členské atributy.



Obr. 2.5 Metoda třídy reprezentující studenta

2.2.3.3 Zobecnění a specializace

Zobecnění a specializace jsou jedním ze základních kamenů objektově-orientovaných technologií. Vezměme třídy `Student` a `Osoba`. Je zřejmé, že třída `Student` je specializací třídy `Osoba` resp. `Osoba` je zobecněním třídy `Student`. Student obsahuje stejné data a metody jako `Osoba` a některé další, které více specializují tuto třídu. Říkáme, že `Student` *dědí* ze třídy `Osoba`. Použití dědičnosti vede jednak ke znuvupoužitelnosti implementace, ale také k tzv. *mnohotvárnosti* (*polymorfismu*). Tam kde je požadována instance třídy, může být totiž předána instance podtřídy. Opačně to není možné, nemůže instanci třídy vydávat za instanci speciálnější podtřídy.

Zdálo by se, že při modelování konceptuálního modelu databáze není pro dědičnost místo. Je jasné, že v relačním datovém modelu tuto vlastnost využijeme je v některých speciálních případech. Musíme si ale uvědomit, že konceptuální model je nezávislý na datovém modelu. Můžeme si tedy dovolit v konceptuálním modelu dědičnost použít a poté ji realizovat v konkrétním datovém modelu. Vezměme v úvahu celou řadu uživatelů v rozsáhlých informačních systémech. Dědičnost v konceptuálním modelu nám usnadní klasifikaci vztahů těchto uživatelů, které může vyústit ve vybudování hierarchie uživatelů.

Příklad 2.5 (Dědičnost).

Na obrázku 2.6 vidíme dva ekvivalentní způsoby nakreslení dědičnosti. Vidíme tedy, že třídy reprezentující studenta resp. učitele dědí ze třídy `Osoba` atributy `login`, `jmeno`, `email` a metodu `isEmailValid()`. Nyní má tedy třída `Student` stejné atributy jako `Osoba` a navíc atribut `rocnik`. V případě třídy `Ucitel` je tímto atributem navíc atribut `uvazek`. Obě třídy obsahují metodu `isEmailValid()`.

2.2.3.4 Asociace a role

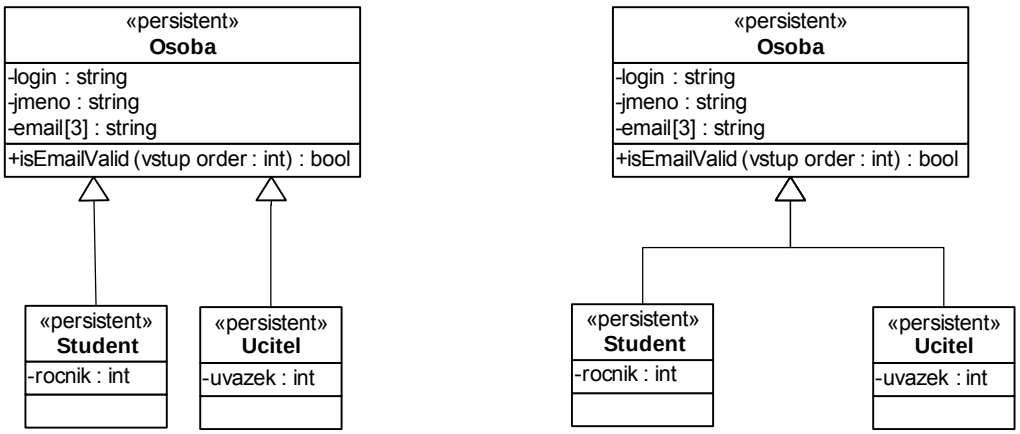
Vazbu mezi třídami modelujeme v UML pomocí *asociací*. Asociace mezi třídami je naznačena hranou mezi těmito třídami. Asociace je definována názvem (popř. směrem), *rolí*

pro každou třídu asociace a *násobností*. Násobnost říká kolik instancí třídy pro niž násobnost definujeme je ve vztahu s instancí druhé třídy. Rozlišujeme tedy násobnosti 0, 1, 0..*, 1..* atp.

Příklad 2.6 (Asociace).

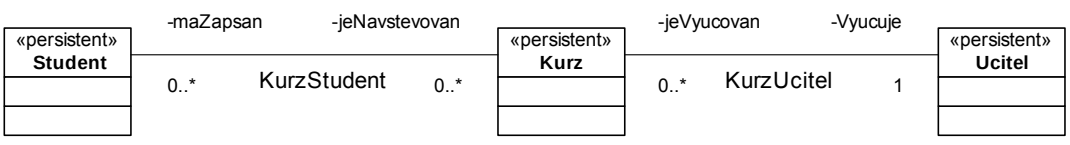
Mějme třídy `Student` a `Ucitel` z předchozích příkladů. Není vezměme třídu `Kurz`, která je abstrakcí kurzu, který si student může zapsat a který učitel vyučuje. Na obrázku 2.7 vidíme třídní digram modelují vztahy mezi těmito třídami. Asociace mezi třídami `Student` a `Kurz` se jmenuje `KurzStudent`. Jedna instance třídy `Student` má vzah s 0–*n* instancemi třídy `Kurz`. Stejná násobnost platí i pro jednu instanci třídy `Kurz`. Roli u studenta jsme označili `maZapsan`, u kurzu `jeNavstevovan`. Asociace mezi třídami `Kurz` a `Ucitel` se nazývá `KurzUcitel`.

Všimněme si, že třídy neobsahují atributy ani metody. Pokud v daném třídním diagramu ukazujeme např. asociace, může být uvedení atributů a metod matoucí. Z tohoto důvodu kreslíme jeden třídní diagram obsahující atributy a metody a druhý ve kterém se zaměřujeme na asociace, který atributy a metody neobsahuje. Takový třídní diagram je potom kompaktnější a přehlednější.



Obr. 2.6 Dva ekvivalentní způsoby nakreslení dědičnosti. Třídy `Student` a `Ucitel` dědí ze třídy `Osoba`.

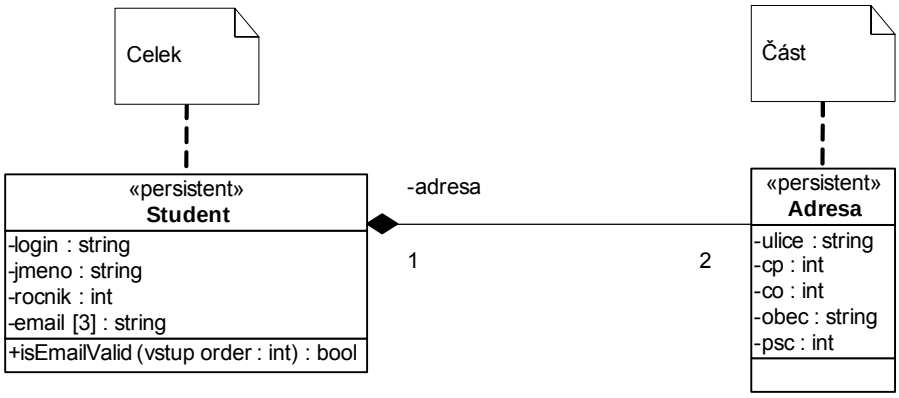
UML rozlišuje celou řadu asociací, *agregace* je asociace, kterou můžeme využít při modelování databází. Této asociaci se také někdy říká asociace typu *skládá se z* (*consist of*). Jedna třída má funkci celku a říkáme, že její instance se skládá z *n* instancí druhé třídy. Agregaci označujeme vyplněným kosodélníkem u třídy, která má funkci celku.



Obr. 2.7 Třídní digram modelující vztahy mezi třídami `Student`, `Kurz` a `Ucitel`.

Příklad 2.7 (Agregace).

Na obrázku 2.8 vidíme agregaci mezi třídami `Student` a `Adresa`. `Student` obsahuje dvě instance třídy `Adresa`.



Obr. 2.8 Ukázka agregace mezi třídami `Student` a `Adresa`.

2.3 Datový model

Datový model představuje zobrazení konceptuálního modelu do modelu, který je blíže fyzické implementaci SŘBD. Tento model nám určuje jakým způsobem budou data modelována. Ze způsobem modelování dat úzce souvisí i jejich dotazování.

2.3.1 Relační datový model

2.3.1.1 Úvod

V roce 1970 publikoval Edgar F. Codd článek "*A Relational Model of Data for Large Shared Data Banks"* [9] uvádějící *relační datový model*. Důvod proč je dnes používán v drtivém množství nasazení SŘBD je v jeho jednoduchosti – základem je jednoduchý koncept: *relace*, kterou si můžeme představit jako dvourozměrnou tabulku. SŘBD založené na relačním datovém modelu budeme nazývat relační SŘBD (*RSŘBD*, angl. *RDBMS*).

2.3.1.2 Základní pojmy

Definice 2.1 (*Relace*).

Nechť $D_1,D_2,...,D_n$ jsou množiny. *N*-ární relaci na množinách $D_1,D_2,...,D_n$ nazýváme libovolnou podmnožinu kartézského součinu, $R \subseteq D_1 \times D_2 \times ... \times D_n$.

Relace je abstrakcí dvou-rozměrné tabulky. Každý řádek obsahuje hodnoty *atributů* entity, každý sloupec obsahuje hodnoty jednoho atributu. V našem případě budou množiny $D_1,D_2,...,D_n$ množiny hodnot jednotlivých *atributů*, tzv. *domény*. Nyní známe tabulku, pro úplný popis dat potřebujeme znát i hlavičku této tabulky – *relační schéma*.

Definice 2.2 (*Relační schéma*).

Relační schéma relace $R \subseteq D_1 \times D_2 \times ... \times D_n$ je zápis $\boldsymbol{R}(A_1,A_2,...,A_n, D_1,D_2,...,D_n)$, kde $\boldsymbol{R}$ je jméno schématu, $A_1,A_2,...,A_n$ jsou jména atributů, $D_1,D_2,...,D_n$ jsou domény jednotlivých atributů.

O relaci říkáme, že je typu $\boldsymbol{R}$ nebo, že je instancí relačního schématu $\boldsymbol{R}$. Konečná množina schémat se nazývá *schéma relační databáze*, množina relací k těmto schématům se nazývá *relační databáze*.

Příklad 2.8 (Relace).

Vezměme relační schéma `Student`(`login`, `jmeno`, `prijmeni`, `rokNarozeni`). V tabulce 2.1 vidíme relaci typu `Student`, která obsahuje řádky s hodnotami atributů `login`, `jmeno`, `prijmeni` a `rokNarozeni`. Vidíme, že pojem atributu u relačního modelu splývá s pojmem atribut u konceptuálních modelů ERD a UML. Sloupce obsahují hodnoty atributů jednotlivých studentů.

Striktně matematicky jedna se o podmnožinu $D_1 \times D_2 \times D_3 \times D_4$, kde D_1 je množina všech jedinečných čísel studentů, D_2 je množina všech jmen, D_3 množina všech příjmení a D_4 množina všech roků narození. Prvky relace jsou čtveřice, můžeme tedy psát `Student`={('svo005', 'Michal', 'Svoboda', 1982), ('lam001', 'Jana', 'Lampová', 1983), (' nov003', 'Jitka', 'Novotná', 1981)}. Obecně je prvkem relace *n-tice* (angl. *tuple*).

Tab. 2.1 Relace `Student`

login	jmeno	prijmeni	rokNarozeni
svo005	Michal	Svoboda	1982
lam001	Jana	Lampová	1983
nov003	Jitka	Novotná	1981

Musíme si uvědomit, že relace je množina n-tic, nikoli uspořádaná množina n-tice a že relační schéma obsahuje množinu názvů atributů, nikoli uspořádanou množinu názvů atributů. Nemůže tedy mluvit o prvním prvku, druhém, atd. prvku resp. sloupci relace. V tabulce 2.2 vidíme relaci `Student`, která je ekvivalentní s relací z tabulky 2.1. V praxi velmi často používáme označení první atribut atd., ale pouze v případě, kdy je jednoznačně přiřazen sloupec relace k názvu atributu.

Tab. 2.2 Relace `Student`

login	rokNarozeni	jmeno	prijmeni
svo005	1982	Michal	Svoboda
nov003	1981	Jitka	Novotná
lam001	1983	Jana	Lampová

V relačním datovém modelu identifikujeme atribut relačního schématu dle jehož hodnoty je n-tice odlišitelná od ostatních. Takový atribut nazýváme *klíčem*. Atribut relačního schématu, který obsahuje klíč jiného schématu nazýváme *cizí klíč*. Klíčem relačního schématu `Student` z příkladu 2.8 je atribut `login`.

2.3.1.3 Od konceptuální modelu k modelu relačnímu

Jak bylo řečeno v předchozí kapitole, pojem atributu v konceptuálním a relačním datovém modelu je ekvivalentní. V následujícím textu budeme používat pojmy vlastní konceptuálnímu modelu UML. Pokud bude použit konceptuální model ERD, čtenář jistě použije ekvivalentní pojmy. Přímý přístup k převodu konceptuálního modelu do relačního datového modelu, může tedy vypadat následovně:

1. Třídy popř. entitní typy a jejich atributy jsou mapovány na relační schémata s příslušnými atributy.
2. Vazby M : N jsou implementovány pomocí relačních schémat, které obsahují cizí klíče – primární klíče tříd v asociaci.

Takovýmto jednoduchým postupem jsem často schopni získat větší část schématu relační databáze. Mohou ovšem nastat situace, které musíme řešit jiným postupem. V kapitole 2.3.1.3 si ukážeme aparát, který nám umožní rozpoznat, zda námi navržené schéma relační databáze je korektní a rovněž nám umožní případné chyby odstranit.

Příklad 2.9 (Převod konceptuálního modelu na schéma relační databáze).

Vezměme konceptuální model reprezentovaný třídním diagramem z příkladu 2.6. V tomto případě jsou vytvořena relační schémata `Student(login, jmeno, email, rocnik)`, `Kurz(id, nazev, kapacita)` a `Ucitel(login, jmeno, email, uvazek)`. ■

Asociace je v relačním modelu reprezentovaná relací. Při transformaci asociace do relačního datového modelu vytvoříme relační schéma obsahující cizí klíče pro všechny třídy v asociaci. Případné atributy asociace vložíme rovněž do schématu. Pokud je nějaká třída přítomna v asociace v různých rolích, musíme do relace reprezentující asociaci přidat pro takovou třídu cizí klíč s odlišným jménem.

Příklad 2.10 (Převod asociace na relaci).

Vezměme konceptuální model reprezentovaný třídním diagramem z příkladu 2.6. Asociace `KurzStudent` je převedena na relační schéma `KurzStudent(kurzId, studentLogin)`. Asociace `KurzUcitel` je převedena na relační schéma `KurzUcitel (kurzId, ucitelLogin)`. Z důvodu přehlednosti byl k názvům atributů přidán prefix naznačující relační schéma, ke kterému atribut náleží. Ukázku relací typu `KurzStudent` a `KurzUcitel` vidíme v tabulce 2.3. Kurz `45613` studují dva studenti, kurz `45601` tři studenti, kurzy `45603` a `45638` studuje jeden student. Jak vyplývá z konceptuálního modelu (ze schématu relační databáze také) jedná se o asociaci typu M:N. Asociace `KurzUcitel` je typu 1:M.

Při převodu dědičnosti do relačního modelu existuje několik variant. Jejich výběr závisí především na množství atributů, které obsahují bázové třídy. První variantou je vytvoření schématu pro každou třídu v hierarchii. Toto řešení je vhodné aplikovat pokud třídy obsahují velké množství různých atributů. Druhou variantou je zařadit všechny atributy do relačního schématu a navíc přidat atribut, který bude indikovat příslušnost n-tice ke třídě.

Tab. 2.3(a) Relace `KurzStudent`

`kurzId`	`studentLogin`
45613	svo005
45613	nov003
45601	svo005
45601	nov003
45601	lam001
45603	lam001
45638	lam001

Tab. 2.3(b) Relace `KurzUcitel`

`kurzId`	`ucitelLogin`
45613	kra102

45601	kra102
45603	dvo005
45638	svo032

Příklad 2.11 (Převod dědičnost do relačního datového modelu).

Vezměme v úvahu dědičnost v třídním diagramu z příkladu 2.4. V tomto případě bude výhodnější použít druhou variantu. Vytvoříme relační schéma `Osoba(login, jmeno, prijmeni, email, rocnik, uvazek)`. Pokud bude n-tice instancí třídy `Student`, bude hodnota atributu `uvazek` rovna `NULL`, pokud bude instancí třídy `Ucitel` bude naopak hodnota atributu `rocnik` rovna `NULL`.

2.3.1.4 Funkční závislosti

V předchozí kapitole jsem popsali proces přímého převodu konceptuálního modelu do modelu relačního. I když je možné použít takovýto postup, není vždy jednoduché najít "dobré" schéma relační databáze. Nalezení takového schématu lze algoritmizovat pomocí různých typů integritních omezení. Jedním z těchto typů je *funkční závislost* (angl. *function dependency* - *FD*).

…

2.3.2 Objektový a objektově-relační datový model

Rozvoj objektově-orientovaných technologií v 90. letech 20. století se nevyhnul ani oblasti databází. *Objektově-orientované SŘBD* (*OOSŘBD,* angl. *OORDBMS*) umožňují používat uživatelské typy, dědičnost, metody tříd, rozlišují pojmy instance a ukazatel na instanci. Pokud jsme schopni definovat uživatelský datový typ, je zřejmé, že podmínka atomičnosti atributu pozbývá v tomto modelu platnost. Tento model nám tedy poskytuje celou řadu výhod, které ovšem 90% aplikací nemusí nutně používat a které je možné více méně elegantně (z pohledu modelování dat), řešit pomocí relačního modelu. Navíc, datový model velmi málo vypovídá o fyzické implementaci dat. Komplikovanější datový model značí komplikovanější a nenutně efektivnější fyzickou implementaci dat. Jinými slovy, pro většinu aplikací je postačující relační datový model. Transformace existujících databází na odlišný datový model by si vyžádala obrovské úsilí a množství financí, bez jakékoli objektivního kladného efektu. Z těchto důvodů se jako většinové prosadilo kompromisní řešení – relační datový model je vhodně doplňován o objektově-orientované prvky, výsledkem je objektově-relační model. Standard SQL obsahující objektově-orientované prvky se nazývá SQL-99 [25, 3].

Příklad 2.20 (Vytvoření schématu v ORSŘBD).

Na tomto příkladu si ukážeme jakým způsobem je relační model obohacen o objektově-orientované rysy. Příkazy jsou určeny pro ORSŘBD Oracle [Ora02]. V SQL-99 můžeme definovat uživatelský typ, atributy tudíž nemusí být atomické. Typ `TPerson` obsahuje instanci typu `TAddress`:

```
CREATE OR REPLACE TYPE TAddress AS OBJECT (
  street VARCHAR2(30),
  city VARCHAR2(30),
  PSC NUMBER(5));
CREATE OR REPLACE TYPE TPerson AS OBJECT (
  login VARCHAR2(6),
  fname VARCHAR2(20),
  sname VARCHAR2(20),
  address TAddress,
  …
) NOT FINAL NOT INSTANTIABLE;
```

Typ `TTeacher` dědí z typu `TPerson`, který je deklarován jako typ jehož instanci není možné vytvořit (pomocí klíčového slova `NOT INSTANTIABLE`) a jako typ, ze kterého je možné dědit (`NOT FINAL`) .

```
CREATE OR REPLACE TYPE TTeacher UNDER TPerson (
  department REF TDepartment,
  title VARCHAR2(30),
  …
  STATIC FUNCTION authentication (login VARCHAR2,
      password VARCHAR2) RETURN REF TTeacher );
```

Klíčové slovo `REF` znamená, že hodnotou atributu je ukazatel na instanci třídy `TDepartment`, která reprezentuje katedru učitele. Vidíme, že třída `TTeacher` obsahuje třídní metodu, která vrací ukazatel na třídu `TTeacher` v případě, že byl zadán korektní login a heslo. Typ `TDepartment` obsahuje ukazatel na vedoucího katedry - instanci `TTeacher`.

```
CREATE OR REPLACE TYPE TDepartment AS OBJECT (
  name VARCHAR2(45),
  shortcut NUMBER(3),
  chief REF TTeacher);
```

2.3.3 Datový model XML

2.3.3.1 Úvod

eXtensible Markup Language (*XML*) [27] je značkovací jazyk specifikovaný konsorciem *World Wide Web Consortium*[1] (*W3C*). Kromě pohledu na XML jako jazyk pro přenos slabě strukturovaných dat, se můžeme na XML dívat jako na jazyk pro modelování dat. Obecně je XML dokument modelován jako graf, pro naše představy bude plně postačující reprezentovat XML dokument jako *strom* (angl. *XML tree*).

XML dokument obsahuje *elementy*, které mohou být libovolně zanořeny a *atributy*, které náleží k elementům. Pokud má být dokument nazván *dobře strukturovaný* (*well-formed*) musí splňovat některá syntaktická pravidla. Element je specifikován *názvem* (někdy mluvíme o *typu elementu*), v textu je otevřen tzv. *počáteční značkou* (*start tag*): název elementu uvezený mezi znaky <>. Element je ukončen tzv. *koncovou značkou* (*end tag*) : název elementu uvezený mezi znaky </ a znak >. Mezi počáteční a ukončovací značkou se nachází *obsah elementu*, ten mohou tvořit dětské elementy a volný text. Pokud element nemá žádný obsah, můžeme koncovou značku vynechat a před znak > počáteční značky vložit znak /.

[1] http://www.w3c.org

Např. `<img src=”img.jpg”/>`. Element může obsahovat 0–*n specifikací atributů*, dvojic *název atributu=hodnota atributu*, které jsou přiřazeny počáteční značce a jsou odděleny mezerami. Obecně platí, že elementy by měly obsahovat data a hodnoty atributů *metadata* (tedy informace o datech).

Příklad 2.21 (XML dokument).

Na obrázku 2.9 vidíme XML dokument reprezentující knihy a jejich autory. Dokument obsahuje jeden kořenový element `books` s *n* podelementy `book`. Element `book` obsahuje dva dětské elementy `title` a `author` a atribut `id`. V případě XML rozlišujeme pořadí elementů, to je velký rozdíl oproti relačnímu datovému modelu. Např. element `book` s `id="003-04312"` je prvním dítětem elementu `books`. Jelikož relace je množina, u relačního datového modelu nerozlišujeme pořadí n-tic v tabulce.

Poznámka: Některé malé SŘBD (typicky založené na *dBASE* [11], např. *FoxPro*, *Clipper*), používají pořadová čísla záznamů v tabulkách. Tato vlastnost ovšem nenachází žádnou podporu v původním relačním datovém modelu a u velkých SŘBD (jako Oracle [22] nebo DB2 [16]) není implementována.

```
<?xml version="1.0" ?>
<books>
 <book id="003-04312">
   <title>The XML Book</title>
   <author>John Smyth</author>
 </book>
 <book id="045-00012">
   <title>The XQuery Book</title>
   <author>Frank Nash</author>
 </book>
</books>
```

Obr. 2.9 XML dokument reprezentující knihy a jejich autory

2.3.3.2 Schéma XML dokumentu

Stejně jako v případě relačního modelu i zde můžeme mluvit o schématu, tentokrát o schématu XML dokumentu. Takové schéma nám tedy říká, jaké typy elementů se v dokumentu vyskytují, jakého typu jsou podelementy jiných elementů atd. Původní jazyk pro popis schémat je *DTD* [27], z novějších uveďme *XML Schema* [28, 29] jakožto standard W3C. Dokument, který splňuje definované schéma označujeme jako *validní*. Úplný popis těchto jazyků je mimo rámec kurzu, uveďme si alespoň příklad schématu XML dokumentu z obrázku 2.9. Na obrázku 2.10 a 2.11 vidíme schéma dokumentu definované v jazyku DTD resp. XML Schema. Na první pohled je patrný rozdíl, zatímco XML Schema je založeno na XML, DTD používá vlastní syntaxi. DTD tedy nezapadá do koncepce XML technologií.

```
<!DOCTYPE books [
 <!ELEMENT books(book)>
 <!ELEMENT book(title,author)>
 <!ATTLIST book id CDATA #REQUIRED>
 <!ELEMENT title(#PCDATA)>
 <!ELEMENT author(#PCDATA)>
]>
```

Obr. 2.10 Schéma dokumentu z obrázku 2.9 definované pomocí DTD

```
<?xml version="1.0"?>
<xsd:schema
xmlns:xsd="http://www.w3.org/2001/XMLSchema">

 <xsd:element name="books">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="book" maxOccurs="unbounded">
     <xsd:complexType>
      <xsd:sequence>
       <xsd:element name="title" type="xsd:string"/>
       <xsd:element name="author" type="xsd:string"/>
      </xsd:sequence>
      <xsd:attribute      name="id"      type="IdType"
use="required"/>
     </xsd:complexType>
    </xsd:element>
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>

 <xsd:simpleType name="IdType">
  <xsd:restriction base="xsd:string">
   <xsd:length value="9"/>
   <xsd:pattern value="[0-1]-[0-1]"/>
  </xsd:restriction>
 </xsd:simpleType>
</xsd:schema>
```

Obr. 2.11 Schémata dokumentu z obrázku 2.9 definované pomocí XML Schema

Vidíme rovněž, že XML Schéma umožňuje definovat datové typy hodnot elementů. Tato vlastnost je výhodná zejména u XML databází. Dále můžeme definovat vzory (viz definice typu `IdType`), které nám mohou pomoci k validaci hodnot elementů.

2.3.3.3 Model XML

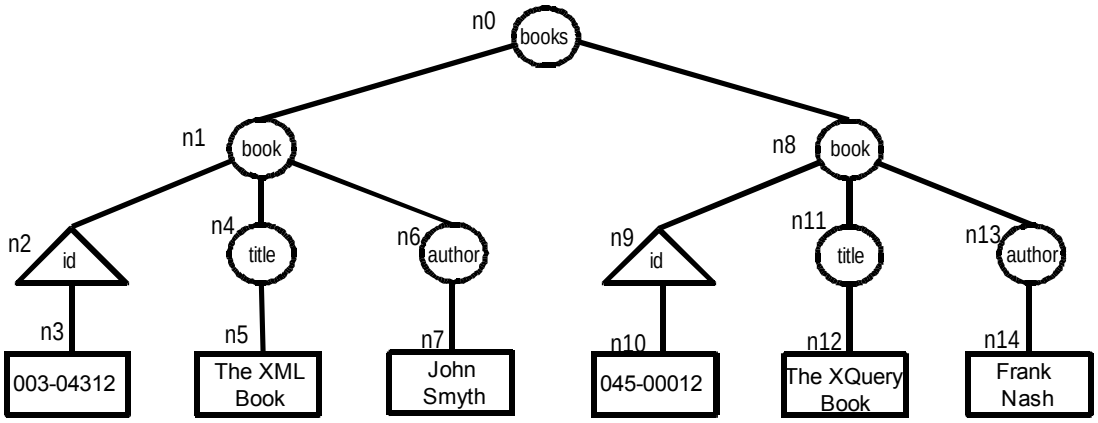
XML dokument můžeme modelovat jako strom. Elementy a atributy modelujeme jako uzly. Vztah *element-podelement* (tzv. *rodič-dítě*) modelujeme hranou mezi příslušnými uzly.

Atribut je modelován jako dítě příslušného elementu. V XML dokumentu je definováno uspořádání na uzlech stromu, tzv. *uspořádání dokumentu* (angl. *dokument ordering*). Toto uspořádání je ekvivalentní s *dopředným uspořádáním* (angl. *preorder*). V tomto uspořádání přiřazujeme každému uzlu stromu hodnotu čítače, který je inkrementován po každém načtení elementu z textové reprezentace dokumentu. Atributy elementu jsou v uspořádání dokumentu umístěny před dětské elementy.

Příklad 2.22 (XML strom).

Na obrázku 2.12 vidíme strom XML dokumentu z obrázku 2.9. Pořadová čísla *i* uzlu n_i odpovídají uspořádání dokumentu. Vidíme, že řetězcové obsahy elementů a hodnoty atributů můžeme také považovat za speciální uzly XML stromu. Upořádaní uzlů dle uspořádání dokumentu je $n_0 < n_1 < ... < n_{14}$. ■

Uvědomme si co plyne z větší složitosti datového modelu XML ve srovnání s relačním datovým modelem. Jazyk vysoké úrovně SQL (viz kapitola 2.4.2) umožňuje pracovat s dvou-rozměrnými tabulkami. Je zřejmé, že z větší složitosti datového modelu XML, plyne větší složitost dotazovacího jazyka nad XML (např. XQuery, viz kapitola 2.4.3) a samozřejmě také vyšší složitost efektivní implementace takového dotazovacího jazyka.



Obr. 2.12 Strom XML dokumentu z obrázku 2.9

2.3.3.4 Nativní XML databáze

Připomeňme myšlenku z předchozí kapitoly. Pro 90% aplikací postačuje relační datový model. Mohou ovšem existovat data, u kterých můžeme využít rysy jazyka XML. Jazyk XML modeluje data slabě-strukturovaná, tedy např. články v digitálních knihovnách, webové stránky apod. V případě těchto dat můžeme data uložit v tzv. nativních XML databázích a dotazovat je pomocí XML dotazovacích jazyků. Vývoj těchto databází je teprve na začátku. Výrobci velkých SŘBD se snaží do svých relačních strojů integrovat podporu XML. Velmi často se jedná o pouhé triviální uložení XML dat. Proto mají takové implementace problém s výkonem při dotazování velkého množství dat. Je zřejmé, že ke skutečné XML databázi vede poměrně dlouhá cesta.

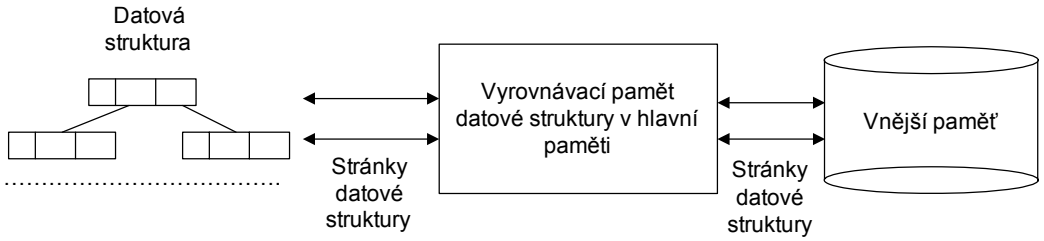
2.5 Fyzická implementace uložení dat

2.5.1 Úvod

Libovolný datový model by byl nepoužitelný bez využití efektivních *datových struktur* a *algoritmů* [32, 13]. Vezměme v potaz tento příklad. Máme v databázi uloženo 10^8 záznamů o klientech bankovního ústavu. Chtějme v těchto záznamech vyhledávat podle jedinečného čísla, které je primárním klíčem. Uvažujme triviální algoritmus, který by sekvenčně procházel záznamy a postupně porovnával hodnotu tohoto atributu všech záznamů. Pokud by porovnání dvou hodnot trvalo 1ms, pak by vyhledání správného záznamu trvalo 27h46m40s. Bezpochyby by byl takový SŘBD v praxi nepoužitelný. Je tedy jasné, že fyzická implementace uložení dat je jedním z podstatných problémů databázových technologií.

2.5.2 Perzistentní datové struktury

Je zjevné, že datové struktury obsahující data musí být *perzistentní* [17], tj. musí být uloženy v nějaké vnější paměti, kde zůstanou uloženy i po nekorektním ukončení programu, restartu počítače apod. Na obrázku 2.14 vidíme způsob implementace perzistentních datových struktur. Datové struktury jsou *stránkované*, tj. skládají se z různého počtu stránek (v případě stromových datových struktur jsou to uzly stromu). Každá stránka je mapována na různý počet stránek *vnější paměti*. V současnosti je nejpoužívanějším typem vnější paměti magnetický disk, který obsahuje tzv. *diskové bloky*. Diskový blok je nejmenší jednotka souborového systému, se kterou může operační systém samostatně pracovat.



Obr. 2.14 Způsob implementace perzistentních datových struktur: stránky datové struktury jsou mapovány na stránky vnější paměti (nejčastěji diskové bloky) a přenášeny mezi vnější a vnitřní pamětí

Pokud chce datová struktura načíst nějakou stránku, ta je načtena z vnější paměti do vyrovnávací paměti v hlavní paměti. Pokud je vyrovnávací paměť plná, z vyrovnávací paměti je vymazána stránka, která byla použita před nejdelší dobou, a na její místo je načtena stránka nová. Pokud byla původní stránka modifikována, je před smazáním z vyrovnávací paměti uložena do vnější paměti.

Míra efektivity, kterou budeme v následujícím textu používat, se označuje jako *složitost algoritmů* [13]. Nejčastěji budeme pro vyjádření kvality používat asymptotickou míru složitostí *O(g(n))*, která nám říká jak je složitost algoritmu omezena shora.

Definice 2.5 (*Míra složitosti O(g(n))*)

Pro každou funkci *g(n)*, označíme zápisem *O(g(n))* množinu funkcí *O(g(n))* = {*f(n)*: takových, že existují kladné konstanty *c* a n_0 tak, že $0 \leq f(n) \leq cg(n)$ pro všechna $n \geq n_0$}. ■

To tedy znamená, že 5n=*O(n)* nebo 120n=*O(n)*. Tato míra nám tedy určuje složitost pouze asymptoticky. Pokud má algoritmus složitost v *O(n)*, víme že počet operací provedených během chodu algoritmu je $\geq n \times$ konstanta *c*. Známe tedy relativně nepřesnou

informaci. Co je ale důležité, víme že počet operací nebude například exponenciální s počtem prvků na vstupu *n*.

2.5.3 Datové struktury a algoritmy

Algoritmus, který jsem si popisovali v kapitole 2.5.1 byl algoritmus vyhledání v poli s asymptotickou složitostí v *O(n)* [13]. Pokud bychom použili algoritmus vyhledávání v setříděném poli, pak bychom dosáhli složitosti *O(log n)*. Pro 10^8 záznamů bychom jeden záznam vyhledali za 8 x 0.001s = 0.008s. Zdálo by se, že takové řešení je vyhovující. Zaměřme se nyní na algoritmus vkládání a modifikace záznamu. Pokud bychom chtěli třídit záznamy při vkládání do pole, museli bychom nejprve nalézt místo v souboru kde bude nový záznam vložen (tedy zatřízen). Poté musíme všechny záznamy ležící za ním překopírovat o jeden záznam ke konci souboru a nový záznam vložit. Je jasné, že takový algoritmus se nedá, kvůli velkým přesunům dat, nazvat efektivní.

V minulosti byly velmi často používané tzv. *indexové soubory*. Záznamy byly vkládány za sebe do datového souboru, tak jak byly uživatelem vkládány. SŘBD obsahoval příkaz, kterým bylo možné k takovému souboru vytvořit indexový soubor, setříděné pole dvojic <hodnota indexovaného atributu, ukazatel do datového souboru>. Takováto technika řeší náš problém pouze částečně. Indexový soubor sice neobsahuje celé záznamy jako datový soubor, pokud by měl být indexový soubor vytvářen při každém vložení záznamů, k žádnému markantnímu zlepšení efektivity nedojde.

Našim cílem je tedy nalézt datovou strukturu s logaritmickými složitostmi operací: *vyhledání* (*find*), *vkládáni* (*insert*), *rušení* (*delete*) a *modifikace* (*update*). Logaritmická složitost ovšem musí být dosažena i v případě, kdy jsou stránky datové struktury ukládány do souboru.

2.5.4 Stromové datové struktury

Nyní se zaměřme na *stromové datové struktury* [1], z nichž některé druhy nám nabízí logaritmické složitosti pro všechny čtyři operace.

Definice 2.6 *(Kořenový strom)*.

Souvislý, acyklický, neorientovaný graf se nazývá *volným stromem* (*free tree*). *Kořenový strom* (*rooted tree*) je volný strom, který obsahuje jeden odlišný uzel (*kořen - root*). *Seřazený strom* (*ordered tree*) je kořenový strom, ve kterém jsou potomci každého uzlu seřazeni.

Uzly, které nemají žádné děti nazýváme *listové uzly*, uzly, které mají děti, pak *vnitřní uzly* stromu. Počet dětí nějako uzlu *u* nazýváme *stupeň uzlu u*. *Cesta* k nějakému uzlu *u*, je posloupnost všech uzlů od kořene k uzlu *u*. *Délka cesty* se je rovna počtu hran, které cesta obsahuje, tedy počtu uzlů posloupnosti – 1. *Výška stromu h* je rovna délce nejdelší cesty ve stromu. Výšku stromu můžeme rovněž definovat přes hloubku uzlu. Hloubka uzlu *u* je definována jako délka cesty k uzlu *u*. Výška stromu je tedy rovna největší hloubce všech uzlů stromu.

Stromů existuje celá řada, nejprve si uvedeme binární vyhledávácí strom, který nám bude sloužit pro demonstraci vlastností stromových datových struktur. V kapitole 2.5.4.2 popíšeme datovou strukturu B-strom, která je používaná v drtivé většině SŘBD.

2.5.4.1 Binární vyhledávací strom

Definice 2.7 (*Binární strom*)

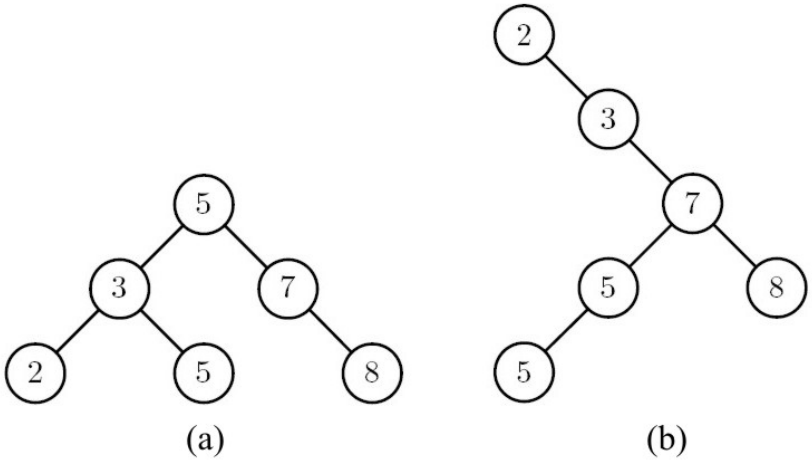
Binární strom je struktura rekurzivně definována nad konečnou množinou uzlů, která buď neobsahuje žádný uzel nebo je složena ze tří disjunktních množin uzlů: *kořene*, binárního stromu zvaného l*evý podstrom* a binárního stromu zvaného *pravý podstrom*.

U binárního stromu, pokud chybí potomek nějakého uzlu, pak pořád rozlišujeme levého a pravého potomka. U seřazeného stromu takové rozlišení není možné, binární strom tedy není seřazený strom.

Každý uzel *binární vyhledávací stromu* obsahuje *klíč*, na jehož doméně je definováno nějaké uspořádání. Pro jednoduchost si představme množinu N a upořádání $\leq$. V pravém podstromu binárního vyhledávacího stromu s kořenem obsahujícím klíč k_0 jsou uloženy uzly s klíči $\leq k_0$, v pravém podstromu uzly s klíči $\geq k_0$.

Příklad 2.33 (Binární vyhledávací strom).

Na obrázku 2.15 vidíme příklady binárních vyhledávacích stromů. Na obrázku (a) vidíme strom s kořenem obsahujícím klíč 5. Levý podstrom má kořen s klíčem 3, pravý podstrom má kořen s klíčem 7. Na obrázku (b) vidíme strom s kořenem obsahující klíč 2. Levý podstrom je prázdný, pravý podstrom má kořen s klíčem 3. Takovýmto způsobem můžeme projít celým stromem až k listům.



Obr. 2.15 Příklady binárních vyhledávacích stromů

Vezměme cestu 5,4,8 ve stromu z obrázku (a) k listovému uzlu s klíčem 8. Délka této cesty je 2. Všechny cesty v tomto stromu mají délku 2, výška stromu $h = 2$. Na obrázku (b) vidíme cestu 2,3,7,8 k listovému uzlu s klíčem 8. Délka cesty je 3. Jelikož druhá cesta k listovému uzlu má délku 4, výška stromu $h = 4$.

Nyní se podívejme na základní operace nad binárním vyhledávacím stromem. Při *vyhledávání* hodnoty k nastavíme kořenový uzel jako aktuální. Pokud je klíč aktuálního uzlu $k_a > k$ nastavíme jako aktuální uzel levé dítě. Pokud je klíč aktuálního uzlu $k_a < k$ nastavíme jako aktuální uzel levé dítě. Pokud $k_a = k$ pak jsem hledanou hodnotu našli. Pokud narazíme na prázdný uzel, hledaná hodnota se ve stromu nenachází.

Základem algoritmů pro vkládání a rušení je algoritmus vyhledávání. Při *vkládání* nejprve nalezneme prázdný uzel, kam má být klíč vložen, vytvoříme nový uzel a vložíme jej na toto místo.

Všimněme si jedné vlastnosti binárních vyhledávacích stromů, cesty k listům mohou být různě dlouhé. V nejhorším případě bude na vyhledání klíče potřeba h porovnání. Pokud budeme do stromu vkládat postupně hodnoty 1,2,3,…,1 000. Dostaneme datovou strukturu u které bude potřeba 1 000 porovnání na vyhledání nějaké hodnoty. Složitost můžeme vyjádřit jako $O(h)$, kde $h = n$. Jinými slovy za určitých okolností se ze stromu stane pole (přesněji seznam) i s jeho složitostmi pro základní operace.

Důležitá vlastnost stromů je tedy tzv. *vyváženost*. Existuje celá řada stromů, které vyváženost definují různým způsobem. V dalším textu budeme považovat strom za vyvážený pokud bude mít délky cest ke všem listům rovny výšce stromu. Tedy hloubka všech listů bude totožná. V takovém případě roste počet uzlů stromu exponenciálně s úrovní stromu. Uvažujme strom, jehož každý uzel má stupeň 2. Tzn. kořenový uzel je jeden, uzlů pod kořenovým uzlem je 2, jejich dětí 4, jejich dětí 8 atd. Počet uzlů roste s exponentem 2, tedy stupněm uzlu c. Nyní se zamysleme co tento rys znamená pro délku cesty k listu, což přeneseně znamená počet porovnání pro základní operace. Řekli jsme, že počet uzlů roste exponenciálně, je zřejmé, že délka cesty k listu je definována funkcí inverzní, tedy logaritmickou: $(c^h)=n \Leftrightarrow h=\log_c n$. Pokud tedy uvažujeme takto vyvážený strom, získáváme složitost všech základních operací nad stromem v $O(\log_c n)$.

2.5.4.2 B-strom

Rudolf Bayer uvedl *B-strom* v [4]. B-strom je vyvážená perzistentní datová struktura, která v uzlech obsahuje $c – 2c$ položek. V případě B-stromu uzly nazýváme stránky.

Definice 2.8 *(B-strom)*

B-strom řádu c je $(2c+1)$-ární strom, který splňuje následující kritéria:

1. Každá stránka (uzel) obsahuje nejvýše $2c$ položek (klíčů).
2. Každá stránka, s výjimkou kořene, obsahuje alespoň n položek.
3. Každá stránka je buď listovou, tj. nemá žádné následovníky nebo má $m+1$ následovníků, kde m je počet klíčů ve stránce.
4. Všechny listové stránky jsou na stejné úrovni.

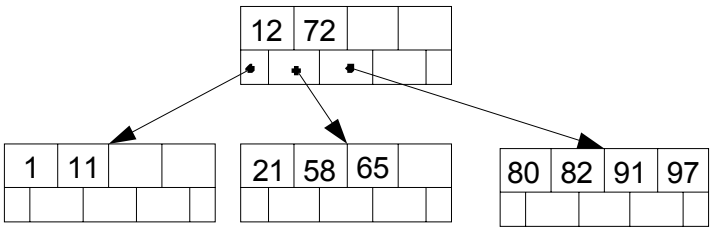
Složitost základních operací je $O(\log_c n)$, kde n je počet položek stromu. Důležitým hlediskem je *faktor využití paměti* (angl. *utilization*), který počítáme jako (počet položek uzlu / c) $\times$ 100 [%]. V případě B-stromu je zaručen faktor využití paměti minimálně 50%. Fyzická velikost stránky se volí v násobcích velikosti diskového bloku (většinou 2048B). Dle velikosti položky stromu (který kromě klíče může obsahovat i neindexovaní atributy) vybereme vhodnou kapacitu stránky c. Typicky se může kapacita pohybovat v rozmezí 20-50. Vidíme tedy, že B-strom má všechny vlastnosti, které požadujeme po indexovací datové struktuře: poskytuje dobrou složitost pro základní operace, je přirozeně perzistentní, a je relativně snadno implementovatelná.

Nejprve si ukažme operaci *vyhledání* položky k. Nastavme kořenový uzel jako aktuální. Nyní v aktuálním uzlu hledáme pořadí položky i takové, že platí $k_i \leq k \leq k_{i+1}$. Pokud takové

pořadí nalezneme, načteme *i*-té dítě uzlu a nastavíme tento uzel jako aktuální. Pokud v listech není hledaná položka nalezena, pak algoritmus končí.

Příklad 2.34 (Vyhledávání v B-stromu).

Na obrázku 2.16 vidíme B-strom řádu 2 s výškou 1. Minimální počet položek uzlu je 2, maximální 4. Počet dětí je roven počtu položek + 1. Pokusme se nyní najít položku 58. V kořenovém uzlu nejprve testujeme zda je $58 \leq 12$. Jelikož $58 > 12$, pokračujeme dále ve hledání korektního dítěte uzlu. V druhém kroku jsme nalezli platnou položku na pořadí i=1, platí totiž $12 \leq 58 \leq 72$. Načteme tedy druhé dítě kořenového uzlu. Listový uzel sekvenčně prohledáme a zjistíme, že obsahuje položku 58.



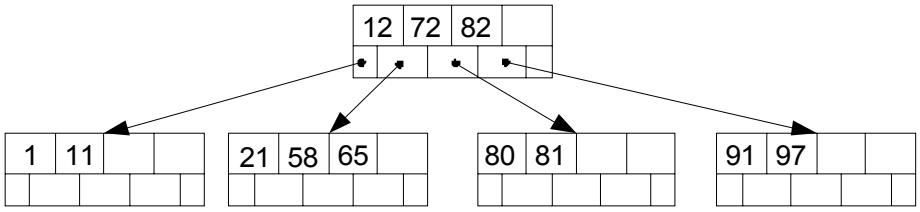
Obr. 2.16 B-strom řádu 2

Při *vkládání* položky *k* do stromu, nejprve algoritmem pro vyhledání nalezneme listový uzel, do kterého má být položka vložena, mohou nastat tyto situace:

1. Počet položek < *2c*, pak položku zatřídíme do uzlu.
2. Počet položek = *2c*. Položku zatřídíme, vyjmeme prostřední prvek k_i, horní polovinu prvků přesuneme do nového uzlu. Tomuto procesu říkáme *štěpení uzlu*. Rodiče uzlu nastavíme jako uzel aktuální. V nelistovém uzlu mohou opět nastat dvě možnosti:
    1. Počet položek < *2c*, pak položku k_i zatřídíme do uzlu. Kromě položky je nutné zatřídit i ukazatel na nově vzniklý uzel. Algoritmus končí.
    2. Počet položek = *2c*. Položku zatřídíme, vyjmeme prostřední prvek k_i, horní polovinu prvků přesuneme do nového uzlu. Pokud má uzel rodiče nastavíme jej jako uzel aktuální. Pokud je uzel kořenem, dojde k vytvoření nového kořene, který obsahuje jednu položku k_i, ukazatel na původní kořenový uzel a ukazatel na nový uzel. Toto je jediná situace, kdy může dojít ke změně výšky stromu. Jelikož se změní hloubka všech listů, strom zůstává stále vyvážený.

Příklad 2.35 (Vkládání do B-stromu).

Na obrázku 2.17 vidíme B-strom po vložení položky 81 do B-stromu z obrázku 2.16. Nejprve vyhledáme listový uzel do kterého má být položka vložena. Tente uzel obahuje položky 80, 82, 91 a 97. Jelikož uzel obsahuje *2c* položek, nemůžeme do něj vložit již žádnou položku. Prostřední prvek 82 je vyjmut a je vytvořen nový uzel obsahující horní polovinu položek původního uzlu (položky 91 a 97). Položka 82 a ukazatel na nový uzel jsou propagovány k rodičovskému uzlu do kterého jsou zatřízeny. Počet položek rodiče je < 2c, algoritmus tedy končí. Obecně může dojít k propagaci k dalším předkům (pokud existují) až ke kořeni, který může být rozštěpen a nahrazen novým kořenovým uzlem.



Obr. 2.17 B-strom z obrázku 2.16 po vložení položky 81

Při *rušení* nejprve položku nalezneme algoritmem pro vyhledání. Mohou nastat dvě situace:

1. Položka se nachází v listovém uzlu, pak je položka smazána.
2. Položka se nachází ve vnitřním uzlu. Mazanou položku musíme nahradit nejbližší menší nebo větší položkou ze stromu. Je zřejmé, že takové položky nalezneme úplně vlevo resp. úplně vpravo v podstromu.

Pokud je počet položek uzlu < c, pak se položky stránky pokusíme vložit do některé ze dvou sousedních stránek. Výsledkem nutně nemusí být smazání původní stránky, ale přenos položek ze sousedního uzlu do uzlu aktuálního. Tzv. *slučování stránek* může být propagováno až do kořenového uzlu, který může být v extrémním případě vyprázdněn a zrušen. Pouze v takovém případě může dojít ke snížení výšky stromu.

Obecně tyto indexovací struktury obsahují složitější informace. Položka může obsahovat hodnoty několika atributů, z nichž pouze jeden je indexován. Při modifikaci položky pak většinou vyhledáme dle indexovaného atributu a modifikujeme hodnoty neindexovaných atributů.

2.5.5 Vícerozměrné datové struktury

Vezměme příklad enitního typu `Student(login, jmeno, prijmeni, rocnik, rokNarozeni)`. Atributy podle kterých budeme chtít vyhledávat budou čtyři: `login`, `prijmeni`, `rocnik` a `rokNarozeni`. Pokud bychom pro implementaci použili B-strom, museli bychom vytvořit čtyři stromy z nichž každý by indexoval jeden atribut. Ve chvíli kdy použijeme dotaz:

```
SELECT * FROM Student WHERE rocnik=4 AND rokNarozeni=1983
```

Musíme v B-stromu indexujícím atribut `rocnik` nalézt záznamy s klíčem rovným hodnotě `4` a v B-stromu indexujícím `rokNarozeni` záznam s klíčem rovným hodnotě `1983`. Takto získané mezivýsledky pak musíme spojit. Výsledkem jsou záznamy, které odpovídají definované podmínce. Mezivýsledky mohou mít daleko větší než velikost celkového výsledku. Je zřejmé, že takovýto algoritmus není efektivní.

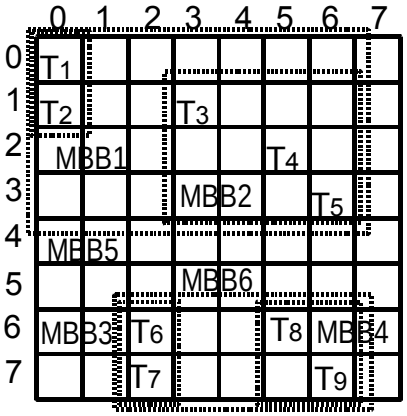
Vícerozměrné datové struktury [BBK00, 33], které jsou vyvíjeny a publikovány od 2. poloviny 80. let, se snaží tento problém řešit. Tyto datové struktury jsou ovšem univerzálnější a jejich použít není zúženo pouze na výše zmíněný problém. Obecně, vícerozměrné datové struktury slouží pro indexování vícerozměrných prostorů. Struktury dělíme na *metrické* a *vektorové*. Metrické datové struktury jsou obecnější, a v současnosti jsou aplikovány na některé speciální problémy, nebudeme se jimi proto v dalším textu zabývat. Hlavním představitelem těchto struktur jsou *M-stromy* [8]. V dalším textu budeme popisovat nejznámější vícerozměrnou datovou strukturu - *R-strom*.

2.5.5.1 R-strom

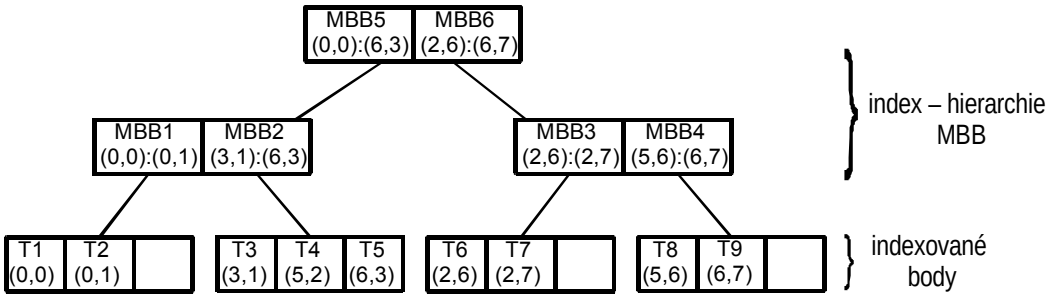
Jednou z prvních vyvinutých vícerozměrných datových struktur je *R-strom*, který byl publikován v roce 1984 [15]. R-strom je založen na shlukování v prostoru blízkých bodů na stejné stránky. V případě R-stromů jsou pro body konstruovány tzv. *minimální ohraničující obdélníky* (angl. *MBB – minimal bounding box*), které shlukují podobné, rozuměj blízké, body. Body jednoho MBB jsou uloženy v listovém uzlu. Vnitřní uzly obsahující hierarchii MBB, tedy jakési nad-MBB.

Příklad 2.36 (R-strom).

Na obrázku 2.18(a) vidíme dvou-rozměrný prostor velikosti 8×8 s osmi body. Na obrázku 2.18(b) vidíme R-strom indexující tento prostor. Listový uzel, který je nakreslen nejvíce vlevo, obsahuje body (0,0) a (0,1). Minimální ohraničující obdélník těchto bodů *MBB1* je obdélník definovaný body QB_l=(0,0) a QB_h=(0,1), v dalším textu budeme zapisovat zkrácené (0,0):(0,1). *MBB5* je minimálním ohraničujícím obdélníkem obsahujícím *MBB1* a *MBB2*.



(a)



(b)

Obr. 2.18 (a) Indexovaný dvourozměrný prostor. (b) R-strom indexující tento prostor

Ve vektorových datových strukturách definujeme několik typů dotazů. Mezi nejpoužívanější patří bodový a rozsahový dotaz. *Bodový dotaz* je definován bodem a vrací `true`, pokud se bod v datové struktuře nachází, v opačném případě vrací `false`. Rozsahový dotaz vrací všechny body indexovaného prostoru, které se nachází v definovaném hyperkvádru, který bývá často nazýván *dotazovací obdélník* (angl. *query box*). Bodový dotaz je speciálním případem dotazu rozsahového, kde oba body definující dotazovací obdélník splývají. Pomocí SQL bychom rozsahový dotaz zapsali takto:

```
SELECT * FROM <table_name> WHERE
```

$$QB_{l1} \leq a_1 \leq QB_{h1} \text{ AND } QB_{l2} \leq a_2 \leq QB_{h2} \text{ AND } \ldots \text{ AND } QB_{ln} \leq a_n \leq QB_{hn}$$

Kde a_i je atribut náležející k *i*-té souřadnici indexovaného prostoru. Je tedy zřejmé, že dotaz na více zaindexovaných atributů, který byl uveden na začátku kapitoly 2.5.5, bychom řešili právě rozsahovým dotazem.

Nyní si popišme algoritmus *rozsahového dotazu*. Nejprve nastavíme kořenový uzel jako aktuální a hledáme zda obsahuje MBB, který protíná dotazovací obdélník QB_l: QB_h. Pokud takový MBB nalezneme, načteme dítě náležející k tomuto MBB a nastavíme je jako aktuální. Znovu hledáme MBB, který protíná dotazovací obdélník. Takto postupujeme až k listovému uzlu, ve kterém kontrolujeme zda neobsahuje body ležící v dotazovacím obdélníku. Pokud ano, zařadíme je do výsledku dotazu. Nyní se vracíme zpět (aktuální cesta je ukládána na zásobník) a pokračujeme v prohledávání nezkontrolovaných částí stromu.

Algoritmus *vkládání* a *rušení* bodu je poměrně komplikovaný a existují různé varianty R-stromů, které se různým způsobem snaží tyto algoritmy řešit. V základní variantě algoritmu, hledáme MBB, který pokrývá i nově vložený bod. Pokud takový nenajdeme, hledáme MBB, který po vložení bodu nejméně zvětší svůj objem. Pokud by po vložení bodu byla překročena kapacita uzlu, je uzel rozštěpen. Způsob štěpení závisí na zvolené variantě R-stromů. Je zřejmé, že změna v minimálním ohraničujícím obdélníku se musí propagovat směrem ke kořenovému uzlu. Z velkého množství variant R-stromů jmenujme například R*-strom [5].

Seznam použité literatury:

[1] AHO, A. V., ULLMAN, J. D. , HOPCROFT, J. E. . *Data Structures and Algorithms*. 1st edition: Addison-Wesley Series in Computer Science, 1983. 253 s. ISBN 0201000237.

[2] APACHE.ORG. *Jakarta Struts* [online]. 2003 [cit. 2006-05-31]. Dostupný z WWW: <http://jakarta.apache.org/struts>.

[3] ANSI. *SQL:99*. ANSI x3.135-1999 Standard, 1999.

[4] BAYER, R. , MCCREIGHT, Edward M. . Organization and Maintenance of Large Ordered Indices. *Acta Informatica*. 1972, vol. 1, s. 173-189.

[5] BECKMANN, N. , et al. The R*-tree: An efficient and robust access method for points and rectangles. In *1990 ACM SIGMOD International Conference on Management of Data*. 1st edition. USA : ACM Press, 1990. s. 322-331.

[6] BENEŠ, M., KRÁTKÝ, M. . *Sylaby předmětu Tvorba informačních systémů* [online]. 2005 [cit. 2006-05-31]. Dostupný z WWW: < http:// www.cs.vsb.cz/kratky/courses/2005-06/ tis/>.

[7] BÖHM, C. , BERCHTOLD, S., KRIEGEL, H., MICHEL, U. . Multidimensional Index Structures in Relational Databases. *Journal of Intelligent Information Systems (JIIS)*. 2000, vol. 15, no. 1, s. 51-70.

[8] CIACCIA, P. , PATELLA, M. , ZEZULA, P. . An Efficient Access Method for Similarity Search in Metric Spaces. In *23rd International Conference on Very Large Databases (VLDB'97)*. 1st edition. Athens, Greece : Morgan Kaufmann, 1997. s. 426-435.

[9] CODD, E.F. . A relational model of data for large shared data banks. *Communications of the ACM*. 1970, vol. 13, no. 6, s. 377-387.

[10] CHEN, P.P. . The Entity-Relationship Model: Toward a Unified View of Data. *ACM on Database Systems*. 1976, vol. 1, no. 1.

[11] DATABASED INTELIGENCE . *dBase* [online]. 1997 [cit. 2006-05-31]. Dostupný z WWW: <http://www.dbase.com/>.

[12] DIETRICH, S.W. , URBAN, S.D. . *An Advanced Course in Database Systems*. 1st edition: Prentice Hall, 2005. 196 s. ISBN 0-13-042898-1.

[13] DVORSKÝ, J. , OCHODKOVÁ, E. , ĎURÁKOVÁ, D. . *Základy algoritmizace* [online]. 2004 [cit. 2006-05-31]. Dostupný z WWW: <http://www.cs.vsb.cz/dvorsky/Download/ ZakladyAlgoritmizace.zip>.

[14] GARCIA-MOLINA, H. , ULLMAN, J.D. , WIDOM J. . *Database systems: the complete book*. 1stedition: Prentice Hall, 2002, 1117s. ISBN 0−13−031995−3.

[15] GUTTMAN, A. . A Dynamic Index Structure for Spatial Searching. In *1984 ACM SIGMOD International Conference on Management of Data*. 1st edition. Boston, USA : ACM Press, 1984. s. 45-57.

[16] IBM. *DB2 Database* [online]. 2002 [cit. 2006-05-31]. Dostupný z WWW: <http://www.ibm.com/db2>.

[17] MANOLOPOULOS, Y. , THEODORIDIS, Y. , TSOTRAS V.J. . *Advanced Database Indexing (Advances in Database Systems)*. 1st edition: Springer, 1999, ISBN 0792377168.

[18] MICROSOFT. .*NET Framework* [online]. 2002 [cit. 2006-05-31]. Dostupný z WWW: <http://msdn.microsoft.com/netframework/>.

[19] MICROSOFT. *Microsoft SQL Server 2005* [online]. 2002 [cit. 2006-05-31]. Dostupný z WWW: < http://www.microsoft.com/sql/>.

[20] MYSQL AB. *MySQL – The Open Source Databáze* [online]. 2001 [cit. 2006-05-31]. Dostupný z WWW: <http://www.mysql.com/>.

[21] OBJECT MANAGEMENT GROUP (OMG). *Unified Modelling Language (UML) 2.0* [online]. 2005 [cit. 2006-05-31]. Dostupný z WWW: <http://www.uml.org/>.

[22] ORACLE. *Oracle Databáze 10g* [online]. 2004 [cit. 2006-05-31]. Dostupný z WWW: <http://www.oracle.com/database>.

[23] POKORNÝ, J. . *Dotazovací jazyky*. 1. vydání: Nakladatelství Univerzity Karlovy. 2002, ISBN 80-246-0497-3.

[24] ISO. *SQL:92*. ISO/IEC 9075:1992 Standard, 1992.

[25] ISO. *SQL:99*. ISO/IEC 9075-1:1999 Standard, 1999.

[26] SUN. *Java 2 Enterprise Edition* [online]. 2002 [cit. 2006-05-31]. Dostupný z WWW: <http://java.sun.com/javaee>.

[27] W3 CONSORTIUM. *eXtensible Markup Language (XML)* [online]. 2002 [cit. 2006-05-31]. Dostupný z WWW: <http://www.w3c.org/>.

[28] W3 Consortium. *XML Schema, W3C Recommendation* [online]. 2001 [cit. 2006-05-31]. Dostupný z WWW: <http://www.w3.org/ XML/Schema>.

[29] W3 Consortium. *XML Schema Tutorial* [online]. 2004 [cit. 2006-05-31]. Dostupný z WWW: <http://www.w3schools.com/schema/>.

[30] W3 Consortium. *XML Path Language (XPath) Version 2.0* [online]. 2002 [cit. 2006-05-31]. Dostupný z WWW: <http://www.w3c.org/>.

[31] W3 Consortium. *XQuery 1.0: An XML Query Language* [online]. 2003 [cit. 2006-05-31]. Dostupný z WWW: <http://www.w3c.org/>.

[32] WIRTH, N. *Algorithms + Data Structures = Programs*. 1st edition: PrenticeHall. 1976.

[33] YU, C. . *High-Dimensional Indexing*. 1st edition : Lecture Notes in Computer Science, Springer-Verlag. 2002.