Databázové systémy

Michal Krátký, Radim Bača

Katedra informatiky, Fakulta elektrotechniky a informatiky
VŠB – Technická univerzita Ostrava
Česká republika

Poslední změny: 11. února 2015

Databázové systémy

Michal Krátký

michal.kratky@vsb.cz
http://www.cs.vsb.cz/kratky/

Radim Bača

radim.baca@vsb.cz
http://www.cs.vsb.cz/baca/

Database Research Group
http://db.cs.vsb.cz/
http://dbedu.cs.vsb.cz/

Katedra informatiky
Fakulta elektrotechniky a informatiky
VŠB – Technická univerzita Ostrava
17. listopadu 15, 708 33 Ostrava–Poruba
Česká republika

http://www.cs.vsb.cz/
http://fei.vsb.cz/
http://www.vsb.cz/

Typeset by PDFLaTeX

Obsah

Úvod

Kniha kterou jste právě začali číst vznikla jako jeden z informačních pramenů pro studenty předmětů zabývajících se databázovými technologiemi. Primárně je text určen studentům bakalářského a magisterského studijního programu Katedry informatiky, Fakulty elektrotechniky a informatiky, VŠB – Technické univerzity Ostrava. Bakalářský studijní program zahrnuje dva povinné předměty a dva předměty volitelné, magisterský program pak zahrnuje tři předměty z oblasti databázových systémů.

Hlavním tématem předmětu **Úvod do databázových systémů** (3. semestr, 2 h přednášek, 2 h počítačových cvičení a 2 h práce na projektu) je relační datový model a SQL, jakožto základní kameny aktuálně nasazených databázových systémů. K tomuto kurzu jsou relevantní kapitoly 1, 2, 3 a 4.

Předmět **Databázové a informační systémy** (4. semestr, 2/3/2) se zabývá především procedurálními nadstavbami nad SQL a transakčním zpracováním. Jelikož cílem předmětu je také návrh a implementace jednoduchého informačního systému, témata kurzu dále zahrnují funkční analýzu a návrh a implementaci datové vrstvy informačního systému. K tomuto kurzu jsou relevantní kapitoly 15.

Každý vývojář datové vrstvy informačního systému by měl mít jasné povědomí o efektivitě dotazu zaslaného na databázi, bez těchto znalostí může jen stěží ladit výkon datové vrstvy. Z těchto důvodů jsou v knize zařazeny kapitoly týkající se fyzického návrhu databáze (kapitola 10) a vykonávání dotazů (kapitola 7). Tato témata jsou součástí volitelného bakalářského předmětu **Databázové systémy** (5. semestr, 2/2/2) a magisterského předmětu **Databázové a informační systémy 2** (1. semestr, 2/2/2).

Hlavním tématem volitelného předmětu **Administrace databázových systémů** (6. semestr, 2/2/0) je (jak název napovídá) administrace databázových systémů. Tento kurz obsahuje jak obecné rysy administrace, tak konkrétní vlastnosti da-

tabázových systémů Oracle a SQL server. K tomuto kurzu je relevantní kapitola 20.

Přestože je kniha určena především studentům univerzitních studijních programů, snahou autorů bylo vytvořit knihu atraktivní i pro odborníky z praxe, pro které jsou databázové systémy každodenní náplní pracovního dne. Kniha ukazuje jako obecné principy, tak konkrétní rysy databázových systémů jako jsou Oracle, SQL Server, DB/2 a MySQL.

V Ostravě 11. února 2015

Michal Krátký, Radim Bača

Část I

Úvod do databázových technologií

Kapitola 1

Úvod do databázových technologií

Obsah

Cíl kapitoly:
V této kapitole bude podán úvod do databázových a informačních systémů především z pohledu historického vývoje od 50. let 20 století do dnešní doby a dále se věnujeme motivaci pro hlubší studium databázových systémů.

1.1 Úvod

Ještě před vývojem výpočetní techniky v její dnešní podobě, vznikla potřeba uložení a dotazování velkého množství dat. Bylo tedy nutné řešit následující problémy. Kam data uložit? V jaké podobě data uložit? Jakým způsobem uložená data získat?

1.2 Historický vývoj

Před vznikem databázové technologie [16, 37, 11] v 70.–80. letech 20. století lidé chtěli ukládat velké množství dat, pro jejich pozdější dotazování a případné analýzy. Příkladem může být zpracování výsledků voleb, zpracovávání mezd ve firmách, vyhledávání informací v knihovnách apod. S rozvojem výpočetní techniky v 50.–70. letech 20. století začaly vznikat programy, které se různým způsobem snažily řešit uložení dat. Typickým rysem těchto programů bylo dávkové zpracování dat, každá funkce byla řešena pomocí samostatného programu tzv. **agendy**. Proto tento princip zpracování dat nazýváme **agendové zpracování dat**. Jednotlivé programy řešily vlastní uložení a dotazování dat. Bylo tedy nemožné dotazovat se uložené databáze bez znalosti datových strukturu úložiště. Dalším problémem byl neexistence dotazovacího jazyka vyšší úrovně.

V 70. letech 20. století začaly vznikat první datové modely specifikující, jakým způsobem budou data modelována a dotazována. Z těchto raných modelů uveďme **síťový** popř. **hierarchický** datový model. U těchto modelů ovšem postrádáme dotazovací jazyk vyšší úrovně. Např. jazyk **CODASYL** obsahoval příkazy, které uživateli umožňovali procházet datovými prvky pomocí ukazatelů mezi nimi.

Třicetiletý vývoj teorie i komerčních databázových aplikací umožnil koncipovat unifikovaný pohled na pojmy, nástroje a metody, které lze nazvat **databázovou technologií**. Základním pojmem je **databáze**, což je strukturovaná množina dat. Aplikace, která umožňuje databázi definovat, konstruovat a manipulovat s databází se nazývá **systém řízení báze dat** (**SŘBD**, angl. **DBMS**). SŘBD spolu s databází tvoří **databázový systém** (**DBS**).

V roce 1970 publikoval Edgar F. Codd článek **A Relational Model of Data for Large Shared Data Banks** [9] uvádějící **relační datový model**. Tento model byl založen na jednoduché myšlence: data jsou modelována pomocí relací, jako relace jsou ukládána, jako relace jsou dotazována. Byl vyvinut jazyk vysoké úrovně SQL, který umožňoval pracovat s relacemi. Navíc byl vyvinut fyzický model umožňující efektivní implementaci relačního datového modelu. V roce 1972 publikovali Rudolf Bayer a E.M. McCreight článek **Organization and Maintenance of Large Ordered Indices** [4] uvádějící datovou strukturu **B-strom**, která se pro fyzickou implementaci relačních SŘBD používá dodnes. Přes vznik nových datových modelů (např. objektového, objektově-relačního), byly za dobu vývoje relační SŘBD (**RSŘBD**, angl. **RDBMS**) natolik zdokonaleny, že přes některé nevýhody, jsou dodnes používány v drtivém množství nasazení SŘBD.

1.3 Proč SŘBD?

Z určitého pohledu by se mohlo zdát, že všechny SŘBD jsou prostě jen určitý druh aplikace, která nám jen trochu usnadňuje psaní informačních systému a dalších aplikací, které potřebují uložit data. Proč se tedy nestačí naučit příkazy pro vložení do databáze a pro čtení z databáze a zbytek nechat na SŘBD? Nebo proč vlastně si data neuložit někde do souboru a SŘBD tedy tím pádem vynechat? Pokusíme se důvody pro nasazení SŘBD vysvětlit na jeho vlastnostech, jenž jsou klíčové pro většinu uživatelů ať už jde o bankovní instituce, nadnárodní korporace, státní firmy, nebo malé firmy zabývající se kupříkladu pouze expedicí zboží, či internetovým obchodem.

Vlastnosti SŘBD:

- Masivnost - SŘBD jsou schopny pracovat s neporovnatelně větším objemem dat než se vleze do hlavní paměti počítače.
- Perzistence - data zůstávají uchovány v databázi za každých okolností (vyjma armagedonu).
- Bezpečnost - v databázi mohou běžet operace složené z mnoha neoddělitelných kroků, kde při náhlém přerušení uprostřed operace by databáze mohla být v nekorektním stavu, jelikož některé kroky operace nebyly provedeny. SŘBD zajišťuje stornování kroků takovéto nedokončené operace a opětovné uvedení databáze do korektního stavu.
- Více-uživatelskost - SŘBD je schopna paralelně zpracovávat velké množství požadavků od uživatelů, přičemž zajišťuje korektnost provedení tím, že předchází nebo řeší případné konflikty.
- Pohodlnost - existuje model, který umožňuje oddělit způsob uložení dat a přístup k datům. Uživatel databáze se tedy příliš nestará o to v jakých datových strukturách jsou jeho data uložena, ale zejména o to jaká je logická struktura dat.
- Efektivnost - SŘBD je schopna provést tisíce operací za sekundu
- Spolehlivost - SŘBD je schopna se velmi rychle zotavit z výpadku OS nebo celého hardware, přičemž samotné SŘBD by nikdy nemělo bezdůvodně selhat. Spolehlivost úzce souvisí s dostupností databáze, což je dnes velmi aktuální téma pro mnoho internetových služeb, které si nemohou dovolit prakticky žádné výpadky. Dostupnost je pak řešena distribucí databáze na více strojů.

Všechny tyto vlastnosti nicméně nepřicházejí samy pouhým nainstalováním vybraného SŘBD a vytvořením databáze. SŘBD jsou komplexní nástroje a pro plné využití jejich vlastností je nutné je správně používat. Pro jeden problém dnešní SŘBD nabízejí celou řadu možností jak jej řešit. Každá varianta má své výhody a svá úskalí. Mnoho uživatelů SŘBD se často spokojí s implicitním nastavením SŘBD, aniž by rozuměli těm úskalím, kterém s tím souvisí. Tato kniha se věnuje hlubšímu pochopení principů SŘBD, které Vám umožní právě ono správné použití možností jednotlivých SŘBD.

1.4 Pro koho je kniha určena

SŘBD jsou natolik komplexní nástroje, že existuje hned několik různých pracovních rolí, které přicházejí se SŘBD do styku. Tyto role se od sebe mohou výrazně lišit proto je podrobněji přestavíme.

- Programátor SŘBD - často velké týmy lidí, které se zabývají přímo implementací nějakého SŘBD. I když se to nemusí zdát pravděpodobné, nové SŘBD vznikají prakticky neustále. Je to dáno zejména novými požadavky souvisejícími s objemy dat, dostupností, či rychlostí odezvy. Další častou motivací pro rozšíření či vznik nového SŘBD jsou nové typy hardware.
- Návrhář databáze - jde o osobu, která stanoví logickou strukturu databáze (schéma databáze). Návrh databáze se většinou provádí v rámci návrhu celého softwarového díla do kterého databáze patří.
- Správce databázových systémů - administrátor SŘBD. Jde o osobu, která se stará o bezproblémový běh celého SŘBD. To v sobě zahrnuje spoustu dílčích úkolů jako například přidělování přístupových práv k datům, rekonstrukce databáze v případě jejího poškození a správné nastavení SŘBD a databáze pro efektivní a bezchybový běh.
- Aplikační programátor - programuje aplikační úlohy nad daty. Jde tedy typicy o programátora, který vyvíjí softwarové dílo, jenž databázi využívá a komunikuje s ní přes stanovené rozhraní (nejčastěji SQL).

V této knize se věnujeme zejména oblastem, které souvisí se správným návrhem databáze a jejím správným využitím. Tento obsah je důležitý zejména pro návrháře databáze a aplikační programátory, kteří představují zdaleka největší počet lidí, kteří se SŘBD pracují. Nicméně v knize je možné nalézt i několik kapitol, jenž by se daly přiřadit spíše k roli programátora SŘBD nebo správce databázových

systémů. Je to dáno i tím, že jednotlivé role nemají ostré hranice a například hlubší znalost SŘBD z pohledu administrátora může být mnohdy užitečná i pro aplikačního programátora a naopak.

1.5 Struktura knihy

Nyní popišme strukturu textu. V kapitole 2 se budeme zabývat dvěma nejvýznamnějšími konceptuálními modely: ERD a UML. Kapitola 3 popisuje nejdůležitější datový model – datový model relační. Dotazovací jazyky pro relační datový model jsou uvedeny v kapitole 4. V kapitole 6 jsou popsány některé moderní datové modely (objektově-relační a datový model XML). Fyzická implementace databází je prezentována v kapitole 10. Kapitola 7 se zabývá vykonáváním dotazů. Přirozeným požadavkem kladeným na ŠRBD je víceuživatelský přístup k databázi. Kapitola 15 popisuje problémy a jejich řešení vzniklé při paralelním přístupu k datům. Specifikace a implementace datové vrstvy informačního systému jsou popsány v kapitole 16. Existující databázové aplikace jako prostorové databáze a datové sklady jsou uvedeny v kapitole 17. Kapitola 20 obsahuje vybrané partie administrace databázových systémů.

Kapitola 2

Konceptuální modelování

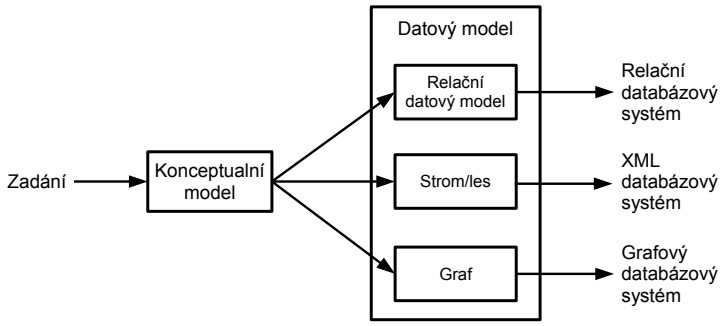
Obsah

Cíl kapitoly:
V této kapitole bude vysvětlen pojem konceptuálního modelování jako procesu sémantického popisu budované databázové aplikace.

2.1 Vývoj informačních systémů

Informační systémy (dále jen IS) jsou speciálním typem softwarového díla. Při jejich vývoji tedy uplatňujeme postupy, které se pro vývoj softwarového díla

doporučují. V této knize se zaměříme pouze na tu část vývoje IS, která se týká databáze.



Obrázek 2.1: Kroky při vytváření databáze

Na obrázku 2.1 můžeme vidět typické kroky, které se provádějí při vývoji databáze. Na začátku tedy máme zadání, které je výchozím bodem pro vytvoření konceptuálního modelu. Konceptuální model představuje reprezentaci zadání, které by ještě mohl zadavatel rozumět a která je mnohem přesnější než slovní popis. V této kapitole se budeme podrobněji zabývat právě reprezentací konceptuálním modelem. Z konceptuálního modelu následně vytváříme datový model pro specifické SŘBD. Jak je vidět na obrázku, existuje celá řada různých datových modelů. Mezi nejpopulárnější dnes nicméně stále patří relační datový model. Relačnímu datovému modelu je z těchto důvodů věnována podstatná část této knihy.

2.2 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 [31] nebo MS SQL Server [26] a také bez ohledu na výsledný datový model (relační, stromový nebo grafový).

S konceptuálním modelem souvisí celá řada základních pojmů:

- Entita

- Entitní typ
- Atribut
- Vztah
- Vztah s informací

V následujícím odstavci si tyto základní pojmy vysvětlíme. Výchozím prvkem je **entita**, čímž rozumíme objekt z reálného světa jehož vlastnosti (**hodnoty atributů**) chceme evidovat v databázi. Příkladem může být entita konkrétní osoby jako třeba Pavel Novák narozený 1990 v Ostravě. Obvykle máme více entit stejného typu. Pak mluvíme o tzv. **entitním typu**. Entitní typ je definován svým jménem a množinou **atributů**, které jej popisují. Dříve uvedené entitě Pavel Novák tedy odpovídá entitní typ `Osoba` kde mezi jeho atributy patří rodné jméno, příjmení, rok a místo narození. Je zejména důležité pochopit, že entita je konkrétní výskyt (instance) entitního typu s konkrétními hodnotami atributů. Mezi atributy entitního typu rozlišujeme tzv. **klíč**, což je množina atributů, dle jejichž hodnot je entita odlišitelná od ostatních entit stejného typu.

Dále můžeme modelovat **vztah** (nebo-li **vazbu**) mezi n entitními typy a pak mluvíme o n-árním vztahu. Vztah je definován názvem a množinou entitních typů, které jsou v příslušném vztahu. Nejčastější jsou vztahy binární a tedy mezi dvěma entitními typy. V dalším textu se zaměříme pouze na binární vztahy. Mezi neintuitivnější vlastnost vztahu patří **kardinalita** (taktéž označovaná za **násobnost**):

1. **1 : M** (angl. **one-to-many**) – vztah, kde entita prvního typu má vztah s **M** entitami druhého typu a kde entita druhého typu má vztah s jednou entitou prvního typu.
2. **M : N** (angl. **many-to-many**) – vztah, kde entita prvního typu má vztah na **M** entitami druhého typu a kde entita druhého typu má vztah s **N** entitami prvního typu.

Další důležitou vlastností vztahu je **povinnost v členství**:

1. Typ není povinen členstvím ve vztahu – existuje entita, která nemá vztah s jinou entitou.
2. Typ je povinen členstvím ve vztahu – každá entita musí mít vztah alespoň s jednou entitou.

Existuje také speciální typ **vztahu s informací**. Takový vztah kromě entitních typů obsahuje také další doplňující atributy.

Příklad 2.1. (Vztah s informací). Mějme vztah `ÚČASTNÍ_SE` mezi entitními typy `Osoba` a `Závod`. Vztah má kardinalitu M:N a chceme uchovat informaci o umístění osoby v daném závodě. Vztah `ÚČASTNÍ_SE` tedy bude obsahovat kromě entitních typů ještě atribut `umístění`.

2.2.1 Lineární zápis konceptuálního modelu

Pro textový popis konceptuálního modelu se často využívá tzv. lineární zápis. S pomocí lineárního zápisu zachycujeme entitní typy a vztahy mezi nimi. Informace, které se obvykle vyskytují v lineárním zápisu jsou:

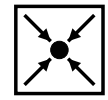
- Entitní typ:
  `EntitníTyp(seznam názvů atributů)`
- Atributy, které tvoří klíč entitního typu jsou podtrženy
- Vztah:
  `JMENO_VZTAHU(EntitníTyp`$_1$`, ..., EntitníTyp`$_n$`)`
- Vztah s informací:
  `JMENO_VZTAHU(EntitníTyp`$_1$`, ..., EntitníTyp`$_n$`, atributy)`
- Vztah s vyznačením kardinality a povinnosti:
  `VZTAH(EntitníTyp`$_1$`:(min,max), EntitníTyp`$_2$`:(min,max))`

V lineárním zápisu se snažíme o dodržení jednoduché konvence: názvy entitních typů začínají velkým písmenem, názvy atributů malým písmenem a vztahy jsou psány velkými písmeny. V lineárním zápisu vztahu je možné vyznačit kardinalitu a povinost pomocí dvojice `(min, max)` uvedené za entitním typem. Tato dvojice se může použít i u grafického vyjádření a proto uvedeme několik typických hodnot spolu s vysvětlením:

- (0,1) - nemusí být ve vztahu, nebo je v jednom vztahu (nepovinost, kardinalita 1)
- (1,1) - musí být právě v jednom vztahu (povinost, kardinalita 1)
- (0,N) - nemusí být ve vztahu, nebo je v jednom či více vztahů (nepovinost, kardinalita N)

- (1,N) - musí být alespoň v jednom nebo více vztahů (povinost, kardinalita N)

Příklad 2.2. (Entitní typy, entity a vztahy) Mějme entitní typy `Student` resp. `Ucitel` s atributy `login`, `jmeno`, `prijmeni`, `rocnik` resp. `login`, `jmeno`, `prijmeni` a `uvazek`. 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 další entitní typ `Kurz(kod, nazev, kapacita)`. Identifikujeme následující vztahy:

- `STUDUJE(Student, Kurz)`
- `GARANTUJE(Ucitel, Kurz)`

Pokud navíc chceme vyznačit kardinalitu a povinnost pak lineární zápis bude vypadat takto:

- `STUDUJE(Student:(0,N), Kurz:(0,N))`
- `GARANTUJE(Ucitel:(0,N), Kurz:(1,1))`

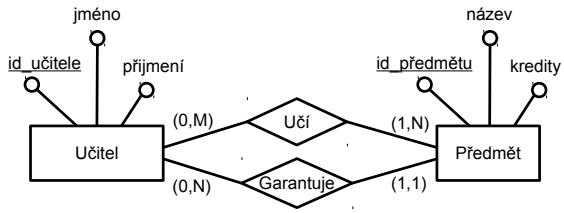
Tedy většina entitních typů nemá ve vztahu povinnost a jejich kardinalita je N. Pouze u entitního typu `Kurz` vyznačujeme, že kurz musí mít právě jednoho garanta.

2.3 Entity Relationship (ER) diagram

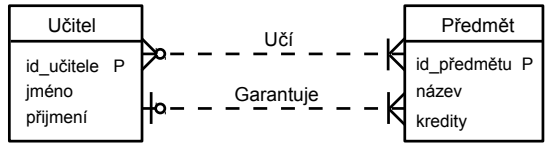
Existuje vícero různých způsobů grafického vyjádření konceptuálního modelu. Jedním z prvních je tzv. **Entity Relationship** (**ER**) publikovaným P. Chenem v roce 1976 [7]. V další letech byl tento model rozšířen (např. o dědičnost) a bylo uvedeno několik variant, které používají výrazně odlišnou notaci.

ER graficky modeluje entitní typy a vztahy databáze. Diagram označujeme **ER diagram** (**ERD**). Pro ER diagram a pro grafické znázornění konceptuálního modelu obecně neexistuje žádný standard a dokonce ani nějaký jeden široce používaný způsob notace. Z těchto důvodů představíme více notací a je velmi pravděpodobné, že se při modelování databáze setkáte s nějakou kombinací těchto notací. Základní přehled jednotlivých notací a jejich srovnání pak je možné nalézt zde [41]. Mezi známější notace patří:

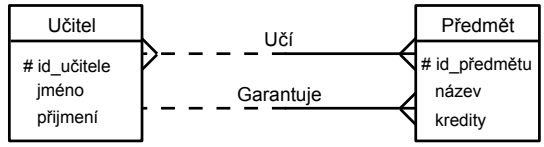
- Chenova notace [7]
- Min/Max
- Crow's foot notace (Information Engineering) [25]
  - Oracle data modeler
  - Toad data modeler
- Bachmanova notace



Obrázek 2.2: Chenova notace



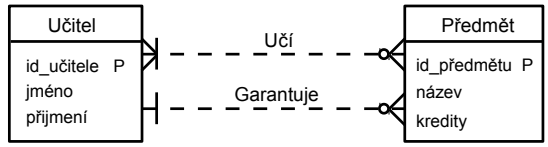
Obrázek 2.3: Crow's foot notace



Obrázek 2.4: Crow's foot notace - Oracle

Příklad 2.3. (ERD pro různé notace)

Na obrázcích 2.2 – 2.5 můžeme vidět výše zmíněné notace pro vztah UČÍ (M:N) a vztah GARANTUJE (1:N) mezi dvěma entitními typy `Učitel` a `Předmět`.



Obrázek 2.5: Crow's foot notace - Toad

Oba entitní typy mají celočíselný klíč (`id_učitele` a `id_předmetu`), přičemž platí že Předmět musí mít alespoň jednoho učitele a učitel nemusí ani nic učit ani nic garantovat. Na tomto příkladu můžeme vidět podobnosti i rozdíly jednotlivých notací. V dalších podkapitolách si jednotlivé rozdíly podrobněji rozebereme.

Existují různé CASE modelovací nástroje (jako např. Oracle nebo Toad data modeler), které umožňují vytvářet ERD. Obvykle se tyto nástroje od sebe použitou notací výrazně liší. Dnešní CASE nástroje se liší zejména v reprezentaci kardinality a povinnosti vztahů, ale při srovnání s Chenovou notací je tam rozdílů mnohem více.

2.3.1 Kardinalita, povinnost a umístění

Jedním z rozdílů mezi ERD notacemi je způsob vyznačení kardinality. Obecně se používají tři způsoby vyznačení kardinality:

- Jedním číslem na každé straně vztahu (Chenova notace),
- dvojící (min, max) (Chenova notace),
- graficky vraní nohou (Crow's foot notace)

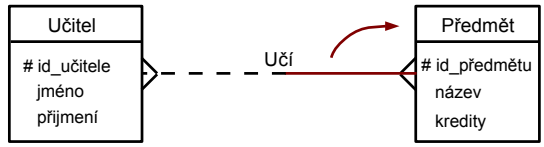
Dále se notace liší podle toho jak se vyznačuje povinnost ve vztahu:

- Informace je obsažena ve dvojici (min, max), která zároveň určuje i kardinalitu,
- Používá se nějaký grafický symbol, který povinnost určuje (povinost/nepovinost):
  - čára / prázdné kolečko

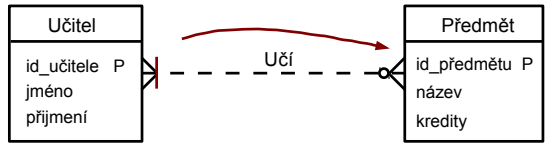
– plné kolečko / prázdné kolečko

– plná čára / čárkovaná čára

Naučení se příslušných symbolů pro notaci, kterou chceme používat, bohužel nemusí vždy stačit. Komplikací je tady umístění symbolu (kardinality nebo povinnosti) vzhledem k entitnímu typu ke kterému se váže. U obou symbolů může být symbol umístěn jak na straně entitního typu ke kterému se váže, tak na druhé straně. Toto je podrobněji demonstrováno v příkladu 2.4. Kardinalita je prakticky u všech notací na straně entitiního typu nicméně u povinnosti tomu tak není a tam je potřeba si dávat pozor.



Obrázek 2.6: Crow's foot notace - Oracle

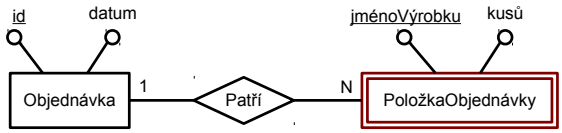


Obrázek 2.7: Crow's foot notace - Toad

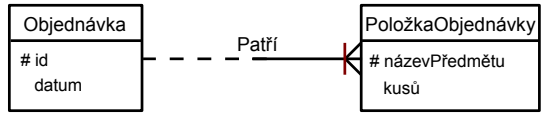
Příklad 2.4. (ERD umístění symbolů) Na obrázcích 2.6 a 2.7 můžeme vidět crow's foot notaci používanou v Oracle a Toad data modeleru pro vztah UČÍ (M:N) mezi dvěma entitními typy `Učitel` a `Předmět`. Platí že předmět musí mít alespoň jednoho učitele a učitel nemusí ani nic učit. Kardinalita je u obou notací vyznačena na straně entitního typu (např. skutečnost, že učitel může učit **mnoho** předmětů je vyznačena vraní nohou u předmětu). Na druhou stranu povinnost je u Oracle data modeleru na straně entitního typu, zatímco u Toad data modeleru je povinnost vyznačena na opačné straně. Na obou obrázcích je červeně a šipkou zvýrazněna skutečnost, že předmět **musí** mít alespoň jednoho učitele.

2.3.2 Slabé entitiní typy

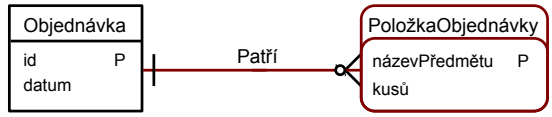
V této kapitole se věnujeme konceptu, který ovlivňuje zejména způsob jakým je tvořen primární klíč u entitního typu. V určitých případech chceme modelovat skutečnost, že určitá entita není jednoznačně definována jen svými atributy, ale i jinou entitou se kterou je vztahu. Mějme dva entitní typy `Objednávka` a `PoložkaObjednávky`, které jsou ve vztahu 1:N. `Objednávka` obsahuje atributy `id` (primární klíč) a `datum` a `PoložkaObjednávky` obsahuje atributy `jménoVýrobku` a `kusů`. Žádný z atributů v entitním typu `PoložkaObjednávky` nemůže tvořit sám osobě primární klíč, protože například stejné jméno výrobku se může objevovat v různých objednávkách. V tomto případě bude jedna položka objednávky jednoznačně identifikována nejen svými atributy, ale i objednávkou ke které patří (resp. primárním klíčem této objednávky). V tomto případě říkáme, že `PoložkaObjednávky` je **slabý entitní typ** a **vztah mezi entitními typy je identifikující**.



Obrázek 2.8: Chenova notace pro slabý entitní typ



Obrázek 2.9: Oracle crow's foot notace pro slabý entitní typ



Obrázek 2.10: Toad crow's foot notace pro slabý entitní typ

Symboly pro znázornění slabého entitního typu se opět mohou mezi jednotlivými notacemi lišit. Na obrázcích 2.8 – 2.10 můžeme vidět několik notací pro

příklad uvedený na začátku této podkapitoly. Symboly, které vyznačují slabý entitiní typ v dané notaci jsou zvýrazněny v ERD červeně.

2.3.3 Dědičnost

Dědičnost využíváme když potřebujeme modelovat situaci, ve které nějaké entitní typy sdílejí více společných rysů, takže se dá říct, že pro ně existuje nějaký předek ze kterého jsou odvozeny. Příkladem může být datový model, kde máme entitiní typ `Student` a `Učitel`. U obou typů budeme chtít evidovat poněkud jiné atributy, ale oba jsou osoby a tedy bude evidovat jméno, přijmení, rodné číslo. Může tedy existovat nadřazený entitiní typ `Osoba`, kde `Student` i `Učitel` jsou jeho potomci. Skutečnost, že jeden entitiní typ je potomek druhého znamená, že potomek má všechny atributy nadřazeného entitiního typu.

Dědičnost můžeme do konceptuálního modelu zahrnout, nicméně hlavní problém je, jak budou tabulky realizovány na úrovni relačního datového modelu. Tři základní varianty jsou vysvětleny v kapitole 3.3

2.4 Konceptuální model UML

2.4.1 Úvod

Unified Model Language (**UML**) [30] 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, některé z nich 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.

2.4.2 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:

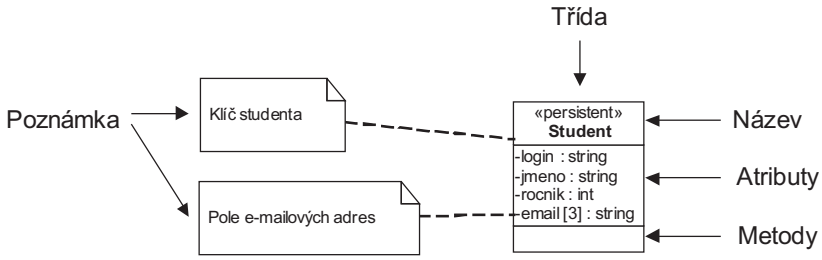
```
[stereotyp] [visibility] [/] <jmenoAtributu> [multiplicity]:
[typ] [=initialValue].
```

Nyní popišme jednotlivé části specifikace:

- `stereotyp`: význam atributu,
- `visibility`: používáme znaky '+' pro **public**; '#' pro **protected** a '-' pro **private**,
- / – označuje odvozený atribut,
- `jmenoAtributu`: jméno atributu,
- `multiplicity`: indikuje vícehodnotový atribut.

Příklad 2.5. (Třída). Vezměme entitní typ `Student` z příkladu **??**. Na obrázku 2.11 vidíme třídu reprezentující studenta. Evidujeme atributy `login`, `jmeno`, `email` a `rocnik`. Viditelnost všech atributů je `private`, tedy na hodnotu atributu

můžeme přistupovat pouze v metodách instance 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.



Obrázek 2.11: 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 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:

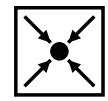
```
[stereotype] [visibility] <jmenoMetody> [parametry]:
[navratovyTyp]
```

Nyní popišme jednotlivé části specifikace:

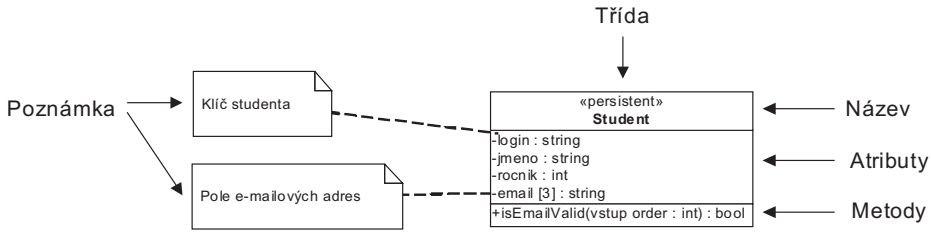
- `stereotype`: význam metody,
- `visibility`: používáme znaky '+' pro **public**; '#' pro **protected** a '-' pro **private**,
- `jmenoMetody`: název metody,

- `parametry`: uspořádaný seznam parametrů oddělených čárkami,
- `navratovyTyp`: návratový typ metody.

Příklad 2.6. (Metody třídy). V případě objektově-orientovaných jazyků definujeme tzv. **nastavovací** a **zpřístupňující** 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 2.11 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 2.12 vidíme způsob zápisu této metody ve třídě.



Obrázek 2.12: Metoda třídy reprezentující studenta

V objektově-orientovaných programovacích jazycích 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.

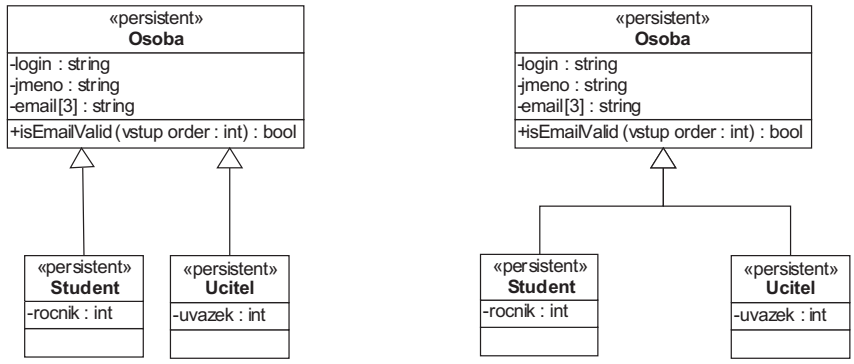
2.4.3 Zobecnění a specializace

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 **znovupoužitelnosti** 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 2.7. (Dědičnost). Na obrázku 2.13 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()`.



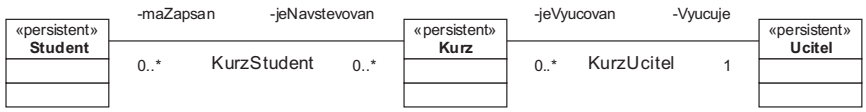
Obrázek 2.13: Dva ekvivalentní způsoby nakreslení dědičnosti třídy `Student` a `Ucitel` dědí ze třídy `Osoba`

2.4.4 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 $0, 1, 0 \cdots *, 1 \cdots *$ atp.

Příklad 2.8. (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 2.14 vidíme třídní digram modelují vztahy mezi těmito třídami. Asociace mezi třídami `Student` a `Kurz` se jmenuje `KurzStudent`. Jedna instance třídy `Student` má vztah 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`.

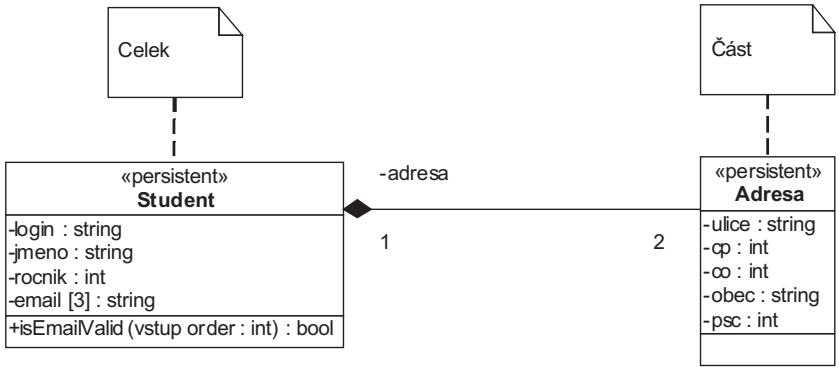


Obrázek 2.14: Třídní digram modelující vztahy mezi třídami `Student`, `Kurz` a `Ucitel`

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.

UML rozlišuje celou řadu asociací, například **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.

Příklad 2.9. (Agregace). Na obrázku 2.15 vidíme agregaci mezi třídami `Student` a `Adresa`. `Student` obsahuje dvě instance třídy `Adresa`.



Obrázek 2.15: Ukázka agregace mezi třídami `Student` a `Adresa`

Část II

Relační datový model

Kapitola 3

Relační datový model

Obsah

Cíl kapitoly:
Datový model představuje zobrazení modelu konceptuálního do modelu, který je blíže fyzické implementaci SŘBD. V této kapitole bude popsán relační 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.

3.1 Úvod

V roce 1970 publikoval Edgar F. Codd článek **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**).

3.2 Základní pojmy

Definice 3.1. *(Relace). Nechť $D_1, D_2, \ldots, D_n$ jsou množiny. N-ární relaci na množinách $D_1, D_2, \ldots, D_n$ nazýváme libovolnou podmnožinu kartézského součinu, $R \subseteq D_1 \times D_2 \times \ldots \times D_n$.*

Relace je abstrakcí dvourozměrné tabulky. Každý řádek obsahuje hodnoty atributů entity, každý sloupec obsahuje hodnoty jednoho atributu. V našem případě budou množiny $D_1, D_2, \ldots, D_n$ množiny hodnot jednotlivých **atributů**, tzv. **domény**. Nyní známe pojem relace, pro úplný popis dat potřebujeme znát i 'hlavičku' relace – **relační schéma**.

Definice 3.2. *(Relační schéma). Relační schéma relace $R \subseteq D_1 \times D_2 \times \ldots \times D_n$ je zápis $R(A_1, A_2, \ldots, A_n, D_1, D_2, \ldots, D_n)$, kde R je jméno schématu, $A_1, A_2, \ldots, A_n$ jsou jména atributů, $D_1, D_2, \ldots, D_n$ jsou domény jednotlivých atributů.*

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

Příklad 3.1. (Relace). Vezměme relační schéma `Student(login, jmeno, prijmeni, rokNarozeni)`. V tabulce 3.1 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.

Formálně se jedná o podmnožinu $D_1 \times D_2 \times D_3 \times \ldots \times D_n$, 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**).

Musíme si uvědomit, že relace je množina n-tic, nikoli uspořádaná množina n-tice. Nemůže tedy mluvit o prvním prvku, druhém, atd. prvku relace. V tabulce 3.2 vidíme relaci `Student`, která je ekvivalentní s relací z tabulky 3.1.

Tabulka 3.1: Relace `Student`

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

Tabulka 3.2: Relace `Student`

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

V relačním datovém modelu definujeme i množinu podmínek, které dále definují vlastnosti jednotlivých atributů. Těmto podmínkám říkáme **integritní omezení (IO)**. Může se jednat o elementární podmínky nad jednotlivými atributy tak o komplexní podmínky zahrnující více atributů. Příkladem může být podmínka, kde definujeme, že hodnota atributu může být NULL (tzn. nemá žádnou hodnotu). Komplexní podmínka může například definovat, že součet všech hodnot určitého atributu musí být menší než je hodnota jiného atributu. Říkáme, že **databáze je konzistentní** pokud všechny relace vyhovují všem integritním omezením.

V relačním datovém modelu identifikujeme atribut relačního schématu dle jehož hodnoty je n-tice odlišitelná od ostatních. Takový atribut nazýváme **primárním klíčem**. Klíčem relačního schématu `Student` z příkladu 3.1 je atribut `login`. Primární klíč je v podstatě také typem integritního omezení.

S primárním klíčem úzce souvisí i tzv. **cizí klíč**, což je atribut(y), který se odkazuje na nějaký primární klíč. Zkusme to popsat více formálně. Mějme dvě relační schémata `R(k,x)` a `S(k,x,ck)`, kde atributy `k` představují klíče relací `R` a `S` a atribut `ck` je cizí klíč odkazující se na atribut `R.k`. Musí platit, že každý prvek relace `S` bude mít hodnotu `S.ck`, která existuje v `R.k`. Tato podmínka se nazývá **referenční integrita** relačních databází a její platnost pro všechny relace je jednou ze základních vlastností relačních databází. Zajištění referenční integrity databázovým systémem je velká pomoc pro programátory používající relační databázový systém, nicméně představuje i určité omezení, když pracujeme s velkými daty. Proto například nový typ databází souhrně nazývaný

NoSQL často ani nemá zajištění referenční integrity mezi svými funkcemi (tzn. nemá cizí klíče).

Příklad 3.2. (Primární a cizí klíče).

Mějme dvě relace Oddělení a Zaměstnanec, které vidíme na obrázku 3.3. Relace `Oddělení` má primární klíč složený ze dvou atributů (id a typ) a relace `Zaměstnanec` se na tuto relaci odkazuje s pomocí cizího klíče. Vidíme, že cizí klíč obsahuje přesně tolik atributů jako primární klíč tabulky na kterou se odkazuje.

Tabulka 3.3: Dvě relace, kde se druhá odkazuje na první

(a) Relace `Oddělení`

id	typ	ulice	obrat
22	export	Slavníkovců	15
22	import	Slavníkovců	16
26	export	Mojmírovců	2

(b) Relace `Zaměstnanec`

rodne číslo	jméno	funkce	id odd.	typ odd.
7905051111	Novák	IT	22	export
6901112233	Kachlička	Manager	22	export
7105029876	Trier	IT	26	export

Na příkladu můžeme vidět, že používání primárních klíčů složených z více atributů může být poněkud nepraktické. Vede to k tomu, že cizí klíče odkazující se na takovou tabulku jsou také složeny z více atributů, což může trochu znepřehlednit schéma databáze. Jednoduchým a často používaným řešením je vytvoření tzv. umělého primárního klíče pro danou relaci, kterým je nejčastěji číslo generované SŘBD při vkládání záznamu. Na obrázku 3.4 můžeme vidět upravené relace z obrázku 3.3, kde byl přidán umělý primární klíč `oID` k relaci `Oddělení` a příslušným způsobem je tedy i upraven cizí klíč relace `Zaměstnanec`.

Tabulka 3.4: Přidání umělého primárního klíče do tabulky oddělení

(a) Relace `Oddělení`

oID	id	typ	ulice	obrat
1	22	export	Slavníkovců	15
2	22	import	Slavníkovců	16
3	26	export	Mojmírovců	2

(b) Relace `Zaměstnanec`

rodne číslo	jméno	funkce	oID
7905051111	Novák	IT	1
6901112233	Kachlička	Manager	1
7105029876	Trier	IT	3

3.3 Od konceptuální modelu k modelu relačnímu

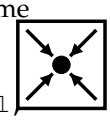
Jak bylo řečeno v předchozí kapitole, pojem atributu v konceptuálním a relačním datovém modelu je ekvivalentní. V následujícím textu budeme používat pojmy vlastní konceptuálnímu modelu ER. Pokud bude použit konceptuální model

UML, čtenář jistě použije ekvivalentní pojmy. Přímý přístup k převodu konceptuálního modelu do relačního datového modelu, může tedy vypadat následovně:

1. Entitní typy a jejich atributy jsou mapovány na relační schémata s příslušnými atributy.
2. Vazby $1 : M$ jsou implementovány vložením primárního klíče schématu s násobností 1 jako cizího klíče do schématu s násobností M.
3. 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 5.1 si ukážeme aparát, který nám umožní rozpoznat, zda námi navržené schéma relační databáze je korektní a rovněž nám umožní případné chyby odstranit.

Příklad 3.3. (Převod konceptuálního modelu na schéma relační databáze). Vezměme konceptuální model reprezentovaný třídním diagramem z příkladu 2.8. V tomto případě jsou vytvořena relační schémata `Student(login, jmeno, email, rocnik)`,`Kurz(id, nazev, kapacita)` a `Ucitel(login, jmeno, email, uvazek)`.



Vazba $M : N$ 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.

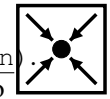
Tabulka 3.5: Relace `KurzStudent`

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

Tabulka 3.6: Relace `Kurz`

id	nazev	kapacita	ucitelLogin
45613	TIS	300	kra102
45601	UDP	300	kra102
45603	ZA	300	dvo005
45638	DBIS	300	svo032

Příklad 3.4. (Převod asociace na relaci). Vezměme konceptuální model reprezentovaný třídním diagramem z příkladu 2.8. Vazba `KurzStudent` (s násobností $M : N$) je převedena na relační schéma `KurzStudent(kurzId, studentLogin)`. Vazba `KurzUcitel` (s násobností $M : 1$) je implementována vložením primárního klíče schématu `Ucitel` do schématu `Kurz`, tedy: `Kurz(id, nazev, kapacita, ucitelLogin)`. Z důvodu přehlednosti byl k názvům atributů přidán prefix naznačující relační schéma, ke kterému atribut náleží. Ukázky relací typu `KurzStudent` a `Kurz` vidíme v tabulkách 3.5 a 3.6. Kurz 45613 studují dva studenti, kurz 45601 tři studenti, kurzy 45603 a 45638 studuje jeden student.



Pokud by násobnost vazby `KurzUcitel` byla $M : N$ museli bychom vytvořit schéma `KurzUcitel(kurzId, ucitelLogin)`, příklad takovéto relace vidíme v tabulce 3.7.

Tabulka 3.7: Relace `KurzUcitel`

kurzId	ucitelLogin
45613	kra102
45601	kra102
45603	dvo005
45638	svo032

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.

Příklad 3.5. (Převod dědičnost do relačního datového modelu). Vezměme v úvahu dědičnost v třídním diagramu z příkladu 2.6. 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`.

Kapitola 4

Dotazovací jazyky

Obsah

Cíl kapitoly:
V této kapitole budou popsány dotazovací jazyky pro relační datový model: relační algebra a SQL.

4.1 Historický vývoj

Dotazovací jazyk bezprostředně souvisí s datovým modelem. Nejdůležitějším a nejrozšířenějším dotazovacím jazykem spjatým s relačním datovým modelem je SQL. Před uvedením SQL se zaměříme na relační algebru, která nám pomůže lépe pochopit jazyk SQL a na které je jazyk SQL založen.

4.2 Relační algebra

Relační algebra [37] je jazyk vysoké úrovně, který nepracuje s jednotlivými n-ticemi[1], ale s celými relacemi. Operátory relační algebry jsou aplikovány na relace, výsledkem jsou opět relace. Jelikož relace jsou množiny, jsou přirozenými operacemi s relacemi množinové operátory sjednocení (označujeme $\cup$), průnik ($\cap$), rozdíl ($-$) a kartézský součin ($\times$). Další používané operace jsou již specifické pro dotazování relací, jedná se o: **selekci**, **projekci** a **spojení**. Čtenář se možná zeptá, proč komplikovat vysvětlování jazyka SQL popisem nějakého abstraktnějšího jazyka, jakým je relační algebra. Velmi často jsou používány SQL dotazy, které jsou triviální (jednoduchý výběr řádků apod.). Pokud ovšem chceme formulovat komplikovanější dotaz, potřebujeme přesně znát tento dotazovací jazyk. K lepšímu pochopení SQL slouží právě relační algebra, která pracuje s relacemi.

4.2.1 Základní operace

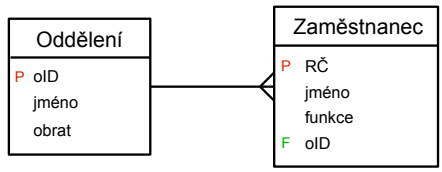
V této kapitole popíšeme základní operace relační algebry jako je projekce, selekce a kartézský součin.

Selekce (**restrikce**), $\sigma_{podmínka}(R)$, operace selekce vybírá z relace R pouze ty řádky, které splňují podmínku. Podmínka je zadána logickým výrazem výrazem, jenž může obsahovat logické spojky (`and`, `or` a `not`), kde prvky logického výrazu jsou ve tvaru $C_1\ \theta\ C_2$. C_1 a C_2 může být jméno atributu, konstanta, či další vnořený výraz vracející skalární hodnotu a $\theta \in \{<, >, =, \leq, \geq, \neq\}$.

Projekce, $\pi_{seznam\ atributů}(R)$, operace projekce vybírá z relace R pouze ty sloupce, které odpovídají atributům uvedeným v seznamu. Seznam atributů pochopitelně musí být podmnožinou nebo roven množině atributů R. Výsledek projekce je tedy relace s daným seznamem atributů a jelikož je výsledek množina, jsou odstraněny případné duplicitní řádky, které mohou vzniknout odstraněním některých sloupců.

Příklad 4.1. (Operace s relacemi). Mějme relace `Oddělení(oID, jméno, obrat)` a `Zaměstnanec(RČ, jméno, funkce, oID)`, jejichž zjednodušené relační schéma vidíme na obrázku 4.1. V tabulce 4.1 vidíme příklad těchto relací a tabulky 4.2 ukazují některé základní operace algebry nad relacemi `Oddělení` a `Zaměstnanec`.

[1] V anglicky psané literatuře se často vyskytuje pojem **tuple**



Obrázek 4.1: Relační model databáze

Tabulka 4.1: Relace `Oddělení` a `Zaměstnanec`

(a) Relace `Zaměstnanec`

RČ	jméno	funkce	oID
7905051111	Novák	IT	1
6901112233	Kachlička	Manager	1
7105029876	Trier	IT	3

(b) Relace `Oddělení`

oID	jméno	obrat
1	účetní	1
3	testování	8

Tabulka 4.2: Základní operace

(a) $\sigma_{\text{funkce = 'IT'}}(\textit{Zaměstnanec})$

RČ	jméno	funkce	oID
7905051111	Novák	IT	1
7105029876	Trier	IT	3

(b) $\pi_{\text{RČ, jméno}}(\textit{Zaměstnanec})$

RČ	jméno
7905051111	Novák
6901112233	Kachlička
7105029876	Trier

(c) *Zaměstnanec* $\times$ *Oddělení*

RČ	jméno	funkce	oID	oID	jméno	obrat
7905051111	Novák	IT	1	1	účetní	1
6901112233	Kachlička	Manager	1	1	účetní	1
7105029876	Trier	IT	3	1	účetní	1
7905051111	Novák	IT	1	3	testování	8
6901112233	Kachlička	Manager	1	3	testování	8
7105029876	Trier	IT	3	3	testování	8

Operace sjednocení, průnik a rozdíl vyžadují, aby relace měly stejné schéma (stejný počet a jména atributů). Pro formální správnost se nám tedy může hodit operace přejmenování, která schéma nastaví do potřebné podoby.

Přejmenování, $\rho_{R'(A_1,\cdots,A_n)}(R)$, operace přejmenování změní původní schéma relace R na schéma $R'(A_1, \cdots, A_n)$.

4.2.2 Kombinace operací a operace spojení

Operátory relační algebry můžeme libovolně kombinovat. Je to dáno tím, že výsledek každé operace nad relacemi je opět relace. Tato základní vlastnost algeber, umožňuje vytvářet složené výrazy relační algebry odpovídající komplikovaným reálným zadáním a potřebám.

Zkusme nejprve jednoduchý příklad využívající relace `Oddělení` a `Zaměstnanec` z příkladu 4.1. Chceme vypsat všechny jména zaměstnanců spolu s obratem oddělení ve kterém zaměstnanec pracuje. S využitím základních operací relační algebry bude výsledný výraz vypadat takto:

$$\pi_{Zaměstnanec.\text{jmeno},\, Oddělení.\text{obrat}}\, \sigma_{Zaměstnanec.\text{oID}=Oddělení.\text{oID}}(Zaměstnanec \times Oddělení)$$

Výsledkem této operace je vlastně spojení odpovídajících záznamů z obou relací podle atributu `oID`. Takovéto spojení je jedna z nejčastějších operací nad relacemi v relační databázi a proto je pro ni zavedeny hned dva podobné operátory: (1) přirozené spojení a (2) theta spojení.

Přirozené spojení, $R \bowtie S$, ze součinu $R \times S$ se vyberou pouze řádky se stejnými hodnotami u stejnojmenných atributů u obou relací.

Theta spojení (zkráceně spojení), $R \bowtie_\Theta S$, ze součinu $R \times S$ se vyberou pouze řádky odpovídající podmínce Θ.

Pokud bychom se vrátili k našemu příkladu pak s pomocí theta spojení bude náš příklad zapsán následovně:

$$Zaměstnanec \bowtie_{Zaměstnanec.oID=Oddělení.oID} Oddělení$$

S pomocí přirozeného spojení tento příklad nezapíšeme, protože *Zaměstnanec* $\bowtie$ *Oddělení* spojuje nejen dle atributu `oID`, ale také dle atributu `jméno`, který nalezneme v obou relacích.

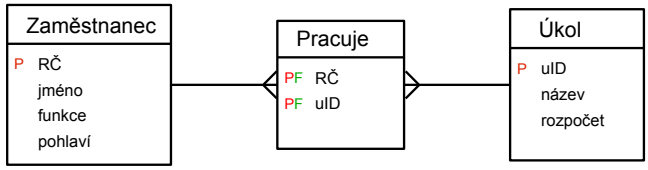
Theta spojení je operací, která odpovídá vnitřnímu spojení v SQL (viz. podkapitola 4.3.1) a jedná se o operaci bez níž se neobejde většina dotazů. V následujících příkladech nám operace spojení také značně zpřehlední zápis dotazů relační algebry.

Příklad 4.2. (Komplexnější příklady). Mějme relace `Zaměstnanec(RČ, jméno, funkce, pohlaví)`,`Pracuje(RČ, uID)` a `Úkol(uID, název, rozpočet)`, jejichž zjednodušené relační schéma vidíme na obrázku 4.2. Uvedeme příklad pro tři trochu komplexnější zadání:

1. Vypiště názvy úkolů, na kterých pracuje nějaká žena.

2. Vypiště názvy úkolů, na kterých pracují jenom ženy.

3. Vypiště názvy úkolů, na kterých pracují zároveň Petr i Hanka.



Obrázek 4.2: Relační model databáze

U všech zadání bude nezbytné provést spojení všech tří relací *Zaměstnanec* ⋈ *Pracuje* ⋈ *Úkol*. Výsledkem tohoto spojení je jedna relace, kde řádek reprezentuje skutečnost, že jeden zaměstnanec pracoval na jednom úkolu. S touto velkou relaci nám stačí pro vyřešení prvního úkolu provést výběr řádků, kde hodnota atributu `pohlaví` je žena a následně vybereme pouze sloupec název. Výraz relační algebry pro první zadání tedy může být

$$\pi_{název}\ \sigma_{pohlaví='žena'}\ (Úkol \bowtie Pracuje \bowtie Zaměstnanec)$$

Všimněme si, že v tomto případě můžeme transparentně změnit pořadí některých operací. Jedná se o provedení operace selekce. Tato operace může být provedena už před spojením všech tří relací, takže korektní výraz relační algebry může vypadat i následovně

$$\pi_{název}(Úkol \bowtie Pracuje \bowtie \sigma_{pohlaví='žena'} Zaměstnanec)$$

Z pohledu vykonávání bude druhá varianta efektivnější (nebo přinejmenším stejně efektivní), jelikož při vyhodnocení nám vzniknou menší mezivýsledky. Takovéto transparentní převody výrazů relační algebry využívají optimalizátory databázových systémů pro dosažení lepšího výkonu.

Druhé zadání můžeme formulovat i jinak: Vypiště názvy úkolů na kterých nepracuje ani jeden muž, ale pracuje na nich nějaká žena. Cílem je tedy provést rozdíl dvou množin: množiny úkolů na kterých pracuje nějaká žena a množiny úkolů na kterých pracuji nějací muži. Výraz tedy bude mít podobu:

$$\pi_{název}(Úkol \bowtie Pracuje \bowtie \sigma_{pohlaví='žena'} Zaměstnanec) -$$

$$\pi_{název}(Úkol \bowtie Pracuje \bowtie \sigma_{pohlaví='muž'} Zaměstnanec)$$

Zkuste se zamyslet proč nestačí u prvního výrazu nalézt seznam všech úkolů? Tedy hledat názvy úkolů na kterých nepracuje ani jeden muž.

U třetího zadání můžeme mít nutkání napsat jako variantu prvního zadání:

$$\pi_{název}(Úkol \bowtie Pracuje \bowtie \sigma_{jméno='Petr' \wedge jméno='Hanka'} Zaměstnanec)$$

Nicméně toto řešení nikdy nemůže vrátit nic jiného než prázdnou množinu. Problém je v podmínce operace selekce, která nikdy nemůže být splněna u jediného zaměstnance. Žádný zaměstnanec nemůže mít zároveň jméno Petr a Hanka. Podobně jako u druhého příkladu je nutné o řešení přemýšlet jako o množinových operacích. Správné řešení bude provádět průnik dvou množin. Množiny úkolů na kterých pracuje Petr a množiny úkolů na kterých pracuje Hanka:

$$\pi_{název}(Úkol \bowtie Pracuje \bowtie \sigma_{jméno='Petr'} Zaměstnanec) \cap$$

$$\pi_{název}(Úkol \bowtie Pracuje \bowtie \sigma_{jméno='Hanka'} Zaměstnanec)$$

4.3 Stuctured Query Language (SQL)

Počátky jazyka SQL (**Stuctured Query Language**) sahají až do roku 1974. Od té doby byl jazyk SQL standardizován jak organizací ISO tak ANSI. Tento standard se v průběhu let vyvíjel a existuje několik hlavních revizí: SQL-86, SQL-89, SQL-92, SQL:1999, SQL:2003, SQL:2006, SQL:2008, SQL:2011.

Bohužel různé SŘBD se více či méně od standardu odlišují. Důvody jsou obvykle historické, či implementační. Při implementaci jádra mnoha SŘBD neobsahoval SQL standard všechny potřebné konstrukce (což v podstatě trvá dodnes) a pro implementaci některých funkcí se výrobci museli rozhodnout pro vlastní řešení. Když byla následně daná konstrukce standardizována v SŘBD již byla někdy ponechána původní verze. To se týká zejména procedurálního rozšíření SQL, které je ve většině SŘBD odlišné, ale rozdíly nalezneme i v základních konstrukcích SQL. Rozdíly najdeme zejména mezi datovými typy, jmény a parametry funkcí a některými konstrukcemi z jazyka pro definici dat.

Důsledkem je špatná přenositelnost databáze mezi jednotlivými SŘBD a nutnost zvýšené pozornosti k syntaxi pokud přecházíme mezi jednotlivými SŘBD. Jelikož je tento text zaměřen i na praktickou výuku databází ve vybraných SŘBD, chceme přehledně uvést i specifika těchto SŘBD. Mezi tyto SŘBD patří zejména SQL Server a Oracle, které v poslední dekádě tvoří velkou část komerčního trhu s databázovými systémy. Naštěstí základní principy jsou u těchto SŘBD shodné se standardem a mnoho částí se obejde bez komentáře pro určité SŘBD. Zejména

základní příkazy SELECT, INSERT, DELETE a UPDATE jsou kompatibilní s SQL standardem.

Podstatná vlastnost SQL je že se jedná o deklarativní jazyk. Příkazy tedy popisují **co chceme** najít nikoli **jak se to má provést**. Tato vlastnost SQL je základem pro nezávislost jazyka na fyzickém uložení dat. Tedy teoreticky nepotřebujeme řešit jak jsou data uložena, pouze nás zajímá jejich logické uspořádání.

Jazyk SQL obsahuje tyto části:

- jazyk pro definici dat (JDD),
- jazyk pro manipulaci s daty (JMD),
- možnost definice přístupových práv,
- možnost definice integritních omezení,
- řízení transakcí.

4.3.1 Příkaz `SELECT`

Naše povídání o SQL začne příkazem `SELECT`, jenž je příkazem pro dotazování databáze. Tomuto příkazu je v každém textu o SQL věnováno oprávněně nejvíce pozornosti. `SELECT` umožňuje deklarativním způsobem vybírat a zpracovávat data v databázi, aniž bychom se museli blíže starat o to jakým způsobem jsou data uložena. Tato jeho velká výhoda na druhou stranu vyžaduje určité vstupní úsilí, jelikož pro psaní `SELECT` příkazů se musíme odklonit od procedurálního chápání zpracování příkazů. Jde tedy hlavně o to naučit se o dotazu přemýšlet jako o operacích nad množinami.

```
SELECT Atr1, ..., AtrN
FROM Tabulka1, ..., TabulkaN
WHERE Podmínka
```

Výpis 4.1: Základní struktura `SELECT` příkazu

`SELECT` je příkaz postavený na relační algebře představené v kapitole 4.2. V některých ohledech se SQL od relační algebry liší, ale základní principy zůstávají stejné. Pro jednoduché dotazy může být příkaz `SELECT` triviální, obecně se ovšem může jednat o velmi komplikovaný příkaz. Ve výpisu 4.1 můžeme vidět pouze základní konstrukce `SELECT FROM WHERE`, které si nyní popíšeme:

- Za klíčovým slovem `SELECT` se vyskytuje seznam atributů, jejichž hodnoty budou výsledkem dotazu.
- Za klíčovým slovem `FROM` se vyskytuje seznam tabulek nad kterými je dotaz definován.
- Za klíčovým slovem `WHERE` se vyskytuje logická podmínka, která realizuje selekci (viz kapitola 4.2 o relační algebře).

Dle standardu jsou `SELECT` a `FROM` povinné části příkazu `SELECT`, nicméně SQL Server umožňuje napsat `SELECT` bez `FROM`. Jelikož nám relační algebra může pomoci lépe pochopit sémantiku příkazu `SELECT`, ukažme si nyní vyjádření dotazu pomocí relační algebry. Mějme dotaz v SQL a ekvivalentní dotaz v relační algebře:

$$\begin{array}{l}\texttt{SELECT } A_1, \ldots, A_n \\ \texttt{FROM } R_1, \ldots, R_m \\ \texttt{WHERE } podminka \end{array}$$

$$\Updownarrow$$

$$\pi_{A_1,\ldots,A_n}\left(\sigma_{podminka}(R_1 \times \cdots \times R_m)\right)$$

Pokud neexistuje podmínka, pak výsledkem dotazu jsou všechny n-tice kartézského součinu $R_1 \times R_2 \times \ldots \times R_m$. Z pohledu duplicit nemusí ale být tyto dva výrazy zcela totožné. To je ukázáno na příkladu 4.4(f). Jde především o to, že SQL duplicity implicitně neodstraňuje zatímco relační algebra ano.

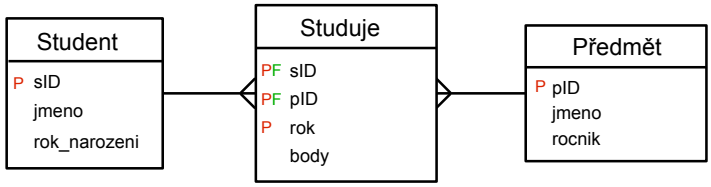
Základní SELECT konstrukce

V příkladech 4.8 4.4, 4.6 a 4.7 jsou představeny základní konstrukce `SELECT`.

Příklad 4.3. (Jednoduché dotazy SQL). Mějme tři relace `Student(sID, jmeno, rok_narozeni)`,`Predmet(pID, jmeno, rocnik)` a `Studuje(sID, pID, rok, body)` jejichž zjednodušené relační schéma vidíme na obrázku 4.4. Tento datový model je používám ve všech příkladech k příkazu `SELECT`.

(a) Vypište celé n-tice všech studentů relace `Student`:

```
SELECT * FROM Student
```



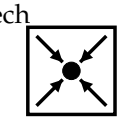
Obrázek 4.3: Relační datový model pro relace `Student`, `Studuje` a `Předmět`

(b) Vypište sID a jméno všech studentů:

```
SELECT sID, jmeno FROM Student
```

Můžeme přidat podmínku za klausuli WHERE kde můžeme použít tyto predikáty: $=$, $<$, $>$, $<=$, $>=$, $<>$ (nebo $!=$ což ale není součástí standardu). Výrazy lze spojovat spojkami `AND`, `OR` a `NOT`. Podmínky napsané za klauzulí WHERE odpovídají operátoru select v relační algebře. Pokud pracujeme s atributy, které nemohou nabývat hodnoty NULL, tak lze vždy napsat podmínku, který pro jeden záznam nabývá jen hodnot `pravda` a `nepravda`.

Příklad 4.4. (Příkazy SELECT s podmínkou a spojení relací). Na několika příkladech ukážeme jak může vypadat dotaz s podmínkou za WHERE.



(a) Vypište všechny studenty z tabulky `Student` narozené po roce 1982.

```
SELECT * FROM Student
WHERE rok_narozeni > 1982
```

(b) Vypište všechny studenty z tabulky `Student` narozené po roce 1982, ale před rokem 1985.

```
SELECT * FROM Student
WHERE rok_narozeni > 1982 AND rok_narozeni < 1985
```

(c) Vypište všechny studenty z tabulky `Student` narozené v roce 1982 a později, ale před rokem 1985 včetně.

```
SELECT * FROM Student
WHERE rok_narozeni >= 1982 AND rok_narozeni <=
    1985
```

V SQL můžeme použít klíčové slovo `BETWEEN`, které zjednodušuje zadávání takovýchto intervalových dotazů.

```
SELECT * FROM Student
WHERE rok_narozeni BETWEEN 1982 AND 1985
```

(d) Vypište všechny studenty, kteří se narodili před rokem 1982 a po roce 1985:

```
SELECT * FROM Student
WHERE NOT (rok_narozeni BETWEEN 1982 AND 1985)
```

(e) Ke každému jménu studenta vypište i `pID` předmětu, který studuje či studoval.

```
SELECT jmeno, pID FROM Student, Studuje
WHERE Student.sID = Studuje.sID
```

Tento dotaz by se v relační algebře dal vyjádřit s pomocí výrazu

$$\pi_{jmeno,pID}\,(Student \bowtie Studuje)$$

Nicméně z pohledu možných duplicit nemusí být výsledky totožné. Jinak řečeno, zatímco `SELECT` může obsahovat duplicity, výraz relační algebry ne jelikož výsledkem je množina, která z definice duplicity obsahovat nemůže. Abychom dostali totožné výsledky musíme do `SELECT` příkazu přidat slůvko `DISTINCT`:

```
SELECT DISTINCT jmeno, pID
FROM Student, Studuje
WHERE Student.sID = Studuje.sID
```

Predikáty se komplikují, když začneme pracovat s atributy, které mohou nabývat hodnoty NULL. Pokud nějaký záznam má v atributu hodnotu NULL tak to obvykle znamená, že hodnota není známa. Pokud píšeme nějaký predikát za WHERE, tak daný záznam je ve výsledku pokud predikát nabývá pro daný záznam hodnoty `pravda`. Nicméně pokud porovnáváme dvě hodnoty a některá z nich je NULL je výsledek porovnání `neznámý` (`uknown`). Pokud je výsledek predikátu `neznámý` pak se do výsledku dotazu nedostane. Negace hodnoty `neznámý` je opět `neznámý`, což může být problém jak uvidíme v příkladu 4.5(b).

Příklad 4.5. (`SELECT` a NULL hodnoty).

(a) Najděte jména všech studentů, kteří nemají uveden rok narození:

```
SELECT jmeno
FROM Student
WHERE rok_narozeni IS NULL
```

Pokud bychom uvedli místo `rok_narozeni is NULL` predikát `rok_narozeni = NULL`, pak by se pro jakoukoli hodnotu `rok_narozeni` predikát vyhodnotil jako `neznámý`. Je to dáno tím, že dochází k porovnání s NULL hodnotou. I u porovnání dvou NULL hodnot je výsledek `neznámý`.

(b) Vypište studenty, kteří nemají uveden rok narození 1985:

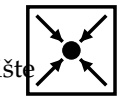
```
SELECT *
FROM Student
WHERE rok_narozeni <> 1985
```

Pokud spustíme tento dotaz tak zjistíme, že ve výsledku nejsou studenti, kteří mají hodnotu NULL v roce narození. Je to dáno tím, že porovnání s NULL hodnotou má výsledek `neznámý` a do výsledku se tedy nedostane. Správný dotaz bude mít tvar:

```
SELECT *
FROM Student
WHERE rok_narozeni <> 1985 OR rok_narozeni IS NULL
```

U dotazů nemusíme jednotlivé atributy používat tak jak jsou, ale může docházet k jejich manipulaci. Manipulací rozumíme například provádění aritmetických operací, nebo volání nějaké funkce příslušné pro daný datový typ (viz. kapitola 4.3.2). K manipulaci s atributem může docházet v jakékoli části příkazu SELECT. Pokud k manipulaci s atributem dochází v predikátu, pak to u většiny SŘBD může mít za následek, že optimalizátor není schopen použít index pro daný atribut.

Příklad 4.6. (`SELECT` používající aritmetické operátory a aliasy).



(a) Vypište, v jakém věku studovali jednotliví studenti jednotlivé předměty (vypište jméno studenta, jeho pID a věk)

```
SELECT jmeno, pID, rok - rok_narozeni
FROM Student, Studuje
WHERE Student.sID = Studuje.sID
```

(b) V předchozím SQL příkazu ve výsledku dostáváme novou hodnotu, kterou jsme získali rozdílem dvou atributů. Takto vznikl nový nepojmenovaný atribut. Aby byl výpis přehlednější můžeme sloupec pojmenovat uvedením náhradního jména (tzv. alias name). Alias se uvádí za klíčovým slovem AS a může být v uvozovkách. Oracle i SQL Server navíc dovolují klíčové slůvko `AS` vynechat.

```
SELECT jmeno, pID, rok - rok_narozeni AS vek
FROM Student, Predmet
WHERE Student.sID = Studuje.sID
```

(c) Aliasy je možné používat i u tabulek. Přidáme alias za jména tabulek čímž zkrátíme zápis podmínky:

```
SELECT jmeno, pID, rok - rok_narozeni AS vek
FROM Student st, Studuje se
WHERE st.sID = se.sID
```

(d) Vypište dvojice (jméno studenta, pID) u studentů jejichž věk v době studia předmětu s daným pID přesáhl 30.

```
SELECT jmeno, pID
FROM Student st, Studuje se
WHERE st.sID = se.sID and rok - rok_narozeni > 30
```

Tento dotaz provádí manipulaci s atributem v predikátu.

(e) V některých případech je použití alias nevyhnutelné. Příkladem může být spojení dvou stejných tabulek (tzv. self-join). Chceme vypsat dvojice všech různých studentů, kteří se narodili ve stejný rok:

```
SELECT S1.jmeno, S2.jmeno, S2.rok_narozeni
FROM Student S1, Student S2
WHERE S1.rok_narozeni = S2.rok_narozeni AND
  S1.jmeno > S2.jmeno AND
  S1.rok_narozeni IS NOT NULL
```

Vidíme, že v tomto případě se v dotazu máme dvě identické tabulky a abychom je od sebe odlišili máme jednu pojmenovánu `S1` a druhou `S2`. Z pohledu dalšího vykonání dotazu je s oběma nakládáno jako by se jednalo o různé tabulky. Tabulky se spojují dle atributu `rok_narození` (nalezení dvojic studentů narozených ve stejný rok) a dále se z výsledku odstraní dvojice odpovídající stejnému studentovi.

Stejně jako relační algebra umožňuje SQL použití množinových operací. Jak uvidíme v dalších podkapitolách množinové operace jde často lehce přepsat do jiného tvaru. Nicméně řešení používající množinové operace jsou často přehlednější a rozhodně se vyplatí je znát a ve vhodných případech použít.

Příklad 4.7. (`SELECT` a množinové operace). V následujících příkladech uvedeme řešení s použitím SQL a relační algebry.

(a) Vypište dohromady jména všech studentů a předmětů.

```
SELECT jmeno FROM Student
UNION
SELECT jmeno FROM Predmet
```

$\pi_{jmeno}\ Student\ \cup\ \pi_{jmeno}\ Predmet$

(b) Vypište studenty, kteří studovali nebo studují oba předměty s pID 1 a 5.

```
SELECT St.jmeno FROM Student St, Studuje Se
WHERE St.sID = Se.sID and Se.pID = 1
INTERSECT
SELECT St.jmeno FROM Student St, Studuje Se
WHERE St.sID = Se.sID and Se.pID = 5
```

$\pi_{jmeno}(Student \bowtie_{pID=1} Studuje) \cap$
$\pi_{jmeno}(Student \bowtie_{pID=5} Studuje)$

(c) Vypište pID všech předmětů, které studují pouze studenti narození po roce 1985.

```
SELECT distinct pID FROM Studuje
EXCEPT
SELECT distinct pID
FROM Student, Studuje
WHERE Student.sID = Studuje.sID AND
  rok_narozeni <= 1985 OR rok_narozeni IS NULL
```

$\pi_{pID}(Studuje) -$
$\pi_{pID}(Student \bowtie_{rok\_narozeni<=1985} Studuje)$

Uspořádání záznamů a omezení velikost výsledku

Výpis 4.2 přestavuje další dvě kostrukce, které můžeme použít v `SELECT`. Jedná se o uspořádání záznamů s pomocí `ORDER BY` a omezení velikost výsledku s pomocí `OFFSET-FETCH`. Pokud nespecifikujeme uspořádání záznamů pak dotaz může vracet záznamy v **libovolném pořadí**. Toto je důležité si uvědomit, jelikož uživatel SŘBD může lehce podlehnout dojmu, že i bez ORDER BY se mu záznamy vracejí vždy v určitém pořadí. Při vykonávání databázový systém nikdy nedělá nadbytečnou práci, takže pokud nespecifikujeme pořadí záznamů, pak jsou záznamy uspořádány tak, jak jdou z posledního operátoru. Tzn. pokud se například v důsledku vytvoření nějakého indexu změní plán vykonání

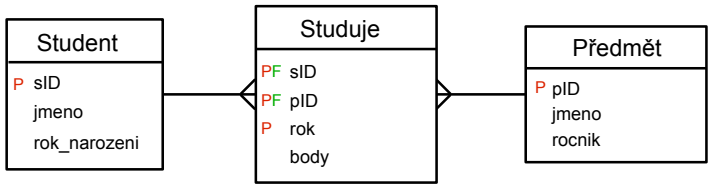
dotazu, může se změnit i pořadí `SELECT` bez `ORDER BY`. Jak vidíme nepovině můžeme specifikovat, zda-li se záznamy budou řadit vzestupně (`ASC`) nebo sestupně (`DESC`), přičemž ASC je implicitní volba. Atributy, které jsou uvedeny za `ORDER BY` nemusí být za `SELECT`, pokud za `SELECT` neuvádíme `DISTINCT`. V `ORDER BY` můžeme třídit i na základě aliasů, vytvořených v klausuli `SELECT`, což je dáno logickým pořadím konstrukcí.

Konstrukce `OFFSET-FETCH` umožňuje specifikovat kolik řádků chceme ve výsledku (`FETCH`) a také od kterého záznamu chceme začít (`OFFSET`). Jinak řešeno, definujeme jakési okno ve výsledku `SELECT`, které se dostane do konečného výsledku. Tedy `m` ve výpisu 4.2 určuje začátek tohoto okna a `n` určuje jeho velikost. Tato konstrukce je u SŘBD podporována až s nejnovějšími verzemi (od SQL Server 2012 a Oracle 12), dříve SŘBD implementovali vlastní konstrukce jako TOP a ROWNUM.

```
SELECT ...
FROM Tabulka1, ..., TabulkaN
ORDER BY Atr1 [ASC|DESC], ..., AtrN [ASC|DESC]
OFFSET m (ROW|ROWS)
FETCH (FIRST|NEXT) n (ROW|ROWS) ONLY
```

Výpis 4.2: Definice uspořádání a výběr podmnožiny výsledku `SELECT`

Příklad 4.8. (Jednoduché dotazy SQL). Mějme tři relace `Student(sID, jmeno, rok_narozeni)`,`Predmet(pID, jmeno, rocnik)` a `Studuje(sID, pID, rok, body)` jejichž zjednodušené relační schéma vidíme na obrázku 4.4. Tento datový model je používám ve všech příkladech k příkazu `SELECT`.



Obrázek 4.4: Relační datový model pro relace `Student`, `Studuje` a `Předmět`

(a) Vypište sID a jméno všech studentů, výsledek bude uspořádán vzestupně dle jmen studentů.

```
SELECT login, jmeno FROM Student
ORDER BY jmeno ASC
```

(b) Vypište věk studentů, kteří studovali UDBS v roce 2010, setřízený sestupně. Vynechte duplicity.

```
SELECT distinct rok - rok_narozeni AS vek
FROM student st
JOIN studuje se ON st.sID = se.sID AND rok = 2010
ORDER BY vek DESC
```

(c) Vypište sID a jméno všech studentů, výsledek bude uspořádán vzestupně dle jmen studentů a budou nás zajímat pouze první tři studenti.

```
SELECT login, jmeno
FROM Student
ORDER BY jmeno ASC
OFFSET 0 ROWS
FETCH NEXT 3 ROWS ONLY
```

SELECT s poddotazy

V této podkapitole si ukážeme další možnosti příkazu SELECT, které budeme souhrně nazývat poddotazy. Poddotazy jsou nějaké vnořené SELECT dotazy jejichž výsledek je využit ve vnějším dotazu. Podle výsledku můžeme poddotazy rozdělit do následujících kategorii:

- Skalární hodnota - dotaz vrací právě jednu hodnotu
- Množina hodnot - výsledkem dotazu je jeden sloupec hodnot
- Tabulka - obecně jakýkoli výsledek SELECT příkazu s více sloupci

Pokud víme, že dotaz bude vrace vždy skalární hodnotu, pak jej v dotazu můžeme jednoduše použít. Místo konkrétní hodnoty v dotazu vepíšeme do závorek dotaz jenž vrací skalární hodnotu. Příklad 4.9(a) ukazuje poddotaz se skalární hodnotou.

Poddotazy dále můžeme rozdělit podle toho, zda-li se odkazují nebo neodkazují na vnější dotaz:

- Nezávislé poddotazy - poddotaz se dá spustit odděleně od vnějšího dotazu.
- Závislé poddotazy - poddotaz se odkazuje na hodnoty vnějšího dotazu a nedá se spustit odděleně.

Nezávislý poddotaz vidíme v příkladu 4.9(a) a závislý poddotaz v příkladu 4.9(b).

Příklad 4.9. (`SELECT` a skalární dotazy).

(a) Vypište studenty s nejvyšším rokem narození.

```
SELECT * FROM student
WHERE rok_narozeni =
   (SELECT MAX(rok_narozeni) FROM Student)
```

Poddotaz používá agregační funkci MAX, která vrací maximální hodnotu atributu ze vstupní množiny (viz. podkapitola agregační funkce). V tomot případě slouží velmi dobře k tomu, abychom dosáhli skalárního výsledku poddotazu.

(b) Vypište studenty, kteří studovali v roce 2010 více než jeden předmět.

```
SELECT * FROM Student st
WHERE
(
  SELECT COUNT(*)
   FROM Studuje se
   WHERE se.sID = st.sID AND rok = 2010
) > 1
```

Poddotaz používá agregační funkci COUNT, která vrací počet záznamů. Tentokrát poddotaz samostatně nespustíme, jelikož se používá `sID` studenta vnějšího dotazu. Tedy pro každého studenta vyhodnotíme vnitřní dotaz a pokud je výsledek větší něž jedna, je daný student ve výsledku. Dále uvidíme, že podobný dotaz lze mnohem jednodušeji řešit s pomocí konsturkce HAVING.

Použití poddotazů se skalárním výsledkem je sice jednoduché, nicméně někdy potřebujeme využít poddotazy vracející celé množiny. K tomu můžeme využít konstrukce schopné pracovat s množinami: `IN`, `EXISTS`, `ANY`, `ALL` a `SOME`, které se mohou objevit v podmínce dotazu.

Začneme konstrukcí IN. Dotaz:

```
SELECT * FROM R WHERE R.b IN (seznam hodnot)
```

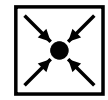
pro každý řádek relace `R` se ověřuje, zda-li je hodnota atributu `b` v seznamu hodnot. Seznam hodnot obvykle dostáváme nějakým poddotazem. Poddotaz tentokrát nemusí vracet jen skalární hodnostu, ale i množinu hodnot. Nejčastěji bude poddotaz vracet hodnoty atributů, který s atributem `R.b` nějak souvisí.

Např. půjde o cizí klíč na `R.b`, nebo naopak `R.b` bude cizí klíč na atribut, který bude vracet poddotaz. Častou chybou jsou dotazy typu:

```
SELECT * FROM Student sID IN (SELECT pID FROM Studuje)
```

V tomto příkladu porovnáváme dva zcela nesouvisející atributy `sID` a `pID` a na nich zakládáme porovnání množin. Výsledek bude zcela náhodný.

Příklad 4.10. (`SELECT` a konstrukce `IN`).



(a) Vypište jména všech studentů, kteří v roce 2010 studovali nějaký předmět.

```
SELECT jmeno FROM Student
WHERE sID IN
(
  SELECT sID FROM Studuje WHERE rok = 2010
)
```

Podmínku lze interpretovat takto jako výběr studentů jejichž sID se nachází v seznamu hodnot, jenž jsou v závorce. Pozorný posluchač si jistě všiml, že se dotaz dá zapsat i pomocí spojení:

```
SELECT jmeno FROM Student, Studuje
WHERE Student.sID = Studuje.sID AND rok = 2010
```

Nicméně v případě, že student má v roce 2010 zapsáno více předmětů, objeví se jeho jméno ve výsledku vícekrát. Je tedy nutné do příkazu přidat slůvko `DISTINCT`. Také z pohledu vykonávání nemusí být oba dotazy totožné, ale rozdíl v čase bude ve většině případů zanedbatelný.

(b) Vypište pID všech předmětů, které studují pouze studenti narození po roce 1985.

```
SELECT distinct pID FROM Studuje
WHERE Studuje.pID NOT IN
(
  SELECT distinct pID FROM Student, Studuje
  WHERE Student.sID = Studuje.sID AND
  (rok_narozeni <= 1985 OR rok_narozeni IS NULL)
)
```

V tomto případě se jedná o alternativní zápis zadání z příkladu 4.7(c).

Další konstrukcí pro práci s poddotazy vracející množinu, či dokonce tabulku je `EXISTS`:

```
SELECT * FROM R WHERE EXISTS (poddotaz)}
```

Pro každý řádek relace `R` se ověřuje, zda-li poddotaz vrací nějaký výsledek (je jedno jaký). Poddotaz musí využívat hodnoty z relace `R` jinak použití `EXISTS` nemá smysl! Jinými slovy poddotaz by měl být závislý. Tedy pokud napíšeme dotaz typu:

```
SELECT * FROM Student S1 WHERE EXISTS
(
  SELECT * FROM Student S2
  WHERE S2.jmeno = 'Petr'
)
```

Pak každý řádek relace `Student` bude podmínku splňovat (pokud poddotaz něco vrací) a výsledek bude stejný jako u dotazu `SELECT * FROM Student`.

Příklad 4.11. (`SELECT` a konstrukce `EXISTS` a `ALL`).

(a) Vypište všechny předměty, pro něž existuje nějaký další předmět vyučovaný ve stejném ročníku.

```
SELECT * FROM Predmet P1
WHERE EXISTS
(
  SELECT * FROM Predmet P2
  WHERE P1.rocnik = P2.rocnik AND
  P1.pID <> P2.pID
)
```

(b) Vypište nejstaršího studenta.

```
SELECT * FROM Student S1
WHERE NOT EXISTS
(
  SELECT * FROM Student S2
  WHERE S1.rok_narozeni < S2.rok_narozeni
) AND rok_narozeni IS NOT NULL
```

U tohoto zadání si ukážeme ještě další možné řešení s využitím konstrukce ALL:

```
SELECT * FROM Student S1
WHERE S1.rok_narozeni <= ALL
(
```

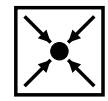
```
    SELECT
    S2.rok_narozeni FROM Student S2
    WHERE S2.rok_narozeni IS NOT NULL
)
```

Poddotazy je možné uvádět i za klausuli `FROM`. Může se opět jednat o libovolný typ poddotazu a návratová hodnota může být i tabulka. S výsledkem tohoto poddotazu je pak zacházeno jako s normální tabulkou, která je pojmenována aliasem uvedeným za závorkami. Ve výpisu 4.3 je tímto aliasem `TabulkaP`. Tyto dotazy vnořené za FROM se někdy nazývají vnořené pohledy (angl. inline views) nebo tabulkové výrazy (angl. table expressions).

```
SELECT Atr1, ..., AtrN
FROM Tabulka1, ..., TabulkaN,
(
  SELECT ... FROM ...
) Alias
WHERE Podmínka
```

Výpis 4.3: SELECT příkaz s poddotazem za FROM

Příklad 4.12. (SELECT s poddotazem za FROM).



(a) Vypište předměty, které studovaly studenti, jenž studovali nějaký předmět v roce 2010. Jinak řečeno mějme množinu studentů S, kteří studovali nějaký předmět v roce 2010. Nalezněme předměty, které někdy studoval nějaký student z množiny S.

```
SELECT distinct P.pID, P.jmeno
FROM Predmet P, Studuje Se,
(
  SELECT distinct St.sID, jmeno, rok_narozeni
  FROM Student St, Studuje Se
  WHERE St.sID = Se.sID AND Se.rok = 2010
) S
WHERE P.pID = Se.pID AND Se.sID = S.sID
```

Řešení se v podstatě skládá ze dvou dotazů, které je rozumné řešit odděleně. Nejprve se tedy důležité správně odladit poddotaz za FROM a jakmile jsme si jisti, že je výsledek správně, můžeme dopsat i vnější dotaz. Jak už to u SQL bývá možných řešení je více a tento dotaz je možné zapsat velmi podobně s využitím konstrukce IN.

Poslední typ jsou poddotazy, který zmíníme jsou skalární poddotazy uvedené za `SELECT`. SQL umožňuje místo atributu uvést celý vnořený `SELECT` dotaz jak to naznačuje výpis 4.4. Jak bylo uvedeno musí se jedná o poddotaz se sklární návratovou hodnotou. U takového `SELECT` dotazu si pak můžeme představit, že se vykoná nejprve dotaz bez poddotazů a pro každý řádek výsledku se vykoná poddotaz(y) a výsledná hodnota se připojí jako nový sloupec k výsledku. Pochopitelně ve skutečnosti se ale takový dotaz vykoná obvykle jinak, což ale není v tuto chvíli podstatné.

```
SELECT Atr1, ...
(
  SELECT ... FROM ...
) Sloupec
FROM Tabulka1, ..., TabulkaN
WHERE Podmínka
```

Výpis 4.4: SELECT příkaz s poddotazem za SELECT

Příklad 4.13. (SELECT s poddotazem za SELECT).

(a) Vypište předměty vyučované ve druhém ročníku a u každého vypište rok narození nejmladšího studenta, který daný předmět studoval v roce 2010.

Pro řešení použijeme agregační funkce popsané v kapitole 4.3.1. Řešení by se mohlo bez těchto funkcí obejít, ale bylo by výrazně složitější. Řešení bude mít dvě části: (1) nalezení předmětů druhé ročníku a (2) nalezení roku narození nejmladšího studenta, který nějaký předmět studoval v roce 2010. Dotaz, který řeší druhou část pro předmět s `pID` jedna může vypadat takto:

```
SELECT MIN(rok_narozeni)
FROM Student St, Studuje Se
WHERE St.sID = Se.sID AND Se.pId = 1 AND Se.rok =
    2010
```

Toto v podstatě bude tvořit poddotaz, pouze místo konkrétní hodnoty bude hodnoty pID vnějšího dotazu, který nachází předměty druhé ročníku.

```
SELECT pID, jmeno,
(
  SELECT MIN(rok_narozeni)
  FROM Student St, Studuje Se
  WHERE St.sID = Se.sID AND Se.pId = P.pID
```

```
    AND Se.rok = 2010
) nejmladsi
FROM Predmet P
WHERE rocnik = 2
```

Spojení tabulek

Spojování tabulek jsem představili už v příkladu 4.4(f) a dále jsme spojování více méně intuitivně používali. Nicméně existuje více možností jak takové spojení tabulek zapsat a dokonce existuje více typů spojení:

- Vnitřní spojení - to jsme používali dosud
- Vnější spojení - představíme v této podkapitole

Nejprve začneme vnitřním spojením. Představíme si možné syntaxe pro zápis vnitřního spojení:

- `SELECT * FROM R, S WHERE R.j = S.j`
  Tento zápis jsme používali doposud
- `SELECT * FROM R INNER JOIN S ON R.j = S.j`
  Tato varianta odpovídá operaci theta spojení $\bowtie_{R.j=S.j}$ uvedené v kapitole 4.2.2. Jedná se o nejčastější způsob zápisu vnitřního spojení.
- `SELECT * FROM R INNER JOIN S USING(j)`
  Opět zápis odpovídající theta spojení $\bowtie_{R.j=S.j}$. Vyžaduje, aby měly atributy dle kterých spojujeme stejné jméno v obou tabulkách. Tato syntaxe je méně využívána než předchozí varianta a není ani podporována všemi SŘBD (např. SQL Server ji nepodporuje).
- `SELECT * FROM R NATURAL JOIN S`
  Tento zápis je ekvivalentní operaci přirozené spojení $\bowtie$. Připomeňme si, že u přirozeného spojení spojujeme dle všech atributů se stejným jménem. To může být trochu problém když například provádíme změny ve schématu jelikož takto můžeme 'nechtěně' změnit i výsledek některých dotazů. Také není podporováno SQL Serverem.

Pokud používáte konstrukci s `INNER JOIN`, pak lze ve většině SŘBD slůvko INNER vynechat, jelikož se jedná o implicitní možnost.

Nyní vysvětlíme rozdíl mezi vnitřním a vnějším spojením. Řekněme že budeme chtít zjistit počet žáků na jednotlivých předmětech. Pokud provedeme vnitřní spojení tabulek `Předmět` a `Studuje` a projekci na atributy `pID`, `sID` a `rok` tak získáme trojice, kde nám stačí pro každý předmět spočítat počet trojic, kde se dané `pID` objevuje. Nicméně co předměty, které žádného studenta nemají? Tyto předměty z nebudou ani v jedné trojici, takže bychom z výsledku vnitřního spojení ani nezjistili, že tyto předměty existují. To můžeme vyřešit právě vnějším spojením. Pokud provedeme vnější spojení dvou tabulek, pak záznamy z jedné tabulky, které nemají žádný odpovídající v druhé tabulce, nejsou z výsledku odstraněny, ale jsou ponechány a atributy druhé tabulky jsou doplněny NULL hodnotami.

U vnějšího spojení navíc můžeme definovat ze které tabulky chceme ponechávat nespárované záznamy:

- `FULL OUTER JOIN` - z obou tabulek.
- `LEFT OUTER JOIN` - z levé tabulky.
- `RIGHT OUTER JOIN` - z pravé tabulky.

Příklad 4.14. (Vnější spojení).

(a) Vypište jména studentů a pID jimi studovaných předmětů, přičemž u studentů, kteří nestudují či nestudovali žádný předmět, vypište NULL.

```
SELECT jmeno, pID
FROM Student
LEFT OUTER JOIN Studuje
  ON Student.sID = Studuje.sID
```

V tomto případě můžeme dát i FULL OUTER JOIN a výsledek se nezmění. Je to dáno tím, že každý záznam v tabulce `Studuje` musí mít odpovídající záznam v tabulc `Student`.

(b) Vypište jména studentů a pID jimi studovaných předmětů v roce 2011, přičemž u studentů, kteří v tomto roce nestudují či nestudovali žádný předmět, vypište NULL.

```
SELECT jmeno, pID
FROM Student
LEFT JOIN Studuje ON Student.sID = Studuje.sID
WHERE rok = 2011
```

Pokud v tomto případě chceme dosáhnout správného výsledku s použitím vnějšího spojení musíme řešit i pořadí provádění operací. Při vykonávání SŘBD začíná vykonáváním klausulí FROM a JOIN. Teprve potom se vykonává WHERE. V řešení které jsme tady uvedli se tedy nejprve provede vnější spojení a dostaneme mezivýsledek shodný s předchozím zadáním. Nicméně v dalším kroku se provede WHERE a vyberou se pouze ty záznamy, které mají v atributu rok hodnotu 2011. Z toho je jasné, že ve výsledku určitě budou pouze ti studentu kteří v roce 2011 něco studují. Což neodpovídá zadání. Správné řešení musí zařadit podmínku rok = 2011 do theta spojení:

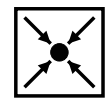
```
SELECT jmeno, pID
FROM Student
LEFT JOIN Studuje
ON Student.sID = Studuje.sID AND rok = 2011
```

Agregační funkce

V SQL můžeme použít i **agregační funkce**, které jsou uvedeny v tabulce 4.3. Parametry těchto funkcí jsou atributy. Výjimkou je funkce `COUNT` jejímž parametrem může být znak `*`, kterým si říkáme o vrácení počtu n-tic včetně duplicit a řádků obsahujících pouze hodnoty `NULL`. Úplný tvar parametru agregačních funkcí je:
`<funkce>([ALL | DISTINCT] <jmeno_atributu>)`.

Pokud použijeme `DISTINCT` nebudou do výsledku počítány duplicitní hodnoty, pokud použijeme `ALL` budou do výsledku zahrnuty i duplicity. Funkce `COUNT(ALL A)` vrátí počet hodnot ve sloupci `A`. V případě funkcí `COUNT`, `SUM` a `AVG` se hodnoty `NULL` ignorují. Agregační funkce se může objevit v konstrukci `SELECT` a také v podmínce za `HAVING` (viz. podkapitola o seskupení záznamů v SQL).

Příklad 4.15. (Dotazy v SQL používající agregační funkce).



(a) Vypište průměrný rok narození studentů.

```
SELECT AVG(rok_narozeni) FROM Student
```

(b) Vypište rok narození nejstaršího studenta.

```
SELECT MIN(rok_narozeni) FROM Student
```

Tabulka 4.3: Agregační funkce v SQL

Agregační funkce	Popis
COUNT	Počet hodnot ve sloupci
SUM	Součet hodnot ve sloupci
MAX	Maximální hodnota ve sloupci
MIN	Minimální hodnota ve sloupci
AVG	Průměrná hodnota ve sloupci

Výsledek můžeme porovnat s příkladem 4.11(b). Řešení používající funkci `MIN` nám vrací pouze rok narození daného studenta, ale další informace už nemáme. Nevíme jeho sID ani jméno. Pokud bychom chtěli výsledek ekvivalentní s řešením používajícím `EXISTS` mohli bychom dotaz rozšířit například takto:

```
SELECT * FROM Student
WHERE rok_narozeni =
(
  SELECT MIN(rok_narozeni) FROM Student
)
```

(c) Jaký je rozdíl mezi následujícími dvěma dotazy?

```
SELECT COUNT(*) FROM Student
SELECT COUNT(rok\_narozeni) FROM Student
```

Je potřeba si uvědomit, že `COUNT` nezapočítává řádky, které obsahuje pouze NULL hodnoty. Zatímco první dotaz vrací počet záznamů v tabulce `Student` (zde žádný řádek nemůže mít jen NULL hodnoty), tak druhý dotaz vrací informaci o tom, kolik studentů má uveden rok narození.

(d) Vypište průměrný rok narození studentů, kteří studují předmět s pID 6. Nejprve se zkusme zamyslet nad řešením které vypadá následovně:

```
SELECT AVG(rok_narozeni)
FROM Student St, Studuje Se
WHERE St.sID = Se.sID and pID = 1
```

Na první pohled se zdá vše v pořádku. Vybíráme studenty, kteří studují předmět s pID 6 a následně z jejich roku narození počítáme průměr. Problémem tohoto řešení jsou duplicity, která nám vzniknou po spojení tabulek. Pokud se podíváme na mezivýsledek takového spojení dostaneme tabulku 4.4.

Tabulka 4.4: Tabulka student studuje předmět s pID 1

sID	rok_narození
1	1986
1	1986
3	1980
4	1983
5	1985
6	1986
6	1986
7	1985
7	1985

V tabulce nalezneme některé studenty vícekrát protože tito studenti ročník opakovali. V důsledku je u těchto studentů jejich rok narození započítán do výsledku vícekrát což ovlivňuje konečný výsledek. Když si toto uvědomíme mohli bychom mít nutkání dát před atribut `rok_narozeni` slůvko DISTINCT, které eliminuje duplicity. V našem případě by to znamenalo, že zůstanou čtyři různá léta (1980, 1983, 1985, 1986), ale studentů je šest. Tedy oproti prvnímu řešení, kde bylo mnoho hodnot, máme naopak hodnot méně. Na tomto příkladu je vidět při použití agregačních funkcí musíme dát pozor zejména na duplicity, které mohou výsledek ovlivnit. Zde je jendo z možných správných řešení:

```
SELECT AVG(rok_narozeni)
FROM Student
WHERE sID IN
(
   SELECT sID
   FROM Studuje
   WHERE pID = 1
)
```

Seskupení záznamů

Agregační funkce představené v předchozí podkapitole se aplikují na celou množinu záznamů. Nicméně někdy je výhodné aplikovat agregační funkce pouze na jednotlivé podmnožiny relace. Kupříkladu nechceme spočítat počet všech studentů, ale třeba počet studentů studujících jednotlivé předměty. V takovém případě je

nutné záznamy nejprve seskupit do jednotlivých podmnožin. U našeho příkladu by se jednalo o seskupení podle předmětu.

```
SELECT atr1, ..., atrN, agregační funkce
FROM ...
GROUP BY atr1, ..., atrN
HAVING podm s agregačními funkcemi
```

Výpis 4.5: Seskupení záznamů v příkazu `SELECT`

Ve výpisu 4.5 vidíme další dvě konstrukce příkazu `SELECT`. K seskupení slouží konstrukt `GROUP BY`, který rozdělí záznamy na podmnožiny. V každé podmnožině mají všechny záznamy stejné hodnoty u atributů `atr1` až `atrN`. Jak je vidět atributů, dle kterých chceme provádět seskupení, může být více než jeden. Obecně platí, že přidáním dalšího atributu za `GROUP BY` vytváříme více menších množin.

Dalším konstruktem, který souvisí se seskupováním a agregačními funkcemi je `HAVING`. V podmínce za `WHERE` se nemohou používat agregační funkce. Jinak řečeno v podmínce za `WHERE` nemůžeme provádět selekci dle výsledků agregovaných funkcí. Možné to je jen v podmínce, kterou uvedeme za slovem `HAVING`. Tedy za `HAVING` můžeme aplikovat podmínku na vytvořené skupiny s využitím agregačních funkcí. Je to dáno pořadím vyhodnocování konstruktů `SELECT`. `WHERE` se vyhodnocuje před `GROUP BY`, zatímco `HAVING` až po něm.

Příklad 4.16. (Seskupení v příkazu `SELECT`).

(a) Vypište počet studentů narozených v jednotlivých letech.

```
SELECT rok_narozeni, COUNT(*)
FROM Student
GROUP BY rok_narozeni
```

Ve výsledku vidíme pouze dvojice (rok, vypočítaný počet studentů). Ze stejného důvodu jako u agregačních funkcí (možnost vzniku duplicit) není někdy špatné ověřit správnost výsledku na datech před provedením seskupení. Když spustíme dotaz:

```
SELECT rok_narozeni, jmeno
FROM Student
ORDER BY rok_narozeni
```

uvidíme tabulku před seskupením a pro každý rok můžeme ručně spočítat počet studentů a porovnat se skutečným výsledkem.

(b) Vypište počet studentů pro jednotlivé předměty v jednotlivých letech.

```
SELECT P.pID, rok, COUNT(*)
FROM Predmet P, Studuje S
WHERE P.pID = S.pID
GROUP BY P.pID, rok
```

(c) Zkuste nyní provést následující dotaz a zamyslet se co je na něm špatně:

```
SELECT jmeno, rok, COUNT(*)
FROM Predmet P, Studuje S
WHERE P.pID = S.pID
GROUP BY P.pID, rok
```

Při provedení dotazu v Oracle a SQL Serveru dojde k chybě. Problém je v tom, že za `SELECT` se mohou mimo agregační funkce vyskytovat pouze atributy, které jsou za `GROUP BY`. Dokonce ani v tomto případě, kdy jméno výrobku je prakticky zaměnitelné za pID, SŘBD nedovolí provedení dotazu. Nicméně toto chování není stejné ve všech SŘBD. Kupříkladu MySQL takovýto provede a dokonce vykoná i následující dotaz:

```
SELECT P.pID, rok, S.sID, COUNT(*)
FROM Predmet P, Studuje S
WHERE P.pID = S.pID
GROUP BY P.pID, rok
```

Problém je ve výpisu S.sID u dané skupiny. Pokud se nad dotazem zamyslíme, může daný předmět v daném roce studovat více různých studentů. Nicméně řádek s daným pID a rokem je jen jeden, takže je otázka jaké sID bude vráceno? Odpověď je, že zcela náhodné sID z dané podmnožiny studentů studující daný předmět. Toto chování MySQL není úplně vhodné, protože méně zkušení uživatelé si nemusí být vědomi toho, že dostávají náhodný výsledek.

(d) Vypište jména všech předmětů spolu s počty studentů, kteří daný předmět studují či studovali. (Pozor! Student může předmět studovat opakovaně.)

```
SELECT jmeno, COUNT(DISTINCT sID)
FROM Predmet P, Studuje S
WHERE P.pID = S.pID
GROUP BY jmeno
```

Na první pohled správné řešení má jednu vadu a to že nevypíše předměty, které nemají ani jednoho studenta. Pokud jste pozorně četli kapitolu 4.3.1

možná vás napadne, že to je způsobeno použitím vnitřního spojení. Pokud tedy chceme všechny předměty jak je uvedeno v zadání, pak je nutné použít vnější spojení, které problém vyřeší:

```
SELECT jmeno, COUNT(DISTINCT sID)
FROM Predmet P
LEFT JOIN Studuje S ON P.pID = S.pID
GROUP BY jmeno
```

(e) Vypište jména všech předmětů spolu s počty studentů, kteří daný předmět studují či studovali v roce 2010.

```
SELECT jmeno, COUNT(DISTINCT sID)
FROM Predmet P
LEFT JOIN Studuje S ON P.pID = S.pID
WHERE rok = 2010
GROUP BY jmeno
```

Podobné zadání jsme měli v příkladu 4.14(b), kde jsme podrobněji popisovali problém s uvedením podmínky `WHERE` po vnějším spojení. Správné řešení musí uvést podmínku `rok = 2010` jako součást vnějšího spojení:

```
SELECT jmeno, COUNT(DISTINCT sID)
FROM Predmet P
LEFT JOIN Studuje S ON P.pID = S.pID AND rok =
    2010
GROUP BY jmeno
```

(f) Vypište pID všech předmětů, které studují nebo studovali alespoň dva studenti.

```
SELECT pID FROM Studuje
GROUP BY pID
HAVING Count(distinct sID) > 1
```

Klausule `HAVING` především usnadňuje a zpřehledňuje zápis takovýchto dotazů, ale dá se nahradit například takto:

```
SELECT distinct pID
FROM Studuje S1
WHERE 1 <
(
  SELECT Count(distinct sID)
  FROM studuje S2
```

```
    WHERE S1.pID = S2.pID
)
```

Pořadí vykonávání konstrukcí příkazu SELECT

Několikrát jsme zmínili, že jednotlivé konstrukce příkazu `SELECT` jsou vyhodnocovány v určitém pořadí. Zároveň jsme ale na začátku psali, že `SELECT` je deklarativní jazyk, který neříká nic o tom jakým způsobem se má dotaz vykonat, ale pouze specifikuje výsledek. Nicméně jak jsme si na příkladech ukázali výsledek je ovlivněn i pořadím vykonání jednotlivých konstrukcí, takže je důležité chápat jaké je toto pořadí. SŘBD pak ve skutku nemusí vyhodnocovat přesně v tomto pořadí, ale **výsledek musí odpovídat vykonání konstruktů v logickém pořadí**. To logické pořadí vykonání konstrukcí je:

1. FROM
2. JOIN
3. WHERE
4. GROUP BY
5. HAVING
6. SELECT
7. ORDER BY

Konstrukce `JOIN` se v seznamu někdy neuvádí, jelikož `JOIN` je v podstatě součástí konstrukce `FROM`.

4.3.2 Datové typy v SQL

Standard SQL obsahuje datové typy atributů. Jak již bylo uvedeno výrobci SŘBD definované datové typy často rozšiřují popř. modifikují. Pokusíme se tedy uvést informace s ohledem jak na standard tak na jednotlivé SŘBD.

Datové typy můžeme rozdělit do několika základních kategorii:

- Celá čísla - tento datový typ bychom měli použít vždy, když plánujeme ukládat přesné číselné hodnoty jejichž rozsah dokážeme předem stanovit.

- Reálná čísla s desetinnou čárkou - pro uložení reálných čísel, kde se mohou vyskytnout i velmi malá, či velmi velká čísla a nevadí nám určitá nepřesnost při uložení.
- Řetezce - sekvence znaků.
- Datum a čas - datový typ pro uložení datumových a časových hodnot.
- Binární řetězce (velké objekty) - pro uložení velkých objektů jako jsou obrázky, signály, či velké dokumenty.

Správný výběr datových typů je klíčový pro veškerou další práci s databázi. Nesprávný výběr datového typu se může vést ke vzniku mnoha zbytečných chyb, či dokonce k implementaci funkcionality, která již byla implementována v SŘBD. Příkladem může být uložení datumu v celočíselných atributech nebo dokonce v řetězci. Takovéto řešení může být z počátku pohodlné, ale problém začne vznikat ve chvíli, kdy potřebujeme vybírat záznamy z určitého časového intervalu. To by vedlo ke komplikovaným konverzím a dokonce chybám.

Vzhledem k mnoha rozdílům mezi jednotlivými SŘBD ve způsobu jakým řeší datové typy vždy konzultujte specifika daného SŘBD s manuálovými stránkami.

Číselné datové typy

Jsou dva základní číselné datové typy:

- Přesné
- Aproximované

Je důležité chápat, že aproximované datové typy nám nemusí uložit číslo přesně tak jak jej ukládáme, ale může dojít k malé odchylce. Mezi tyto datové typy patří `FLOAT`, `REAL` a `DOUBLE`.

- Celočíselné datové typy jsou implementovány jako různě dlouhá binární čísla. Rozlišujeme tyto datové typy:
  - `SMALLINT` – celé číslo kratší než `INTEGER`, typicky 16b,
  - `INTEGER` – typicky 32b, můžeme použít zkratku `INT`.
- Reálné datové typy s pevnou řadovou čárkou:

  - `DECIMAL(p,q)` – reálné číslo s p ciframi a desetinou čárkou q cifer zprava. Pokud $q = 0$ můžeme tento parametr vynechat. Můžeme použít zkratku `DEC`.
  - `NUMERIC` – podobné jako `DECIMAL`, rozdíly závisí na implementaci.

- Reálné datové typy s pohyblivou řadovou čárkou:
  - `FLOAT(p)` – reálné číslo s pohyblivou řadovou čárkou s přesností p,
  - `REAL`
  - `DOUBLE` – číslo delší než REAL.

Řetězcové datové typy

Standard definuje, že řetězce mohou být

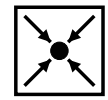
- pevné délky, `CHARACTER(n)` – řetězec délky n, menší řetězce jsou zprava doplněny prázdnými znaky. Můžeme použít zkratku `CHAR`.
- proměnlivé délky, `CHARACTER VARYING(n)` – řetězec s maximální délkou n, tj. řetězce nejsou zprava doplněny prázdnými znaky. Můžeme použít zkratku `VARCHAR`.

Oba typy řetězců mohou mít své výhody a úskalí. Pevná délka řetězce nám snižuje dobu aktualizace řetězce. Pokud máme proměnlivou délku a provedeme aktualizaci kde dojde k rozšíření hodnoty řetězce, pak SŘBD musí nejprve posunout všechny údaje, které se ve stránce nacházejí za aktualizovanou hodnotou. Je tedy nutné nejprve udělat místo pro vložení. Na druhou stranu použití proměnlivé délky může zásadním způsobem zmenšit objem uložených dat v databázi. Menší objem dat znamená menší počet načtených stránek při zpracování dotazu což může mít za následek urychlení dotazu.

Z praktických důvodů SŘBD řeší i kódování řetězce. Proto Oracle i SQL Server implementují unicode řetězcové datové typy, které nejsou součástí standardu. Tabulka 4.5 obsahuje přehled jednotlivých řetězcových datových typů, který je platný pro obě databáze. Oracle má navíc datový typ `VARCHAR2`, který je synonymem `VARCHAR` a který Oracle doporučuje používat.

Mezi běžné operace s řetězci patří jejich zřetězení. Zatímco SQL Server používá pro zřetězení operátor +, Oracle používá operátor ||.

Příklad 4.17. (Práce s řetězci).



	pevná délka	proměnlivá délka
8-bitový kód	`CHAR[n]`	`VARCHAR[n]`
UNICODE	`NCHAR[n]`	`NVARCHAR[n]`

Tabulka 4.5: Řetězcové datové typy

(a) Proveďte následující skripty a zkuste říct proč příkaz SELECT nic nevrací:

```
CREATE TABLE Osoba
(
  jmeno CHAR(20),
  prijmeni CHAR(20)
);
INSERT INTO Osoba VALUES ('Petr', 'Hudecek');
SELECT * FROM Osoba WHERE jmeno LIKE 'Petr';
```

Skript má tři kroky: vytvoření tabulky, vložení do tabulky a vyhledání vloženého záznamu. Pokud skript spustíme v Oracle dostaneme prázný výsledek. Pokud jej spustíme v SQL Serveru výsledek bude jeden záznam. V tomto případě je na vině operátor LIKE, který v Oracle porovnává hodnoty přesně. Používáme pevnou délku řetězce, takže dle definice je při vkládání hodnota 'Petr' doplněna o 16 prázdných znaků. Po vložení je tedy hodnota 'Petr '. Naštěstí Oracle (i SQL Server) umožňuje porovnávat řetězce pomocí operátoru =. Vyhledání Petra v Oracle může vypadat takto:

```
SELECT * FROM osoba WHERE jmeno = 'Petr';
```

Datové typy pro datum a čas

Časové datové typy jsou dvou typů:

- zohledňující časové pásmo
- bez informace o časovém pásmu

Datové typy bez časového pásma jsou:

- `DATE` - ukládá rok, měsíc a den.
- `TIME` - ukládá hodinu, minutu a sekundu.

- `TIMESTAMP` - ukládá rok, měsíc, den, hodinu, minutu a sekundu. Jedná se tedy o kombinaci DATE a TIME

SQL Server používá místo `TIMESTAMP` klíčové slovo `DATETIME2`.

Datové typy s časovými pásmy jsou:

- `TIME WITH TIME ZONE` - rozšíření datového typu `TIME` o časové pásmo
- `TIMESTAMP WITH TIME ZONE` - rozšíření datového typu `TIMESTAMP` o časové pásmo

SQL Server nabízí pro uložení informace o časovém pásmu jediný datový typ `DATETIMEOFFSET`.

Častým problémem je formát data nebo času, který se snažíme uložit do databáze. Textových formátů pro uložení dat a času je celá řada a při vkládání s pomocí INSERT máme dvě možnosti: (1) buď použijeme výchozí formát pro danou databázi nebo (2) explicitně řekneme jaký formát pro vložení použijeme. V příkladu 4.18 jsou obě varianty rozebrány pro SQL Server.

Příklad 4.18. (Vkládání datumu). Výchozím formátem datumu je u SQL Serveru YYYY-MM-DD, ale bez problémů je schopen správně interpretovat i jiné formáty jako DD/MM/YYYY. Pro explicitní konverzi datumu se používá funkce `CONVERT` nebo `PARSE`. Mějme následující tabulku s osobami:

```
Osoba(id : INT, jmeno : VARCHAR, narozen : DATE)
```

(a) Vložte záznam do tabulky `Osoba` se jménem Petr a datumem narození 1.2.2000

```
INSERT INTO Osoba VALUES (1, 'Petr', '2000-02-01')
```

Pokud si nejsme jisti formátem, který by SQL Server správně interpretoval tak můžeme využít funkci DATEFROMPARTS.

```
INSERT INTO Osoba VALUES (1, 'Petr',
  DATEFROMPARTS(2000, 2, 1))
```

Pro jiné datové typy jako DATETIME2, či TIME jsou k dispozici DATETIME2FROMPARTS, či TIMEFROMPARTS.

(b) Vložte záznam do tabulky `Osoba`, kde datumová hodnota bude mít tvar DD/MM/YY.

```
INSERT INTO Osoba VALUES (2, 'Jakub', '01/02/00')
```

SŘBD	Datové typy
SQL Server	`BINNARY`, `VARBINNARY`, `IMAGE`
Oracle	`CLOB`, `NCLOB` a zastaralý `LONG`
MySQL	`TEXT`, `BLOB` a jeich varinaty s prefixem `MEDIUM` a `LONG`

Tabulka 4.6: SŘBD a jejich datové typy pro velká data

Takovéto řešení povede k výjimce, jelikož SQL Server není schopen datum správně interpretovat. Abychom vložili datum v tomto formátu je nutné formát explicitně specifikovat. To můžeme provést použitím funkce `CONVERT(datový_typ, hodnota, styl)`, kde styl umožňuje specifikovat požadovaný formát. Seznam formátu nalezneme zde: http://msdn.microsoft.com/en-us/library/ms187928.aspx. Styl, který je specifikován v zadání, má kód 3.

```
INSERT INTO Osoba VALUES (3, 'Jakub',
  CONVERT(DATE, '01/02/00', 3))
```

Datové typy pro velké objekty

Maximální velikost dat uložených v základních datových typech je ve všech SŘBD omezena. Kupříkladu `VARCHAR` v MySQL může mít maximálně 255 znaků, v SQL Serveru 8000 znaků a v Oracle 4000 znaků. Občas ale potřebujeme uložit objekt, pro který je taková velikost nedostačující. Typicky se může jednat o nějaký signál, obrázek, či velký textový dokument.

V tomto případě je situace u jednotlivých SŘBD zcela odlišná. Tabulka 4.6 uvádí několik SŘBD spolu se seznamem datových typů používaných pro ukládání velkých dat. Pro informace vztahující se konkrétní SŘBD čtenáře laskavě odkazujeme na manuálové stránky SŘBD.

Konverze mezi hodnotami

V SQL jsou definovány implicitní konverze mezi hodnotami různých datových typů, navíc můžeme použít funkci `CAST(výraz AS datový_typ)` pro explicitní konverzi.

Příklad 4.19. (Explicitní konverze s pomocí `CAST`).

(a) Vypište průměrný rok narození studenta.

```
SELECT AVG(CAST(rok_narozeni as FLOAT))
FROM Student
```

Toto zadání už jsme mohli vidět v příkladu 4.15(a), kde jsme datové typy ještě příliš neřešili. Nicméně agregační funkce typicky vracejí výsledek v tom datovém typu, který má výraz, jenž použijeme jako parametr. Abychom tedy ve výsledku neměli výsledek zaokrouhlený na celé číslo je dobré nejprve přetypovat na desetinné číslo.

Funkce v SQL

Podobně jako procedurální jazyky nabízí i SQL celou řadu funkcí, pro práci s hodnotami jednotlivých datových typů. Je mimo možnosti této knihy nějak podrobněji jednotlivé funkce rozebírat. Pouze čtenáře odkazujeme na manuálové stránky SŘBD, který se chystáte použít. Mezi funkcemi nalezneme funkce pro manipulaci s hodnotami různých datových typů, ale také i systémové funkce, které vracejí například systémový čas. Funkce lze používat jak v projekci příkazu SELECT, tak v podmínkách, či v DML operacích, kde pracujeme s hodnotami.

4.3.3 Definice dat v SQL

Základní příkazy pro práci se schématy tabulek jsou:

- `CREATE TABLE` -– vytvoření tabulky,
- `ALTER TABLE` -– změna schématu tabulky,
- `DROP TABLE` -– zrušení tabulky.

Příkaz CREATE TABLE

Příkaz `CREATE TABLE` je obvykle jeden z prvních příkazů, které spouštíme v nové databázi. Příkazem specifikujeme schéma tabulky (atributy a jejich datové typy), její integritní omezení a často je součástí příkazu i definice určitých aspektů fyzického uložení tabulky. Konstrukce související s fyzickým uložením jsou často specifické pro dané SŘBD a proto je v této kapitole záměrně zcela opomineme. Fyzickému uložení i s ohledem na jednotlivé SŘBD se pak podrobně věnuje kapitola 10.

Příkaz `CREATE TABLE` má následující tvar:

```
CREATE TABLE Tabulka (
  jmeno_atributu datový_typ [IO_atributu,]
   ...
   IO_tabulky [,]
   ...
)
```

V příkazu tedy definujeme jméno tabulky, seznam atributů a seznam integritního omezení tabulky. U každého atributu definujeme jeho integritní omezení a datový typ (což je vlastně také forma integritního omezení). Datové typy jsme rozebírali v kapitole 4.3.2, nyní se podíváme ještě na integritní omezení atributů (IO_atributu):

- `NOT NULL` – sloupec nesmí mít hodnotu NULL,
- `DEFAULT hodnota` – implicitní hodnota atributu,
- `UNIQUE` – hodnoty atributu musí být jedinečné v rámci celé tabulky. Pouze jedna hodnota ve sloupci může být NULL,
- `PRIMARY KEY` – atribut je primárním klíčem tabulky. Atribut nemůže nabývat hodnoty NULL. Pokud primární klíč obsahuje více atributů, definujeme primární klíč jako IO_tabulky a ne jako IO_atributu.
- `REFERENCES tab(atr)` – atribut je cizím klíčem a odkazuje se na atribut `atr` tabulky `tab`.
- `CHECK` – integritní oemezení je zadáno logickým výrazem. Logickou podmínkou můžeme omezit hodnotu atributu, např. pomocí `CHECK(cena >= 0.0)` zajistíme, že cena nemůže být záporná.

Příklad 4.20. (Vytvoření tabulky). Vezměme relační schéma z obrázku 4.4 a zkusme některé tabulky vytvořit.

(a) Vytvoříme nejprve tabulku `Student` kde definujeme primární klíč a specifikujeme, že atribut `jmeno` nesmí být NULL:

```
CREATE TABLE Student (
  sID int PRIMARY KEY,
  jmeno VARCHAR(20) NOT NULL,
  rok_narozeni INT
)
```

(b) Nyní vytvoříme tabulku `Studuje`, kde je primární klíč tvořen trojicí atributů `sID`, `pID` a `rok`:

```
CREATE TABLE Studuje (
  pID INT REFERENCES Predmet,
  sID INT REFERENCES Student,
  rok INT,
  body INT,
  PRIMARY KEY (pID, sID, rok)
)
```

Vidíme, že v tomto případě musel být primární klíč, složený ze tří atributů, mimo definici atributu jako integritní omezení tabulky. Zmíníme ještě že z pohledu efektivity může záležet i na pořadí atributů, jelikož se pro primární klíč obvykle implicitně vytváří index. Více v kapitole 10. V samostatné položce můžeme zapsat i integritní omezení týkající se ze cizího klíče:

```
CREATE TABLE Studuje (
  pID INT,
  sID INT,
  rok INT,
  body INT,
  PRIMARY KEY (pID, sID, rok),
  FOREIGN KEY (pID) REFERENCES Predmet,
  FOREIGN KEY (sID) REFERENCES Student
)
```

Vidíme, že v obou případech se klauzule `REFERENCES` trochu liší od definice. Tedy za jménem tabulky není v závorkách uveden atribut na který se v tabulce odkazujeme. V takovém případě se odkazujeme na primární klíč.

(c) Nakonec vytvoříme ještě tabulku `Předmět`, kde specifikujeme, že atribut `jmeno` nesmí být `NULL` a `rocnik` nesmí mít hodnotu vyšší než 5:

```
CREATE TABLE Predmet (
  pID INT PRIMARY KEY,
  jmeno VARCHAR(30) NOT NULL,
  rocnik INT CHECK(rocnik <= 5)
)
```

Příkaz `ALTER TABLE`

Příkazem `ALTER TABLE` můžeme modifikovat schéma tabulky. Příkaz může mít několik podob, kde mezi základní patří:

- přidání sloupce (`ALTER TABLE tab ADD`)
- odebrání sloupce (`ALTER TABLE tab DROP COLUMN`)
- modifikace sloupce (`ALTER TABLE tab ALTER COLUMN`)
- přidání integritního omezení tabulce (`ALTER TABLE tab ADD CONSTRAINT`)
- odebrání integritního omezení tabulce (`ALTER TABLE tab DROP CONSTRAINT`)

U modifikace sloupce se situace u jednotlivých SŘBD liší. Syntaxe `ALTER COLUMN` je používaná v SQL Serveru. Oracle a MySQL používá `MODIFY` místo `ALTER COLUMN` a verze Oracle starší než 10g používá `MODIFY COLUMN`. Příkaz `ALTER COLUMN` je jedním z příkazů, kde se jednotlivé SŘBD často liší v syntaxi. Je tedy vhodné při použití složitější syntaxe zkontrolovat manuálové stránky.

V případě příkazů `DROP` u příkazu `ALTER` můžeme použít klíčová slova `RESTRICT` a `CASCADE`. Pokud použijeme `RESTRICT` pak se zrušení nepovede, pokud je na atribut nebo integritní omezení odkazováno. Pokud použijeme `CASCADE`, dojde ke zrušení a s ním jsou zrušeny všechny objekty, které se na integritní omezení nebo atribut odkazují. Při použití `CASCADE` je tedy nutné být velmi opatrný.

Příklad 4.21. (Změna schématu tabulky). Mějme tabulky vytvořené v příkladu 4.20 a tedy tabulky odpovídající schématu z obrázku 4.4. Na příkladech si ukážeme jak by vypadalo přidání, modifikace a odebrání atributu v SQL Serveru.

(a) Přidáme do tabulky `Student` sloupec pro stipendium `NUMERIC(6,2)`.

```
ALTER TABLE Osoba
ADD stipendium NUMERIC(6,2)
```

(b) Sloupec pro plat už však nestačí, rozsah je třeba zvětšit o 2 číslice.

```
ALTER TABLE Osoba
ALTER COLUMN stipendium NUMERIC(8,2)
```

(c) Nakonec se rozhodneme, že informace o platu odporuje zákonu o ochraně osobních údajů a z databáze jej odstraníme.

```
ALTER TABLE Osoba
DROP COLUMN stipendium
```

Příkaz DROP TABLE

Odstranění tabulky z databáze provedeme pomocí `DROP TABLE`, např. `DROP TABLE Student`. Chování příkazu `DROP TABLE` může měnit pomocí klíčových slov `RESTRICT` a `CASCADE`. Funkce těchto klíčových slov jako obdobná jako u příkazu `ALTER TABLE`. `RESTRICT` zabrání odstranění tabulky pokud je někde odkazována. `CASCADE` odstraní všechny pohledy a indexy, v jejichž definicích je uvedena rušená tabulka.

Příkaz CREATE SCHEMA

Pomocí příkazu `CREATE SCHEMA jmeno_schematu seznam_objektu_schematu` vytvoříme schéma pojmenované `jmeno_schematu`. V seznamu objektů mohou být uvedeny: tabulky, pohledy, domény, globální IO, přístupová práva, procedury a triggery. Na rozdíl od konvencí relačního datové modelu, je databáze v SQL tvořena množinou tabulek a pohledů, které jsou definovány jedním nebo více schématy.

Pro zrušení schématu použijeme příkaz `DROP SCHEMA`.

Práce s indexy v SQL

Pro efektivní přístup k datům (tedy efektivnějším než je sekvenční vyhledávání) jsou určeny indexy (viz kapitola 10). Index je datová struktura, kterou vytvoříme příkazem `CREATE INDEX` a která je SŘBD využívána automaticky podle potřeby. SQL umožňuje vytvořit a zrušit index na jeden nebo více atributů. Syntaxe příkazu je následující:

```
CREATE INDEX jmeno_indexu
ON jmeno_tabulky (seznam_atributu)
```

Tento příkaz není standardizován nicméně tato podoba příkazu je používaná ve všech SŘBD. Příkaz má obvykle mnoho parametrů, které jsou už specifické pro jednotlivé SŘBD. Pokud neuvedeme jinak (nebo použijeme klíčové slovo

ASC), jsou záznamy indexovány vzestupně. Pomocí klíčového slova DESC definujeme sestupné uspořádání záznamů. Index zrušíme příkazem DROP INDEX jmeno_indexu.

Příklad 4.22. (Vytvoření indexu). Vezměme tabulku Student z příkladu 4.20. Index na atribut jmeno vytvoříme příkazem:

```
CREATE INDEX jmeno ON Student (jmeno)
```

4.3.4 Manipulace s daty v SQL

Příkazy pro manipulaci s daty zahrnují příkazy pro výběr dat a příkazy pro aktualizaci a rušení dat. Výběr dat je realizován příkazem SELECT, mezi aktualizační příkazy patří INSERT, UPDATE a DELETE.

Příkaz INSERT

Příkaz INSERT vkládá n-tice do tabulky. Obecný tvar příkazu je uveden ve výpisu 4.6. Pokud jsme u sloupce nespecifikovali NOT NULL, nemusíme hodnotu takové sloupce uvádět a hodnota v n-tici bude NULL.

```
INSERT INTO Tabulka (Atribut1, ... AtributN)
VALUES (Hodnota1, ..., HodnotaN)
```

Výpis 4.6: Příkaz INSERT

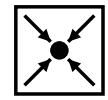
Závorky se seznamem atributů za jménem tabulky jsou v určitých případech nepovinné. Tento seznam je nepovinný pokud seznam hodnot v závorkách za VALUES odpovídá pořadí a počtu atributů tabulky. Z praktických důvodů je ale vhodné seznam atributů uvádět. Pokud například dojde k přidání nepovinného atributu do tabulky, pak všechny příkazy INSERTmající seznam atributů nad touto tabulkou zůstanou platné.

Druhou možností jak realizovat vkládání do tabulky je kombinace příkazů INSERT a SELECT jak je naznačeno ve výpisu 4.7. Výsledek dotazu musí mít stejný počet atributů jako Tabulka a také datové typy musí korespondovat.

```
INSERT INTO Tabulka (Atribut1, ... AtributN)
SELECT ... FROM ...
```

Výpis 4.7: Příkaz INSERT s příkazem SELECT

Příklad 4.23. (Vkládání do tabulky).



(a) Vložte do databáze studenta se jménem Alexandr.

```
INSERT INTO Student(sID, jmeno)
VALUES (9, 'Alexandr')
```

Připoměňme si, že tabulka `Student` má stanoven `sID` jako primární klíč. Proto jsme `sID` studenta jsme stanovili prozatím ručně jako první dostupné (nevyužité) `sID` v tabulce `Student`. Při opakovaném spuštění tohoto příkazu dostaneme chybu, která hlásí porušení intergitního omezení primárního klíče. Ekvivalentní zápis vynechávající seznam atributů by mohl být i tento:

```
INSERT INTO Student VALUES (9, 'Alexandr', NULL)
```

(b) Přiřaďte do předmětu s pID 4 všechny studenty pro rok 2012.

```
INSERT INTO Studuje
SELECT sID, 4, 2012, NULL
  FROM Student
```

Ještě jednou připomeneme že výsledek `SELECT` dotazu musí odpovídat schématu tabulky `Studuje`. Příkaz `SELECT` tedy musí vracet čtyři atributy, kde všechny jsou typu číslo.

Příkaz DELETE

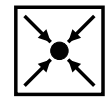
Příkaz `DELETE` odstraňuje na základě podmínky jeden nebo více řádků z tabulky. Výpis 4.8 ukazuje jeho základní strukturu.

```
DELETE FROM Tabulka
WHERE Podminka
```

Výpis 4.8: Příkaz `DELETE`

Podmínka může být poměrně komplikovaná. Může obsahovat poddotazy, agregační funkce atd. Obvyklejší je nicméně mít u příkazu `DELETE` jednoduché podmínky a nejčastější je mazání jednotlivých záznamů na základě klíče. Pokud není definována žádná podmínka, pak jsou zrušeny všechny záznamy z tabulky.

Příklad 4.24. (Mazání záznamů).



(a) Vymažte z databáze studenta s sID 1.

```
DELETE FROM Studuje WHERE sID = 1
DELETE FROM Student WHERE sID = 1
```

Jak vidíme v tomto případě obsahuje řešení příkazy dva. Přičemž první příkaz maže z tabulky `Studuje` a ne z tabulky `Student` jak uvádí zadání. Problémem je v tomto případě cizí klíč a referenční integrita vyžadovaná SŘBD. Pokud by jsme zkusili smazat studenta který studoval nějaké předměty dojde k výjimce (to se může lišit podle použitého SŘBD). Je tedy nejprve nutné smazat záznamy z tabulky `Studuje`, které se na studenta odkazují. Druhou možností je definovat u cizího klíče `Studuje.sID` vlastnost `ON DELETE CASCADE`, která povede k tomu, že při mazání studenta se odstraní i související záznamy z tabulky `Studuje`.

(b) Vymažte z databáze všechny studenty, kteří v nějakém předmětu dvakrát dostali méně než 51 bodů.

```
DELETE FROM Studuje WHERE sID IN
(
  SELECT Se.sID FROM Student S, Studuje Se
  WHERE S.sID = Se.sID AND Se.body < 51
  GROUP BY Se.sID, pID
  HAVING COUNT(*) >= 2
)
```

Příkaz UPDATE

Příkaz `UPDATE` mění hodnotu ve sloupci (nebo sloupcích) v jednom nebo více řádcích na základě podmínky. Výpis 4.9 obsahuje strukturu příkazu `UPDATE`. Příkaz `UPDATE` se vždy provádí nad jednou tabulkou. Výraz je nová hodnota, kterou nastavujeme atributu. Podmínka pak specifikuje pro které řádky chceme nové hodnoty nastavit.

```
UPDATE Tabulka
SET Atribut1 = Výraz1, ... AtributN = VýrazN
WHERE Podminka
```

Výpis 4.9: Příkaz `UPDATE`

Příklad 4.25. (Aktualizace záznamů).

(a) Nastavte všem studentům body na 90 u všech předmětů, které studovali v roce 2011.

```
UPDATE Studuje
SET body = 90
WHERE rok = 2011
```

(b) Nastavte všem studentům u předmětů, které studovali v roce 2011, body na minimum bodů, kterého bylo dosaženo (mezi všemi studenty) v roce 2010.

```
UPDATE Studuje
SET body =
(
  SELECT MIN(body) FROM Studuje
  WHERE rok = 2010
)
WHERE rok = 2011
```

Všimněme si, že druhá podmínka se váže k příkazu `UPDATE`, zatímco první je součástí `SELECT` příkazu.

(c) Nastavte všem studentům u předmětů, které studují v roce 2011, body na minimum bodů, které obdržel daný student v roce 2010.

```
UPDATE s1
SET s1.body =
(
  SELECT MIN(body) FROM Studuje
  WHERE rok = 2010 and sID = s1.sID
)
FROM Studuje s1
WHERE s1.rok = 2011
```

Na tomto příkladu vidíme že do `UPDATE` příkazu může být přidána i konstrukce `FROM`, která nám umožní pojmenovat relaci se kterou pracujeme. Díky toho můžeme přistoupit k aktuální hodnotě měněného záznamu (t.j., s1.sID) v `SELECT` příkazu. Můžeme tedy použít `sID` studenta pro nalezení jeho minima z minulého roku.

4.3.5 Další rysy jazyka SQL

Systémový katalog

Systémový katalog je množina tabulek obsahující informace o:

- tabulkách a jejich sloupcích,
- pohledech,
- uživatelích,
- právech.

Standard SQL92 tyto tabulky definuje, nicméně standard byl přijat až poté co výrobci do svých implementací zavedli vlastní tabulky. V následující tabulce uvedeme názvy tabulek s podobným významem pro SQL92 a Oracle.

SQL92	Oracle	Popis
TABLES	USER_TABLES	Obsahuje jeden řádek pro každou tabulku kterou vlastní aktuální uživatel.
	ALL_TABLES	Obsahuje jeden řádek pro každou tabulku ke které má aktuální uživatel alespoň jedno přístupové právo.
COLUMNS	USER_TAB_COLUMNS	Obsahuje jeden řádek pro každý sloupec tabulky kterou vlastní aktuální uživatel.
	ALL_TAB_COLUMNS	Obsahuje jeden řádek pro každý sloupec tabulky ke které má aktuální uživatel alespoň jedno přístupové právo.
USERS	ALL_USERS	Obsahuje jeden řádek pro každého uživatele v systému.
VIEWS	USERS_VIEW	Obsahuje jeden řádek pro každý pohled který vlastní aktuální uživatel.
	ALL_VIEW	Obsahuje jeden řádek pro každý pohled ke kterému má aktuální uživatel alespoň jedno přístupové právo.

TABLE_PRIVILEGES	USER_TAB_PRIVS	Obsahuje jeden řádek pro každé právo přidělené uživatelem.
	ALL_TAB_PRIVS	Obsahuje jeden řádek pro každé právo kde aktuální uživatel je vlastníkem, dárcem nebo příjemcem.

Příklad 4.26. Následujícím příkazem v SŘBD Oracle získáme názvy tabulek a vlastníka tabulek ke kterým má aktuální uživatel alespoň jedno přístupové právo.

```
SELECT TABLE_NAME, OWNER
  FROM ALL_TABLES;
```

Příklad 4.27. Následujícím příkazem v SŘBD Oracle získáme záznamy o všech sloupcích tabulky USER_TABLES.

```
SELECT * FROM ALL_TAB_COLUMNS
  WHERE TABLE_NAME='USER_TABLES';
```

Komentáře

SŘBD DB2 a Oracle umožňují přidat komentáře k jednotlivým tabulkám a jejich sloupcům příkazem COMMNENT ON který ovšem nebyl zařazen do standardu. DB2 navíc umožňuje přidat komentáře k dalším objektům databáze (uložené procedury apod.). Tyto komentáře pak opět můžeme získat ze systémového katalogu. Např. v Oracle máme k dispozici tabulky USER_TAB_COMMENTS, ALL_TAB_COMMENTS, USER_COL_COMMENTS a ALL_COL_COMMENTS.

Příklad 4.28. Následujícím příkazem přidáme v SŘBD Oracle komentář k tabulce student.

```
COMMENT ON TABLE student IS 'Tabulka␣obsahuje␣zaznamy␣
   o␣studentech␣bakalarskeho,␣magisterskeho␣a␣
   doktorskeho␣studia'
```

Příklad 4.29. Následujícím příkazem přidáme v SŘBD Oracle komentář k atributu fname tabulky student.

```
COMMENT ON COLUMN student.fname IS 'Jméno␣studenta'
```

Ochrana dat

Příkazem GRANT přidělíme práva uživateli na příslušnou databázovou operaci nad tabulkou:

```
GRANT seznam_operaci
  [ ON jmeno_tabulky ]
  TO uzivatel | PUBLIC

seznam_operaci := operace [, seznam_operaci] | ALL
    PRIVILEGES
operace := SELECT | INSERT | UPDATE | DELETE |
    REFERENCES
```

Kde REFERENCES značí právo odkazovat se na záznamy tabulky cizím klíčem, ALL PRIVILEGES pak zahrnuje všechny operace.

Následujícím příkazem přiřadíme uživateli kra28 právo na výběr dat z vlastní tabulky student:

```
GRANT SELECT
  ON student
  TO kra28;
```

Pokud se uživatel jmenuje kra226, pak uživatel kra28, může získat data z tabulky:

```
SELECT * FROM kra226.student;
```

V SQL92 umožňuje definovat právo pouze pro sloupec tabulky pro operace UPDATE, INSERT a REFERENCES. Následujícím příkazem umožníme aktualizovat hodnotu atributu login tabulky student všem uživatelům.

```
GRANT UPDATE(login) ON student to PUBLIC;
```

Poznamenejme, že uživatel který vytvořil tabulku automaticky získává práva na všechny operace nad touto tabulkou.

Pokud za příkazem GRANT uvedeme WITH GRANT OPTION, pak daný uživatel může toto právo předávat dalším uživatelům.

Příkazem REVOKE naopak zrušíme práva uživateli na příslušnou databázovou operaci nad tabulkou:

```
REVOKE seznam_operaci
  [ ON jmeno_tabulky ]
```

```
  FROM uzivatel | PUBLIC

seznam_operaci := operace [, seznam_operaci] | ALL
    PRIVILEGES
operace := SELECT | INSERT | UPDATE | DELETE |
    REFERENCES
```

Následujícím příkazem zrušíme uživateli kra28 právo na výběr dat z vlastní tabulky `student`:

```
REVOKE SELECT
  ON student
  FROM kra28;
```

Vestavěné funkce

Součástí SQL92 jsou i vestavěné funkce, které nejčastěji slouží pro práci s hodnotami jednotlivých datových typů. V tabulce 4.8 vidíme jejich soupis. Pro úplnost dodejme, že jednotlivé SŘBD obsahují další funkce, popř. funkce ze standardu s mírně odlišnou syntaxí.

Funkce	Popis
UPPER(retezec)	Zkonvertuje řetězec na velká písmena.
LOWER(retezec)	Zkonvertuje řetězec na malá písmena.

Tabulka 4.8: Vestavěné funkce v SQL92.

Příklad 4.30. Některá atributy systémového katalogu obsahují názvy tabulek a atributů konvertované na velká písmena. Nemůže tedy např. použít příkaz:

```
SELECT * FROM USER_TAB_COLUMNS
  WHERE TABLE_NAME='kurzy';
```

V tomto případě můžeme s výhodou použít:

```
SELECT * FROM USER_TAB_COLUMNS
  WHERE TABLE_NAME=UPPER('kurzy');
```

Kapitola 5

Normální formy

Obsah

Cíl kapitoly:
Představení teorie definující správný návrh databáze. Představení pojmů funkční závislost, dekompozice a normální forma

5.1 Návrh schématu relační databáze

V kapitole 3.3 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. Existuje mnoho způsobů, jak navrhnout schéma databáze k jednomu zadání. Některá řešení jsou srovnateně dobrá, jiná výrazně horší. V této kapitole prezentujeme elegantní teorii k návrhu databází a na základě této teroie můžeme jasně definovat jak vypadá špatný návrh databáze a dobrý návrh databáze.

5.1.1 Funkční závislosti

Definice 5.1. *(Funkční závislost (FZ)). Atributy $B_1\ B_2 \dots B_m$ jsou funkčně závislé na atributech $A_1\ A_2 \dots A_n$ pokud pro libovolné dvě n-tice platí: jestliže jsou hodnoty atributů $A_1\ A_2 \dots A_n$ totožné, pak jsou totožné i hodnoty atributů $B_1\ B_2 \dots B_m$. Funkční závislost zapisujeme:*

$$A_1\ A_2 \dots A_n \to B_1\ B_2 \dots B_m$$

Uvědomme si, že se jedná o implikaci, nikoli ekvivalenci, obráceně tedy výraz neplatí. Někdy také říkáme, že atributy $A_1\ A_2 \dots A_n$ **funkcionálně určují** atributy $B_1\ B_2 \dots B_m$. Čtenáře možná napadne, proč se funkcionální závislost jmenuje funkcionální. Zjevně existuje funkce, která hodnotám atributů $A_1\ A_2 \dots A_n$ přiřazuje hodnoty atributů $B_1\ B_2 \dots B_m$. Pokud by pro hodnoty atributů $A_1\ A_2 \dots A_n$ existovaly různé hodnoty atributů $B_1\ B_2 \dots B_m$, nejednalo by se o funkci a samozřejmě ani funkční závislost.

Řekneme, že závislost $\overline{A} \to \overline{B}$ je

- triviální, pokud $\overline{B} \subset \overline{A}$
- netriviální, pokud $\overline{B} \not\subset \overline{A}$
- úplně netriviální, pokud $\overline{B} \cap \overline{A} = \emptyset$

Příklad 5.1. (Funkční závislosti).

(a) Mějme relaci `Osoba(RČ, jméno, email, věk)`, zkuste určit všechny úplně netriviální FZ.

`RČ` $\to$ `jméno`
`RČ` $\to$ `email`
`RČ` $\to$ `věk`
`RČ` $\to$ `jméno email`
`RČ` $\to$ `jméno věk`
`RČ` $\to$ `email věk`
`RČ` $\to$ `jméno email věk`
`RČ jméno` $\to$ `email`
`RČ jméno` $\to$ `věk`
`RČ jméno` $\to$ `email věk`
`email` $\to$ `jméno`
`email` $\to$ `věk`
`email` $\to$ `RČ`

`email` → `jméno věk`
...

Seznam není úplný protože by obsahoval několik desítek pravidel. Při bližším zkoumání pravidel zjistíme, že některá pravidla by se mohla odvodit z předchozích. Takové odvození popisují tzv. armstrongovy axiomy.

(b) Mějme další relaci `Film(název, rok, délka, studio)` na které platí FZ `název, rok` → `délka, studio`. V podstatě to znamená, že v jednom roce nikdy nebyl vyroben film se stejným názvem. Zkuste se zamyslet, zda-li bude platit funkční závislost `název` → `délka, studio`.

Pokud bychom se pokusili interpretovat význam druhé funkční závislosti tak bychom řekli, že aby FZ platila, nesměl by existovat dva různé filmy se stejným názvem. Našli bychom asi spoustu protipříkladů, takže druhá FZ neplatí.

Jak bylo zmíněno pro funkční závislosti existují určitá odvozovací pravidla. Tato pravidla se obvykle nazývají **Armstrongovy axiomy**. Představme si ty nejdůležitější z nich:

- Dekompozice FZ:
  $\overline{A} \to B_1B_2 \Longrightarrow \overline{A} \to B_1,\ \overline{A} \to B_2$
  Říkáme, že FZ $\overline{A} \to B_1B_2$ je rozložena na elementární FZ, tj. takové, které mají na pravé straně pouze jeden atribut
- Sjednocení FZ:
  $\overline{A} \to B_1,\ \overline{A} \to B_2 \Longrightarrow \overline{A} \to B_2B_2$
- Rozšíření FZ:
  $\overline{A} \to \overline{B} \Longrightarrow \overline{AZ} \to \overline{BZ}$ pro libovolnou množinu Z
- Tranzitivita:
  $\overline{A} \to \overline{B}$ a $\overline{B} \to \overline{C} \Longrightarrow \overline{A} \to \overline{C}$

S množinou funkčních závislostí F nad relaci R souvisí **uzávěr** množiny atributů $\overline{A} \subset R$. Uzávěr množiny atributů $\overline{A}$, vzhledem k funkčním závislostem F, je množina atributů $\overline{B}$, kde $\overline{A} \to \overline{B}$. Množina $\overline{B}$ se nazývá uzávěr $\overline{A}$ a značíme ji $\overline{A}+$.

Algoritmus pro hledání uzávěru vidíme v algoritmu 1.

Příklad 5.2. (Uzávěr množiny atributů). Uvažujme relační schéma s atributy: A, B, C, D a E. Relace má funkční závislosti: $AB \to C$, $BC \to AD$ a $CE \to B$. Jaký je uzávěr $\{AB\}^+$?

Algoritmus 1: Hledání uzávěru $\overline{A}^+$

Vstup : Množina atributů $\overline{A}$, Množina FZ
Výstup: Množina atributů $\overline{A}^+$ které jsou uzávěrem $\overline{A}$
$\overline{X} = \overline{A}$;
while dokud dochází ke změně $\overline{X}$ **do**
  if $\exists \overline{Y} \rightarrow \overline{B} : \overline{Y} \subset \overline{X}$ **then** přidej $\overline{B}$ do $\overline{X}$;
$\overline{A}^+$ je roven $\overline{X}$;

Řešení: Nejprve $X = \{A, B\}$. Jelikož funkční závislost $AB \rightarrow C$ obsahuje na levé straně pouze atributy z X, můžeme do X přidat atributy z levé strany. Po této iteraci dostáváme $X = \{A, B, C\}$. V dalším kroku přidáme, na základě FZ $BC \rightarrow AD$, do X atribut D. V této chvíli se pomocí žádné FZ nepodaří rozšířit X o další atribut a nalezli jsem tedy uzávěr: $\{A, B\}^+ = \{A, B, C, D\}$.

Definice 5.2. *(Klíč relace). Množina atributů $\overline{A}$ se nazývá **klíčem** relace R pokud:*

1. *Všechny ostatní atributy relace jsou funkčně závislé na této množině.*
2. *Obvykle se navíc uvádí, že neexistuje podmnožina $\overline{A'}$ (různá od $\overline{A}$), která by byla klíčem R (mluvíme o tom, že **klíč je minimální**)*

Ověření zda-li je množina $\overline{A}$ klíčem relace souvisí s uzávěrem. Pokud uzávěr $\overline{A}^+$ obsahuje všechny atributy R, pak se jedná o klíč R. Pokud ovšem chceme, aby klíč byl minimální, pak musíme ještě ověřit, zda-li neobsahuje nějakou podmnožinu, která by byla také klíčem.

Relace může mít více klíčů, v takovém případě vybíráme jednu množinu atributů, kterou nazveme **primární klíč**. V konceptuálním modelu jsme požadavek, aby klíč byl minimální, nezaznamenali. Tento požadavek se vyskytuje až v relačním datovém modelu.

Víme jak poznat klíč, ale nejobvyklejší bývá hledání klíčů pro nějakou relaci. Teoreticky bychom měli určit uzávěr každé podmnožiny atributů. Prakticky ale začínáme od nejkratších podmnožin a pokračujeme k delším. Po nalezení nějakého klíče už nemusíme testovat nadmnožiny tohoto klíče, protože ty budou klíčem také (i když ne minimálním klíčem).

Množinu atributů, která obsahuje klíč se nazývá **nadklíč** (zkratka pro nadmnožinu klíče). Každý klíč je nadklíč, ovšem ne každý nadklíč je klíčem. Nadklíč splňuje první podmínku definice klíče, tj. funkcionálně určuje všechny ostatní atributy. Nesplňuje ovšem druhou podmínku – podmínku minimálnosti.

Báze nazýváme množinu funkčních závislostí ze kterých je možné odvodit všechny ostatní funkční závislosti. Pokud neexistuje podmnožina báze, ze které by bylo možné odvodit všechny ostatní funkční závislosti, řekneme že množina funkčních závislostí je **minimální a neredundantní**. Při sestavování množiny FZ pro nějaké relační schéma obvykle intuitivně vytváříme množinu, která tuto podmínku splňuje. V dalším textu budeme popisovat formální postup, jak tuto množinu nalézt, či jak ověřit že naše množina funkčních závislostí je neredundantní a minimální. Celý postup se skládá ze dvou kroků:

1. Hledání redundantních FZ.
2. Hledání redundantních atributů na levých stranách FZ.

Začneme tedy ověřením, jestli je FZ $\overline{A} \rightarrow B$ redundantní nebo ne. Máme množinu FZ F a chceme určit, zda-li $\overline{A} \rightarrow B$ vyplývá z F. V zásadě máme dvě možnosti, jak to zjistit:

- Nalézt uzávěr $\overline{A}$ na základě pravidel v F; pokud uzávěr obsahuje B, pak je pravidlo $\overline{A} \rightarrow B$ redundantní
- Druhou možností je zkusit odvodit $\overline{A} \rightarrow B$ přímo z F na základě Armstrongových pravidel

Příklad 5.3. (Hledání redundantních FZ).

Mějme $R(X, Y, Z)$ a množinu FZ: $\{X \rightarrow YZ, Y \rightarrow XZ\}$. Určete neredundantní množinu FZ.

Nejprve vytvoříme množinu elementárních FZ: $\{X \rightarrow Y, X \rightarrow Z, Y \rightarrow X, Y \rightarrow Z\}$. Budeme postupně testovat každou FZ a zkusit zjistit, zda-li je redundantní:

- Začneme pravidlem $X \rightarrow Y$. Zbývající FZ jsou $\{X \rightarrow Z, Y \rightarrow X, Y \rightarrow Z\}$. Dále hledáme uzávěr levé strany pravidla, které ověřujeme a nesmíme zapomenout, že uzávěr hledáme jen na základě zbývajících FZ. Tedy $\overline{X}+$ = $\{X, Z\}$, což neobsahuje Y, takže pravidlo není redundantní.
- Pokračujeme pravidlem $X \rightarrow Z$. Zbývající FZ jsou $\{X \rightarrow Y, Y \rightarrow X, Y \rightarrow Z\}$. V tomto případě je $\overline{X}^+$ = $\{X, Y, Z\}$, což obsahuje Z, takže pravidlo je redundantní. Můžeme si všimnout, že pravidlo $X \rightarrow Z$ lze odvodit z $X \rightarrow Y$ a $Y \rightarrow Z$ na základě tranzitivity.
- Můžeme zkusit další pravidlo $Y \rightarrow X$. Analogicky s předchozím postupem nalezneme uzávěr levé strany. $\overline{Y}^+$ = $\{Y, Z\}$, což neobsahuje X, takže pravidlo není redundantní.

- A nakonec pravidlo $Y \rightarrow Z$. $\overline{Y}^+ = \{X, Y, Z\}$, což obsahuje Z, takže pravidlo je také redundantní. Opět lze toto pravidlo odvodit na základě tranzitivity.

Po odstranění jedné FZ bychom měli na zbylé množině opět ověřit, zda-li neexistuje nějaké redundantní FZ. Nicméně v tomto příkladu lze odstranit pouze jednu FZ, co6 lze lehce analogicky ověřit. Takže výsledkem je, že máme dvě možné neredundantní množiny FZ:

1. $\{X \rightarrow Y, Y \rightarrow X, Y \rightarrow Z\}$
2. $\{X \rightarrow Y, X \rightarrow Z, Y \rightarrow X\}$

Druhým krokem při hledání báze FZ je hledání redundantních atributů na levých stranách FZ. V tomto případě je pravidlo poměrně jednoduché. Máme $\overline{A} \rightarrow \overline{B}$ a pro nějaké $C \in \overline{A}$ platí $(\overline{A} - C)^+ = \overline{A}^+$, pak pro tuto FZ je atribut C redundantní.

Příklad 5.4. (Hledání redundantních FZ).

Mějme $R(A, B, C, D, E)$ a tuto množinu FZ:
$\{ABC \rightarrow D, E \rightarrow C, AB \rightarrow E, C \rightarrow D\}$. Nyní chceme odstranit redundantní atributy.

Provedeme kontrolu pouze pro FZ $ABC \rightarrow D$. Nejprve musíme určit, že $ABC^+ = \{A, B, C, D, E\}$. Poté určíme uzávěry $BC^+ = \{B, C, D\}$, $AC^+ = \{A, C, D\}$ a $AB^+ = \{A, B, C, D, E\}$. Zjevně je tedy atribut C redundantní, jelikož $ABC^+ = AB^+$.

5.1.2 Dekompozice

Teorie, která bude dále prezentována, se snaží algoritmizovat získání korektního schématu relační databáze. Na vstupu je jediné relační schéma (tzv. **univerzální relační schéma**) obsahující všechny atributy a výstupem je schéma relační databáze, které má určité dobré vlastnosti (budeme prezentovat později).

Příklad 5.5. (Vytvoření univerzálního schématu z konceptuálního modelu). Vezměme v úvahu relační schémata z příkladu 3.3. Pro zjednodušení u schématu `Ucitel` neuvažujme atribut `uvazek`, u schématu `Student` pak neuvažujeme atribut `rocnik`. Z důvodu přehlednosti byl k názvům atributů přidán prefix naznačující relační schéma, ke kterému atribut náleží. Vytvoříme univerzální schéma `Kurz(id, nazev, kapacita, ucitelLogin, ucitelJmeno, ucitelEmail,`

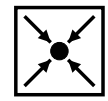
studentLogin, studentJmeno, studentEmail). V tabulce 5.1 vidíme příklad relace Kurz. Vidíme, že kurz TIS studují dva studenti, kurz UDP 3 studenti, kurzy ZA a DBIS studuje jeden student.

Tabulka 5.1: Hodnoty atributů relace typu Kurz

id	nazev	kapacita	ucitelLogin	ucitelJmeno	ucitelEmail
45613	TIS	300	kra102	Milan Kratochvíl	milan.kratochvil@vsk.cz, milan.kratochvil@seznam.cz
45613	TIS	300	kra102	Milan Kratochvíl	milan.kratochvil@vsk.cz, milan.kratochvil@seznam.cz
45601	UDP	300	kra102	Milan Kratochvíl	milan.kratochvil@vsk.cz, milan.kratochvil@seznam.cz
45601	UDP	300	kra102	Milan Kratochvíl	milan.kratochvil@vsk.cz, milan.kratochvil@seznam.cz
45601	UDP	300	kra102	Milan Kratochvíl	milan.kratochvil@vsk.cz, milan.kratochvil@seznam.cz
45603	ZA	300	dvo005	Petr Dvořák	petr.dvorak@vsk.cz
45638	DBIS	300	svo032	Tomáš Svoboda	tomas.svoboda@vsk.cz

id	...	studentLogin	studentJmeno	studentEmail
45613	...	svo005	Michal Svoboda	michal.svoboda.stud@vsk.cz, msvoboda@centrum.cz
45613	...	nov003	Jitka Novotná	jitka.novotna.stud@vsk.cz, supergirl@post.cz
45601	...	svo005	Michal Svoboda	michal.svoboda.stud@vsk.cz, msvoboda@centrum.cz
45601	...	nov003	Jitka Novotná	jitka.novotna.stud@vsk.cz, supergirl@post.cz
45601	...	lam001	Jana Lampová	jana.lampova.stud@vsk.cz, jlampa@ume.cz
45601	...	lam001	Jana Lampová	jana.lampova.stud@vsk.cz, jlampa@ume.cz
45601	...	lam001	Jana Lampová	jana.lampova.stud@vsk.cz, jlampa@ume.cz

Příklad 5.6. (Funkční závislosti relace Kurz). Vezměme v úvahu relaci typu Kurz z příkladu 3.3. V relaci můžeme nalézt například tyto funkční závislosti:



$$\texttt{id} \rightarrow \texttt{nazev}$$
$$\texttt{id} \rightarrow \texttt{kapacita}$$
$$\texttt{id} \rightarrow \texttt{ucitelLogin}$$

Můžeme tedy psát: $\texttt{id} \rightarrow \texttt{nazev kapacita ucitelLogin}$

Je zjevné o čem funkční závislosti vypovídají. id je klíč relace Kurz, proto musí být hodnoty atributů nazev a kapacita stejné pro stejnou hodnotu atributu id. Mezi relací Kurz a Ucitel je vazba M:1, proto existuje funkční závislost

`id` $\rightarrow$ `ucitelLogin`. `id` $\rightarrow$ `studentLogin` není funkční závislost, protože více studentů může studovat jeden kurz – mezi původními relacemi `Kurz` a `Student` je vazba $M : N$. Funkční závislosti vypovídají o relačním schématu nikoli o nějaké relaci relačního schématu. Při identifikaci funkčních závislostí musí vnímat obecná omezení plynoucí z reality, nikoli závislosti plynoucí z nějaké relace.

Pokud uvažujeme univerzální relaci (například tu z příkladu 5.5), můžeme v této relaci identifikovat následující anomálie:

1. **Redundance** : informace se mohou v n-ticích opakovat.
2. **Anomálie při modifikaci**: pokud je v relaci redundance, je zřejmé, že když změníme nějaký redundantní údaj, zůstane na dalších místech výskytu nezměněn.
3. **Anomálie při rušení**: pokud budeme chtít zrušit nějakou informaci z takové relace, vedlejším efektem může být zrušení jiné významné informace.

Doposud jsme mluvili o nutnosti vytvořit korektní schéma relační databáze, zjevně se jedná o schéma neobsahující tyto anomálie. Přestože jsme v mnoha případech intuitivně schopni vytvořit korektní schéma, v následujícím textu si ukážeme jak je možné vytvoření takového schématu formalizovat a tedy i algoritmizovat.

Rozklad (**dekompozice**) relačního schématu je jeden ze způsobů jak odstranit výše uvedené anomálie z relace. Rozklad relačního schématu R s atributy $A_1A_2 \ldots A_n$ je takové rozdělení na dvě schémata S a T s atributy $\{B_1B_2 \ldots B_m\}$ resp. $\{C_1C_2 \ldots C_k\}$ pro které platí:

1. $\{A_1A_2 \ldots A_n\} = \{B_1B_2 \ldots B_m\} \cup \{C_1C_2 \ldots C_k\}$
2. n-tice relace S jsou projekcí na atributy $\{B_1B_2 \ldots B_m\}$. To znamená, že pro každou n-tici $t \in$ R je n-tice $t' \in S$ tvořena hodnotami atributů $\{B_1B_2 \ldots B_m\}$. Jelikož relace je množina, pak jedna n-tice z relace S může být projekcí dvou n-tic z relace R.
3. Podobně n-tice relace T jsou **projekcí** na atributy $\{C_1C_2 \ldots C_k\}$.

Příklad 5.7. (Rozklad relace). Uvažujme relaci z příkladu 5.5, kterou rozdělíme na relace `Kurz1` a `Kurz2`. Schéma relace `Kurz1` obsahuje všechny atributy kromě atributů studenta, tedy `Kurz1(id, nazev, kapacita, ucitelLogin,`

ucitelJmeno, ucitelEmail). Schéma relace Kurz2 obsahuje atributy id a všechny atributy studenta, tedy Kurz2(id, studentLogin, studentJmeno, studentEmail). Příklad relací typu Kurz1 a Kurz2, které vzniknou rozkladem relace typu Kurz z tabulky 5.1 vidíme v tabulkách 5.2 a 5.3.

Tabulka 5.2: Relace Kurz1

id	nazev	kapacita	ucitelLogin	ucitelJmeno	ucitelEmail
45613	TIS	300	kra102	Milan Kratochvíl	milan.kratochvil@vsk.cz, milan.kratochvil@seznam.cz
45601	UDP	300	kra102	Milan Kratochvíl	milan.kratochvil@vsk.cz, milan.kratochvil@seznam.cz
45603	ZA	300	dvo005	Petr Dvořák	petr.dvorak@vsk.cz
45638	DBIS	300	svo032	Tomáš Svoboda	tomas.svoboda@vsk.cz

Tabulka 5.3: Relace Kurz2

id	studentLogin	studentJmeno	studentEmail
45613	svo005	Michal Svoboda	michal.svoboda.stud@vsk.cz, msvoboda@centrum.cz
45613	nov003	Jitka Novotná	jitka.novotna.stud@vsk.cz, supergirl@post.cz
45601	svo005	Michal Svoboda	michal.svoboda.stud@vsk.cz, msvoboda@centrum.cz
45601	nov003	Jitka Novotná	jitka.novotna.stud@vsk.cz, supergirl@post.cz
45601	lam001	Jana Lampová	jana.lampova.stud@vsk.cz, jlampa@ume.cz
45603	lam001	Jana Lampová	jana.lampova.stud@vsk.cz, jlampa@ume.cz
45638	lam001	Jana Lampová	jana.lampova.stud@vsk.cz, jlampa@ume.cz

Vidíme, že tyto relace splňují podmínky, které jsme definovali pro rozklad relačního schématu. Tento rozklad řeší všechny tři anomálie pouze částečně. Relace typu Kurz1 obsahuje opakující se jména a emaily učitelů. Podobně relace typu Kurz2 rovněž obsahuje opakující se jména a emaily studentů.

5.1.3 Boyce-Coddova normální forma

Jak již bylo řečeno, cílem dekompozice je rozložit univerzální schéma na více schémat, jejichž relace nebudou obsahovat redundanci. V relačním modelu existuje jednoduchá podmínka, která nám zaručí, že v relacích splňujících tuto podmínku nebudou existovat tyto anomálie. Tato podmínka se jmenuje **Boyce-Coddova**

normální forma (**BCNF**). V textu velmi často volně přecházíme od pojmu relace k pojmu relační schéma, čtenář jistě zná striktní definice těchto pojmů.

Relační schéma R je v BCNF pokud pro každou netriviální FD na `R` $A_1 A_2 \ldots A_n \rightarrow B_1 B_2 \ldots B_m$ platí, že $A_1 A_2 \ldots A_n$ je **nadklíčem** schématu `R`. Jinými slovy levá část každé netriviální FD musí obsahovat klíč.

Příklad 5.8. (Rozklad relace). Vidíme, že relační schéma `Kurz` z příkladu 5.5 není v BCNF. Klíčem relace je {`id, studentLogin`}, ovšem platí funkční závislost `id` → `nazev kapacita ucitelLogin`. Tzn. existuje FD, která na levé straně neobsahuje nadklíč. V tomto případě to znamená, že `id` funkčně neurčuje atributy `studentLogin, studentJmeno` a `studentEmail`.

Relační schémata `Kurz1` a `Kurz2` z příkladu 5.7 rovněž nejsou v BCNF. Klíčem schématu `Kurz1` je atribut `id`, ovšem FD `ucitelLogin` → `ucitelJmeno ucitelEmail` nesplňuje podmínku. Relační schéma `Kurz2` rovněž není v BCNF, jelikož platí FD `loginStudent` → `studentJmeno studentEmail`, ale klíčem relace je {`id, loginStudent`}.

Aby schémata byla v BCNF, museli bychom provést další dekompozici, jejímž výsledkem by byla schémata splňující podmínku BCNF.

Rozklad relace do BCNF

Rozklad je realizován algoritmem, který rozkládá univerzální schéma na množinu relací, které splňují následující vlastnosti:

1. Jednotlivá schémata jsou v BCNF.
2. Vzniklá relační schémata obsahují všechna data obsažená v původním schématu. Tzn. ze vzniklých schémat můžeme zpětně získat původní relační schéma. Tato podmínka je triviální, je obsažena v definici rozkladu relačního schématu.

Příklad 5.9. (Rozklad relace). Mějme relační schéma `Kurz` z příkladu 5.5. Platí funkční závislost `id` → `nazev kapacita ucitelLogin ucitelJmeno ucitelEmail`. `id` není nadklíčem, tato FD je tedy výjimkou z BCNF. V tomto případě atributy pravé strany funkčně závisí na atributu `id`. Využijeme tuto výjimku pro rozklad schématu na dvě schémata:

1. Schéma obsahující všechny atributy funkční závislosti: `Kurz1`{`id, nazev, kapacita, ucitelLogin, ucitelJmeno, ucitelEmail`}.

2. Schéma se všemi atributy s výjimkou atributů na pravé straně FD: `Kurz2{id, studentLogin, studentJmeno, studentEmail}`.

Vidíme, že rozklad odpovídá intuitivnímu rozkladu z příkladu 5.7. Žádné schéma není v BCNF. Všimněme si, že v prvním schématu je zahrnuta vazba 1:N mezi schématy `Ucitel` a `Kurz` a ve druhém vazba M:N mezi schématy `Student` a `Kurz`. V prvním případě platí FD:

`id` → `ucitelLogin`
`ucitelLogin` → `ucitelJmeno`
`ucitelLogin` → `ucitelEmail`
Tedy: `ucitelLogin` → `ucitelEmail ucitelJmeno`

Pokud použijeme tzv. **tranzitivní pravidlo**, dostaneme:
`id` → `ucitelJmeno`
`id` → `ucitelEmail`

Redundanci tedy zapříčiňuje **tranzitivní závislost**, kterou identifikujeme u schémat zahrnujících vazbu 1:N. Využijeme FD `ucitelLogin` → `ucitelEmail ucitelJmeno`. Schéma `Kurz1` je rozloženo na:

1. Schéma obsahující atributy FD: `Kurz11(ucitelLogin, ucitelEmail ucitelJmeno)`.
2. Schéma obsahující všechny atributy původního schématu kromě atributů na pravé straně: `Kurz12(id, nazev, kapacita, ucitelLogin)`.

Obě schémata jsou v BCNF. V tabulkách 5.4 a 5.5 vidíme konkrétní relace typu `Kurz11` a `Kurz12`.

Tabulka 5.4: Relace typu `Kurz11`

id	nazev	kapacita	ucitelLogin
45613	TIS	300	kra102
45601	UDP	300	kra102
45603	ZA	300	dvo005
45638	DBIS	300	svo032

Podívejme se nyní na relační schéma `Kurz2` s klíčem {`id, studentLogin`}. V této relace je zahrnuta vazba M:N. Platí FD:

Tabulka 5.5: Relace typu `Kurz12`

ucitelLogin	ucitelJmeno	ucitelEmail
kra102	Milan Kratochvíl	milan.kratochvil@vsk.cz, milan.kratochvil@seznam.cz
dvo005	Petr Dvořák	petr.dvorak@vsk.cz
svo032	Tomáš Svoboda	tomas.svoboda@vsk.cz

$$\texttt{studentLogin} \rightarrow \texttt{studentEmail studentJmeno}$$

`studentLogin` není klíčem, přesto je na levé straně FD. Problém vyřešíme rozkladem na:

- Schéma obsahující atributy této funkční závislosti: `Kurz21(studentLogin studentEmail, studentJmeno)`
- Schéma obsahující všechny atributy kromě atributů na pravé straně: `Kurz22(id, studentLogin )`

Obě schémata jsou v BCNF. V tabulkách 5.6 a 5.7 vidíme konkrétní relace typu `Kurz21` a `Kurz22`. Vidíme, že rozklad relačního schématu `Kurz` na schémata `Kurz11`, `Kurz12`, `Kurz21` a `Kurz22` řeší všechny anomálie. Toho jsme dosáhli postupným rozkladem na relační schémata v BCNF.

Tabulka 5.6: Relace `Kurz21`

id	studentLogin
45613	svo005
45613	nov003
45601	svo005
45601	nov003
45601	lam001
45603	lam001
45638	lam001

Tabulka 5.7: Relace `Kurz22`

studentLogin	studentJmeno	studentEmail
svo005	Michal Svoboda	michal.svoboda.stud@vsk.cz, msvoboda@centrum.cz
nov003	Jitka Novotná	jitka.novotna.stud@vsk.cz, supergirl@post.cz
lam001	Jana Lampová	jana.lampova.stud@vsk.cz, jlampa@ume.cz

5.1.4 První, druhá a třetí normální forma

Relační schéma R je v **první normální formě** (**1NF**), pokud obsahuje pouze atomické atributy. Tato podmínka je přirozená pro relační model. Relační schéma R je ve **druhé normální formě** (**2NF**) pokud je v 1NF a neobsahuje netriviální závislosti obsahující na levé straně vlastní podmnožinu klíče.

Relační schéma R je ve **třetí normální formě** (**3NF**) pokud je ve 2NF a pokud je $A_1A_2 \dots A_n \rightarrow B_1B_2 \dots B_m$ netriviální FD a buď $A_1A_2 \dots A_n$ je nadklíč nebo $B_1B_2 \dots B_m$ je podmnožinou klíče. Můžeme najít rozklad, jehož relace jsou ve 3NF, ale nejsou v BCNF. Schéma ve 3NF obsahuje všechny informace jako schéma původní, může ovšem obsahovat redundanci.

5.2 Test

1. V tabulce 5.8 vidíme relaci `Person`. Ukažte:
   a) Atributy relačního schématu.
   b) N-tice relace.
   c) Hodnoty atributů vybrané n-tice.
   d) Relační schéma relace.
   e) Vypište platné funkční závislosti.
   f) Je schéma v BCNF?

Tabulka 5.8: Relace `Person`

`rc`	`jmeno`	`prijmeni`
5608065344	Václav	Novák
7751011313	Květoslava	Horká
8352091825	Zuzana	Molavcová

Část III

Ostatní datové modely a dotazovací jazyky

Kapitola 6

Ostatní datové modely

Obsah

Cíl kapitoly:
V této kapitole jsou vysvětleny další datové modely jako je objektový, objektově-relační a datový model XML.

6.1 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 [3, 20, 37].

Příklad 6.1. (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 [31]. V SQL-99 můžeme definovat uživatelský typ, atributy tudíž nemusí být atomické. Typ `TPerson` obsahuje atribut `address` 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`.

SQL

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

6.2 Datový model XML

6.2.1 Úvod

Zatímco releační data považujeme za **data strukturovaná**, čistě textové dokumenty (angl. **plain text document**) za data **nestrukturovaná**, někde mezi těmito dvěma typy dat se nachází data **slabě strukturovaná**. Slabě strukturovaná data mají nějakou strukturu, často rekurzivní, nicméně schéma těchto dat může být volnější než v případě pevně nastaveného schématu relační databáze. Specifikace XML (a další navazující specifikace) nám umožní vytvářet a zpracovávat značkovací jazyky pro popis slabě strukturovaných dat.

eXtensible Markup Language (XML) [44, 29] je značkovací jazyk specifikovaný organizací World Wide Web Consortium[1] (W3C) v roce 1998[2]. Na XML se můžeme dívat ze dvou pohledů. Z pohledu prvního se nejedná o jazyk s pevnou množinou značek, jako např. HTML, ale umožňuje vytvářet vlastní množinu značek (obecně definovat vlastní schéma XML dokumentu). Vývojáři tedy nemusí pro různé aplikaci vytvářet vlastní soubory, implementovat pro ně parsery atd. Z druhého pohledu nabízí komplikovanější datovým model oproti relačnímu datovému modelu. Obecně je XML dokument modelován jako graf, pro naše představy bude plně postačující reprezentovat XML dokument jako **strom** (angl. **XML tree**).

Obecně se jedná jazyk popisující slabě strukturovaná data, který reprezentuje informace elementy, které mohou obsahovat další elementy a atributy. S XML se setkáme buď ve formě jazyka dodavatele třetí strany, např. MathML[3],

[1] `http://www.w3c.org/`
[2] Aktuální je 5. edice specifikace 1.0 z roku 2008.
[3] `http://www.w3.org/TR/MathML/`

SVG[4], XHTML[5], nebo ve formě vlastního jazyka, který potřebujeme pro uložení či export dat z vlastní aplikace. Podívejme se nyní na příklady prezentující dvě XML aplikace. V následujícím dokumentu MathML je popsán vzorec $[a+b]^{260}$:

```
<?xml version="1.0" encoding="UTF-8"?>
<math xmlns="http://www.w3.org/1998/Math/MathML">
  <mrow>
    <msup>
      <mfenced open="[" close="]">
        <mrow>
          <mi>a</mi>
          <mo>+</mo>
          <mi>b</mi>
        </mrow>
      </mfenced>
      <mn>260</mn>
    </msup>
  </mrow>
```

Výpis 6.1: Ukázka dokumentu MathML

V tomto dokumentu SVG je popsán obrázek 6.1:

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE svg PUBLIC "-//W3C//DTD␣SVG␣1.1//EN"
"http://www.w3.org/Graphics/SVG/1.1/DTD/svg11.dtd">

<svg width="100%" height="100%" version="1.1"
xmlns="http://www.w3.org/2000/svg">

<rect x="20" y="20" rx="20" ry="20" width="250" height
    ="100"
style="fill:red;stroke:black;stroke-width:5;opacity:0
    .5"/>
</svg>
```

Výpis 6.2: Ukázka dokumentu SVG

Aby byl dokument syntaktický korektní (označujeme jej jako dobře strukturovaný) musí splňovat určitá syntaktická pravidla.

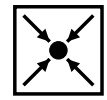
[4] http://www.w3.org/TR/SVG/
[5] http://www.w3.org/TR/xhtml1/

Obrázek 6.1: Obrázek vygenerovaný SVG souborem.

6.2.2 Dobře strukturovaný XML dokument

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 nebo značce), v textu je otevřen tzv. **počáteční značkou** (**start tag**), což je název elementu uvezený mezi znaky $<$ a $>$. Element je ukončen tzv. **koncovou značkou** (**end tag**): název elementu uvezený mezi znaky $</$ a $>$. 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 `/`. 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 6.2. (XML dokument). Na obrázku 6.3 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, což je podstatný 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 [10], např. FoxPro, Clipper), používají pořadová čísla záznamů v tabulkách. Tato vlastnost ovšem nemá žádnou podporu v původním relačním datovém modelu a u velkých SŘBD (jako Oracle [31] nebo DB2 [19]) 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>
```

Výpis 6.3: XML dokument reprezentující knihy a jejich autory

6.2.3 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** [44], z novějších uveďme **XML Schema** [47] 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 výpisu 6.3. Ve výpisech 6.4 a 6.5 vidíme schémata 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)>
]>
```

Výpis 6.4: Schéma XML dokumentu z výpisu 6.3 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>
```

Výpis 6.5: Schéma XML dokumentu z výpisu 6.3 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.

Validní dokument

Validní dokument

- XML dokument splňující definované schéma nazýváme **validní**.
- Obdoba schématu relační databáze:
  relační databáze - schéma relační databáze
  vs
  dokument - schéma dokumentu

- K čemu to je? XML sice umožňuje definovat libovolný jazyk, nicméně je vhodné zkontrolovat, zda konkrétní dokument obsahuje strukturu kterou od něj očekáváme.
- Jazyky popisující schéma: DTD, XML Schema (W3C), ...
- **Výhoda:** není nutné psát parser pro každý XML dokument. Pouze definujeme schéma ke kterému je dokument validní.

6.2.4 DTD

Document Type Definition - DTD

- DTD bylo specifikováno přímo ve specifikaci XML[6].
- Přestože časem vznikly sofistikovanější specifikace, pro svou jednoduchost je stále často využíváno pro definici schématu XML dokumentů.
- DTD je umístěno přímo v XML dokumentu nebo používáme referenci na externí zdroj s DTD.

[6] Aktuální je 5. editace specifikace 1.0: `http://www.w3.org/TR/REC-xml/`

Část IV

Vykonávání dotazů a fyzická implementace DBS

Kapitola 7

Zpracování dotazů v databázových systémech

Obsah

Cíl kapitoly:
V této kapitole je podrobněji vysvětlen způsob vykonávání dotazů v relačních SŘBD. Zaměřujeme se především na pojem plán vykonání dotazu.

7.1 Úvod

Čtenáře jistě napadlo jakým způsobem databázový systém vykonává uživatelský dotaz a zda je možné ovlivnit rychlost vykonávání takového dotazu. Vysvětlení této problematiky je náplní této a následujících kapitol. Existuje řada technik umožňují zrychlení provedení databázových operací, můžeme například použít parametrizované dotazy, hromadné operace nebo vhodně nastavit úroveň izolace transakce. Další možností jak ovlivnit čas vykonání dotazu je tzv. **fyzický návrh databáze**, tedy výběr datové struktury pro uložení dat, která nejlépe odpovídá operacím vykonávaným nad touto tabulkou. S fyzickým návrhem sou-

visí pojem vyhodnocení dotazu, tedy proces kdy SŘBD (konkrétně **optimalizátor**) hledá cestu jak nejrychleji provést uživatelský dotaz a dotaz poté vykoná. Podívejme se nejprve na motivační příklad.

```
SELECT Kurz.* FROM Student, Kurz, Student_Kurz
  WHERE Student.lname='Sobota' AND
        Student.login=Student_Kurz.login AND
        Student_Kurz.rok = 2009 AND
        Student_Kurz.idKurz = Kurz.idKurz
```

V souvislosti s vykonáním dotazu v SŘBD nás mohou napadnout některé otázky. Jak je tento dotaz vykonán? Jaké je pořadí jednotlivých operací? Může pořadí operací ovlivnit rychlost vykonání dotazu? Jak jsou data v uložena v databázi? Je možné uložení dat měnit a můžeme tak ovlivnit rychlost provedení dotazu? Odpovědi na tyto otázky se dozvíme v této a následujících kapitolách.

Jazykové okénko:
V souvislosti s vyhodnocením dotazu můžeme nalézt v české literatuře celou řadu pojmů, např. vykonání, vykonávání, provedení, zpracování, vyhodnocení, spuštění. Dle našeho názoru se ale jedná o různé překlady tří anglických slov používaných ve dvou významech. Prvním slovem je **processing** v kontextu query processing (tedy zpracování dotazu). V tomto případě používáme české slovo vykonání nebo zpracování. Dalšími slovy jsou **evaluation** a **execution** v kontextu query evaluation/execution plan (plán vyhodnocení dotazu). V tomto případě používáme český překlad vyhodnocení.

SŘBD se při zpracování dotazu nejprve snaží odstranit duplicitní syntaxi jazyka SQL k čemuž můžeme použít model jazyka SQL – relační algebru. Relační algebra byla popsána dříve v kapitole 4.2, nyní si pouze připomeneme základní operace, které uvidíme později v tzv. plánech vyhodnocení dotazu.

Příklad 7.1. (Operace relační algebry)

Mějme následující tabulky: `Dodavatel` (softwarových aplikací), `Produkt` (softwarová aplikace), `DodavatelProdukt` (vazební tabulka nesoucí informaci o objemu prodeje produktu dodavatelem).

`Dodavatel D`

D#	Nazev	Zeme
D1	IBM	US
D2	Oracle	US
D3	Microsoft	US

`Produkt P`

P#	Typ	Prodej
P1	RDBMS	1 000
P2	OS	1 000
P3	IDE	100

`DodavatelProdukt DP`

D#	P#	Prodej
D1	P1	200
D1	P2	100
D1	P3	10
D2	P1	350
D3	P1	200
D3	P2	900
D3	P3	80

Nyní se podívejme na základní operace, které později uvidíme v plánech vykonávání dotazů.

Selekce (restrikce): vraťˊ všechny prodejce s objemem prodeje $>= 1\,000$:

P

P#	Typ	Prodej
P1	RDBMS	1 000
P2	OS	1 000
P3	IDE	100

$\Rightarrow$

$\sigma_{P.Prodej>=1000}$

P#	Typ	Prodej
P1	RDBMS	1 000
P2	OS	1 000

Projekce: vraťˊ jména dodavatelů:

D

D#	Nazev	Zeme
D1	IBM	US
D2	Oracle	US
D3	Microsoft	US

$\Rightarrow$

$\Pi(D.Nazev)$

Nazev
IBM
Oracle
Microsoft

Spojení (join): vraťˊ dodavatele a jejich prodeje jednotlivých produktů, které prodávají:

DP

D#	P#	Prodej
D1	P1	200
D1	P2	100
D1	P3	10
D2	P1	350
D3	P1	200
D3	P2	900
D3	P3	80

$\bowtie$

D

D#	Nazev	Zeme
D1	IBM	US
D2	Oracle	US
D3	Microsoft	US

$=$

DP $\bowtie$ D

D#	P#	Prodej	Nazev	Zeme
D1	P1	200	IBM	US
D1	P2	100	IBM	US
D1	P3	10	IBM	US
D2	P1	350	Oracle	US
D3	P1	200	Microsoft	US
D3	P2	900	Microsoft	US
D3	P3	80	Microsoft	US

Příklad 7.2. (Dotaz: vraťjména dodavatelů, kteří dodávají produkt P1).

Výraz relační algebry: ((DP $\bowtie$ D) $\sigma_{P\#='P1'}$) $\Pi(D.Nazev)$

DP $\bowtie$ D

D#	P#	Prodej	Nazev	Zeme
D1	P1	200	IBM	US
D1	P2	100	IBM	US
D1	P3	10	IBM	US
D2	P1	350	Oracle	US
D3	P1	200	Microsoft	US
D3	P2	900	Microsoft	US
D3	P3	80	Microsoft	US

((DP $\bowtie$ D) $\sigma_{P\#='P1'}$) $\Pi(D.Nazev)$

Nazev
IBM
Oracle
Microsoft

Nyní se podívejme na různé varianty vykonání dotazu z příkladu 7.2. Uvažujme databázi obsahující 100 dodavatelů, 10 000 záznamů v tabulce `DP` z nichž pouze 50 se týká produktu `P1`. Pro jednoduchost uvažujme, že tabulky `DP` a `D` jsou reprezentovány dvěma soubory na disku.

Vykonání dotazu, varianta 1:

Pokud nebudeme uvažovat žádnou optimalizace, pak se dotaz vykoná v tomto pořadí $\bowtie$, σ, Π (tedy spojení, selekce, projekce):

1. Spojení `DP` a `D`[1]: tento krok zahrnuje načtení každého ze 100 dodavatelů 10 000$\times$ (každý dodavatel pro 10 000 záznamů `DP`). Výsledek (spíše mezivýsledek), který obsahuje 10 000 záznamů, je zapsán na disk nebo uložen do hlavní paměti[2].

2. Selekce `P#='P1'` na výsledek kroku 1: nyní načteme 10 000 záznamů (výsledek předchozího kroku) a provedeme selekci, výsledkem je 50 záznamů.

3. Projekce `Nazev` na výsledek kroku 2: výsledkem je 50 záznamů.

Vykonání dotazu, varianta 2:

Dotaz provedený dle následujícího plánu vrací stejný výsledek, avšak je proveden efektivněji. V tomto případě jsou operace vykonány v pořadí σ, $\bowtie$, Π (tedy selekce, spojení, projekce):

[1] Uvažujeme nejjednodušší algoritmus zahnízděnými cykly, v kapitole **??** jsou popsány další algoritmy.

[2] Problémem mezivýsledků je v tom, že se nemusí vlézt do hlavní paměti, především v případě rozsáhlých dat nebo víceuživatelského přístupu.

1. Selekce `P#='P1'` na `DP`: tento krok zahrnuje čtení 10 000 záznamů, výsledkem je 50 záznamů.
2. Spojení výsledku kroku 1 a relace `D` pro atribut `D#`: tento krok zahrnuje načtení 100 dodavatelů a provedení operace spojení. Výsledkem je 50 záznamů.
3. Projekce `Nazev` na výsledek kroku 2: výsledkem je 50 záznamů.

Nyní porovnejme obě varianty vykonání dotazu. Uvažujeme, že data jsou v souborech uložena ve **stránkách** o velikosti 2 kB, jeden přístup na disk pak znamená načtení nebo zapsání jedné stránky (v kapitole **??** je popsáno proč jsou datové struktury používané v databázových systémech stránkované). Pro zjednodušení uvažujeme, že stránka obsahuje 100 záznamů (v případě obou tabulek). Tabulka `DP` je tedy uložena ve 100 stránkách, tabulka `D` v 1 stránce.

Abychom byli schopni porovnat různé plány vyhodnocení dotazu, musíme nějakým způsobem měřit cenu vykonání dotazu. V databázových systémech se používají zejména ceny vstupně-výstupní (**IO Cost**) a procesorová (**CPU Cost**). Cena IO je měřena počtem přístupů ke stránkám tabulek, cena CPU je měřena počtem operací provedených během vykonávání dotazu. Pozorného čtenáře napadne, že čas vykonání dotazu je dán také tím, zda jsou stránky tabulek umístěny v hlavní paměti nebo na disku. V kapitole **??** ukazujeme, že rozlišujeme tzv. logické a fyzické přístupy ke stránkám datových struktur a rozlišujeme tak oba druhy přístupů.

Porovnání obou plánů je následující (viz tabulka 7.1):

- IO cena: v případě 1. varianty se provede 302 přístupů ke stránkám, v případě 2. varianty 104 přístupů.
- CPU cena: v případě 1. varianty se provede $10\,000 \times 100 = 10^6$ operací, v případě 2. varianty $50 \times 100 = 5 \times 10^3$ operací.

Přestože jsou oba plány ekvivalentní (tedy dávají stejné výsledky); 2. varianta je více než $3\times$ efektivnější z pohledu počtu přístupů ke stránkám a $1\,000\times$ efektivnější z pohledu počtu operací.

Databázový systém může provádět další optimalizace. Například zavedením indexu na tabulku `DP` vykonáme selekci na méně diskových přístupů (50 diskových přístupů namísto 10 000), což znamená že dostaneme plán vykonání dotazu, který je $7\,000\times$ efektivnější než původní plán z pohledu přístupů ke stránkám. Další možností je provést projekci před selekcí a spojením, v tomto

Tabulka 7.1: Porovnání dvou plánů vykonání dotazu.

	((DP ⋈ D) $\sigma_{P\#='P1'}$) $\Pi(Nazev)$	((DP $\sigma_{P\#='P1'}$) ⋈ D) $\Pi(Nazev)$
IO cena	302	104
CPU cena	1 030 000	10 100

případě je ovšem nutné provést projekci na všechny atributy kromě `Prodej` a `Zeme`, jelikož budou použity v selekci a spojení.

7.2 Zpracování dotazu

Zpracování dotazu má v SŘBD na starost **procesor dotazu** (angl. **query processor**), zpracování zahrnuje **překlad dotazu** a **vykonání dotazu**. Dotaz je překládán ve třech fázích (viz obrázek 7.1):

1. parsování dotazu,
2. výběr logického plánu dotazu,
3. výběr fyzického plánu dotazu.

Parsování dotazu zahrnuje převod do nějaké interní formy, snahou je eliminovat syntaxi dotazovacího jazyka (SQL). Interní forma je nejčastěji nějaký druh **stromu dotazu** (angl. **query tree**), což je reprezentace výrazu relační algebry (viz obrázek 7.2). V této fázi probíhá také zpracování pohledů: pohledy jsou nahrazeny definicí.

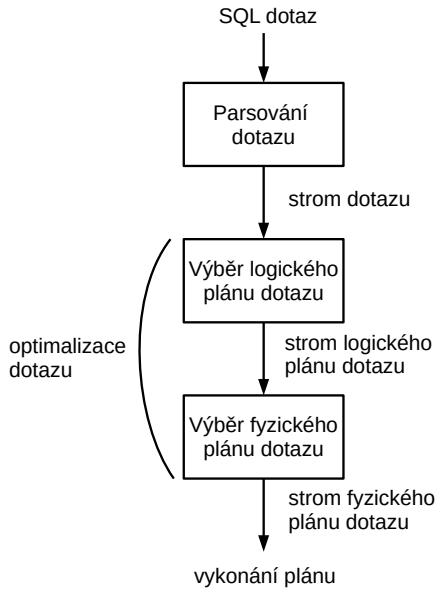
SQL umožňuje dotaz vyjádřit mnoha způsoby, ve fázi **Výběr logického plánu dotazu** hledáme efektivnější tvar dotazu než nabízel původní dotaz, mluvíme o **přepsání dotazu** (angl. **query rewritting**). Můžeme například použít transformační pravidla přepisují výraz na výraz ekvivalentní (tedy výraz který vrací stejný výsledek), např. výraz:

```
(A JOIN B) WHERE selekce na A
```

může být přepsán na ekvivalentní výraz:

```
(A WHERE selekce na A) JOIN B
```

Který bude pravděpodobně zpracovávat méně záznamů při operaci spojení. Je tedy zřejmé, že dotaz není ve skutečnosti vykonán tak jak jej zadá uživatel. Výsledkem této fáze je strom logického plánu dotazu.

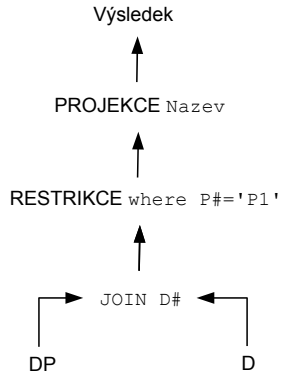


Obrázek 7.1: Fáze překladu dotazu

Fáze **Výběr fyzického plánu dotazu** zahrnuje výběr algoritmů implementujících operátory logického plánu, tzv. fyzické operace, a výběr pořadí těchto operací. Procesor v této fázi uvažuje velikost tabulek, existenci indexů, distribuci hodnot, shlukování uložených dat a ostatní informace o datech, které načítá ze systémového katalogu. Každé fyzická operaci je přiřazena hodnota IO ceny a CPU ceny. Tyto ceny (nejčastěji IO cena) určují celkovou cenu fyzického plánu. Nejlevnější fyzický plán jak poté vykonán.

7.3 Struktura následujících kapitol

V následujících kapitolách podrobně vysvětlíme problematiku vykonávání dotazů a fyzické implementace DBS. V následující kapitole je popsáno vytváření plánu vyhodnocení dotazu v databázovém systému. V kapitole 9 jsou popsány jednotlivé operace plánu vyhodnocení dotazu spolu se základní fyzickou implementací databázových systémů. V kapitole 10 jsou popsány obecné prin-



Obrázek 7.2: Stromu dotazu z předchozí kapitoly

cipy fyzického návrhu zároveň s konkrétní implementací v Oracle a SQL Server. Kapitola 11 popisuje jednotlivé datové struktury využívané v databázových systémech pro uložení dat. Kapitola 12 popisuje algoritmy operací v plánech vyhodnocení dotazu a jejich složitost. V kapitole 13 jsou popsány základní vlastnosti disků a diskových polí.

Uvedené pořadí kapitol provází čtenáře od jednodušších příkladů a popisu obecných koncepcí až po detailní popis algoritmů a datových struktur. Pokud čtenář preferuje opačný postup, tedy rád se seznamuje nejprve s detaily a teprve poté s obecnými koncepcemi, měl by nejprve přečíst kapitoly věnující se diskům a diskovým polí (kapitola 13), datovým strukturám (kapitola 11), algoritmům operací plánu vyhodnocení dotazu (kapitola 12) a teprve po jejich přečtení by měl věnovat pozornost popisu plánu vyhodnocení dotazu (kapitola 9) a fyzické implementaci DBS (kapitola 10).

Kapitola 8

Úvod do fyzické implementace databázových systémů

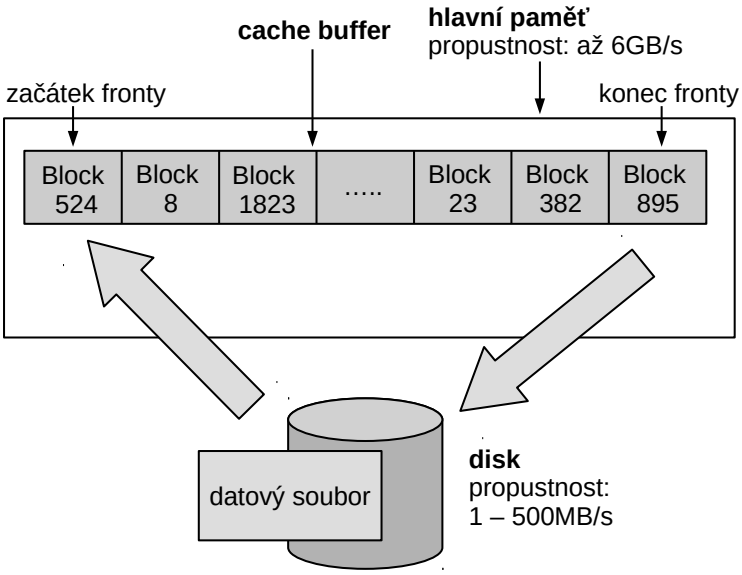
Obsah

Cíl kapitoly:
V této kapitole jsou popsány obecné principy související s plánem vyhodnocení dotazu, především operace plánu a základní fyzická implementace DBS. Příklady pak demonstrují možnosti ladění fyzického návrhu databáze.

8.1 Stránkování datových struktur

Data (např. záznamy), které uživatel vkládá do databáze, jsou uložena v různých datových strukturách, které budou popsány v následujících kapitolách. Tyto da-

tové struktury se skládají ze **stránek** (uzlů, bloků), které jsou umístěny na disku, tak aby databázový systém zaručil trvalost dat. Velikost stránky je volena jako násobek 2 kB, nejčastěji 8kB. Proč? Stránky jsou primárně umístěny na disku, stránka musí tedy mít velikost násobku diskového sektoru (512 B) a alokační jednotky souborového systému (nejčastěji 2 kB).



Obrázek 8.1: Uložení stránek s daty v hlavní paměti a na disku.

Čtení a zápis na disk jsou řádově pomalejší operace než čtení a zápis do paměti (v následující kapitole si ukážeme, že diskové operace mohou být až $1000\times$ pomalejší než přístupy do paměti), proto je tedy vhodné aby většina stánek s daty (nejlépe všechny) byla umístěny v hlavní paměti. Pro tento účel alokuje databázový systém oblast v hlavní paměti, která se nazývá **cache buffer**. Cache buffer si můžeme představit jako frontu stránek (viz obrázek 8.1), kde první stránka byla použita jako poslední a poslední stránka jako první. Pokud DBS požaduje načtení nějaké stránky z disku a cache je plná, pak dojde k uložení poslední stránky na disk a na volné místo je načtena požadovaná stránka. Odložení této stránky má následující význam: k ostatním stránkám přistupoval DBS až po přístupu k odkládané stránce a můžeme se oprávněně domnívat, že jsou v daný okamžik důležitější než odkládaná stránka. Tento algoritmus nazýváme **LRU** (Least Recently Used), tedy nejméně aktuálně používaná (stránka je od-

straněna z hlavní paměti). **Přístup ke stránce** (angl. **page access**) nazýváme operaci, která provede vyhledání stránky v cache případně načtení stránky z disku po kterém typicky následuje čtení nebo aktualizace záznamů stránky.

Na obrázku 8.1 vidíme, že naposledy zpracovávaná stránka je blok s pořadovým číslem 524, poslední stránkou ve frontě je naopak blok 895 (ten bude odstraněn z cache při dalším požadavku na stránku, která není v hlavní paměti). Pořadová čísla stránek nejčastěji definují pořadí stránky v datovém souboru, stránka 8 je tedy umístěna v souboru od pozice $8 \times 8\,kB$, tedy od pozice 65 536. Pozorného čtenáře napadne, co se stane s trvalostí dat, pokud budou všechna data umístěna v hlavní paměti (což je bezpochyby žádoucí s ohledem na výkon DBS) a dojde k pádu databázového systému. V takovém případě odkazujeme na kapitolu 14 popisující funkci log souboru.

Jelikož propustnost paměti je řádově $10 - 1000\times$ vyšší než propustnost diskových operací, rozlišujeme v databázových systémech dva typy přístupů ke stránkám: **logické** a **fyzické přístupy** (angl. **logical** and **physical accesses**). Logický přístup vyjadřuje přístup ke stránce libovolné datové struktury, fyzický přístup značí načtení nebo zápis stránky na disk. Pokud je při požadavku stránka nalezena v cache buffer mluvíme o **cache hit**, v opačném případě o **cache miss**. Míru úspěšnosti použití cache buffer pak měříme ukazatelem **cache hit rate**: cache hit rate = (cache hits / počet logický čtení) * 100 [%].

Pokud budou všechna data databáze umístěna v cache buffer, bude počet fyzických přístupů u všech databázových operací roven 0 a cache hit rate = 100%. Pokud naopak bude databáze v porovnání s velikostí cache buffer velká, bude se počet fyzických přístupů blížit počtu logických přístupů a cache hit rate bude klesat k 0. Z pohledu výkonu je vhodné aby byla celá databáze umístěna v paměti, což není ale vždy možné. V takovém případě mluvíme o pravidle 90:10, tzn. pokud chceme dosáhnout rozumného výkonu DBS, pak musí být nejméně 90% často používaných umístěno v hlavní paměti a jen 10% na disku.

8.2 Propustnost diskových operací

Výkon diskových operací se značně liší v závislosti na tom, zda se jedná o čtení nebo zápis a především zda se jedná o sekvenční nebo náhodné operace. Příkladem sekvenčního čtení může být čtení souboru od začátku do konce, náhodné čtení znamená čtení souboru po částech s přeskakováním na náhodné pozice v souboru. V tabulce 8.1 vidíme výkon sekvenčních a náhodných přístupů

pro různé disky[1]: dva disky SATA a SAS v diskovém poli RAID1 a RAID10, SSD disk zapojený v notebooku bez RAID a SDHC kartu. Zatímco první dva disky jsou klasické disky obsahující mechanické části (např. magnetické disky pro záznam dat, čtecí a zapisovací hlavičky), další dva disky používají pro záznam dat polovodičové paměti.

Operace	SATA RAID1	SAS RAID10	Samsung SSD 840 Pro Series	SDHC
Sekvenční čtení	159.3	416.8	516.4	12.1
Sekvenční zápis	177.9	249.6	494.3	11.0
Náhodné čtení 512KB	173.5	239.0	465.5	11.8
Náhodný zápis 512KB	213.1	237.3	475.7	1.5
Náhodné čtení 4KB	2.5	4.2	30.3	3.7
IOPS	613.3	1 020.8	7 392.4	902.6
Náhodný zápis 4KB	12.1	15.9	63.8	1.4
IOPS	2 946.4	3 868.8	15 574.8	344.3

Tabulka 8.1: Výkon sekvenčních a náhodných operací pro různé disky [MB/s].

V tabulce 8.1 vidíme, že SSD dosahuje nejvyššího výkonu pro sekvenční čtení i zápis, propustnost se pohybuje kolem 500MB/s. SAS RAID10 dosahuje propustnosti kolem 420MB/s pro sekvenční čtení, výkon sekvenčního zápisu ale klesá k 250MB/s. SDHC karta dosahuje přibližně $40\times$ nižšího výkonu pro zápis i čtení než SSD. Náhodné čtení po blocích o velikosti 512 kB většinou kopíruje výkon sekvenčních operací. U náhodných operací po blocích o velikosti 4 kB vidíme prudký pokles výkonu. K tomuto poklesu dochází přestože disky obsahují vlastní vyrovnávací paměť (o velikosti jednotek MB): úspěšnost cachování je v případě náhodných operací nízká a klesá k nule. Jelikož se jedná řádově o stejnou velikost bloku jako je používaná v databázových systémech, podíváme se nyní na výkon těchto náhodných operací více podrobně.

U disků SATA RAID1 a SAS RAID10 klesl výkon náhodného čtení až o dva řády, u SSD klesl výkon čtení $17\times$. Vidíme, že výkon náhodných operací klasických disků (SAS, SATA) je výrazně nižší v porovnání s disky obsahujícími polovodičové paměti (SSD, SDHC). Důvodem je především doba přesunu hlavičky a natočení disku. V tabulce jsou pro náhodné operace rovněž uvedeny **počty přístupů k blokům za sekundu** (angl. **IO operations per second - IOPS**). Vidíme, že v nejhorším případě klesá počet přístupů k blokům u operace čtení až na hodnotu 613.

[1] Měřeno aplikací CrystalDiskMark – `http://crystalmark.info/software/CrystalDiskMark/index-e.html`

Co to znamená? Zatímco sekvenční načtení 10 GB souboru na SSD disku bude trvat 19,8 s, načtení souboru náhodnými operacemi bude trvat 338 s. U klasických disků je situace ještě výrazně horší. V následujících kapitolách tedy budeme často pozorovat, že databázový systém využívá sekvenční čtení stránek namísto náhodného čtení menšího počtu stránek. Dle výše popsaných vlastností diskových operací, bude čas vykonání operace i tak nižší. Je zřejmé, že pro konkrétní disk je DBS schopen spočítat kolik stránek se ještě vyplatí načíst náhodně namísto sekvenčního čtení všech stránek datové struktury. Podobný jev můžeme pozorovat i při čtení a zápisu do paměti, nicméně v kapitole **??** ukážeme, že náhodné a sekvenční operace v hlavní paměti nevykazují v případě databázových systémů výrazný rozdíl.

8.3 Základní fyzická implementace relační databáze

Ve většině databázových systémů najdeme tyto datové struktury pro uložení dat:

- **tabulka typu halda** (angl. **heap table**) – jedná se o stránkované sekvenční pole (dále v textu budeme mluvit jen o tabulce),
- **index** – nejčastěji stránkovaný B-strom nebo nějaká jeho varianta (např. B^+-strom).

Později si ukážeme, že existují i další datové struktury implementované v některých databázových systémech. Důležité je, že neexistuje datová struktura, která by měla pro všechny situace optimální vlastnosti, volba vhodné datové struktury tedy vždy závisí na konkrétních datech a dotazech.

Nyní se podívejme na základní operace těchto datových struktur a především na složitosti a ceny těchto operací. V následujícím textu budeme uvažovat n jako počet položek uložených v datové struktuře (např. záznamů v tabulce) a N jako počet stránek datové struktury. Složitosti operací budeme uvádět jak z pohledu počtu operací (kde parametrem složitosti bude n), tak z pohledu přístupů ke stránkám (kde parametrem složitosti bude N). Tyto složitosti pak budou určovat ceny operací: **cenu procesorovou** (angl. **CPU cost**) a **cenu vstupně výstupní** (angl. **IO Cost**).

8.3.1 Operace nad tabulkou

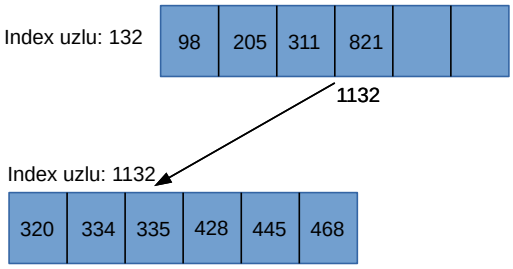
Při **vkládání** záznamu do tabulky je záznam vložen nejčastěji na první volné místo v poslední stránce tabulky. Pokud v poslední stránce již není místo, je vytvořena nová stránka a záznam je vložen do této stránky. V nejlepším případě je cena IO rovna 1, efektivita vkládání je tedy silnou stránkou této datové struktury. Každému záznamu je přiřazeno jedinečné číslo, tzv. **ROWID** (nebo RID), které je nejčastěji 32bitové a obsahuje jedinečné číslo stránky a pořadí záznamu v bloku. Pokud tedy databázový systém získá ROWID (např. z indexu jak si později ukážeme), může na jeden logický přístup získat kompletní záznam z tabulky typu halda. V tomto případě mluvíme o operaci **přístup ke stránce tabulky pomocí ROWID**.

Sekvenční průchod tabulku (angl. **table scan**) postupně načítá stránky tabulky, které dále zpracovává (např. provádí selekci pokud ta je součástí dotazu uživatele). Výhodou této fyzické operace je maximální výkon čtení z disku (pokud data nejsou uložena v hlavní paměti), kde dosahujeme propustnosti až 500 MB/s. Nevýhodou je přístup ke všem stránkám, přestože výsledkem dotazu může být například jen jeden záznam. Složitost operace je $O(n)$ resp. $O(N)$.

8.3.2 Operace nad indexem (B-stromem)

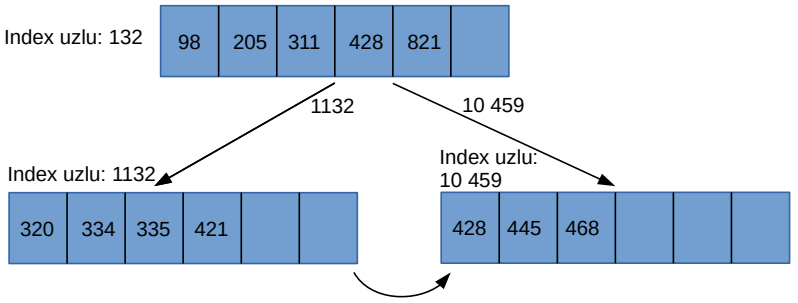
V této kapitole budeme popisovat vybrané vlastnosti B-stromu, které pak zužitkujeme při popisu jednotlivých fyzických operací plánu dotazu. Datová struktura B-strom je popsána v kapitole **??**, v následujícím textu počítáme se základními znalostmi čtenáře o této datové struktuře.

Vložení do B-stromu znamená průchod stromem od kořene k listu do kterého je vložena položka (klíč a ROWID záznamu v tabulce). Pokud do některého uzlu není možné vložit další položku, dojde k rozštěpení tohoto uzlu, což v nejhorším případě může znamenat rozštěpení kořenového uzlu, vytvoření nového kořenového uzlu a zvýšení výšky stromu o 1. Pokud nedojde ke štěpení stránek, přistupujeme přesně k $h + 1$ stránkám datové struktury, kde h je výška stromu. Při 8 kB stránce se počet položek v uzlech pohybuje až ve stovkách a výška stromu se pak pohybuje řádově v jednotkách pro strom s milióny položek. Na první pohled je tedy operace vkládání relativně levná (ale dražší než vkládání do tabulky), nicméně přístupy ke stránkám jsou náhodné, což zejména u přístupů na disk představuje velký výkonnostní problém. Jak k těmto náhodným přístupům v B-stromu dochází a proč je to vlastně výkonnostní problém? Podívejme se na následující příklad.



Obrázek 8.2: Část B-stromu se dvěma uzly.

Příklad 8.1. Podívejme se na obrázek 8.2 kde je zobrazena část B-stromu obsahující dva uzly: rodičovský uzel s indexem 132 a dětský uzel s indexem 1 132 pro interval klíčů $[311, 821)$. Pokud chceme do tohoto stromu vložit klíč 421, pak musí dojít k rozštěpení stránky. Pokud poslední uzel vytvořený ve stromu má index 10 458, pak nově vytvořený uzel bude mít index 10 459. Tato situace je ukázána na obrázku 8.3. Je tedy zřejmé, že stránky ve vztahu rodič-dítě nebo sourozenci nemusí být na disku umístěny vedle sebe, důsledkem jsou náhodné přístupy na disku při průchodu stromem.



Obrázek 8.3: Část B-stromu z obrázku 8.2 po vložení klíče 421.

Bodový dotaz v B-stromu (angl. **point query** nebo **unique scan**) hledá zda ve stromu je či není uložen klíč k. Pro každou úroveň stromu je zpracován právě jeden uzel; cena IO je tedy $h + 1$ (jedná se řádově o jednotky přístupů). **Rozsa-**

hový dotaz v B-stromu (angl. **range query** nebo **range scan**) vrací všechny klíče ze stromu z intervalu $[k_l, k_h]$. Tento dotaz je vykonán ve dvou fázích: nejprve se bodovým dotazem najde klíč k_l a pak se prochází všechny sousední listové uzly (každý listový uzel obsahuje ukazatel na následující listový uzel) dokud jsou klíče $\leq k_h$. Cena IO je tedy rovna $h + 1$ + počet načtených listových uzlů, kde počet načtených listových uzlů je přímo úměrný velikosti výsledku. Bodový i rozsahový dotaz jsou vykonány náhodnými přístupy ke stránkám. Zatímco u bodového dotazu bude počet přístupů řádově v jednotkách, u rozsahového dotazu se může, pro dotazy s rozsáhlými výsledky, jednat o velký počet náhodných přístupů.

8.3.3 Srovnání tabulky a B-stromu

Náhodné přístupy vykonávané při provádění rozsahových dotazů vedou k významných důsledkům při provádění SQL dotazu: jakmile optimalizátor zjistí, že výsledek rozsahového dotazu v B-stromu bude obsahovat větší počet klíčů, využije pro realizaci selekce sekvenční průchod tabulkou namísto rozsahového dotazu ve stromu. Pokud budeme uvažovat SAS disk z předchozí kapitoly, kde sekvenční přístupy jsou $100\times$ rychlejší než přístupy náhodné, pak optimalizátor použije rozsahový dotaz v indexu jen v případě, že při provádění dotazu načte mnohem méně stránek než je počet stránek tabulky / 100. Čtenáře by jistě zajímalo, jak může optimalizátor znát velikost výsledku resp. počet načtených stránek před vykonáním samotné operace. Optimalizátory používají metody odhadu velikosti výsledku, především na základě statistik uložených v databázi.

Až doposud jsme porovnávali výkon operací tabulky a indexu v případě, kdy jsou stránky datových struktur načítány z disku. Pojďme se nyní zamyslet nad počtem operací, tedy ceně CPU, která je kromě ceny IO, také používána v databázových systémech pro stanovení ceny plánu provedení dotazu.

Plán obsahuje jednotlivé operace provedené při vykonávání dotazu a jejich ceny, typicky používáme dva typy cen:

- Častěji **cenu přístupů ke stránkám** (IO cena, angl. **IO Cost**) – tedy počet přístupů (operací čtení nebo zápisu) ke stránkám datových struktur.
- Méně často **cenu mikroprocesorového času** (CPU cena, angl. **CPU Cost**) – počet operací vykonaných CPU při provádění dotazu.

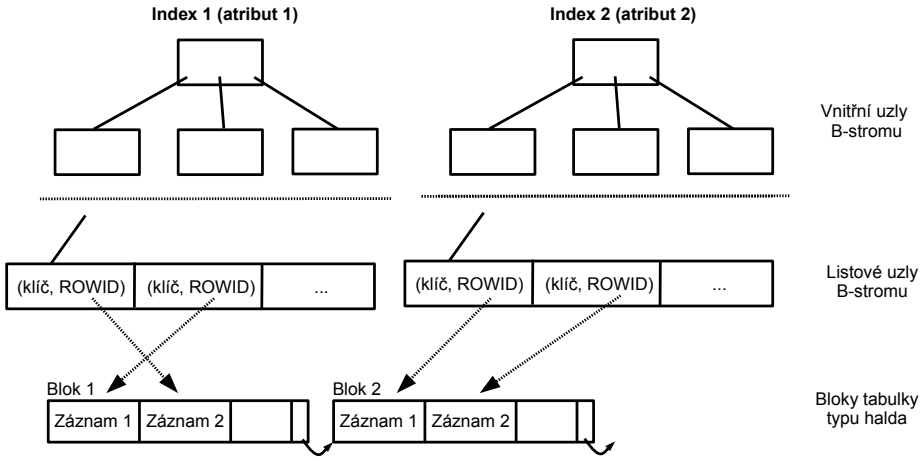
Důvodem proč se častěji používá IO cena je obecně dražší operace čtení/zápisu stránky z disku než operace procesoru (zejména v případě náhodného čtení). Nyní obě datové struktury popišme:

- **Tabulka typu halda** (angl. **heap table**): jedná se o pole obsahující bloky (stránky) ve kterých jsou uloženy záznamy. Každý záznam je identifikován pomocí **ROWID** (také RID). Vyhledání je sekvenční, složitost vyhledávání je tedy $O(n)$, kde n je počet záznamů v tabulce. IO složitost (tedy počet načtených stránek) je $O(n/C)$, kde C je průměrný počet záznamů ve stránkách tabulky. Pro jednoduchost uvažujme že záznamy jsou přidávány vždy na konec tabulky, v takovém případě je IO cena vkládání 1.
- **Index:** jedná se nejčastěji o B^+-strom kde klíčem je indexovaný atribut, ROWID přiřazené ke každému klíči odkazuje na příslušný záznam v tabulce typu halda (index tedy neobsahuje hodnoty všech atributů). Dotaz na klíč (tzv. **bodový dotaz**) má IO cenu $h + 1$, kde výška stromu $h = \lfloor \log_C(n) \rfloor$ (typicky 3–5), C je průměrný počet dětí uzlů (typicky 100–500).

Na první pohled se zdá, že provádění bodového dotazu v indexu je vždy rychlejší než sekvenční průchod tabulkou protože $n/C \gg h + 1$. Jelikož logické pořadí uzlů B-stromu neodpovídá vždy fyzickému pořadí uzlů na disku, dotazy jsou prováděny pomocí náhodných přístupů. V případě bodového dotazu je tedy $h+1$ přístupů ke stránkám provedeno náhodně. Podrobnější popis výkonu sekvenčních a náhodných přístupů najdeme v kapitole 13, nyní zjednodušeně konstatujme, že náhodné přístupy jsou až $100\times$ pomalejší než sekvenční čtení či zápis. Problém náhodných přístupů se projeví zejména u rozsahových dotazů, který vrací větší počet záznamů ve výsledku. V tomto případě B-strom náhodně čte větší počet listových uzlů B-stromu. Pokud optimalizátor zjistí, v rámci odhadu ceny dotazu, že by mělo dojít k náhodnému čtení většího počtu stránek B-stromu (např. k více než 10% listových stránek indexu), raději je příslušná operace provedena sekvenčním průchodem tabulkou typu halda. V plánech by nás tedy nemělo překvapit použití sekvenčního průchodu tabulkou, přestože je pro daný atribut k dispozici index.

Pozorný čtenář jistě upozorní na to, že DBS se snaží držet co největší počet stránek v hlavní paměti (v tzv. **cache bufferu**), kvůli maximálnímu výkonu přístupu k datům. Pokud je celá datové struktura uložena v hlavní paměti, nejsou náhodné přístupy tak problematické: rychlá L2 cache CPU je maximálně využita díky 8 kB stránkám. Databázový systém, ale nemusí nutně vždy dopředu vědět, které stránky datové struktury jsou umístěny v paměti nebo pouze na disku (pokud neuvažujeme triviální případ kdy jsou všechny stránky umístěny v hlavní paměti nebo naopak pouze na disku).

Na obrázku 8.4 vidíme základní fyzickou implementaci relačního datového modelu kde záznamy jsou uloženy v blocích tabulky typu halda a k tabulce jsou vytvořeny dva indexy (v tomto případě B-stromy). Indexy v listových uzlech obsahují klíče a ROWID, které ukazuje na příslušný záznam v tabulce typu



Obrázek 8.4: Základní fyzická implementace relačního datového modelu.

halda[2]. Pokud dotaz obsahuje selekci na hodnotu klíče jednoho z indexů, pak je bodovým nebo rozsahovým dotazem nalezen klíč (nebo klíče) a jeho ROWID. Pomocí ROWID pak dalším přístupem ke stránce tabulky typu halda získáme příslušný záznam. V tomto případě jsou všechny přístupy ke stránkám náhodné. Přestože velikost indexu je obecně menší než velikost tabulky (index obsahuje pouze hodnoty klíč a ROWID, tabulka obsahuje hodnoty všech záznamů), tato velikost roste lineárně s počtem indexů a velikost většího počtu indexů tak může snadno překročit velikost tabulky. Později popíšeme další problémy související s vytvářením většího počtu indexů.

Příklad 8.2. (Základní fyzická implementace pro relační datový model 1).

Uvažujme tabulku s počtem záznamů $n = 10^6$, velikost záznamu bude 8B pro tabulku (celý záznam) i index (klíč + ROWID), velikost bloku 8 kB. Počet bloků tabulky typu halda bude 2 439. Při provádění selekce nad tabulkou typu halda bude provedeno 2 439 sekvenčních přístupů ke stránkám. Počet bloků indexu bude 4 878 pokud uvažujeme 50% využití stránek. Jelikož výška stromu je 2, dotaz na klíč se provede 3 náhodnými přístupy ke stránkám B-stromu.

Příklad 8.3. (Základní fyzická implementace relačního datového modelu 2).

Mějme následující tabulky:

[2]ROWID je nejčastěji hodnota složená z jedinečného čísla bloku a pořadí položky v bloku na kterou odkazuje, nejedná se tedy o ukazatel do paměti.

```
CREATE TABLE Producer (
  id INT PRIMARY KEY,
  name VARCHAR(50) NOT NULL,
  address VARCHAR(50) NOT NULL)

CREATE TABLE Store_item (
  id INT PRIMARY KEY,
  name VARCHAR(50) NOT NULL,
  idProducer INT REFERENCES Producer NOT NULL)
```

Tabulka `Producer` obsahuje 100 000 záznamů, velikost stránky je 8 kB, počet stránek tabulky je 512[3], průměrná délka záznamu je 24B, průměrný počet záznamů ve stránce je 195. Tabulka `Store_item` obsahuje 1 000 000 záznamů: počet stránek tabulky je 4 352, průměrná délka záznamu je 21 B, průměrný počet záznamů ve stránce je přibližně 230. Z těchto hodnot jsou pro nás zajímavé nejvíce počty bloků tabulek, jelikož IO cenu plánu budeme vztahovat k tomuto počtu.

[3]Počet bloků tabulky `Producer` v například v DBS Oracle zjistíme příkazem `SELECT blocks FROM user_segments WHERE segment_name = 'PRODUCER';`

Kapitola 9

Plán vyhodnocení dotazu

Obsah

Cíl kapitoly:
V této kapitole jsou popsány obecné principy související s plánem vyhodnocení dotazu, především operace plánu a základní fyzická implementace DBS. Příklady pak demonstrují možnosti ladění fyzického návrhu databáze.

9.1 Úvod

Jakmile optimalizátor vybere nejlevnější (nejrychlejší) **plán vyhodnocení dotazu** (pozorný čtenář ví, že se jedná o plán blízký nejlevnějšímu), dotaz je pro-

veden a uživateli je vrácen výsledek. V DBS máme často možnost zobrazit tento plán. V plánu jsou především uvedeny jednotlivé fyzické operace implementující operátory relační algebry. Například selekce může být provedena porovnáním hodnot všech záznamů tabulky nebo bodovým dotazem v indexu (nejčastěji v B-stromu). Zobrazení plánu není jen jakási zajímavost, kterou nás může DBS ohromit, ale jedná se o mocný nástroj, který může sloužit k odladění (tedy zrychlení) provádění dotazu. Proč? V dnešních databázových systémech můžeme vybrat z celé řady datových struktur pro uložení dat (viz následující kapitoly), které poskytují různé výhody v závislosti na datech a dotazech. Vhodnou volbou uložení dat (tzv. **fyzický návrh databáze**) ovlivníme výběr operací plánu a tedy i rychlost provedení dotazu. Zobrazení plánu nám pak slouží k identifikaci neefektivních operací, tak abychom mohli zvolit jiný fyzický návrh. V plánu můžeme například vidět, že pro dotaz obsahující selekci došlo k sekvenčnímu průchodu tabulkou a je nutné vytvořit index na tento atribut atd. Přestože jsou základní koncepty související s plány dotazů společné pro všechny databázové systémy, zobrazení a obsah plánu je závislý na konkrétním systému, v této kapitole se podíváme na zobrazení a čtení plánu pro některé databázové systémy. Pro základní přiblížení problematiky plánu popíšeme jednoduchou fyzickou implementaci databáze, tak jak je dostupná v drtivé většině DBS.

9.2 Operace plánu vyhodnocení dotazu

Plán vyhodnocení dotazu může obsahovat celou řádu operací, nyní se změříme na ty nejčastější, které pracují se základní fyzickou implementací popsanou v předchozí kapitole. Operace obecně popíšeme a uvedeme jejich názvy v DBS Oracle a SQL Server, v ostatních systémech jsou názvy těchto operací variací na uvedené názvy. Nejčastější operace plánu vyhodnocení dotazu jsou:

- **Sekvenční průchod tabulkou typu hlada** – operace postupně přistupuje ke všem blokům tabulky a ukládá záznamy do výsledku. Pokud je součástí této operace selekce resp. selekce, nad záznamem se před zařazením do výsledku provede filtrace.

  Označení v Oracle: `TABLE ACCESS (FULL)`
  Označení v SQL Server (dále jen SQLS): `TABLE SCAN`[1]

- **Přístup k záznamu tabulky** typu halda pomocí ROWID získané z indexu.

  Oracle: `TABLE ACCESS (BY INDEX ROW ID)`
  SQLS: `RID Lookup (Heap)`[2]

- **Bodový a rozsahový dotaz v indexu.** Pokud uvažujeme B-strom, pak je bodový dotaz proveden průchodem od kořene k listu kde se nachází nebo nenachází hledaný klíč. Rozsahový dotaz nejprve provede stejný sestup pro hodnotu $>=$ dolní mez rozsahu a poté pokračuje průchodem listovými uzly dokud nenarazí na klíč $>$ horní hodnota rozsahu,pak prohledání ukončí.

  Oracle: `INDEX (UNIQUE SCAN), INDEX (RANGE SCAN)`
  SQLS: `INDEX SEEK`

  Přestože je počet logických přístupů rozsahového dotazu nejčastěji nižší než počet logických přístupů při sekvenčním průchodu tabulkou, při vyhodnocení dotazu optimalizátor spíše zvolí sekvenční průchod tabulkou, jelikož přístupy rozsahového dotazu v indexu jsou náhodné.

- **Selekce**, označení σ
- **Projekce**, označení Π
- **Spojení** (angl. **join**), označení $\bowtie$ – dle situace používá optimalizátor více algoritmů spojení, v kapitole 12.2.1 jsou tyto algoritmy popsány.
- **Třídění** (angl. **sort**)

Technologické okénko:
Zobrazení plánu vyhodnocení dotazu v Oracle

V Oracle je zobrazení plánu součástí každého klientského prostředí (např. v SQL Developer[3] se plán zobrazí graficky jak později uvidíme), pokud nechceme využívat tyto klientské aplikace, máme k dispozici příkaz `explain plan`, který plán dotazu uloží do tabulky `PLAN_TABLE`, např.

```
explain plan for select * from student where prijmeni='Poe';
```

Pokud chceme plán uložit do jiné tabulky použijeme příkaz `explain plan into <table_name> for ...`. Záznamy v tabulce `PLAN_TABLE` nejsou automaticky mazány, musíme je tedy mazat ručně, např. `DELETE PLAN_TABLE`.

Pro zobrazení plánu použijeme dotaz:

```
SELECT substr (lpad('␣', level-1) ||
       operation || '␣(' || options ||
       ')',1,30 ) "Operation",
      object_name "Object"
```

[3] http://www.oracle.com/technetwork/developer-tools/sql-developer/downloads/index.html

```
FROM PLAN_TABLE
START with id = 0
CONNECT by prior id=parent_id;
```

Příklad 9.1. (Zobrazení plánu vykonávání dotazu v Oracle).

Mějme tabulku `Student`:

```
CREATE TABLE Student (
  login CHAR(6) PRIMARY KEY,
  jmeno VARCHAR2(30) NOT NULL,
  prijmeni VARCHAR2(50) NOT NULL,
  email VARCHAR2(40),
  konto NUMBER);
```

Zobrazení plánu se provede v těchto krocích:

1. Smažeme záznamy z tabulky `PLAN_TABLE`: `delete PLAN_TABLE;`
2. Uložíme plán dotazu do tabulky `PLAN_TABLE`: `explain plan for select * from student where prijmeni='Poe';`
3. Vypíšeme plán dotazu: `SELECT ... FROM PLAN_TABLE;`

Výsledkem je výpis, který ukazuje, že operace SELECT byla provedena sekvenčním průchodem tabulkou typu halda **(m: IO Cost?)**.

```
Operation                      Object
------------------------------ ------------------------------
SELECT STATEMENT ()
 TABLE ACCESS (FULL)           STUDENT
```

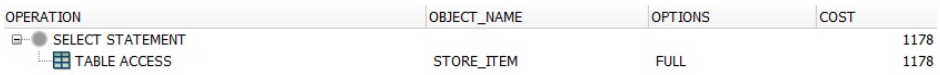
Technologické okénko:
Zobrazení plánu vyhodnocení dotazu v SQL Server

9.3 Příklady plánu vyhodnocení dotazu

Sekvenční průchod tabulkou

Na obrázku 9.1 vidíme plán sekvenčního průchodu tabulkou pro dotaz `SELECT * FROM Store_item` pro Oracle (operace `TABLE ACCESS (FULL)`). Při tomto

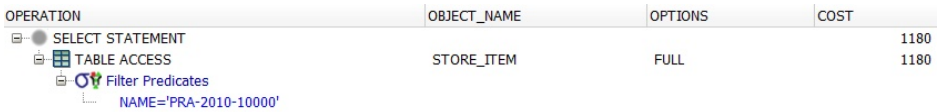
sekvenčním průchodu jsou postupně procházeny všechny bloky tabulky typu halda. Bloky jsou umístěny do cache v hlavní paměti, pokud při požadavku na stránku není stránka umístěna v paměti, je načtena z disku do cache. Sloupec COST obsahuje cenu plánu, která je pouze orientační[4] a je v tomto případě řádově stejná jako počet stránek tabulky (4 352).



Obrázek 9.1: Sekvenční průchod tabulkou v Oracle.

Sekvenční průchod tabulkou se selekcí

Na obrázku 9.2 vidíme plán sekvenčního průchodu tabulkou se selekcí pro dotaz `SELECT * FROM Store_item WHERE name='PRA-2010-100000'` pro Oracle v případě, že pro atribut `name` není vytvořen index (operace `TABLE ACCESS (FULL)` a selekce). Při tomto sekvenčním průchodu jsou postupně procházeny všechny stránky tabulky typu halda a je hledán záznam vyhovující podmínce. Cena plánu je opět řádově stejná jako počet bloků tabulky. V tomto případě je ale velikost výsledku 1, přesto muselo být prohledáno 4 352 stránek. Tento rozpor nám většinou signalizuje nevhodný fyzický návrh, který by měl být přepracován. I při tak relativně velkém objemu zpracovávaných dat je dotaz proveden řádově v desítkách ms pokud jsou stránky umístěny v hlavní paměti. Kouzlo tkví především v sekvenčním průchodu stránkami, při kterém dosahují mikroprocesory, paměti a disky maximálních propustností.

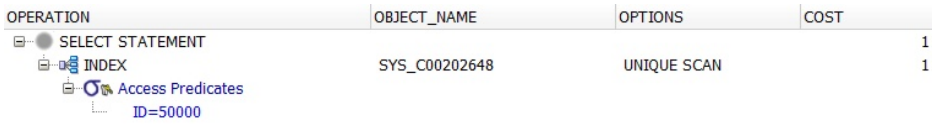


Obrázek 9.2: Sekvenční průchod tabulkou se selekcí v Oracle.

[4] Oracle zobrazuje v plánu dotazu pouze orientační cenu, přesné údaje získáme příkazem `SET AUTOTRACE ON`.

Bodový dotaz v B-stromu

Na obrázku 9.3 vidíme plán bodového dotaz v indexu pro dotaz `SELECT id FROM Store_item WHERE id=50000` pro Oracle (operace `INDEX (UNIQUE SCAN)`). Tato operace je vykonána pokud dotaz obsahuje selekci na atribut pro který je vytvořen index. Jelikož chceme aby se v plánu objevila pouze operace bodového dotazu v indexu, je v dotazu projekce pouze na indexovaný atribut. Projekce na další atributy by měla za následek další operaci – přístup do tabulky typu halda pro získání hodnot těchto atributů.



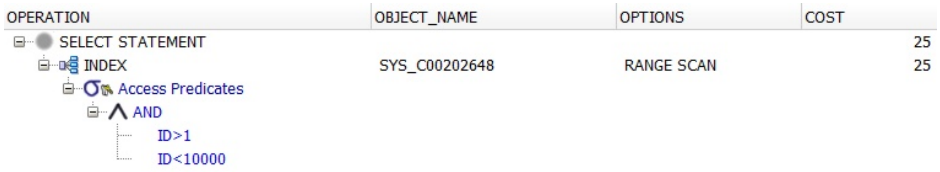
Obrázek 9.3: Bodový dotaz nad indexem v Oracle.

Velikost výsledku bodového dotazu je nejvýše 1; tato operace se tedy provádí pouze v případě dotazů na hodnotu jedinečného atributu (tedy atributu který je v `CREATE TABLE` označen jako `DISTINCT/UNIQUE`). Poznamenejme, že tuto operaci uvidíme v případě dotazů na primární klíč, jelikož ten je indexován vždy (téměř ve všech SŘBD) a je z definice jedinečný. Ačkoli počet stránek indexu je xyz, počet logických přístupů je 3 (výška stromu je 2, musíme tedy načíst 3 stránky, abychom se dostali od kořene k listu stromu, kde se nachází hledaná hodnota). Ke každému klíči stromu je přiřazeno ROWID, které může sloužit pro další operace prováděné během vykonávání dotazu.

Rozsahový dotaz v B-stromu

Na rozdíl od předchozí operace, v tomto případě hledáme více než jednu hodnotu v B-stromu. Na obrázku 9.4 vidíme plán rozsahového dotazu v indexu pro dotaz `SELECT id FROM Store_item WHERE id > 1 AND id < 10000` pro Oracle (operace `INDEX (RANGE SCAN)`). Tato operace je vykonána pokud dotaz obsahuje selekci na atribut pro který je vytvořen index. Jelikož chceme aby se v plánu objevila pouze operace bodového dotazu v indexu, je v dotazu projekce pouze na indexovaný atribut.

Rozdíl oproti bodovému dotazu je v tom, že předem neznáme maximální velikost výsledku (tedy počet klíčů které musí být v listech stromu porovnány). V tomto případě se tedy hledá první hodnota rozsahu, poté se prochází listové

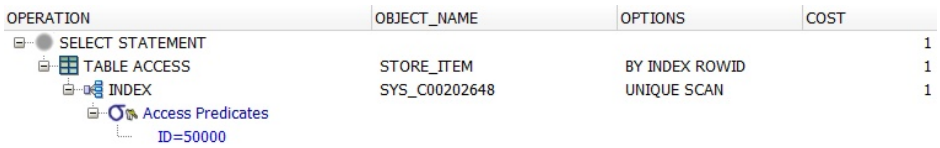


Obrázek 9.4: Rozsahový dotaz nad indexem v Oracle.

uzly dokud se nenalezne hodnota $>=$ vyšší hodnotě rozsahu. Operace je vykonána i v případě, kdy je v dotazu definována pouze jedna hodnota (např. `... WHERE id=50000`), ale indexovaný atribut není jedinečný, tzn. výsledek dotazu může být > 1. Jelikož logické a fyzické uspořádaní stránek může být u B-stromu odlišné, přístupy ke stránkám stromu jsou přístupy náhodné. Větší počet náhodně procházených stránek by znamenal radikální pokles času provedení dotazu, v takovém případě volí optimalizátor raději sekvenční průchod tabulkou než provedení rozsahového dotazu v indexu.

Přístup do tabulky typu halda pomocí ROWID

Na obrázku 9.5 vidíme plán dotazu `SELECT name FROM Store_item WHERE id=50000` pro Oracle. Při provádění dotazu se nejprve provede bodový dotaz v indexu (operace `INDEX (UNIQUE SCAN)`), poté se pomocí získaného ROWID čtou hodnoty ostatních atributů z tabulky (operace `TABLE ACCESS (BY INDEX ROWID)`). Jelikož součástí ROWID je číslo bloku, je operace přístupu do tabulky provedena na jeden logický přístupy.



Obrázek 9.5: Přístup do tabulky typu halda pomocí ROWID v Oracle.

Spojení

Databázové systémy disponují celou řadou algoritmů implementující spojení tabulek. Obecně se jedná o operaci, která prochází dvě tabulky (s počtem prvků

m a n) a spojuje záznamy z obou tabulek dle hodnot vybraných atributů (které jsou umístěny v obou tabulkách). Typicky využíváme přirozeného spojení, tj. kontrolujeme rovnost hodnot atributů z obou tabulek. Nyní si některé algoritmy operace spojení ukážeme na konkrétních dotazech a jejich plánech. Mějme dotaz:

```
SELECT S.id, S.name, P.name
  FROM Store_item S, Producer P
  WHERE S.name='PRA-2010-10000' AND S.idProducer=P.id;
```

Na obrázku 9.5 je uveden plán využívající algoritmus s**pojení vnořenými cykly s indexem** (operace `NESTED LOOPS` a `INDEX (UNIQUE SCAN)`). Plán je vyhodnocen v těchto krocích:

1. Sekvenčním průchodem se selekcí se v tabulce `Store_item` hledají záznamy vyhovující podmínce `S.name='PRA-2010-10000'`.
2. Pro každý nalezený záznam se provede bodový dotaz do indexu `Producer(id)`. Výsledkem jsou tady záznamy vyhovující podmínce z tabulky `Store_item` spolu s ROWID ukazující na záznamy tabulky `Producer`.
3. Pomocí těchto ROWID získáme hodnotu atributu `name` přístupem ke stránkám tabulky `Producer`.

OPERATION	OBJECT_NAME	OPTIONS	COST
SELECT STATEMENT			1220
NESTED LOOPS			
NESTED LOOPS			1220
TABLE ACCESS	STORE_ITEM	FULL	1177
Filter Predicates			
STORE_ITEM.NAME='PRA-2010-10000'			
INDEX	SYS_C00203250	UNIQUE SCAN	0
Access Predicates			
STORE_ITEM.IDPRODUCER=PRODUCER.			
TABLE ACCESS	PRODUCER	BY INDEX ROWID	1

Obrázek 9.6: Operace spojení vnořenými cykly s přístupem do indexu v Oracle.

Algoritmus je výhodný pouze v případě menšího počtu bodových dotazů do indexu. V případě většího počtu selektovaných hodnot v 1. kroku algoritmu, dochází k velkému počtu bodových dotazů, tedy k velkému počtu náhodných přístupů do indexu. V takovém případě optimalizátor volí variantu s**pojení vnořenými cykly** (bez použití přístupu do indexu) nebo **spojení hašováním**. Tyto algoritmy se využívají i v případě, kdy tabulka procházená ve vnitřním

cyklu spojení obsahuje malý počet záznamů a sekvenční průchod tabulkou je tak relativně levná operace. Tento případ si ukážeme v následujícím příkladě.

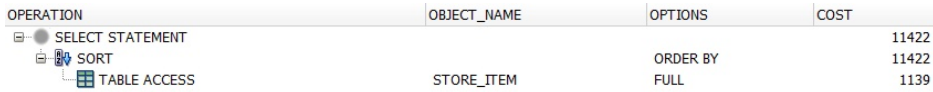
Mějme stejný dotaz s tím, že počet záznamů v tabulce `Producer` byl snížen ze 100 000 na 10 000. Na obrázku 9.7 vidíme plán s operací spojení hašováním. Systém prochází sekvenčně obě tabulky, u tabulky `Store_item` provádí selekci `S.name='PRA-2010-10000'`. Záznamy jsou poté spojeny pomocí hašování.

Obrázek 9.7: Operace spojení hašováním v Oracle.

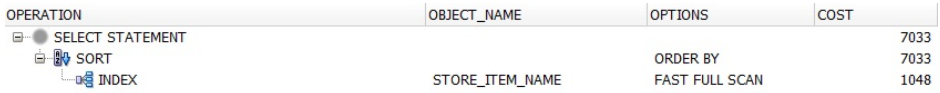
Cílem této kapitoly není podrobně popsat výše zmíněné algoritmy, zveme zvídavého čtenáře k přečtení kapitoly xyz, kde se dozví detaily algoritmů operací plánů. V této chvíli můžeme ovšem, i bez detailní znalosti algoritmů, identifikovat jejich obecné vlastnosti: některé algoritmy využívají index (spojení vnitřními cykly s indexem), některé algoritmy potřebující záznamy setřízené (**spojování slučováním**, angl. **Merge Join**), jiné mohou pracovat i bez nesetřízení záznamů (spojení hašováním, spojení vnitřními cykly) atd. Z počtu logických čtení je patrné, že spojení je nákladná operace se složitostí v nejhorším případě $O(n \times m)$ (v případě vnořených cyklů), doba provádění dotazu tedy roste poměrně rychle s velikostí spojovaných množin. Při fyzickém návrhu tedy často dbáme na zrychlení či eliminaci této operace, zejména u častých operací.

Třídění

Dotaz: `SELECT * from Store_item ORDER BY name;`



Poznámka: Index (obsahuje setříděné klíče) by částečně zamezil třídění pouze v případě dotazu `SELECT name from Store_item ORDER BY name;`



OPERATION	OBJECT_NAME	OPTIONS	COST
SELECT STATEMENT			7033
SORT		ORDER BY	7033
INDEX	STORE_ITEM_NAME	FAST FULL SCAN	1048

9.3.1 Pravidlo pro vytvoření indexu

Pravidlo 1: Vytvoř index pro všechny atributy, které se v dotazech objevují za WHERE.

Poznámka: Později si řekneme, že to vždy není vhodné, máme zde totiž druhou stranu problému – narůstající velikost indexu a cenu aktualizace.

V tomto případě tedy použijeme:

```
CREATE INDEX Store_Item_name ON Store_item(name);
```

JOIN

```
SELECT S.id, S.name, P.name
  FROM Store_item S,Producer P
  WHERE S.name='PRA-2010-10000' and S.idProducer=P.id;
```

OPERATION	OBJECT_NAME	OPTIONS	COST
SELECT STATEMENT			5
NESTED LOOPS			
NESTED LOOPS			5
TABLE ACCESS	STORE_ITEM	BY INDEX ROWID	4
INDEX	STORE_ITEM_NAME	RANGE SCAN	3
Access Predicates			
STORE_ITEM.NAME='PRA-2010-100(			
INDEX	SYS_C00203257	UNIQUE SCAN	0
Access Predicates			
STORE_ITEM.IDPRODUCER=PRODUCER.			
TABLE ACCESS	PRODUCER	BY INDEX ROWID	1

Dotaz je proveden s IO cenou 5 (namísto 1 155), výsledkem je neměřitelný čas provedení dotazu (v předchozím případě to bylo 0,05s – znamená max 20 dotazů/s). V případě reálných dat s větší velikostí záznamu by se jednalo o nepoměrně vyšší hodnotu.

9.3.2 Statistiky databáze

Optimalizátor pro výpočet ceny dotazu využívá statistiky databáze uložené v katalogu.

- Pro tabulku ukládá:
  - Kardinalitu (mohutnost, tedy počet záznamů).
  - Počet diskových stránek tabulky.
- Pro sloupec ukládá:
  - Počet totožných hodnot ve sloupci, min a max hodnoty.
  - 10 nejfrekventovanějších hodnot a počet výskytů.
- Pro index ukládá:
  - Počet listových stránek indexu.
  - Výšku indexu (pokud se jedná o strom).

Poznámky

- Tyto údaje nejsou, kvůli režii systému, aktualizovány po každé aktualizaci záznamu.
- Nejčastěji jsou aktualizovány systémovou utilitou, kterou spouští administrátor databáze (u DB2 se jedná o utilitu RUNSTATS[5]).
- **Opět tedy vidíme, že maximální výkon SŘBD není nastaven automaticky!**

Poznámky

- Oracle od verze 10g tyto statistiky počítá v době nízké zátěže serveru, pokud si ovšem nejsme jisti hodnotami, provedeme ruční smazání cache a přepočítání statistik před novým provedením dotazu:

```
alter system flush shared pool;
alter system flush buffer cache;
execute dbms stats.gather table stats(ownname=> 'TEST',
  tabname=> 'OBJEDNAVKY',
  estimate_percent=> DBMS_STATS.AUTO_SAMPLE_SIZE);
```

[5] http://www.ibm.com/developerworks/data/library/techarticle/dm-0412pay/

- Proč SŘBD používají právě tyto hodnoty a navíc používají drobně odlišné hodnoty? Každý SŘBD používá odlišný algoritmus odhadu ceny operací plánu.

9.3.3 Oracle – AUTOTRACE

Statistiky provedení dotazu

- Příkaz `EXPLAIN PLAN FOR` uvádí odhad ceny vykonání dotazu, která je ovlivněna čtením z cache buffer a dalšími optimalizacemi aplikovanými během provedení dotazu.
- Pokud v nějaké konkrétní situaci získává SŘBD data z cache namísto disku, nemáme jistotu, že tomu tak bude za každé situace.
- Kromě plánu vykonání dotazu a odhadu ceny potřebujeme mít k dispozici přesné statistiky provedení dotazu.
- V Oracle tuto podrobnou informaci získáme pomoci příkazu `SET AUTOTRACE ON` (viz obrázek 9.8).

Z hodnot které poskytuje příkaz AUTOTRACE nás pak zajímají především tyto hodnoty:

- **consistent gets**

  Počet přístupů k blokům (tabulky nebo indexu). Mluvíme o tzv. **logických přístupech** (nebo logickém čtení).

- **physical reads**

  Celkový počet bloků načtených z disku. Mluvíme o tzv. fyzických přístupech (nebo **fyzickém čtení**).

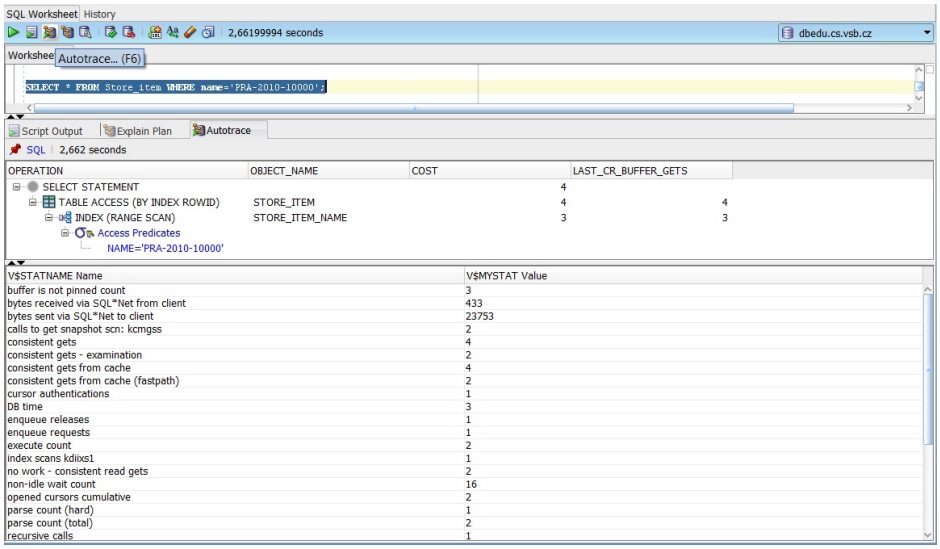
- Pokud pro logický přístup neexistuje fyzické čtení, mluvíme o **cache hit** (úspěšném čtení z cache).

  Míra úspěšnosti použití cache buffer:
  cache hit rate = (cache hits / počet logický čtení) * 100 [%].

- V opačném případě mluvíme o **cache miss** (neúspěšném čtení z cache).

- **sorts (memory)**

  Počet operací třídění, které byly vykonány v paměti bez zápisu na disk.



Obrázek 9.8: AUTOTRACE v SQL Developer

- **sorts (disk)**

  Počet operací třídění, které požadovaly nejméně jeden zápis na disk.

- **rows processed**

  Počet záznamů zpracovaných během operace.

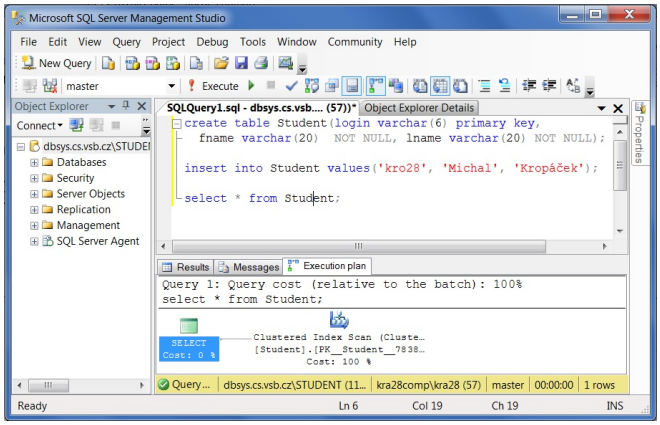
- **redo size**

  Celkový objem redo záznamů v bytech.

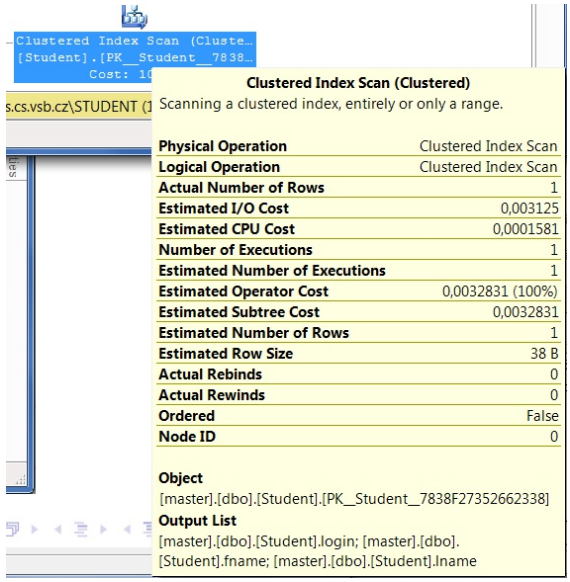
9.3.4 SQL Server

SQL Server - zobrazení plánu

Management Studio
menu: Query/Include Actual Execution Plan



SQL Server - detail operace



SQL Server - SET STATISTICS IO ON

```
SET STATISTICS IO ON;
select * from Student;
```

Výstup:

```
Table 'Student'. Scan count 1, logical reads 2, physical
reads 0, read-ahead reads 0, lob logical reads 0, lob physical
reads 0, lob read-ahead reads 0.
```

- logical reads – počet logických přístupů (consistent gets v Oracle)
- physical reads – počet fyzických přístupů (physical reads v Oracle)

9.3.5 Vyhodnocení logických a fyzických přístupů

Vyhodnocení logických a fyzických přístupů

- **Pravidlo 1:** Pokud počet záznamů výsledku je mnohem nižší než počet logických přístupů, musíme uvažovat o optimalizaci dotazu (např. vytvoření indexu).
- **Pravidlo 2:** Pokud se počet fyzických čtení blíží počtu logických přístupů, můžeme efektivitu dotazů zlepšit zvětšením **cache** (data cache v SQLS, cache buffer v Oracle):
  - Velikost cache je omezena velikostí hlavní paměti.
  - Cache obsahuje často používaná data, není nutné aby cache obsahovala všechna data (velikost cache tedy nemusí být stejná jako velikost dat).
  - Pravidlo 90:10 – často používaná data byl měla být z 90% umístěna v hlavní paměti.

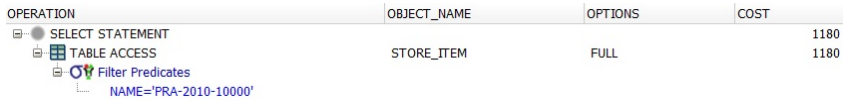
9.3.6 Příklad

TABLE ACCESS (FULL) 1/2

- Velikost tabulky `Store_item` je 4 352 bloků[6] (4 352 × 8 kB = 34 MB).

[6] `SELECT blocks FROM user_segments WHERE segment_name = 'STORE_ITEM';`

- **Dotaz:** `SELECT * FROM Store_item WHERE name='PRA-2010-10000';`



- Výsledek: 1 záznam.

TABLE ACCESS (FULL) 2/2

- **Výsledek AUTOTRACE:**
  - consistent gets: 4 283 (logické přístupy)
  - physical read: 0 (fyzické čtení)

- **Komentář:**
  - Logické přístupy přibližně odpovídají počtu bloků tabulky, všechny bloky jsou v cache, proto je čas provedení dotazu relativně nízký (0,05 s).
  - 0.05 s? ⇒ Podobných dotazů SŘBD provede nejvýše 20/s. V reálných prostředích často požadujeme propustnost stovky – tisíce transakcí za s.
  - Výsledek 1 záznam vs 4 352 bloků tabulky ⇒ je nutné provedení dotazu optimalizovat.

INDEX (UNIQUE SCAN) 1/2

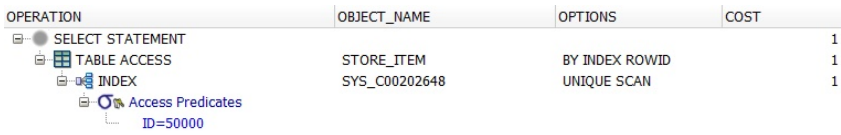
- Vytvoříme index na atribut `name`:

```
CREATE INDEX Store_Item_name ON Store_item(name);
```

- Velikost indexu je 3 840 bloků[7] (velikost tabulky je 4 352 bloků).

[7] `SELECT blocks FROM user_segments WHERE segment_name = 'STORE_ITEM_NAME';`

- **Dotaz:** `SELECT * FROM Store_item WHERE name='PRA-2010-10000';`



INDEX (UNIQUE SCAN) 2/2

- **Výsledek AUTOTRACE:**
  - consistent gets: 4 (logické přístupy)
  - consistent gets - examination: 2
  - consistent gets from cache: 4
  - physical reads: 2 (fyzické čtení)
  - physical read total bytes: 16 384

- **Komentář:**
  - Během dotazu byly požadovány 4 bloky, z toho 2 byly načteny z disku (2 byly v cache bufferu).
  - Stránky mají 8 kB, celkem bylo načteno 16 384 B.
  - V předchozím případě (bez indexu) bylo provedeno 4 283 logických přístupů, v tomto případě 4, což odpovídá velikosti výsledku 1.

9.3.7 Jednoduchý/složený index

Jednoduchý index

- Vytvoříme index na atribut `name` tabulky `Producer` (obsahuje 100 000 záznamů):

  ```
  CREATE INDEX Producer_name ON Producer(name);
  ```

- Velikost indexu je 384 bloků[8] (velikost tabulky je 640 bloků).

[8] `SELECT blocks FROM user_segments WHERE segment_name = 'PRODUCER_NAME';`

- **Dotaz:** `SELECT * FROM Producer WHERE address='prod7452';`

  Velikost výsledku: 1

- consistent gets: 3 (logické přístupy)

Složený index

- Pokud se chceme dotazovat na hodnoty dvou a více atributů v jednom dotazu, můžeme vytvořit tzv. **složený index**.
- Např.: `CREATE INDEX Producer_name_addr ON Producer(name,address);`
- Velikost indexu je 512 bloků[9] (velikost indexu Producer(name) je 384 bloků, velikost tabulky je 640 bloků).

- **Dotaz:** `SELECT * FROM Producer WHERE name='prod7452' AND address='address560`
- consistent gets: 2 (logické přístupy)

Dotazy na jednotlivé atributy 1/3

- **Dotaz:** `SELECT * FROM Producer WHERE name='prod7452';`

OPERATION	OBJECT_NAME	COST
SELECT STATEMENT		3
TABLE ACCESS (BY INDEX ROWID)	PRODUCER	3
INDEX (RANGE SCAN)	PRODUCER_NAME_ADDR	2
Access Predicates		
NAME='prod7452'		

  consistent gets: 3 (logické přístupy)

- **Dotaz:** `SELECT * FROM Producer WHERE address='address56000';`

OPERATION	OBJECT_NAME	COST	LAST_CR_BUFFER_GETS
SELECT STATEMENT		172	
TABLE ACCESS (FULL)	PRODUCER	172	552
Filter Predicates			
ADDRESS='address56000'			

  consistent gets: 574 (logické přístupy) !!!

[9] `SELECT blocks FROM user_segments WHERE segment_name = 'PRODUCER_NAME_ADDR';`

Dotazy na jednotlivé atributy 2/3

- Ačkoli byl atribut `address` součástí složeného klíče `(name,address)`, byl dotaz na něj proveden sekvenčním průchodem tabulkou ⇒ 574 namísto 3 logických přístupů k blokům.
- **Důvod?** Složený index je implementován B-stromem se složeným klíčem `(name,address)` v tomto pořadí. S logaritmickou složitostí jsou tedy vykonány dotazy na atribut `name` nebo atributy `(name,address)`.

Dotazy na jednotlivé atributy 3/3

Řešením je (pokud se dotazujeme atributů `name`, `(name,address)` a `address`) vytvořit složený index `(name,address)` a `(address)`, **vlastnosti**:

- Celková velikost obou indexů je 896 (velikost tabulky je 640 bloků).
- Aktualizace atributu `name` (operace INSERT, UPDATE, DELETE) bude znamenat aktualizaci tabulky a jednoho indexu.
- Aktualizace atributu `address` bude znamenat aktualizaci tabulky a dvou indexů.

9.4 Reference

Reference

- Thomas Kyte. Expert Oracle Database Architecture: 9i and 10g Programming Techniques and Solutions.
- Oracle. CREATE TABLE manual. `http://download.oracle.com/docs/cd/B19306_0 server.102/b14200/statements_7002.htm`
- Oracle. CREATE CLUSTER manual. `http://download.oracle.com/docs/cd/B1930 server.102/b14200/statements_5001.htm`
- Microsoft. Clustered Index Design Guidelines. `http://msdn.microsoft.com/en-us/library/ms190639.aspx`

Kapitola 10

Fyzická implementace databázových systémů

Fyzická implementace

- Fyzická implementace definuje datové struktury a způsob uložení pro základní logické objekty:
  - Uložení tabulek.
  - Typy indexů.
  - Materializované pohledy.
- Fyzická implementace tedy řeší uložení dat na nejnižší úrovni databáze.
- Dostupných datových struktur je celá řada, implicitně nabízí SŘBD administrátorovi nějakou volbu.
- Např. CREATE TABLE značí vytvoření tabulky typu halda, CREATE INDEX značí vytvoření B^+-stromu.

Proč řešit fyzickou implementaci?

- Je lehké navrhnout systém, který sice funguje, ale není příliš efektivní.

- Standardní řešením je prohlášení databáze za pomalou a nákup nového hardware.

- Dnešní databázové systémy ovšem nabízí řadu možností jak ušít fyzickou implementaci naši databázi na míru.
- Podmínkou pro odladění databáze je ovšem potřeba mít poměrně hluboké znalosti datových struktur a algoritmů vykonávání dotazů.

10.0.1 Tabulka

Typy tabulek

- **Tabulka typu halda** (**Heap table**) – záznamy v tabulce nejsou uspořádány. Je to implicitní volba CREATE TABLE.
- **Shlukování záznamů** (**Data clustering**) – záznamy jsou v datovém souboru seřazeny podle zvoleného klíče.
- **Hašovaná tabulka** (**Hash table**) - záznamy se stejnou hašovanou hodnotou jsou uloženy ve stejném bloku, nebo ve velmi blízkém bloku.
- **Zhmotněné pohledy** (**Materialized views**) – uložené výsledky dotazů, které bývají často v databázi vyhodnocovány. V tomto případě se nejedná přímo o tabulku, ale spíše o fragment z tabulky, či několika tabulek.

Tabulka typu heap (Heap table)

- Jde o základní typ tabulky, jenž se vytvoří po zadání příkazu CREATE TABLE.
- Jedná se o stránkované perzistentní pole s velikostí bloku nejčastěji 8kB.
- Záznamy v tabulce nejsou nijak uspořádány:
  - Záznamy nejsou fyzicky mazány, jsou pouze označeny jako smazané.
  - Při vkládání je záznam vždy umístěn na první nalezenou volnou pozici v tabulce nebo na konec pole.
- $\Rightarrow$ Není možné se spoléhat na uspořádání záznamů v tabulce ($\Rightarrow$ neefektivní vyhledávání).

- ⇒ Tento typ tabulky je velmi efektivní z pohledu operace INSERT a využití místa.

Shlukování záznamů (Data clustering) 1/2

- Záznamy jsou v datovém souboru seřazeny podle nějakého klíče.
- Pro implementaci je nejčastěji využita nějaká varianta B-stromu.
- Listové uzly stromu (bloky) obsahují kromě klíče i další záznamy tabulky (nemusí to být nutně všechny atributy).
- **Shlukované záznamy vs index implementovaný B-stromem**: index obsahuje v listech pouze klíč a ROWID, ostatní atributy musí DBMS přečíst z tabulky.
- Rozlišujeme shlukování pro jednu a více tabulek.

Shlukování záznamů (Data clustering) 2/2

- Eliminace náhodných přístupů do tabulky (v Oracle se jedná o operaci TABLE ACCESS (BY INDEX ROWID)) pro získání hodnot dalších atributů (kromě klíče).
- ⇒ **Výhodné** všude kde potřebujeme hodnoty dalších atributů:
  - operace SELECT s projekcí neklíčových atributů,
  - spojování tabulek,
  - třídění podle klíče.
- **Nevýhoda:** Zhoršený výkon zejména u operace insert (data se musí 'zatřizovat').
- ⇒ Výhodné zejména v případě převahy čtení proti aktualizaci.

Shlukování záznamů - Příklad 1/2

- Uveďme příklad shlukování pro dvě tabulky.
- Mějme dvě tabulky:
  - `Zaměstnanec (id, jméno, pozice, telefon)`

  - `Objednávka(idZaměstnance, idZbozi, idFirmy, datum)`

- Tabulka `Zaměstnanec` bude typu heap.
- Tabulka `Objednávka` se bude plnit záznamy tak, jak budou firmy objednávat zboží.
- Každá objednávka je automaticky přiřazena zaměstnanci.

Shlukování záznamů - Příklad 2/2

```
  Zaměstnanec (id, jméno, pozice, telefon)
Objednávka(idZaměstnance, idZbozi, idFirmy, datum)
```

- **Problém:** Pokud budeme chtít zjistit všechny objednávky zaměstnance, pak v případě tabulky typu halda mohou být objednávky rozptýleny po celé tabulce.
- $\Rightarrow$ Dotaz bude realizován často nákladnou operací spojení.
- V případě shlukování budou záznamy objednávek zaměstnance uloženy ve stejném bloku jako záznam zaměstnance. $\Rightarrow$ Minimalizujeme počet logických přístupů při provedení dotazu.

Hašované tabulky

- Záznamy se stejnou hašovací hodnotou jsou uloženy ve stejném bloku nebo ve velmi blízkém bloku.
- Výhoda hašovaných tabulek je podobná jako tabulek shlukujících záznamy. Máme ve stejných blocích uloženy data k nimž je často přistupováno společně.
- Hašování poskytuje v některých případech lepší složitost než B-strom (složitost $O(k)$ vs $O(\log n)$), může ovšem dojít k plýtvání místem.

10.0.2 Index

Typy indexů 1/2

- **B$^+$-strom** – nerozšířenější typ indexu.

- **Složený index** (**composite index**) – index, kde je klíčem více než jeden atribut.
  Problém: Při indexu `login,jmeno` a dotazu na jméno, bude docházet k sekvenčnímu průchodu!
- **Pokrývající index** (**covering index**) – index, který obsahuje dostatek informací, aby pokryl dotaz bez nutnosti přistupovat do datového souboru relace, které se index týká.

  V případě eliminace operace TABLE ACCESS (BY INDEX ROWID)) v Oracle se jedná o řešení ekvivalentní se shlukování záznamů.

Typy indexů 2/2

- **Hustý a řídký index** (**dense and sparse index**) – hustý index má ukazatel na každý řádek tabulky a řídký index obsahuje ukazatele pouze na bloky tabulky.
- **Bitmapový index** – index, který pro každou hodnotu h indexovaného atributu x a jeden záznam obsahuje jeden bit. Bit je nastaven na jedna pokud má záznam hodnotu h v atributu x, jinak je nula.

  Nevýhoda: Problematická aktualizace indexu, používá se v případě malého či žádného počtu aktualizací.

10.1 Fyzická implementace databáze v Oracle

10.1.1 Tabulka v Oracle

Správa prostoru segmentu

- Podrobný popis CREATE TABLE najdete v dokumentaci:
  `http://download.oracle.com/docs/cd/B19306_01/server.102/b14200/statements_7002.htm`
- V Oracle, každá tabulka je uložena ve vlastním datovém **segmentu**, index je uložen ve vlastním indexovém segmentu. Segment je pak uložen v tzv. **tablespace**.

Správa prostoru segmentu 2/2

- Oracle nabízí dva způsoby **správy prostoru segmentu** (**space segment management**):
  - **Manuální** (**manual space segment management**) – správa prostoru segmentu se provádí na základě několika parametrů jako je FREELIST, PCTUSED, PCTFREE, INITTRANS.
  - **Automatická** (**automatic space segment management**) – je součástí Oracle od verze 9i. Odpadá nutnost konfigurovat některé parametry.

Parametry tabulky

- **PCTFREE** - udává hodnotu (v procentech), kdy je blok považován za plný.
- **LOGGING / NOLOGGING** - nastavuje generování redo log záznamů pro data v daném segmentu.
- NOLOGGING může generování redo log záznamů potlačit, ale pouze pro některé operace.
- **INITTRANS** - každý blok obsahuje v hlavičce pole záznamů o tom, které záznamy v bloku jsou zamknuty.
- Nastavení INITTRANS udává jaká je počáteční velikost tohoto pole. Implicitně je hodnota nastavena na 2.

10.1.2 IOT

Indexově organizovaná tabulka 1/3

- **Indexově organizovaná tabulka** (**index organized table**, IOT) je variantou shlukování záznamů.
- IOT ukládá záznamy v blocích uspořádaně dle primárního klíče.
- Oracle umožňuje nastavit dva parametry specifické pro IOT: OVERFLOW a INCLUDING.

Indexově organizovaná tabulka 2/3

- **OVERFLOW** stanovuje maximální velikost záznamu tabulky uloženého v listovém uzlu. Maximální velikost je stanovena parametrem PCTTRESHOLD x, kde hodnota x udává maximální velikost záznamu v procentech celkové velikosti bloku.

- Část záznamu, která se do listového bloku nevešla, je uložena v odděleném segmentu a data v listu se na ně pouze odkazují.

Indexově organizovaná tabulka 3/3

- Parametr INCLUDING $attr_1, attr_2, ...$ udává atributy, jenž mají být uloženy v listovém uzlu tabulky.

- Zbývající atributy jsou opět uloženy v odděleném segmentu.

- Parametr PCTFREE je u IOT využit jen při vytváření tabulky. Udává místo v blocích, které je ponecháno volné pro další vkládání.

Indexově organizovaná tabulka, Příklad 1/2

```
create table heap_addresses (
  empId references emp(empId) on delete cascade,
  addrType varchar2(10),
  street varchar2(20),
  city varchar2(20),
  state varchar2(2),
  zip number,
  primary key(empId, addrType));
```

Indexově organizovaná tabulka, Příklad 2/2

```
create table iot_addresses (
  empId references emp(empId) on delete cascade,
  addrType varchar2(10),
  street varchar2(20),
  city varchar2(20),
  state varchar2(2),
  zip number,
  primary key(empId, addrType))
  ORGANIZATION INDEX;
```

10.1.3 Heap Table vs IOT

Heap Table vs IOT, Zadání[1]

- Mějme evidenci zaměstnanců, u každého zaměstnance budeme chtít evidovat několik adres (práce, domov atd.)
- Typický dotaz nad oběma tabulkami bude vracet hodnoty atributů zaměstnance a všechny jeho adresy.
- Ukážeme si řešení pomocí tabulky typu halda (**Heap Table**) a indexově organizované tabulky (**IOT**).

Vytvoření tabulky zaměstnanců

```
CREATE TABLE Employee AS
  SELECT
    object_id empid,
    object_name ename,
    created hiredate,
    owner job
  FROM all_objects;

ALTER TABLE Employee
  ADD CONSTRAINT employee_pk PRIMARY KEY (empid);
```

Vytvoříme tabulku zaměstnanců vložením záznamů z tabulky `ALL_OBJECTS` systémového katalogu (tabulka obsahuje všechny objekty databáze).

Tabulka `Employee`

```
EMPID  ENAME                    HIREDATE     JOB
-----  ------------------------ ------------ -------
100    ORA$BASE                 30.03.10     SYS
116    DUAL                     30.03.10     SYS
117    DUAL                     30.03.10     PUBLIC
278    MAP_OBJECT               30.03.10     SYS
279    MAP_OBJECT               30.03.10     PUBLIC
364    SYSTEM_PRIVILEGE_MAP     30.03.10     SYS
366    SYSTEM_PRIVILEGE_MAP     30.03.10     PUBLIC
...
```

[1] Thomas Kyte. Expert Oracle Database Architecture: 9i and 10g Programming Techniques and Solutions.

```
SELECT count(*) FROM Employee;
```

```
COUNT(*)
----------------------
57920
```

Vytvoření tabulky Address_heap

Vytvoříme tabulku adres typu halda.

```
CREATE TABLE Address_heap (
  empid references Employee(empid) on delete cascade,
  addr_type varchar2(10),
  street varchar2(20),
  city varchar2(20),
  state varchar2(2),
  zip number,
  PRIMARY KEY (empid,addr_type));
```

Vytvoření tabulky Address_iot

Vytvoříme tabulku adres typu indexově organizovaná tabulka.

```
CREATE TABLE Address_iot (
  empid references Employee(empid) on delete cascade,
  addr_type varchar2(10),
  street varchar2(20),
  city varchar2(20),
  state varchar2(2),
  zip number,
  PRIMARY KEY (empid,addr_type))
ORGANIZATION INDEX;
```

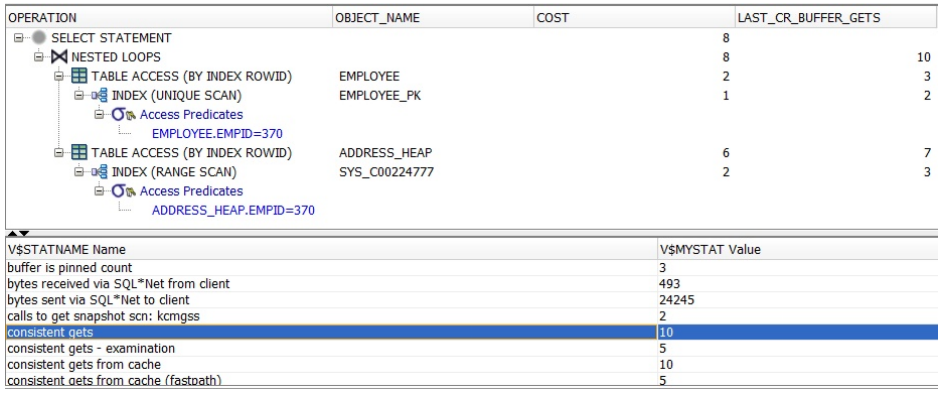
Vložení adres

```
INSERT INTO Address_heap
  SELECT empid, 'WORK', '123 main street',
       'Washington', 'DC', 20123
    FROM Employee;

INSERT INTO Address_iot
  SELECT empid, 'WORK', '123 main street',
       'Washington', 'DC', 20123
    FROM Employee;
```

Stejné záznamy byly vloženy pro typ adres HOME, PREV a SCHOOL. ⇒ Do každé tabulky bylo vloženo 231 680 záznamů.

Dotaz, Heap table

```
SELECT * FROM Employee, Address_heap
  WHERE Employee.empid = Address_heap.empid
        AND Employee.empid = 370;
```

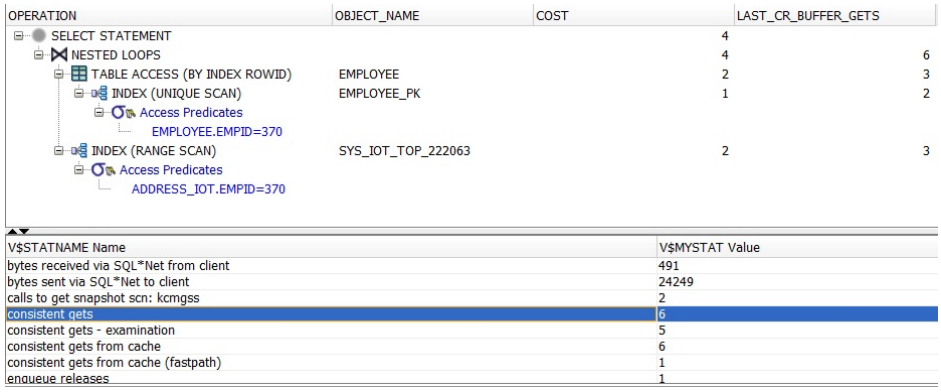


OPERATION	OBJECT_NAME	COST	LAST_CR_BUFFER_GETS
SELECT STATEMENT		8	
NESTED LOOPS		8	10
TABLE ACCESS (BY INDEX ROWID)	EMPLOYEE	2	3
INDEX (UNIQUE SCAN)	EMPLOYEE_PK	1	2
Access Predicates			
EMPLOYEE.EMPID=370			
TABLE ACCESS (BY INDEX ROWID)	ADDRESS_HEAP	6	7
INDEX (RANGE SCAN)	SYS_C00224777	2	3
Access Predicates			
ADDRESS_HEAP.EMPID=370			

V$STATNAME Name	V$MYSTAT Value
buffer is pinned count	3
bytes received via SQL*Net from client	493
bytes sent via SQL*Net to client	24245
calls to get snapshot scn: kcmgss	2
consistent gets	10
consistent gets - examination	5
consistent gets from cache	10
consistent gets from cache (fastpath)	5

Dotaz, IOT

```
SELECT * FROM Employee, Address_iot
  WHERE Employee.empid = Address_iot.empid
        AND Employee.empid = 370;
```

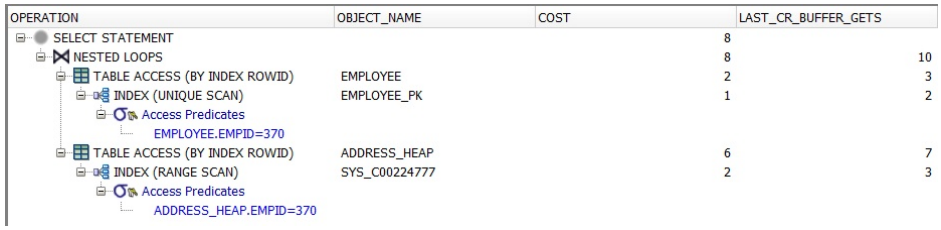


OPERATION	OBJECT_NAME	COST	LAST_CR_BUFFER_GETS
SELECT STATEMENT		4	
NESTED LOOPS		4	6
TABLE ACCESS (BY INDEX ROWID)	EMPLOYEE	2	3
INDEX (UNIQUE SCAN)	EMPLOYEE_PK	1	2
Access Predicates			
EMPLOYEE.EMPID=370			
INDEX (RANGE SCAN)	SYS_IOT_TOP_222063	2	3
Access Predicates			
ADDRESS_IOT.EMPID=370			

V$STATNAME Name	V$MYSTAT Value
bytes received via SQL*Net from client	491
bytes sent via SQL*Net to client	24249
calls to get snapshot scn: kcmgss	2
consistent gets	6
consistent gets - examination	5
consistent gets from cache	6
consistent gets from cache (fastpath)	1
enqueue releases	1

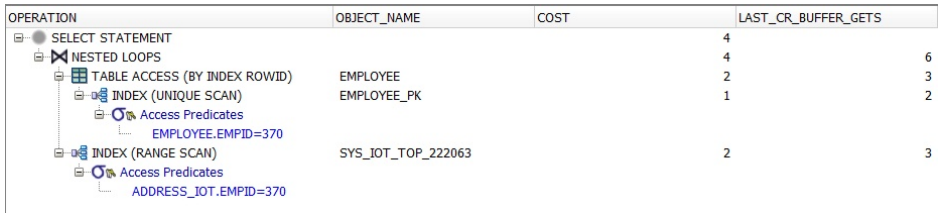
Porovnání

Heap Table



OPERATION	OBJECT_NAME	COST	LAST_CR_BUFFER_GETS
SELECT STATEMENT		8	
NESTED LOOPS		8	10
TABLE ACCESS (BY INDEX ROWID)	EMPLOYEE	2	3
INDEX (UNIQUE SCAN)	EMPLOYEE_PK	1	2
Access Predicates			
EMPLOYEE.EMPID=370			
TABLE ACCESS (BY INDEX ROWID)	ADDRESS_HEAP	6	7
INDEX (RANGE SCAN)	SYS_C00224777	2	3
Access Predicates			
ADDRESS_HEAP.EMPID=370			

Index Organized Table



OPERATION	OBJECT_NAME	COST	LAST_CR_BUFFER_GETS
SELECT STATEMENT		4	
NESTED LOOPS		4	6
TABLE ACCESS (BY INDEX ROWID)	EMPLOYEE	2	3
INDEX (UNIQUE SCAN)	EMPLOYEE_PK	1	2
Access Predicates			
EMPLOYEE.EMPID=370			
INDEX (RANGE SCAN)	SYS_IOT_TOP_222063	2	3
Access Predicates			
ADDRESS_IOT.EMPID=370			

Zhodnocení

- Použitím indexově organizované tabulky jsme snížili počet logických přístupů na 60%.
- Při větším počtu dotazů, pokud by dotazy nad tabulkou typu halda trvaly 1,7s, pro IOT by byly provedeny za 1,02s.

10.1.4 Shlukované tabulky

Shlukované tabulky

- V Oracle můžeme nastavit shlukování pro více záznamů pro stejný klíč.
- U shlukovaných tabulek ukládáme záznamy se stejnou hodnotou atributu do stejného bloku, přičemž záznamy mohou být z různých tabulek.
- Rozlišujeme:
  - Indexově shlukované tabulky (Index Clustered Tables)

  - Hashované shlukované tabulky (Hash Clustered Table), Sorted Hash Clustered Table

- Podrobný popis viz dokumentace, např.
  `http://download.oracle.com/docs/cd/B19306_01/server.102/b14200/statements_5001.htm`

Indexově shlukovaná tabulka

Jméno	Odd.
Pepa Chvátal	100
Tonda Překliška	100
Jana Drábová	100
Karel Nedělá	200
Franta Rychlý	200

Tabulka 10.1: Zaměstnanci

Jméno Oddělení	Odd.
Pro styk s veřejností	100
Ekofyziologie	200

Tabulka 10.2: Oddělení

⇒

Pro styk s veřejností	100
Pepa Chvátal	100
Tonda Překliška	100
Jana Drábová	100
Ekofyziologie	200
Karel Nedělá	200
Franta Rychlý	200

Indexově shlukovaná tabulka, Příklad 1/2[2]

```
CREATE CLUSTER sc_srvr_id (
srvr_id NUMBER(10))
SIZE 1024;

SELECT cluster_name, tablespace_name, hashkeys,
     degree, single_table
```

[2] `http://psoug.org/reference/clusters.html`

```
FROM user_clusters;

CREATE INDEX idx_sc_srvr_id ON CLUSTER sc_srvr_id;

SELECT index_name, index_type, tablespace_name
FROM user_indexes;
```

Indexově shlukovaná tabulka, Příklad 2/2[3]

```
CREATE TABLE cservers (
srvr_id NUMBER(10),
network_id NUMBER(10),
status  VARCHAR2(1),
latitude FLOAT(20),
longitude FLOAT(20),
netaddress VARCHAR2(15))
CLUSTER sc_srvr_id (srvr_id);
CREATE TABLE cserv_inst (
siid       NUMBER(10),
si_status  VARCHAR2(15),
type       VARCHAR2(5),
installstatus VARCHAR2(1),
location_code NUMBER(10),
custacct_id VARCHAR2(10),
srvr_id    NUMBER(10),
ws_id      NUMBER(10))
CLUSTER sc_srvr_id (srvr_id);
```

IOT vs indexově shlukovaná tabulka

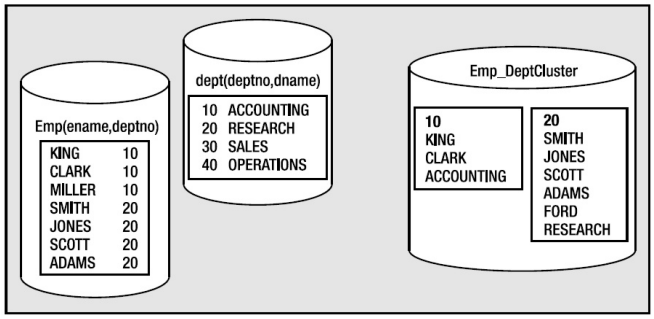
- Je indexově shlukovaná tabulka pro jednu tabulku ekvivalentní s indexově organizovanou tabulkou?
- **Není, např.:**
  - Indexově organizovaná tabulkou obsahuje v bloku záznamy ke klíčům: 1, 2, 3, 4 a 5 v tomto pořadí.
  - Indexově shlukovaná tabulka obsahuje záznamy ke klíčům: 1, 5, 2, 4, 3.
- Indexově shlukovaná tabulka zaručuje, že záznamy k jednomu klíči budou uloženy ve stejném (nebo blízkém) bloku, nezaručuje uspořádání klíčů.
- $\Rightarrow$ IOT má větší problém s aktualizacemi než indexově shlukovaná tabulka. IOT tedy využíváme jen v případě minimálního počtu aktualizací a velkého využití setřízení záznamů dle klíče.

[3] http://psoug.org/reference/clusters.html

10.1.5 Indexově shlukovaná tabulka (Index Clustered Table)

Index Clustered Table, Zadání[4]

- Budeme chtít evidovat pracoviště a jejich zaměstnance pomocí indexově shlukovaná tabulky.
- Typický dotaz bude získávat zaměstnance oddělení.



Vytvoření shluku

```
CREATE CLUSTER Employee_dept_cluster (
  deptno number(2)
)
SIZE 1024;
```

- Záznamy budou shlukovány dle atributu `deptno` (číslo pracoviště).
- Očekáváme okolo 1024bytů v záznamech asociovaných s každou hodnotou shlukovaného klíče.

Vytvoříme shlukovaný index pro shluk.

```
CREATE INDEX Employee_dept_cluster_idx
  ON CLUSTER Employee_dept_cluster;
```

[4]Thomas Kyte. Expert Oracle Database Architecture: 9i and 10g Programming Techniques and Solutions

Vytvoření shlukovaných tabulek

Vytvoříme tabulky pracovišť a zaměstnanců.

```
CREATE TABLE Department (
  deptno number(2) primary key,
  dname varchar2(14),
  loc varchar2(13))
CLUSTER Employee_dept_cluster(deptno);

CREATE TABLE Employee (
  empno number primary key,
  ename varchar2(10),
  job varchar2(9),
  mgr number,
  hiredate date,
  sal number,
  comm number,
  deptno number(2) references Department(deptno))
CLUSTER Employee_dept_cluster(deptno);
```

Dotaz

- Dotaz

```
SELECT * FROM Department d, Employee e
  WHERE d.deptno = 10 AND
        d.deptno = e.deptno;
```

  značí přístup k jednomu či několika bloků.

- V případě indexu či IOT musíme provést operací spojení.

Vložení záznamů

Pro vložení záznamů použijeme ukázkové tabulky `scott.dept` a `scott.emp`.

```
BEGIN
  for x IN (select * FROM scott.dept)
  LOOP
    INSERT INTO Department
      VALUES (x.deptno, x.dname, x.loc);
    INSERT INTO Employee
      SELECT * FROM scott.emp
        WHERE deptno = x.deptno;
  END LOOP;
END;
/
```

Dotaz

```
SELECT * FROM Department d, Employee e
 WHERE d.deptno = 10 AND
       d.deptno = e.deptno;
```

OPERATION	OBJECT_NAME	COST	LAST_CR_BUFFER_GETS
SELECT STATEMENT		1	
NESTED LOOPS		1	1
TABLE ACCESS (BY INDEX ROWID)	DEPARTMENT	0	1
INDEX (UNIQUE SCAN)	SYS_C00225178	0	1
Access Predicates			
D.DEPTNO=10			
TABLE ACCESS (CLUSTER)	EMPLOYEE	1	0
INDEX (UNIQUE SCAN)	EMPLOYEE_DEPT_CLUSTER_IDX	0	0
Access Predicates			
E.DEPTNO=10			

V$STATNAME Name	V$MYSTAT Value
bytes sent via SQL*Net to client	23961
calls to get snapshot scn: kcmgss	2
consistent gets	1
consistent gets - examination	1
consistent gets from cache	1

Dotaz

```
SELECT * FROM Department d, Employee e
 WHERE d.deptno = 10 AND
       d.deptno = e.deptno;
```

OPERATION	OBJECT_NAME	COST	LAST_CR_BUFFER_GETS
SELECT STATEMENT		1	
NESTED LOOPS		1	1
TABLE ACCESS (BY INDEX ROWID)	DEPARTMENT	0	1
INDEX (UNIQUE SCAN)	SYS_C00225178	0	1
Access Predicates			
D.DEPTNO=10			
TABLE ACCESS (CLUSTER)	EMPLOYEE	1	0
INDEX (UNIQUE SCAN)	EMPLOYEE_DEPT_CLUSTER_IDX	0	0
Access Predicates			
E.DEPTNO=10			

V$STATNAME Name	V$MYSTAT Value
bytes sent via SQL*Net to client	23961
calls to get snapshot scn: kcmgss	2
consistent gets	1
consistent gets - examination	1
consistent gets from cache	1

Hašovaná shlukovaná tabulka 1/2

- Nabízí podobný koncept jako u indexově shlukované tabulky. Místo varianty B-stromu je implementována pomocí hašování.
- Řádky se stejnou hašovací hodnotou jsou mapovány do stejného bloku. Bloky náležící jedné hašovací hodnotě jsou propojeny do seznamu.

- Z toho důvodu je nutné dobře zvolit nastavení tabulky:
  - SIZE – velikost položky.
  - HASHKEY – počet různých položek v hašovacím poli (tedy velikost hašovací tabulky).
- Hashovací tabulka alokuje HASHKEY / (Velikost bloku / SIZE) bloků.
- Můžeme použít uložení záznamů jedné nebo více tabulek v bloku.

Hašovaná shlukovaná tabulka 2/2

- Co je potřeba vědět:
  - Nelze použít na rozsahové dotazy (`WHERE key BETWEEN 50 AND 100`) (hašování nezachovává uspořádání).
  - Musíme dobře odhadnout (či znát) počet položek, které se budou v hašovací tabulce ukládat.
  - Aktualizace klíče je problematická (dochází k přesouvání do jiného bloku)
- Kdy je vhodné použít:
  - Pokud máme nějaký horní odhad počtu položek v tabulce.
  - V takovém případě získáme větší rychlost vkládání a vyhledávání než v případě B-stromu ($O(k)$ vs $O(\log n)$).

Hašovaná shlukovaná tabulka, Příklad 1/2

```
CREATE CLUSTER hcl_srvr_id (
si_clustercol NUMBER(10))
PCTFREE 0
TABLESPACE uwdata
HASHKEYS 141
ROWDEPENDENCIES;

SELECT object_name, object_type
FROM user_objects
ORDER BY object_type;
```

Hašovaná shlukovaná tabulka, Příklad 2/2

```
CREATE TABLE cservers (
srvr_id    NUMBER(10),
network_id NUMBER(10),
status     VARCHAR2(1),
latitude   FLOAT(20),
longitude  FLOAT(20),
netaddress VARCHAR2(15))
CLUSTER hcl_srvr_id (srvr_id);

CREATE TABLE cserv_inst (
siid          NUMBER(10),
si_status     VARCHAR2(15),
type          VARCHAR2(5),
installstatus VARCHAR2(1),
location_code NUMBER(10),
custacct_id   VARCHAR2(10),
srvr_id       NUMBER(10),
ws_id         NUMBER(10))
CLUSTER hcl_srvr_id (srvr_id);
```

Sorted Hash Clustered Table

- Rozšíření hašované shlukované tabulky.
- Data můžou být navíc v bloku setřízena podle nějakého atributu.
- Dotazy typu:
  `Select * FROM WHERE` **`Key`**`=hodnota ORDER BY Sort_Atribut`

10.1.6 Bitmapový index

Bitmapový index 1/2

- Pokud se dotazujeme více atributů tabulky s malými doménami (často se udává 7 a méně), je výhodné použít bitmapový index.
- Mějme tabulku `Employee` s atributy: `fname`, `lname`, `gender` (doména: {M,F}), `rating` (doména: {1,2,...,6}) a `department` (doména: {1,2,...,5}) nad kterou často používáme dotazy s aributy `gender`, `rating` a `department` v podmínce, pak je vhodné vytvořit na tyto atributy bitmapový index.
- Index založený na B-stromu by byl velký: musíme vytvořit tolik indexů kolik je dotazovaných atributů nebo jeden složených index, ale pak v dotazu nesmí první atributy složeného klíče chybět.

Bitmapový index 2/2

- **Vytvoření indexu:** `CREATE BITMAP INDEX Employee_bitmap ON Employee (gender,rating,department);`
- Vyhodnocení dotazu `SELECT * FROM Employee WHERE gender='M' AND rating=6 AND department=1` bude znamenat sekvenční průchod relativně malým polem bitových řetězců (při porovnání s velikostí tabulky) a hledání relevantních záznamů.

10.1.7 Dočasné tabulky

Dočasné tabulky

- Generují minimální množství redo log záznamů (především DELETE a UPDATE).
- Vhodné pokud například chceme v proceduře předzpracovat dat, která načítáme z externího zdroje.
- Používáme pro dočasná data generovaná nějakým algoritmem v databázi.
- Vytváříme příkazem `CREATE GLOBAL TEMPORARY TABLE`

10.1.8 ▷ Indexy

Index v Oracle

- Oracle umožňuje 'otáčet' jednotlivé byty v klíči a měnit tak distribuci klíčů (pomocí funkce `reverse()`) [5].
- Může být výhodné u indexů na primárním klíči, kde se primární klíč tvoří ze sekvence.

[5] Thomas Kyte. Expert Oracle Database Architecture: 9i and 10g Programming Techniques and Solutions, str. 429.

Oracle	SQL Server
Heap Table	Heap Table
Index organized table	Clustered index
Index	Unclustered index

  - ⇒ Při vkládání se neustále vkládá na konec indexu.
  - ⇒ U více-uživatelské aplikace budou všichni klienti přistupovat ke stejným stránkám indexu (pokud budou najednou vkládat).

Statistiky indexu

- Pohled `USER_INDEXES`.
- **Clustering factor** - jak moc pořadí záznamů v tabulce odpovídá pořadí klíčů v indexu:
  - Colocated – pokud je tato hodnota velmi blízká počtu bloků v indexu.
  - Disorganized – pokud je tato hodnota velmi blízká počtu řádků tabulky.

10.2 Fyzická implementace databáze v SQL Serveru

Indexy a tabulky v SQL Serveru

- **Heap tabulka** – záznamy v tabulce nejsou nijak uspořádány.
- **Shlukovaná tabulka** (**cluster index**) – záznamy jsou v datovém souboru uspořádány v indexu podle nějakého klíče. Obdoba indexově organizované tabulky (IOT) v Oracle.
- **Index** (**unclustered index**) – klasický B^+strom, kterých může být na jedné tabulce vytvořeno více.

Porovnání s Oraclem

10.3 Návrh indexů a typů tabulek

Potencionální kandidáti na index

- Index se vytváří obvykle pro klíče a cizí klíče.
- Existují dvě základní pravidla pro vytvoření indexu:
  - V případě, že index bude používán k nalezení malého počtu záznamů v tabulce.
  - V případě, že index pokryje jeden nebo více dotazů.
- Atributy, které se často vyskytují v klausuli `where`, jsou potenciálními kandidáty na index.
- Každý index znamená zvýšení počtu operací při změnách v databázi. Vytvoření nového indexu tedy musíme vždy pečlivě zvažovat.
- Musíme pečlivě vybírat atribut, podle kterého se provádí shlukování záznamů v tabulce. Měl by to být atribut (atributy), který odpovídá nejdůležitějším dotazům.

Ladění databáze

- Obecně rozlišujeme dva způsoby ladění:
  - Proaktivní – snažíme se analyzovat fyzický návrh databáze a provádět změny směrem k lepšímu fungování.
  - Reaktivní – reagujeme na nějaké problémy v databázi.
- Ladění databáze je iterativní proces.
- Vždy bychom měli mít dobře zmapovaný stávající stav.
- Po provedené změně bychom měli zkontrolovat dopad na celkový výkon databáze.

10.4 Analýza a návrh indexů

Analýza a návrh indexů 1/4

1. Sestavíme tabulku (ne v databázi, ale v textovém dokumentu) pro všechny dotazy (operace SELECT) a aktualizace (operace INSERT, UPDATE a DELETE) se sloupci:
   - Dotaz nebo aktualizace (označíme zkratkou, např. A1 nebo U1).
   - Četnost (např. častý, méně častý, používán výjimečně; případně označíme čísly).
   - Průměrná selektivita dotazu (% k celkovému počtu záznamů).
2. Jednotlivé dotazy a aktualizace seskupíme dle tabulek se kterými pracují.

Analýza a návrh indexů 2/4

- Navrhneme indexy pro:
  - Primární klíče (jsou vytvořeny automaticky).
  - Cizí klíče (pokud dle nich vyhledáváme nebo je používáme při spojování celých tabulek).
  - Atributy které se nacházejí v dotazech za WHERE (kromě dotazů na nerovnost hodnot atributů ¡¿).
- Sledujeme přitom tato **kritéria**:
  - Maximální zlepšení dotazování.
  - Minimální zhoršení aktualizace (operací INSERT, UPDATE a DELETE).
  - Minimální velikost indexů.

Analýza a návrh indexů 3/4

- Musíme vyhodnotit četnost dotazování a aktualizací pro daný atribut. Pokud např. bude UPDATE častější operace než dotazování, pak vytvořením indexu zhoršíme propustnost systému (aktualizace pro indexovaný atribut znamená aktualizaci tabulky i indexu).
- Neplatí vždy: pokud je součástí aktualizace podmínka s indexovaným atributem, pak index zrychluje aktualizaci záznamu, resp. vyhledání aktualizovaného záznamu.

Analýza a návrh indexů 4/4

1. Vytvoříme indexy a sestavíme tabulku (ne v databázi, ale v textovém dokumentu) pro všechny dotazy se sloupci:
   - Název indexu.
   - Typ indexu (B-strom, bitmapový index).
   - Dotazy (jejich zkratky), které zrychluje, počet logických operací před a po vytvoření indexu.
   - Aktualizace (jejich zkratky), které zpomaluje, počet logických operací před a po vytvoření indexu.
   - Velikost indexu v blocích (pro testovací data).
   - Zhodnocení, zda je na základě výsledků nutné index vytvářet.
   - Celková velikost indexů pro tabulku a velikost tabulky v blocích.
2. Pokud pro některé dotazy nebudeme vytvářet index, zdůvodníme.

Reference

- Thomas Kyte. Expert Oracle Database Architecture: 9i and 10g Programming Techniques and Solutions.
- Oracle. CREATE TABLE manual. `http://download.oracle.com/docs/cd/B19306_0 server.102/b14200/statements_7002.htm`
- Oracle. CREATE CLUSTER manual. `http://download.oracle.com/docs/cd/B1930 server.102/b14200/statements_5001.htm`
- Microsoft. Clustered Index Design Guidelines. `http://msdn.microsoft.com/en-us/library/ms190639.aspx`

Kapitola 11

Datové struktury využívané v ŠRBD

Obsah

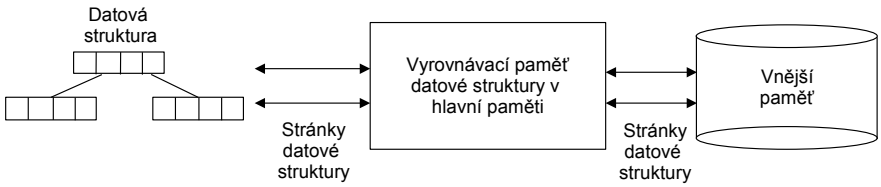
11.1 Úvod

Libovolný datový model by byl nepoužitelný bez využití efektivních **datových struktur** a **algoritmů** [48, 13, 35]. 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 načtení dvou záznamů z disku a porovnání dvou hodnot trvalo 0.1 ms, pak by vyhledání správného záznamu trvalo 2h 46m 40s. Bezpochyby by byl takový SŘBD v praxi nepoužitelný. Je tedy jasné, že fyzická implementace uložení dat je jedním z podstatných problémů databázových technologií. Optimalizace dotazů realizovaná v SŘBD plánovačem dotazů (viz kapitola 7) je nejčastěji založena na využití vhodných indexů.

11.2 Perzistentní datové struktury

Je zjevné, že datové struktury obsahující data musí být **perzistentní** [24], tj. musí být uloženy v nějaké vnější paměti, kde zůstanou uloženy i po nekorektním ukončení systému, restartu počítače apod. Na obrázku **??** 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** (nejčastěji o velikost 512 B). Souborové systémy pak používají **alokační jednotky**, tedy nejmenší jednotku na disku se kterou pracuje souborový systém, o velikosti násobků velikosti diskových bloků (nejčastěji 2 048 nebo 4 096 B).



Obrázek 11.1: 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. Nejčastěji používaným mechanismem rozhodujícím, které stránky budou uloženy ve vyrovnávací paměti jsou

časová razítka. Pokud je vyrovnávací paměť plná, z vyrovnávací paměti je vymazána stránka, která byla použita nejdříve, a na její místo je načtena stránka nová. Pokud byla původní stránka modifikována, je před smazáním z vyrovnávací paměti uložena do vnější paměti.

11.3 Složitost algoritmů

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

Definice 11.1. *(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 $0 \leq f(n) \leq c \cdot g(n)$ pro všechna $n \geq n_0\}$.*

To tedy znamená, že $5 \cdot n = O(n)$, $120 \cdot n = O(n)$ ovšem $n^2 \neq O(n)$. Tato míra nám tedy určuje složitost pouze asymptoticky. Pokud má algoritmus složitost v $O(n)$, víme že počet operací provedených během chodu algoritmu je $\leq c \cdot n$. Známe tedy relativně nepřesnou informaci. Co je ale důležité, víme že počet operací nebude například exponenciální s počtem prvků vstupu.

11.4 Datové struktury

V případě datových struktur se zajímáme především o složitost následujících operací: **vyhledání** (**find**), **vkládáni** (**insert**), **rušení** (**delete**) a **modifikace** (**update**). Velmi často je operace vyhledání využívána i ostatními operacemi, za základní tedy budeme považovat operaci vyhledání.

Algoritmus, který jsme si popisovali v kapitole 11.1 byl algoritmus vyhledání v poli s asymptotickou složitostí v $O(n)$ [13]. Pokud bychom použili algoritmus binární vyhledávání v setříděném poli, pak bychom dosáhli složitosti $O(\log n)$. Pro 10^8 záznamů a doby jednoho porovnávání 0.1ms bychom jeden záznam vyhledali za $8 \cdot 0.001s = 0.008\,s$. Zdálo by se, že takové řešení je vyhovující. Zaměřme se nyní na algoritmy vkládání a modifikace záznamu. Pokud bychom chtěli třídit záznamy při vkládání do takového **pole**, museli bychom nejprve nalézt místo v souboru, kde bude nový záznam vložen (tedy zatřízen). Poté musíme všechny záznamy ležící za ním přesunout o jeden záznam ke konci souboru a nový záznam vložit. Je jasné, že takový algoritmus se nedá, kvůli

velkým přesunům dat, nazvat efektivní. Nyní si ukažme datové struktury, které se pokusí tento problém řešit efektivněji (z pohledu časové složitosti) než takové pole.

11.4.1 Stránkovaný seznam

Pozorný čtenář jistě namítne, že by celý problém bylo možné řešit **stránkovaným seznamem**. Tj. seznam by obsahoval stránky (tzv. **bloky**) o velikosti alokační jednotky souborového systému (např. 2kB) a samotné záznamy by byly uloženy v těchto blocích. Jelikož záznam může mít proměnnou délku, kapacita bloku (měřeno maximálním počtem uložených záznamů) nebude konstantní. Každý blok bude mít explicitní jedinečné číslo (pořadí bloku, zkráceně id) a bude obsahovat ukazatel na následující blok. Tímto způsobem bychom vyřešili problém s přesouváním všech záznamu při zatřizování: záznam bychom zatřídili do nalezeného bloku a pokud by došlo k přetečení (situace kdy pro nový záznam není v bloku místo, angl. **blok overflow**), pak by došlo k rozštěpení bloku (angl. **blok split**). Při štěpení by první polovina záznamů zůstala v původním bloku, pro druhou polovinu záznamu bychom vytvořili nový blok a ukazatel na následující blok v původním bloku by byl nastaven na blok nový. Nový blok by převzal ukazatel na následující blok od původního bloku.

Problém této datové struktury je komplikovanější vyhledávání. Při půlení intervalu musíme vybírat prostřední prvek z intervalu (v tomto případě tedy nějaký prvek s prostředního bloku) a porovnávat jej s prvkem prohledávaným. Pokud je hledaný prvek $\leq$ pak budeme prohledávat levou část intervalu, pokud je hledaný prvek $>$, pak budeme prohledávat pravou část intervalu. Problém je ale v tom, že nemáme k dispozici náhodný přístup k blokům (jako v případě pole). Jak je to možné? Uvažujme tento příklad. V seznamu budeme mít 10 bloků a bloky budou uspořádány, tedy všechny klíče záznamů bloku s jedinečným číslem i budou $\leq$ než všechny klíče záznamů bloku s jedinečným číslem $i + 1$, kde $1 \leq i \leq 9$. Nyní budeme chtít vložit záznam do 5. bloku, který musí být rozštěpen: vznikne nový uzel s id 11, který ale z pohledu uspořádání klíčů leží mezi bloky 5 a 6. Uspořádání jedinečných čísel bloků neodpovídá uspořádání klíčů a binární vyhledávání by selhalo ve výběru prostředních bloků z prohledávaného intervalu.

V tomto případě tedy musíme navíc udržovat pole, která bude obsahovat jedinečná čísla bloků setřízena dle uspořádání klíčů, v tomto případě by tabulka vypadala takto: $\{1, 2, 3, 4, 5, 11, 6, 7, 8, 9, 10\}$. Při výběru prostředního bloku bychom tedy jedinečné číslo uzlu našli v této datové struktuře. Jelikož počet bloků $<<$ než počet záznamů, režie práce s tímto polem bude neporovnatelně nižší než

režie práce se seznamem. Další nevýhodou je časová složitost vyhledávání $O(\log_2 n)$, kde n je počet záznamů. Je otázka zda neexistuje datová struktura, která by umožnila vyhledávání s logaritmickou časovou složitostí o vyšším základu než 2.

11.4.2 Indexový soubor

V minulosti byly velmi často používané tzv. **indexové soubory**[1]. Záznamy byly vkládány za sebe do datového souboru, tak jak byly uživatelem vloženy. SŘBD obsahoval příkaz, kterým bylo možné k takovému souboru vytvořit indexový soubor, setříděné pole dvojic ⟨hodnota indexovaného atributu, ukazatel do datového souboru⟩. Při vložení musí být do indexového souboru zatřízen vkládaný záznam. Jelikož režie zatřizování do pole bude vysoka (především kvůli posunu záznamů), pro implementaci musíme zvolit datovou strukturu podobnou stránkovanému seznamu z předchozí kapitoly.

V následující kapitole si představíme datovou strukturu, která řeší výše popsané problémy.

11.5 Stromové datové struktury

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

Definice 11.2. *(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.*

Pokud má uzel seřazeného stromu k potomků, lze určit prvního potomka, druhého potomka, až k-tého potomka. Uzly, které nemají žádné děti nazýváme **listové uzly**, uzly, které mají děti, pak **vnitřní uzly** stromu. Počet dětí uzlu **u** nazýváme **stupeň uzlu u**. **Cesta** k nějakému uzlu **u**, je posloupnost všech uzlů od kořene k uzlu **u**. **Délka cesty** je rovna počtu hran, které cesta obsahuje, tedy počtu uzlů posloupnosti - 1. **Výška stromu h** je rovna délce nejdelší cesty ve

[1] Tento název nás může zmást, můžeme se totiž domnívat, že indexový soubor je jiný název pro index a např. příkaz CREATE INDEX v SQL znamená vytvoření indexového souboru. Není to pravda, indexem rozumíme libovolnou datovou strukturu s lepší složitostí operací než lineární, tedy např. B-strom či R-strom.

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 vyhledávací strom, který nám bude sloužit pro demonstraci vlastností stromových datových struktur. V kapitole 11.5.2 popíšeme datovou strukturu B-strom, která je používaná v drtivé většině SŘBD.

11.5.1 Binární vyhledávací strom

Definice 11.3. *(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* ***levý podstrom*** *a binárního stromu zvaného* ***pravý podstrom****.*

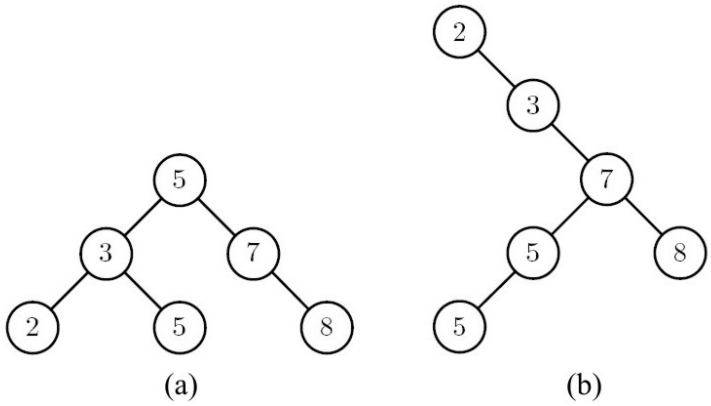
U binárního stromu, pokud chybí potomek nějakého uzlu, pak stále rozlišujeme levého a pravého potomka. U seřazeného stromu takové rozlišení není možné, binární strom tedy není seřazený strom.

Každý uzel **binární vyhledávací stromu** obsahuje **klíč**, na jehož doméně je definováno nějaké uspořádání. Pro jednoduchost si představme množinu N a uspořádání $\leq$. V levém 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 11.1. (Binární vyhledávací strom). Na obrázku 11.2 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.

Vezměme cestu 5,7,8 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 pravé dítě. Pokud



Obrázek 11.2: Příklady binárních vyhledávacích stromů

$k_a = k$, 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 $1, 2, 3, \ldots, 1000$. 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 $O(\log_c n)$.

V případě binárního vyhledávacího stromu můžeme výsledný strom ovlivnit pořadím vkládaných prvků. Necháme na čtenáři aby vymyslel jaké pořadí způsobí, že výsledný strom bude vyvážený.

11.5.2 B-strom

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

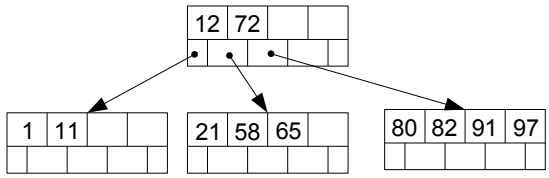
Definice 11.4. *(B-strom). B-strom řádu c je $(2 \cdot c + 1)$-ární strom, který splňuje následující kritéria:*

- *Každá stránka (uzel) obsahuje nejvýše $2 \cdot c$ položek (klíčů).*
- *Každá stránka, s výjimkou kořene, obsahuje alespoň n položek.*
- *Každá stránka, s výjimkou kořene, obsahuje alespoň n položek.*
- *Každá stránka je buď listovou, tj. nemá žádné následovníky nebo má $m + 1$ následovníků, kde m je počet klíčů ve stránce.*
- *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 2048 B). 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 11.2. (Vyhledávání v B-stromu). Na obrázku 11.3 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.



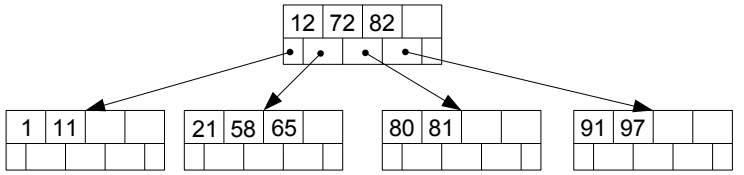
Obrázek 11.3: B-strom řádu 2

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 $< 2 \cdot c$, pak položku zatřídíme do uzlu.
2. Počet položek $= 2 \cdot c$. 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:
   a) Počet položek $< 2 \cdot c$, pak položku k_i zatřídíme do uzlu. Kromě položky je nutné zatřídit i ukazatel na nově vzniklý uzel. Algoritmus končí.
   b) Počet položek $= 2 \cdot c$. 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 11.3. (Vkládání do B-stromu). Na obrázku 11.4 vidíme B-strom po vložení položky 81 do B-stromu z obrázku 11.3. Nejprve vyhledáme listový uzel do kterého má být položka vložena. Tento uzel obsahuje položky 80, 82, 91 a 97. Jelikož uzel obsahuje $2 \cdot c$ 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 $< 2 \cdot c$, 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.



Obrázek 11.4: B-strom z obrázku 11.3 po vložení položky 81

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

1. Položka se nachází v listovém uzlu: položka je 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.

11.6 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
```

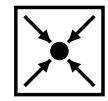
Musíme v B-stromu indexujícím atribut `rocnik` nalézt záznamy s klíčem rovným hodnotě 4 a v B-stromu indexujícím `rokNarozeni` záznam s klíčem rovným hodnotě 1983. Takto získané mezivýsledky pak musíme spojit. Výsledkem jsou záznamy, které odpovídají definované podmínce. Mezivýsledky mohou mít daleko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

Vícerozměrné datové struktury [49], 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**.

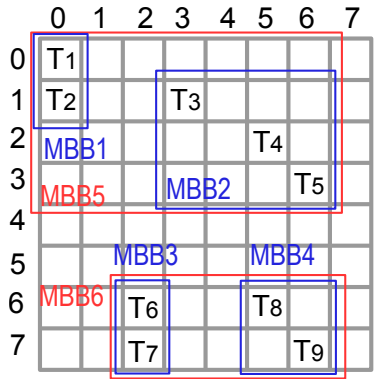
11.6.1 R-strom

Jednou z prvních vyvinutých vícerozměrných datových struktur je **R-strom**, který byl publikován v roce 1984 [17]. 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.

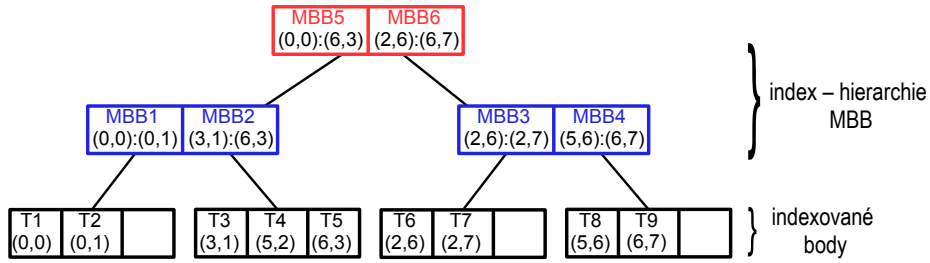
Příklad 11.4. (R-strom). Na obrázku 11.5 vidíme dvou-rozměrný prostor velikosti 8×8 s osmi body. Na obrázku 11.6 vidíme R-strom indexující tento prostor. Listový uzel, který je nakreslen nejvíce vlevo, obsahuje body (0,0) a (0,1). Minimální ohraničující obdélník těchto bodů MBB1 je obdélník definovaný 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.



Ve vektorových datových strukturách definujeme několik typů dotazů. Mezi nejpoužívanější patří bodový a rozsahový dotaz. **Bodový dotaz** je definován



Obrázek 11.5: Indexovaný dvourozměrný prostor



Obrázek 11.6: R-strom indexující tento prostor

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} ≤ a_1 ≤ QB_{h1} AND QB_{l2} ≤ a_2 ≤ QB_{h2} AND ...
AND QB_{ln} ≤ a_n ≤ 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 11.6, 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].

Kapitola 12

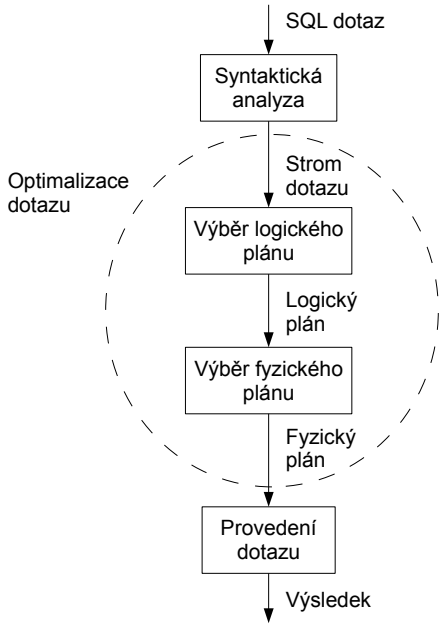
Vyhodnocování dotazů

Obsah

Cíl kapitoly:
V této kapitole je podrobněji vysvětlen způsob vykonávání dotazů v relačních SŘBD. Zaměřujeme se zejména na sestavení plánu vykonání dotazu a na implementaci operace spojení.

12.1 Úvod

V kapitole 4 jsme se seznámili s relační algebrou a jazykem SQL. Nyní se podrobně seznámíme se způsobem vyhodnocení dotazu v SŘBD. O tento proces se stará **část pro vykonání dotazů** (angl. **Query processor**) složená z několika dílčích kroků, které převádějí dotaz do elementárních operací relační algebry a provádějí tyto operace [16]. Na obrázku 12.1 můžeme vidět schéma vyhodnocování dotazu.



Obrázek 12.1: Části vyhodnocování dotazu

Nejprve probíhá syntaktická analýza a převod dotazů z SQL do relační algebry (**plán pro vyhodnocení dotazu**). Bohužel tento převod není jednoznačný a z toho důvodu hraje významnou roli **optimalizace dotazu**, která se snaží vybrat plán pro vyhodnocení dotazu, který proběhne za co nejkratší dobu.

Rozlišujeme dva druhy plánů pro vykonání dotazu. První je **logický plán vykonání dotazu**, který je vlastně stromem pro vykonání dotazu s použitím relační algebry. Druhý je **fyzický plán vykonání dotazu**, který už vybírá konkrétní algoritmy implementující jednotlivé operátory logického plánu. Sestavení obou plánů je netriviální část celého vyhodnocení.

V následujících kapitolách si představíme základní algoritmy implementující operátory relační algebry a některé techniky pro optimalizaci plánu vykonání.

12.2 Fyzický plán vykonání dotazu

Operátory fyzického plánu vykonávání dotazu udávají algoritmy jenž se použijí pro implementaci daného relačního operátoru. Některé fyzické operátory ovšem mohou popisovat operace, které s relačními operátory přímo nesouvisí. Může to být například třídění nějaké relace pro lepší efektivitu spojení, nebo sekvenční průchod relací.

Jednotlivé záznamy relace jsou uloženy v blocích na disku (vnější paměti). Počet přístupů, které vyžaduje daný algoritmus pak nazýváme I/O složitostí a toto je hlavním parametrem při posuzování složitosti. Vedlejším parametrem pak je tzv. procesorová složitost, tedy doba běhu algoritmu v hlavní paměti, která je obvykle ve srovnání s I/O složitostí nevýznamná. Procesorová složitost se bere do úvahy pouze v případě, že se jedná o netriviání algoritmus s nelineární složitostí.

Použité značení v následujících kapitolách:

- $B(\mathtt{X})$ – počet bloků, které zabírá relace X.
- M – počet bloků, které můžeme uložit v hlavní paměti.
- Block_Size – velikost jednoho bloku. Obvykle $4 - -16$ [kB].

Prozatím budeme předpokládat, že nemáme k dispozici žádný index, který by umožňoval rychlejší přístup k záznamům relace.

12.2.1 Spojení relací

Operace spojení dvou relací je obecně nejkritičtější operací při vykonání dotazu, proto je této operaci v plánu pro vykonání věnována speciální pozornost. Obecně rozdělujeme algoritmy pro spojení dvou relací na tři typy:

1. **Hnízděné cykly** (angl. **Nested-loop joins**)
2. **Třídění-slévání**
3. **Hašovaná spojení**

Hnízděné cykly

Tento algoritmus představuje triviální verzi algoritmu spojení, kde je brána v úvahu architektura s vnější pamětí popsaná na začátku této kapitoly. Algoritmus 2 popisuje spojení dvou relací `S(X,Y)` a `R(Y,Z)`, které mají velikost přesahující M bloků a $B(\mathtt{S}) \geq B(\mathtt{R})$. Atribut `Y` představuje společný atribut obou relací, přes který se provádí spojení. Algoritmus používá také termín **svazek bloků**, což je prostě množina za sebou ležících bloků relace.

Algoritmus 2: Hnízděné cykly

```
1 foreach svazek M − 1 bloků relace S do
2     Načtení i-tého svazku M − 1 bloků do hlavní paměti;
3     Setřízení záznamů v tomto svazku podle atributu Y;
4     foreach blok b relace R do
5         Načtení bloku b do hlavní paměti;
6         foreach n-tici t ∈ b do
7             Nalezení všech záznamu s atributem t.Y ve svazku relace S;
8             Vytvoření výsledné n-tice pro každou shodu a zapsání výsledku;
```

Algoritmus 2 je zapsán s využitím třech vnořených cyklů, nicméně vnější cyklus provádí průchod relací `S` a další dva vnitřní provádí průchod relací `R`. Z pohledu I/O složitosti jde o dva vnořené cykly. I/O složitost tohoto algoritmu je

$$\frac{B(\mathtt{S})}{M-1}(M-1+B(\mathtt{R})))$$

Obecně není tento algoritmus příliš efektivní a je vhodný v případě, že spojujeme malé relace.

Příklad 12.1. (Hnízděné cykly). Mějme dvě relace, které chceme spojit: `Student(loginStudenta, jmeno)` a `Studuje(predmet, loginStudenta)`, kde $B(\mathtt{Student}) = 500$, $B(\mathtt{Studuje}) = 50\,000$ a $M = 101$. I/O složitost takového spojení s využitím hnízděného cyklu lze lehce spočítat. Ve vnějším cyklu je vždy menší relace (`Student`) tedy I/O složitost 250 500. V případě že by ve vnějším cyklu byla relace `Studuje` pak by byl počet diskových přístupů algoritmu byl 300 000.

Třídění-slévání

Při spojování dvou relací `S(X,Y)` a `R(Y,Z)` hraje důležitou roli také další parametr. Je to nejvyšší počet záznamů se stejnou hodnotou v atributu `Y`. V tom

nejhorším případě mají všechny záznamy stejnou hodnotu tohoto atributu. V tomto případě se nevyhneme hnízděným cyklům i v případě velkých relací. Dále budeme u tohoto algoritmu předpokládat, že nám počet záznamů se stejnou hodnotou atributu `Y` vejde do paměti o velikosti M.

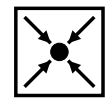
Algoritmus 3 nejprve setřídí každou relaci podle spojovacího atributu `Y` pomocí **dvoufázového třídění vícecestným sléváním** (**two-phase multiway merge sort**). Každou z těchto setřízených relací pak procházíme s využitím operací `eof()` a `advance()` (jejich implementace zde není důležitá). Každá relace má přiřazen kurzor, který je na začátku slévání nastaven na první n-tici setřízené relace. Aktuální n-tici na kterou ukazuje kurzor značíme v algoritmu malým písmenem. Operace `advance()` posune kurzor na další n-tici s jinou hodnotou atributu `Y` než je ta aktuální. Operace `eof()` vrací true pokud jsme se dostali za poslední n-tici relace.

Algoritmus 3: Třídění-slévání

```
1  Seřídění relace R pomocí dvoufázového třídění vícecestným sléváním;
2  Seřídění relace S pomocí dvoufázového třídění vícecestným sléváním;
3  while ¬R.eof() and ¬R.eof() do
4      if r.y == s.y then
5          Kartézský součin všech n-tic z relací R a S s touto hodnotou;
6          Zapsání výsledků;
7          R.advance();
8          S.advance();
9      else
10         if r.y < s.y then
11             R.advance();
12         else
13             S.advance();
```

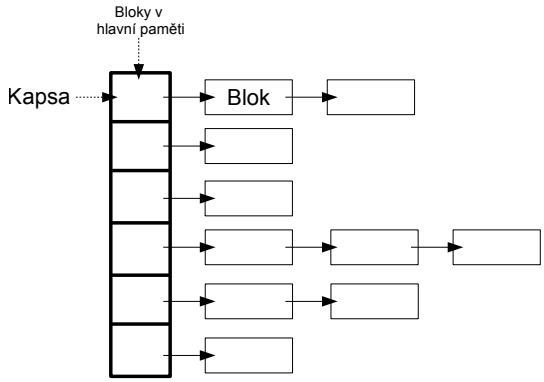
I/O složitost dvoufázového třídění relace `R` je $4 \cdot B(\mathtt{R})$. Přesněji to platí jen při splnění nerovnosti $B(\mathtt{R}) \leq M^2$, v tomto případě tuto skutečnost budeme mít za splněnou. Fáze slévání se provede jedním průchodem obou relací. Ve výsledku je tedy I/O složitost algoritmu 3: $5(B(\mathtt{R}) + B(\mathtt{S}))$

Příklad 12.2. (Třídění-slévání). Mějme dvě relace `Student` a `Studuje` z příkladu 12.1, které budeme chtít spojit s využitím algoritmu třídění-slévání. Složitost takového algoritmu bude 252 500 což je nepatrně více než v případě hnízděných cyklů. Musíme si však povšimnout, že složitost algoritmu třídění-slévání roste lineárně s velikosti relací, zatímco hnízděné cykly mají kvadratickou složitost. S rostoucí velikostí relací bude tedy algoritmus třídění-slévání efektivnější.



Hašovaná spojení

Hlavní myšlenkou hašovaných spojení je rozdělení relace R do M disjunktních podmnožin (**kapsy**) pomocí hašovací funkce. **Hašovací funkce** rozděluje n-tice se stejnou hodnotou atributu který hašujeme (může jich být i více) do stejné kapsy. Naopak se ale také může stát (a v praxi velmi často), že se do stejné kapsy mapují záznamy s různými hodnotami. Schématické znázornění takového rozdělení relace do jednotlivých kapes lze vidět na obrázku 12.2. Z obrázku je patrné, že jedna kapsa se může skládat s většího množství bloků v závislosti na počtu záznamů v jedné kapse. V každé fázi algoritmu vytváření kapes pak držíme v hlavní paměti pouze jeden blok náležící jedné kapse. Algoritmus 4 popisuje algoritmus vytváření kapes pro relaci R.



Obrázek 12.2: Rozdělení relace do kapes

Algoritmus 4: Vytváření kapes

```
1 foreach blok b ∈ R do
2     čti blok b;
3     foreach n-tici t ∈ b do
4         if blok kapsy h(t) je plný then
5             zapiš blok na disk;
6             inicializuj nový prázdný blok pro kapsu h(t);
7         vlož t do kapsy h(t);
8 Zapsání zbývajících bloků na disk;
```

Algoritmus hašovaného spojení dvou relací R a S pak probíhá v následujících dvou krocích:

1. Vytvoření kapes pro každou relaci.
2. Slévání i-té kapsy relace R s i-tou kapsou relace S

Na vytvoření kapes pro jednu relaci X je potřeba $2 \cdot B(\mathtt{X})$ přístupů na disk (načtení každého bloku a jeho následné zapsání). Při slévání jednotlivých kapes (druhý bod) pak čteme každý blok pouze jednou, jelikož opět předpokládáme, že se nám n-tice se stejnou hodnotou spojovacího atributu vejdou do M bloků v hlavní paměti. I/O složitost hašovaného algoritmu spojení je tedy: $3(B(\mathtt{R}) + B(\mathtt{S}))$.

Příklad 12.3. (Hašované spojení). Mějme dvě relace `Student` a `Studuje` z příkladu 12.1, které budeme chtít spojit s využitím algoritmu hašované spojení. Složitost takového algoritmu bude 151 500 což je nejméně ze všech předchozích algoritmů. Odhad složitosti hašované spojení ovšem předpokládá, že existuje hašovací funkce, která nám rovnoměrně rozdělí n-tice do kapes. To nemusí být vždy pravda. Hnízděné spojení bude navíc efektivnější pro dostatečně malé relace.

12.2.2 Selekce

Složitost operace selekce závisí na parametrech relace a na typu zvolené selekce. Předpokládáme jednoduchou selekci `Atr` θ `hodnota`, kde $\theta = \{=, <, >, \leq, \geq\}$. Nejprve definujeme následující konstanty pro relaci X:

- `V(Atr, X)` - počet různých hodnot atributu `Atr` v relaci X.
- `N(Atr, X)` = `B(X) / V(Atr, X)` - velikost výsledku porovnání (v průměrném případě)
- `h(Atr)` - výška B^+-stromu pro atribut `Atr`.

V případě, že záznamy v relaci nejsou v blocích shlukovány podle atributu selekce a nemáme k dispozici ani index pro daný atribut pak pochopitelně nezbývá, než procházet celou relaci sekvenčně. Samozřejmě, I/O složitost takové operace je $B(\mathtt{X})$.

Dále stanovíme I/O složitost operace selekce pro selekci s rovností. Podle toho o jaký atribut jde a zda máme k dispozici index, nebo záznamy shlukované do bloků, můžeme jednotlivé případy rozdělit takto:

- *`Atr` je primární klíč a máme k dispozici index*

$$\mathtt{h(Atr)} + 1$$

- *Máme k dispozici index pro* `Atr`

  $$\mathtt{h(Atr)} + \mathtt{N(Atr, X)}$$

  Pokud nejsou záznamy shluknuty vedle sebe do bloků podle atributu `Atr`, může každá n-tice nalezená v indexu ležet v jiném bloku. Počet načtených bloků tedy může být až `N(Atr, X)`.

- *n-tice jsou shlukovány do bloků podle hodnot atributu* `Atr` *a máme k dispozici index*

  $$\mathtt{h(Atr)} + \frac{\mathtt{N(Atr, X)}}{Block\_Size}$$

  Počet načtených bloků relace je v tomto případě menší, protože n-tice jsou umístěny v bloku za sebou.

Podobně můžeme stanovit složitost selekce také pro další typy operací. Pochopitelně stěžejní je v tomto případě odhad velikosti výsledku takové operace.

12.2.3 Další operace

Relační algebra obsahuje množství dalších operací (množinové operace, operace projekce, `GROUP BY` atd.), které je nutné implementovat pomocí efektivních algoritmů. Na některé lze využít techniky popsané výše (třízení-slévání, hašování). Pro podrobnější seznámení s těmito algoritmy je možné nahlédnout do [16, 37].

12.3 Optimalizace dotazů

V zásadě dochází ke dvěma typům optimalizací:

- Optimalizace plánu vykonání dotazu na základě pravidel relační algebry. Dochází k **minimalizaci plánu** a k vynechaní redundantních částí [16, 37].
- Cenová optimalizace plánu vykonání dotazu na základě statistik relací. Optimalizátor odhaduje **velikosti výsledku**, **cenu operací** a **cenu celého plánu vykonání** (angl. **cost-based analysis**). [16, 37].

Alternativních plánu vykonání daného dotazu existuje obvykle velké množství (jejich počet roste exponenciálně s velikostí dotazu). Cena plánu je ale odhadována jen pro několik alternativních plánů, které mají potenciálně nízkou

cenu vykonání. Například selekce se téměř vždy provádí před operací spojení pro zmenšení množin, které do operace spojení vstupují. V dotazech často bývá několik operací spojení, takže musíme určit pořadí jejich provádění, jelikož operace spojení je binární. Pokud je to možné, nejprve se spojují relace s menší odhadnutou velikostí. I přes to, že výběr optimálního plánu znamená NP-úplný problém jedná se, vzhledem k malé velikosti výrazu, o problém zvládnutelný ve velmi krátkém čase. U cenové optimalizace je potřeba také zohlednit, že doba cenové optimalizace by neměla překročit dobu ušetřenou touto optimalizací.

Kapitola 13

Úložiště dat

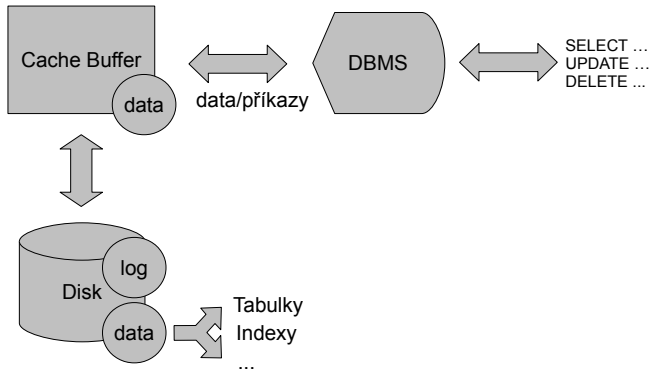
Obsah

Cíl kapitoly:
V této kapitole jsou podrobně popsána úložiště dat jako jsou disky a disková pole. Popsané vlastnosti disků pak vysvětlují proč je často v plánu vyhodnocení dotazu použito sekvenční namísto náhodného čtení. U diskových polí budou popsány jednotlivé specifikace RAID.

13.1 Úvod

Pro uložení dat v databázových systémech musíme řešit dva zásadní problémy. Prvním problémem je zajištění persistence dat pro zaručení **trvalosti** změn (tedy vlastnosti D - durability z ACID). Tento cíl je řešen uložením dat na disku. Druhým problémem je zajištění maximálního výkonu (propustnosti) databázového systému. Jelikož paměť poskytuje řádově vyšší propustnosti než disk (6 GBps vs 3 – 500 MBps), SŘBD se snaží maximální objem dat umístit v hlavní paměti (v tzv. **cache buffer**). Odhadovaná hranice je 90%, tzn. v nejhorším případě by mělo být 10% dat umístěno na disku, 90% v paměti. Výkon databázového systému tedy do jisté míry souvisí s výkonem disku (obecně úložiště dat). Kvůli persis-

tenci a zotavení se pak systém musí starat o správu logu a aktualizaci datových souborů na disku.



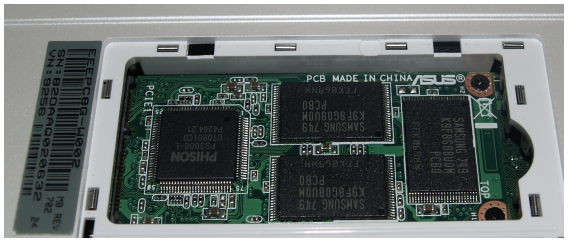
Obrázek 13.1: Uložení dat SŘBD na disku a v hlavní paměti.

Na obrázku 13.1 vidíme uložení dat SŘBD na disku a v hlavní paměti. Jakmile uživatel požaduje nějakou stránku (tabulky, indexu apod.), SŘBD zjišťuje zda stránka není umístěna v cache buffer, pokud ne, je stránka načtena z disku do paměti. Pokud požadovaná stránka tabulky nebo indexu je v cache buffer, mluvíme o **cache hit**. Pokud požadovaná stránka není v cache buffer, mluvíme o **cache miss**.

13.2 Disky

V současné době rozlišujeme především dva typy disků: **HDD** (hard drive, hard disk) a **SSD** (solid state drive). **HDD** (viz obrázek 13.2) je magnetický disk s mechanickým pohybem čtecích a zapisovacích hlav. Data jsou umístěna ve stopách na jednotlivých plotnách. Nejmenší jednotkou, kterou je možné na disku alokovat (číst nebo zapisovat), je sektor o velikosti 512 B. Tyto disky obecně poskytují větší kapacity a nižší výkon především pro náhodné čtení a zápis. **SSD** (viz obrázek 13.3) naproti tomu neobsahuje mechanické části, mezi nejčastěji používané paměti těchto disků pro uložení dat patří: flash, SRAM, DRAM. Tyto disky obecně poskytují nižší kapacity a vyšší výkon zejména náhodného čtení a zápisu, cena na 1 GB je vyšší než v případě HDD. Operace čtení a zápis nazýváme **vstupně-výstupní operace** nebo **přístupy** (angl. input-output (IO) accesses nebo disk accesses).

Obrázek 13.2: HDD, převzato z `http://en.wikipedia.org/wiki/Hard_disk_drive`

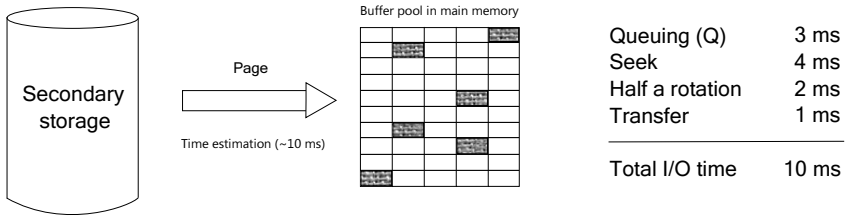


Obrázek 13.3: SDD, převzato z `http://cs.wikipedia.org/wiki/Solid-state_drive`

Jak již bylo řečeno nejmenší alokační jednotkou disku je sektor o velikost 512 B. Souborové systémy jako ext2, NTFS apod. používají nejmenší alokační jednotku o velikosti násobku velikosti sektoru, nejčastěji 2 kB. Stránky datových struktur použité v SŘBD pro uložení dat pak mají velikost v násobcích těchto alokační jednotek, nejčastěji 8 nebo 16 kB.

Nyní se podívejme na problematiku čtení a zápisu, přesněji sekvenčního a náhodného čtení a zápisu. Na obrázku 13.4 jsou uvedeny jednotlivé operace disku při čtení dat [22]. Vidíme, že samotný přenos dat představuje 10% z celkové času čtení 10 ms. Největší režii tedy představuje příprava na čtení spojená s přenastavením čtecí hlavy. V takovém případě provedeme pouze 100 čtení za sekundu, tzv. **IOPS** (vstupně-výstupních operací za sekundu). V případě sekvenčního čtení kdy čteme jednotlivé sektory za sebou, tato režie odpadá a

získáme tak maximální propustnost (až 500 MB/s). Při náhodném čtení stránek, např. při bodovém nebo rozsahovém dotazu v B-stromu, je každá stránka načtena s výše popsanou režií. Výsledkem je minimální propustnost: pokud budeme uvažovat stránku o velikost 8 kB a 100 IOPS, je výsledná propustnost 800kB/s, tedy 625× nižší než v případě sekvenčních čtení a zápisu.



Obrázek 13.4: Jednotlivé operace disku nutné pro náhodné čtení dat [22].

Problém náhodných přístupů je maskován pomocí vyrovnávací paměti disku (tzv. **diskové cache**), která má kapacitu až jednotky MB. Zejména u velkých souborů (daleko větších než je velikost vyrovnávací paměti) je ale tento mechanismus neúspěšný a databázový systém bude například upřednostňovat sekvenční čtení v poli (např. v tabulce typu halda) před procházením indexem při kterém jsou stránky čteny náhodně (např. při provádění bodového nebo rozsahového dotazu v B-stromu). Tato volba se nám může jevit více paradoxní, když si uvědomíme, že sekvenční čtení může být rychlejší ačkoli načte z disku více dat než čtení náhodné. Pro úplnost dodejme, že podobný problém můžeme vidět i v případě přístupů do hlavní paměti, kde sekvenční čtení je řádově rychlejší než náhodné čtení. V tomto případě se o rychlé čtení stará L2 cache procesoru, do které se ukládá malá část aktuálně používané paměti.

V tabulce **??** vidíme propustnosti některých typů disků[1]: dvou HDD (SATA a SAS) a dvou polovodičových (SSD a SDHC). V případě sekvenčního čtení dosáhl nejvyššího výkonu SSD, přičemž SAS disk se mu ve výkonu čtení přiblížil. Vidíme také, že výkon SDHC je řádově nižší než výkon ostatních disků. V případě náhodného čtení stránek o velikost 512 kB se výkon obou operací snížil jen nepatrně. V případě náhodného přístupu ke stránkám o velikosti 4 kB (což je velikost odpovídající velikosti stránek používaných databázovými systémy), dostáváme radikální pokles výkonu: téměř 100× nižší propustnost u HDD a 10× nižší propustnost v případě SSD a SDHC. Ačkoli vidíme, že SSD do jisté míry eliminuje problém náhodných přístupů, v žádném případě se u něj výkon náhodných

[1] Měřeno aplikací CrystalDiskMark – `http://crystalmark.info/software/CrystalDiskMark/index-e.html`

Tabulka 13.1: Propustnosti operací čtení a zápisu pro některé disky [MB/s, (IOPS)]

Operace	SATA RAID1	SAS RAID10	Samsung SSD 840 Pro Series	SDHC
Sekvenční čtení	159.3	416.8	516.4	12.1
Sekvenční zápis	177.9	249.6	494.3	11.0
Náhodné čtení 512kB	173.5	239.0	465.5	11.8
Náhodné zápis 512kB	213.1	237.3	475.7	1.5
Náhodné čtení 4kB (QD=1)	2.5 (613.3)	4.2 (1 020.8)	30.3 (7 392.4)	3.7 (902.6)
Náhodné zápis 4kB (QD=1)	12.1 (2 946.4)	15.9 (3 868.8)	63.8 (15 574.8)	1.4 (344.3)

přístupů nepřibližujeme výkonu sekvenčních přístupů, jak s oblibou výrobci SSD tvrdí.

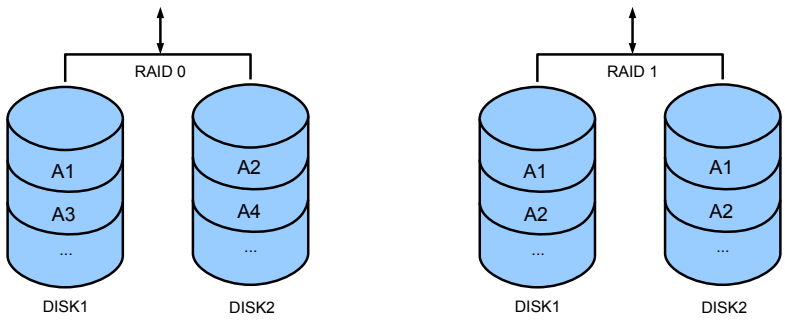
Přestože výkon SSD jednoznačně favorizuje tyto disky pro uložení dat v databázových systémech není tomu vždy tak. Cena SSD je poměrně vysoká a kapacita nízká v porovnání s HDD, z tohoto důvodu je tedy dnes v diskových polích často použita kombinace HD a SD disků.

13.3 RAID

RAID (**Redundant Array of Independent/Inexpensive Disks**) [34] popisuje různé kombinace disků do jedné logické jednotky, která pomocí redundance zabraňuje ztrátě dat. Data jsou distribuovaná/rozdělena na disky různým způsobem, označujeme jako úroveň RAID - **RAID level**. Úroveň RAID volíme dle požadavků na bezpečnost dat, výkon a výslednou kapacitu. Redundance zabraňuje ztrátě dat, ačkoli dojde k poškození jednoho (pro některé úrovně i více) disků. V souvislosti s RAID používáme tyto pojmy:

- tolerance selhání (**fault tolerance**) – o data nepřijdeme i když dojde ke zničení jednoho (v některých případech i více) disků.
- proužkování (**striping**) – data jsou ukládána střídavě na dva nebo více disků,
- zrcadlení (**mirroring**) – data jsou ukládána na dva nebo více disků zároveň.

V dalším textu popíšeme nejznámější úrovně RAID. V případě **RAID 0** se data ukládají postupně v **proužcích** (**stripes**) na jednotlivé disky (viz obrázek 13.5), proto tuto úroveň RAID nazýváme také striping (proužkování). Pokud bude velikost proužku (**stripe size**) 2 kB, pak se 2 kB se uloží na disk 1, další 2 kB na disk 2, atd. Poznamenejme, že velikost proužku je parametrem RAID. **Stripe unit** je pak tvořen všemi proužky v daném pořadí na disku. Pokud je například počet disků 4, jedná se o kilobyty 1-8, 9-16 atd.



Obrázek 13.5: Rozdělení proužků na dva disky v RAID 0 a RAID 1.

Vlastnosti RAID 0 jsou následující:

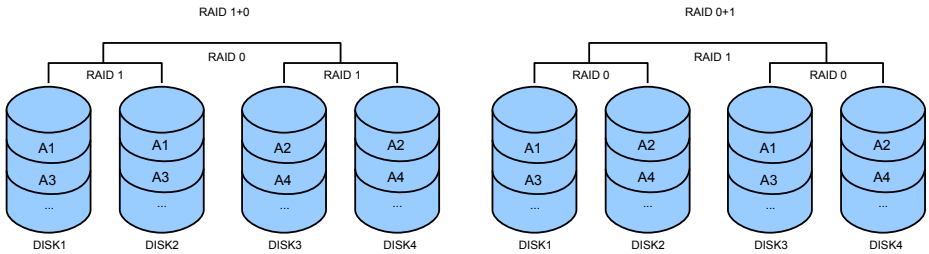
- maximální kapacita – kapacita je daná součtem kapacit disků,
- nulová tolerance selhání – při ztrátě jednoho disku přijdeme o všechna data,
- zvýšený výkon - možné paralelní čtení/zápis proužků v jednotkách.

V případě **RAID 1** se data zapisují na stejné místo dvou disků (viz obrázek 13.5), jedná se o tzv. **zrcadlení dat** (**mirroring**). Jelikož se zápis provádí na oba disky, výkon závisí na výkonu pomalejšího disku. Čtení je realizováno načtením dat z jednoho nebo obou disků v závislosti na parametrech a zaneprázdněnosti disků.

Vlastnosti RAID 1 jsou následující:

- poloviční kapacita při toleranci selhání,
- **multiplexing** – paralelní čtení i zápis na všechny disky v poli.

RAID 1+0 a **RAID 0+1** představuje jednoduché rozšíření RAID 1 pro více než dva disky. Specifikace **RAID10** (někdy také RAID 1+0) kombinuje zrcadlení (RAID 1) s proužkováním (RAID 0), proto také mluvíme o zrcadleném proužkování (mirrored stripping). Proužek dat je zapsán na disk a zároveň zrcadlen na disk jiný. V RAID 0+1 jsou proužky nejprve zapsány na polovinu disků a pak zrcadleny na druhou polovinu disků. Na obrázku 13.6 vidíme konfiguraci 4 disků zapojených v RAID 1+0 a RAID 0+1. Pokud je soubor uložen ve dvou proužcích, pak v případě RAID 1+0 jsou stejné proužky uloženy na discích 1,2 a 3,4, v případě RAID 0+1 jsou uloženy na discích 1,3 a 2,4.



Obrázek 13.6: Rozdělení proužků A1 – A4 v RAID 1+0 a RAID 0+1.

Vlastností obou úrovní RAID je zlepšená propustnost čtení daná proužkováním (pokud řadič umožňuje čtení z více disků, tedy multiplexing) s tolerancí selhání danou zrcadlením. Zápis se provádí na dva disky, výkon diskového pole závisí na výkonu pomalejšího disku (viz RAID1). Nevýhodou této úrovně je podobně jako v případě RAID 1 ztráta poloviny diskové kapacity.

RAID 5 se snaží řešit toleranci selhání jiným způsobem než RAID 1 (a RAID 1+0 či RAID 0+1) u kterého je obětována polovina diskové kapacity. Tolerance selhání je zde řešena korekcí chyby namísto plné redundance použité při zrcadlení. Stejně jako v případě RAID 0 jsou data rozdělena na více disků (striping), na rozdíl od RAID 0, ale každá jednotka (stripe unit) obsahuje tzv. **paritní proužek** (**parit stripe**). Tento paritní proužek se neukládá v každé jednotce na stejný disk, v případě RAID 5 tedy mluvíme o tzv. proužkování s rotací parity.

Pokud uvažujeme RAID 5 s n disky (např. 9), dostaneme $n - 1$ (např. 8) datových proužků a 1 paritní proužek. Datové a paritní proužky jsou v každé jednotce na různých discích. Zápis proužku S_{new} znamená zápis proužku a změnu paritního proužku v dané jednotce na hodnotu P_{new}: $P_{new} = (S_{old}$ XOR $S_{new})$ XOR P_{old}. Zápis jednoho proužku tedy znamená čtení dvou proužků a zápis dvou proužků (porovnejme s RAID1 nebo RAID 1+0 kde zapisujeme, pokud možno paralelně, dva použky). Zejména hardwarové řešení RAID 5 se snaží

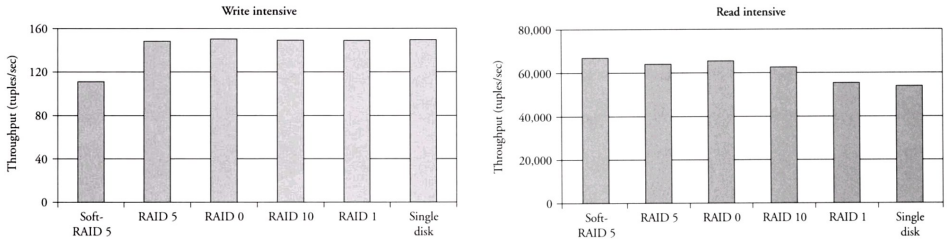
potlačit tento výkonnostní problém. V **RAID 6** jsou použity 2 paritní proužky namísto jednoho. Výsledkem je menší disková kapacita, ale vyšší tolerance selhání než u RAID 5 (je možné selhání dvou disků bez ztráty dat).

V tabulce 13.2 vidíme souhrn popisovaných úrovní RAID. Mezi parametry RAID patří nastavení velikosti proužku (**stripe size**). V případě uložení datových souborů SŘBD v RAID je vhodné volit velikost proužku stejně velkou jako velikost bloku (nejčastěji 8kB). Výkon RAID je definován úrovní, nicméně výsledný výkon je značně závislý na implementaci (softwarové nebo hardwarové). Zásadní například je, zda řadič umožňuje multiplexing – paralelní čtení i zápis na všechny disky.

Tabulka 13.2: Úrovně RAID a tolerovaný výpadek disků

RAID	**Metoda**	**Tolerovaný výpadek disků**
RAID 0	proužkování	0
RAID 1	zrcadlení	1
RAID 5	proužkování s rotací parity	1
RAID 6	proužkování s rotací dvojité parity	2
RAID 1+0, 0+1	kombinace zrcadlení a proužkování	$2 - n/2$

V následující části ukážeme jak volba úrovně RAID ovlivňuje výkon čtení a zápisu. Výsledky z obrázku 13.7 jsou převzaty z knihy [40], pro testy byl použit SQL Server 7 na Windows 2000.



Obrázek 13.7: Testování zápisu a čtení pro jednotlivé úrovně RAID [40].

Negativní vlastnosti většího počtu zápisů se projevují zejména u softwarového RAID 5. Hardwarový RAID 5 řeší tyto problémy pomocí cache. Pro ostatní úrovně RAID dostáváme přibližně stejný výkon zápisu, v případě čtení vidíme znatelně nižší výkon RAID 1, který ovšem překonává bez RAIDové

řešení. RAID 0, RAID 5 a RAID 10 pak zlepšují výkon čtení pomocí distribuce na více disků (striping). Z uvedených výsledků je patrné, že samotná volba RAID není nijak zásadní z pohledu výkonu, je ale zásadní z pohledu tolerance selhání a dostupné diskové kapacity. Pro výkon úložiště je důležitější volba typu disku (SATA, SAS, SSD), jak bylo ukázáno v předchozí kapitole. Pro úplnost dodejme, že test neobsahuje testování náhodného a sekvenčního, synchronního a asynchronního čtení a zápisu atd.

Na základě popsaných vlastností můžeme stanovit vhodnost jednotlivých úrovní RAID pro různé soubory SŘBD:

- Log soubor – využívá především sekvenční synchronní zápis, proto využijeme toleranci chyb RAID 1, RAID 1+0, RAID 0+1 a jejich vyšší propustnosti zápisu (při porovnání s RAID 5).
- Pomocné soubory – použijeme RAID 0 jelikož tolerujeme ztrátu dat.
- Datové a indexové soubory – v tomto případě převažuje čtení nad zápisem, RAID 5 je tedy postačující. Pokud máme dostatek disků, použijeme RAID 1+0 (nebo 0+1).

Část V

Řízení souběhu

Kapitola 14

Transakce a zotavení z chyb

Obsah

14.1 Úvod

Ačkoli víceuživatelský přístup k SŘBD a zotavení spolu souvisí a sdílí společné koncepty, v této knize jsme je rozdělili do dvou kapitol především z důvodu jasnějšího výkladu. V této kapitole budeme popisovat základní problémy zotavení SŘBD z chyb a v kapitole 15 se budeme zabývat problémy víceuživatelského přístupu k SŘBD. Oba problémy se týkají velkých SŘBD, malé SŘBD často zotavení a víceuživatelský přístup vůbec nepodporují. Mohlo by se tedy zdát, že se

jedná o jakousi nadstavbu databázového systému, kterou není nutné používat. Opak je ovšem pravdou; to že některé SŘBD transakce a zotavení nepodporují, plyne jednoduše z toho, že ignorují problémy vzniklé chybami a víceuživatelským přístupem. Často se jedná o pragmatický přístup k implementaci; pokud výrobce nepředpokládá práci více uživatelů, je zbytečné řešit problémy vzniklé ve víceuživatelském provozu.

Zotavení (angl. **recovery**) [11] znamená zotavení databáze z nějaké chyby. Výsledkem zotavení musí být korektní stav databáze a k dosažení tohoto stavu často využíváme nějaké redundantní informace. Pozorný čtenář si jistě pamatuje, že redundanci je nutné se vyhnout při návrhu databáze. V tomto případě se ale jedná o redundanci, která není propagována do logické vrstvy a pro uživatele je tedy za normálních okolností skryta. Při zotavení pak SŘBD využívá tuto redundantní informaci pro rekonstrukci databáze v nekorektním stavu. Ačkoli většina prací, které se zabývají zotavením z chyb, popisuje zotavení v relačních ŠRBD, tyto principy jsou platné pro SŘBD s různými datovými modely.

14.2 Transakce

Transakce je logická (nedělitelná, atomická) jednotka práce s databází, která začíná operací BEGIN TRANSACTION a končí operacemi COMMIT nebo ROLLBACK.

Uvažujme příklad transakce z výpisu 14.1 jejíž úkolem je převést 100Kč z účtu číslo 345 na účet číslo 789. Je zřejmé, že převod musí být proveden jako jedna nedělitelná operace, ačkoli se jedná o dvě operace `UPDATE`. V tomto případě je databáze po provedení první operace `UPDATE` v nekorektním stavu – databáze nereflektuje stav reálného světa – 100Kč zmizelo. Obecně tedy transakce jako logická jednotka práce s databází nezahrnuje jednu databázovou operaci, ale častěji posloupnost operací. Úkolem transakce je převést korektní stav databáze na jiný korektní stav, přičemž nemusí zanechat databázi v korektním stavu po jednotlivých operacích transakce. Abychom splnili podmínku korektnosti, musíme v tomto případě provést obě nebo žádnou operaci `UPDATE`.

```
BEGIN TRANSACTION;

try
{
  UPDATE Account 345 { balance -= 100; }
  UPDATE Account 789 { balance += 100; }
  COMMIT;
```

```
}
catch(SqlException)
{
  ROLLBACK;
}
```

Výpis 14.1: Příklad transakce popsané pseudokódem.

Otázkou zůstává z jakého důvodu může dojít k chybě při provádění transakcí. Jako příklady uveďme přetečení/podtečení hodnoty atributu zadané uživatelem, syntaktické chyby v SQL příkazu, pád ŠRBD nebo operačního systému, nedostatek místa na disku apod. Při jakékoli chybě musí být databáze v korektním stavu; nemůže se tedy stát, aby po zotavení z chyby, databáze obsahovala jen část aktualizací provedených v rámci nějaké transakce. Jako motivaci prezentujeme tabulku uvedenou v článku [18] z roku 1983, která ukazuje frekvence výskytů a čas zotavení pro základní typy chyb, které se vyskytují ve velkých databázových systémech:

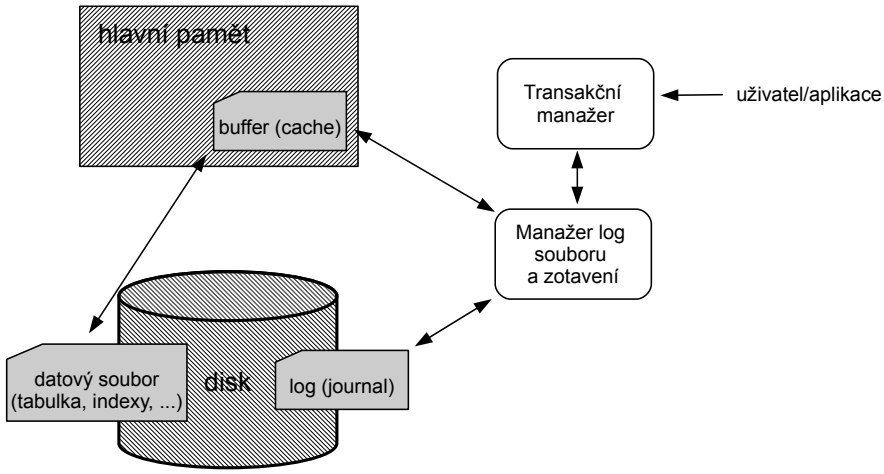
Typ chyby	Popis chyby	Frekvence výskytu	Čas zotavení
Lokální	Přetečení hodnoty atributu zadané uživatelem, syntaktická chyba v SQL příkazu	10-100/min	Stejný jako je čas provedení transakce
Systémová	Pád systému	Několik za týden	Několik minut
Chyba média	Chyba disku	1-2/rok	1-2 hodiny

Komponenta SŘBD, která se stará o řízení transakcí se nazývá **manager transakcí** (angl. **transaction management**) nebo **monitor transakčního zpracování** (angl. **transaction processing monitor**) – viz kapitola **??**. Programátor pak transakce řídí pomocí operací:

- COMMIT – signalizuje úspěšné ukončení transakce. Programátor oznamuje transakčnímu manageru, že transakce byla úspěšně dokončena, databáze je nyní v korektním stavu, a všechny změny provedené v rámci transakce mohou být trvale uloženy v databázi. Jinými slovy všechny změny jsou **potvrzeny** (angl. **comitted**) pro trvalé uložení v databázi.
- ROLLBACK – signalizuje neúspěšné provedení transakce. Programátor oznamuje transakčnímu manageru, že databáze může být v nekorektním stavu a všechny změny provedené v rámci transakce musí být **zrušeny** (angl. **roll back** nebo **undo**).

V případě transakce z výpisu 14.1 jsou operací COMMIT zaznamenány nové částky na obou účtech; v případě nějaké chyby, operace ROLLBACK nastaví částky na obou účtech na hodnoty platné před začátkem transakce.

Čtenáři se může zdát podivné, že operace ROLLBACK umožňuje zrušit všechny změny provedené v rámci transakce a možná se ptát zda se nejedná o nějaká kouzla. Musíme si uvědomit, že programátor připojením k SŘBD nepracuje přímo s datovým souborem a tudíž jím provedené změny se neprovádí v tomto datovém souboru. SŘBD se skládá z celé řady komponent (viz kapitola **??**) a programátor, naštěstí, nemůže přímo pracovat s fyzickou reprezentací dat. Pro podporu operace ROLLBACK má systém k dispozici **log** nebo **journal** na disku kde jsou zaznamenány detaily o všech provedených operacích (především o hodnotách objektů před a po provedení operace). V případě ROLLBACK operace je systém na základě záznamu v logu schopen vrátit hodnoty příslušného záznamu na původní hodnoty.



Obrázek 14.1: Architektura manažeru zotavení.

Nyní si uveďme několik poznámek k transakcím:

- Transakce nemůže být uvnitř jiné transakce, tedy BEGIN TRANSACTION se nemůže vyskytovat bez toho aby předchozí transakce byla ukončena operací COMMIT nebo ROLLBACK. Tato vlastnost plyne z nedělitelnosti transakce.
- Položme si nyní otázku proč mluvíme v případě transakcí o korektním stavu databáze, proč nemluvíme o konzistentním stavu databáze? Pojem konzistentní databáze znamená přesně to, že v databázi neexistují žádné

výjimky z daných integritních omezení. Vezměme v úvahu transakci z výpisu 14.1, zde bezpochyby nejsme schopni žádné integritní omezení definovat. Z tohoto důvodu tedy říkáme, že transakce je posloupnost operací převádějící databázi v korektním stavu do jiného korektního stavu. Pojem korektní má tedy ten význam, že respektujeme výsledek operací, který jsou vykonány v reálném světě. V případě převodu peněz z jednoho účtu na druhý, musí být součet sum na obou účtech stejný. Integritní omezení nejsme tedy schopni definovat: obecně toto tvrzení neplatí, např. když chci převést peníze z prvního účtu na účet třetí.

14.3 Zotavení transakce

Jak bylo řečeno v předchozí kapitole, transakce začíná vykonáním operace BEGIN TRANSACTION a končí vykonáním COMMIT nebo ROLLBACK. Operace COMMIT zavádí tzv. **potvrzovací bod** (angl. **commit point**, u starších systémů mluvíme o **synchpoint**, tedy o **synchronizačním bodu**). Potvrzovací bod odpovídá úspěšnému ukončení logické jednotky práce s databází a představuje tedy bod ve kterém je databáze v korektním stavu. Naproti tomu ROLLBACK vrací databázi do stavu ve kterém byla při vykonání BEGIN TRANSACTION, jinými slovy vrací databázi k předchozímu potvrzovacímu bodu.

V okamžiku zavedení potvrzovacího bodu jsou:

- všechny změny provedené v rámci transakce trvale uloženy v databázi,
- všechny adresace (např. ty nastavené pomocí kurzorů) a zámky entic uvolněny (viz kapitola 15.4).

Z předchozího výkladu je zřejmé, že transakce není jen jednotkou práce s databází, ale rovněž jednotkou zotavení: pokud systém zhavaruje, uživatel musí mít k dispozici databázi, která bude obsahovat všechny potvrzené změny. Nyní popíšeme několik implementačních detailů souvisejících z transakcemi. Z důvodu efektivity používá SŘBD **vyrovnávací paměť** umístěnou v hlavní paměti (angl. **cache** či **buffer**) obsahující aktuální záznamy z databáze. Může se tedy stát, že potvrzené záznamy nebyly před pádem systému fyzicky zapsány na disk. I v tomto případě musí systém provést zotavení a databáze bude obsahovat všechny potvrzené záznamy. Aby byl systém schopen zaručit takovéto chování, všechny změny jsou zapsány do logu před samotným zápisem změn do databáze. Před ukončením vykonávání operace COMMIT je do logu zapsán tzv. COMMIT záznam. Pravidlo které stanovuje, že zápis do logu musí být proveden před samotným zápisem záznamu na disk nazýváme **pravidlo dopředného**

zápisu do logu (angl. **write-ahead log rule**). Systém je pak schopen na základě informací z logu provést zotavení databáze.

Vlastnost ACID

Každá transakce musí splňovat vlastnost ACID: **atomičnost** (angl. **atomicity**), **korektnost** (angl. **correctness**), **izolovanost** (angl. **isolation**) a **trvalost** (angl. **durability**):

- **A - Atomičnost** – transakce musí být atomická: jsou provedeny všechny operace transakce nebo žádná.
- **C - Korektnost** – transakce převádí korektní stav databáze do jiného korektního stavu databáze, mezi začátkem a koncem transakce nemusí být databáze v korektním stavu.
- **I - Izolovanost** – transakce jsou navzájem izolovány: změny provedené jednou transakcí jsou pro ostatní transakce viditelné až po provedení COMMIT.
- **D - Trvalost** – jakmile je transakce potvrzena, změny v databázi se stávají trvalými i po případném pádu systému.

14.4 Zotavení systému

Zotavení není vázáno pouze na jednu transakci, ale i na celý databázový systém. Kromě **lokální chyb** popisovaných v předchozích kapitolách, mohou nastal při provozu databázového systému i tzv. **chyby globální**. Nejčastěji tyto chyby rozdělujeme na **chyby systémové** (někdy také **soft crash**, např. výpadek proudu) a **chyby média** (někdy také **hard crash**). V obou případech chyba ovlivňuje všechny transakce ve kterých se vyskytla, na rozdíl od chyb lokálních, které se týkají pouze transakce jedné. V případě chyby média ovšem navíc dochází ke zničení části databáze.

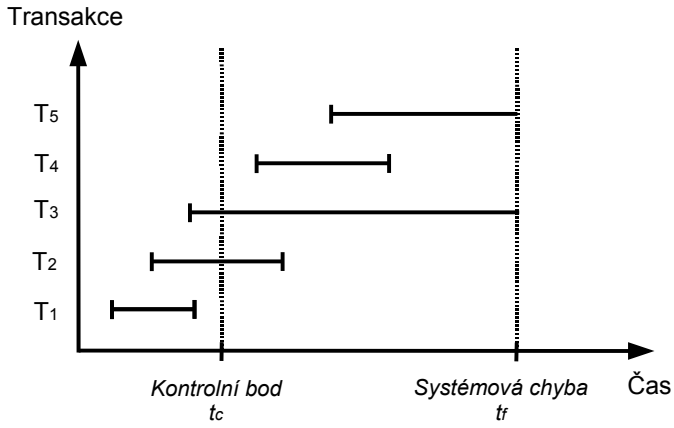
14.4.1 Zotavení po systémové chybě

Základním problémem vzniklým při systémové chybě, je ztráta obsahu hlavní paměti, tedy ztráta obsahu vyrovnávací paměti SŘBD. Přesný stav transakce

přerušené chybou není tedy znám a transakce musí být **zrušena** (angl. **undo**) – pro tuto akci budeme často používat označení UNDO. V některých případech je transakce úspěšně ukončena, ovšem změny nejsou přeneseny z vyrovnávací paměti na disk. V tomto případě musí být po restartu systému transakce **přepracována** (angl. **redo**) – pro tuto akci budeme často používat označení REDO. Aby SŘBD věděl, které operace musí být pro danou transakci provedeny, vytváří tzv. **kontrolní body** (angl. **check point**). Kontrolní body jsou vytvářeny např. po určitém počtu záznamů, které byly zapsány do logu a zahrnují:

- zápis obsahu vyrovnávací paměti na disk,
- zápis záznamu o kontrolním bodu do logu.

Záznam o kontrolním bodu zahrnuje všechny transakce vykonávané v době vytvoření kontrolního bodu. Na následujícím příkladu si ukážeme jak jsou v případě zotavení tyto záznamy využity.



Obrázek 14.2: Pět kategorií transakcí vzniklých při systémové chybě.

Z obrázku 14.2 můžeme číst:

- Systémová chyba nastala v čase t_f.
- Kontrolní bod t_c byl vytvořen ze všech kontrolních bodů nejpozději před tím než nastala systémová chyba.

- Transakce typu T_1 byla úspěšně dokončena před časem t_c.
- Transakce typu T_2 začala před časem t_c, byla úspěšně dokončena po t_c a před t_f.
- Transakce typu T_3 začala před časem t_c nebyla ale dokončena před t_f.
- Transakce typu T_4 začala po t_c a byla dokončena před t_f.
- Transakce typu T_5 začala po t_c, ale nebyla dokončena před t_f.

Je zřejmé, že po restartu systému musí být transakce typu T_3 a T_5 zrušeny a transakce typu T_2 a T_4 musí být přepracovány. Jelikož změny provedené transakcí T_1 byly provedeny před kontrolním bodem t_c, tuto transakci při zotavení vůbec neuvažujeme. Po restartu systém tedy aplikuje tento algoritmus:

1. Vytvoř dva seznamy transakcí: UNDO a REDO.
2. Do UNDO vlož všechny transakce, které nebyly úspěšně dokončeny před posledním kontrolním bodem. Seznam REDO je prázdný.
3. Začni procházet záznamy v logu od záznamu posledního kontrolního bodu.
   (a) Pokud je pro transakci T nalezen v logu záznam BEGIN TRANSACTION, ponech T v seznamu UNDO.
   (b) Pokud je pro transakci T nalezen v logu záznam COMMIT, přesuň T ze seznamu UNDO do seznamu REDO.

Po skončení algoritmu obsahuje seznam UNDO transakce T_3 a T_5 a seznam REDO transakce T_2 a T_4. Nyní systém prochází log zpětně a ruší transakce ze seznamu UNDO (tedy jejich jednotlivé operace), poté prochází logem dopředu a přepracovává transakce ze seznamu REDO. Po zpracování obou seznamů je zotavení ukončeno a systém je připraven k dalšímu provozu.

14.4.2 Zotavení po chybě média

Zotavení v případě chyby média začíná obnovením databáze ze záložní kopie popř. **dump** souboru[1]. V dalším kroku je procházen log a všechny transakce, které byly dokončeny po čase vytvoření zálohy jsou přepracovány. V tomto případě nejsou žádné transakce zrušeny: aktualizace byly zrušeny prostou ztrátou dat.

[1] Tento soubor obsahuje úplný obraz databáze a je vytvářen z databáze popř. využit pro rekonstrukci databáze nejčastěji utilitami **dump**/**restore** nebo **unload**/**reload**.

14.5 Základní techniky zotavení

V předchozích kapitolách jsem popsali obecné koncepty zotavení, nyní si představíme dvě základní techniky zotavení: zotavení odloženou aktualizací a zotavení okamžitou aktualizací [14].

14.5.1 Zotavení odloženou aktualizací

Zotavení odloženou aktualizací (angl. **deferred update**) neprovádí aktualizaci databáze na disku dokud transakce nedosáhne potvrzovacího bodu. Před dosažením potvrzovacího bodu, všechny aktualizace databáze jsou zaznamenány do lokálního bufferu. Jakmile transakce dosáhne potvrzovacího bodu, aktualizace jsou nejprve zaznamenány do logu a následně do databáze. Vidíme tedy aplikaci pravidla dopředného zápisu do logu. Je zřejmé, že pokud transakce selže před dosažení potvrzovacího bodu, pak není nutné provést UNDO, protože databáze nebyla aktualizovaná. REDO bude provedeno v případě kdy systém zapsal aktualizace do logu, k zapsání změn do databáze ale nedošlo. Tato technika zotavení se tedy nazývá **NO-UNDO/REDO algoritmus**.

Do logu jsou v tomto případě zapsány nové hodnoty, což umožní systému při zotavení realizovat operaci REDO.

Ačkoli se tato technika zdá výkoná (zejména z pohledu minimalizace I/O), v praxi se používá pouze v případě, kdy systém provádí krátké transakce a každá transakce mění pouze několik málo položek. Pro ostatní typy transakcí hrozí přetečení popř. velká expanze lokálních bufferů, které obsahují operace jednotlivých transakcí.

14.5.2 Zotavení okamžitou aktualizací

Zotavení okamžitou aktualizací (angl. **immediate update**) může aktualizovat databázi na disku před tím než transakce dosáhne potvrzovací bod. Operace jsou v tomto případě zapsány do logu vynuceným zápisem a poté je aktualizována databáze. Opět tedy vidíme aplikaci pravidla dopředného zápisu do logu. Pokud transakce selže před dosažením potvrzovacího bodu, a během provádění transakce došlo k aktualizaci databáze, pak je nutné provést UNDO. Je zřejmě, že v tomto případě budou pro zotavení aplikovány jak operace UNDO tak operace REDO, proto tuto techniku nazýváme **UNDO/REDO algoritmus**. Ve speciálním případě, kdy jsou všechny aktualizace transakce zapsány do databáze před dosažením

potvrzovacího bodu (a při zotavení tedy odpadá REDO), pak mluvíme o **UNDO/NO-REDO algoritmu**.

Na rozdíl od odložené aktualizace, kde ukládáme do logu nové hodnoty, v případě okamžité aktualizace ukládáme do logu původní hodnoty. Uložení původních hodnot umožní systému provést při zotavení operaci UNDO.

14.6 Záchranné body (savepoints)

Ačkoli jsem si v předchozích kapitolách popsali transakci jako nedělitelnou jednotku práce s databází, přece jen existuje koncept, který transakci rozděluje na menší části. Tento koncept se nazývá **záchranné body** (angl. **savepoints**) a byl zaveden v SQL99 [3]. V případě operace ROLLBACK nedochází ke zrušení celé transakce, ale k návratu na záchranný bod. Musíme si uvědomit, že záchranný bod není ekvivalentní potvrzení změn: změny provedené transakcí nejsou stále viditelné pro ostatní transakce dokud ta neprovede operaci COMMIT.

14.7 Transakce v SQL

Nyní popíšeme podporu transakcí v SQL. Nejprve, všechny SQL příkazy jsou atomické (s výjimkou příkazů CALL a RETURN). Operace BEGIN TRANSACTION se v SQL provede příkazem `START TRANSACTION`, operace COMMIT příkazem `COMMIT WORK` a operace ROLLBACK příkazem `ROLLBACK WORK`. Syntaxe příkazu `START TRANSACTION` je následující:

```
START TRANSACTION <volitelné parametry>;
```

Kde <**volitelné parametry**> specifikují **režim přístupu** (angl. **access mode**), **úroveň izolace** (angl. **isolation level**) a **velikost diagnostikované oblasti** (angl. **diagnostics area size**):

- Úroveň izolace zadáváme ve tvaru `ISOLATION LEVEL` <**izolace**>, kde <**izolace**> je `READ UNCOMMITED`, `READ COMMITED`, `REPEATABLE READ` a `SERIALIZABLE`. Jelikož úroveň izolace úzce souvisí s víceuživatelským přístupem do databáze, podrobnosti najdete v následující kapitole 15.
- Režim přístupu může být `READ ONLY` nebo `READ WRITE`. Pokud nespecifikujeme žádný režim, je nastaven `READ WRITE`. Pokud ovšem zvolíme úroveň izolace `READ UNCOMMITED`, pak je nastaven režim `READ ONLY`.

Pokud tedy zvolíme režim `READ WRITE`, pak úroveň izolace nesmí být `READ UNCOMMITED`.

- Velikost diagnostikované oblasti se nastavuje jako celé číslo uvedené za klíčovým slovem `DIAGNOSTICS SIZE` a specifikuje kolik výjimek bude systém ukládat na zásobník.

Syntaxe COMMIT a ROLLBACK je následující:

```
COMMIT [WORK] [AND [NO] CHAIN];

ROLLBACK [WORK] [AND [NO] CHAIN];
```

Vidíme tedy, že `WORK` je doplňkové slovo a implicitní volba je `AND NO CHAIN`. `AND CHAIN` automaticky vykoná `START TRANSACTION` se stejnými parametry jako v předchozím případě po provedení `COMMIT`. Po potvrzení transakce, jsou automaticky uzavřeny všechny kurzory, je tedy proveden `CLOSE` (viz kapitola 18.5.4), s jedinou výjimkou kurzoru deklarovaného jako `WITH HOLD`.

Nyní popíšeme práce se záchrannými body v SQL. Záchranný bod je vytvořen příkazem: `SAVEPOINT` <**`jméno záchranného bodu`**>`;`

Příkazem `ROLLBACK TO` <**`jméno záchranného bodu`**>`;` provedeme zrušení všech operací provedených za specifikovaným záchranným bodem. Příkaz `RELEASE` <**`jméno záchranného bodu`**>`;` zruší specifikovaný záchranný bod, po tomto příkazu tedy nemůže provést ROLLBACK k tomuto bodu. Po ukončení transakce jsou automaticky zrušeny všechny záchranné body.

Příklad 14.1. (Transakce v SQL).

Vezměme v úvahu tabulku `Student`:

```
CREATE TABLE Student (
  login CHAR(5) PRIMARY KEY,
  fname VARCHAR(30) NOT NULL,
  lname VARCHAR(30) NOT NULL,
  email VARCHAR(40) NOT NULL
);
```

Nyní chceme vložit do databáze záznamy dvou studentů, v případě chyby nebude vložen ani jeden záznam. Jelikož potřebujeme zachytit výjimky signalizující neprovedený příkaz, musíme použít nějakou procedurální nadstavbu nad SQL nebo hostitelský programovací jazyk, v tomto případě použijeme PL/SQL SŘBD Oracle (viz kapitola 18.5).

```
BEGIN
  INSERT INTO Student VALUES('sob28', 'Jan', 'Sobota',
    'jan.sobota@vsb.cz');
  INSERT INTO Student VALUES('sob28', 'Jan', 'Neděle',
    'jan.nedele@vsb.cz');
  COMMIT;
EXCEPTION
  WHEN OTHERS THEN
    ROLLBACK;
END;
```

V tomto případě se nepodaří vložit druhý záznam (omylem jsme jako login použili login prvního studenta, ten má být ovšem jedinečný), celá transakce bude tedy zrušena.

Příklad 14.2. (Transakce v SQL).

Vezměme v úvahu část schématu systému evidujícího autory a recenzenty článků. V takovém systému může být jedna osoba v celé řadě rolí (autor, recenzent, administrátor apod.). Vezměme v úvahu tabulku `Person`:

```
CREATE TABLE Person (
  login         VARCHAR(20) PRIMARY KEY,
  email         VARCHAR(50) UNIQUE NOT NULL,
  password      VARCHAR(20) NOT NULL,
  firstName     VARCHAR(20) NOT NULL,
  middleName    VARCHAR(20),
  secondName    VARCHAR(20) NOT NULL,
  email2        VARCHAR(50),
  web           VARCHAR(70));
```

, tabulku `Role`:

```
CREATE TABLE Role (
  id            INT NOT NULL PRIMARY KEY ,
  name          VARCHAR(50) NOT NULL UNIQUE);
```

a tabulku evidující role osob:

```
CREATE TABLE PersonRole (
  idPerson      VARCHAR(20) REFERENCES Person NOT NULL,
  idRole        INT REFERENCES Role NOT NULL,
  UNIQUE(idPerson, idRole));
```

Při vložení nové osoby chceme zároveň přidat této osobě roli "Autor"(pro jednoduchost stanovme pro tuto roli id=1[2]). Transakce tedy bude vypadat takto:

```
BEGIN
  INSERT INTO Person VALUES('sob28', 'jan.sobota@vsb.cz', 'heslo',
      'Jan', NULL, 'Sobota', NULL, NULL);
  INSERT INTO PersonRole VALUES('sob28', 1);
  COMMIT;
EXCEPTION
  WHEN OTHERS THEN
    ROLLBACK;
END;
```

Pokud selže vložení osoby (např. osoba je už v systému evidována), pak vložení role osoby nedává smysl a celá transakce bude zrušena. Pro úplnost dodejme, že v reálném případě bychom namísto konstant u příkazu INSERT použili proměnné obsahující data získaná např. z webových formulářů.

[2]Záznam musí být uložen v tabulce `Role`.

Kapitola 15

Víceuživatelský přístup k databázi

Obsah

15.1 Úvod

Jedním z kritérií jak klasifikovat databázové systémy je počet uživatelů, kteří mohou používat tento systém současně [14]. SŘBD je **jednouživatelský** pokud

jej může využívat pouze jeden uživatel v daném čase. SŘBD který může být využíván více uživateli současně se nazývá **víceuživatelský**. Pokud k ŠRBD přistupuje více uživatelů současně, mluvíme o **souběhu** (angl. **concurency**). Jinými slovy, souběh umožňuje SŘBD zpřístupnit databázi mnoha transakcím ve stejném čase [11]. Souběh přináší celou řadu problémů, které musí řešit i programátor na aplikační úrovni (ačkoli se tak často neděje).

Stejně jako v případě zotavení (viz kapitole 14) poznamenejme, že koncepce souběhu jsou nezávislé na datové modelu, byť největší počet článků řeší souběh tohoto nejpoužívanějšího datového modelu. Problematika souběhu je velmi komplikovaná a je rozhodně nad rámec této knihy popsat ji úplně. Tato kapitola se pokusí objasnit čtenáři základní problémy souběhu a jejich řešení.

Pro demonstraci problémů souběhu a jejich řešení budeme používat plány provádění transakcí. **Plán** je posloupnost operací transakce, pro zjednodušení budeme používat pouze dvě operace:

- READ namísto SELECT,
- WRITE namísto UPDATE, nicméně tato operace může zahrnovat i INSERT a DELETE.

Pro zjednodušení budeme také pracovat pouze s enticí, tedy záznamem, tedy prvkem relace. Pokud jsou plány provedeny souběžně mluvíme o **plánu souběžném** nebo také o **plánu paralelním**.

15.2 Problémy nastávající při souběhu

Nyní popišme tři problémy, které mohou nastat při souběhu:

- **ztráta aktualizace** (angl. **lost update**),
- **nepotvrzená závislost** (angl. **uncommitted dependency**),
- **nekonzistentní analýza** (angl. **inconsistent analysis**).

Nyní tyto problémy podrobně popíšeme.

15.2.1 Problém ztráty aktualizace

Uvažujme situaci na obrázku 15.1. Transakce A získá v čase t_1 z databáze (např. příkazem SELECT) entici t [1]. Transakce B získá stejnou entici v čase t_2. Transakce A změní hodnotu entice v čase t_3 (např. příkazem UPDATE), ovšem na základě hodnoty v čase t_1. Nakonec transakce B aktualizuje entici t v čase t_4. Jelikož transakce B provedla aktualizaci pro hodnotu v čase t_2 (která je stejná jako hodnota v čase t_1), dojde ke ztrátě aktualizace provedené transakcí A v čase t_3.

Transakce A	**Čas**	**Transakce B**
`READ` t	t_1	-
-	t_2	`READ` t
`WRITE` t	t_3	-
-	t_4	`WRITE` t

Obrázek 15.1: Aktualizace transakce A je ztracena v čase t_4.

15.2.2 Problém nepotvrzené závislosti

Problém nepotvrzené závislosti nastane v případě kdy jedna transakce načte nebo v horším případě aktualizuje entici, která byla aktualizována dosud nepotvrzenou transakcí. Jelikož tato transakce nebyla potvrzena, existuje možnost, že potvrzena nebude a naopak transakce bude zrušena operací ROLLBACK. V tomto případě ale první transakce pracuje s hodnotami které nejsou platné.

Na obrázku 15.2 vidíme, že transakce A načetla v čase t_2 nepotvrzenou aktualizaci entice t provedenou transakcí B. V čase t_3 je transakce B zrušena. Transakce A pak pracuje s chybnými hodnotami entice t, tedy s hodnotami získanými v čase t_2, ačkoli platné hodnoty jsou hodnoty z času před t_1.

Na příkladu z obrázku 15.3 můžeme dokonce vidět ztrátu aktualizované hodnoty. Transakce A se stala závislou na nepotvrzené změně z času t_1, navíc došlo ke ztrátě aktualizace provedené v čase t_2: ROLLBACK provedený v čase t_3 zapříčiní nastavení entice na hodnoty z času před t_1.

[1] Dle definice je **entice** (angl. **tuple**) prvkem relace, slangově ovšem můžeme mluvit o záznamu jako jednom řádku tabulky.

Transakce A	Čas	Transakce B
-	t_1	`WRITE` t
`READ` t	t_2	-
-	t_3	`ROLLBACK`

Obrázek 15.2: Transakce A se stala v čase t_2 závislou na nepotvrzené změně transakce B.

Transakce A	Čas	Transakce B
-	t_1	`WRITE` t
`WRITE` t	t_2	-
-	t_3	`ROLLBACK`

Obrázek 15.3: Transakce A aktualizovala v čase t_2 nepotvrzenou změnu provedenou transakcí B v čase t_1.

15.2.3 Problém nekonzistentní analýzy

Na obrázku 15.4 vidíme dvě transakce A a B, které pracují s účty acc_1, acc_2 a acc_3. Před časem t_1 mají účty hodnotu 30, 20 a 50. Transakce A počítá součet konečných zůstatků na účtech, transakce B převádí částku 10 z účtu 1 na účet 3. V tomto případě má transakce A k dispozici nekonzistentní databázi a proto vykoná nekonzistentní analýzu (spočítá nekorektní součet 110 namísto hodnoty 100). Pokud porovnáme tento problém s předchozím problémem nepotvrzené závislosti, vidíme, že transakce A není závislá na nepotvrzených změnách transakce B, protože B potvrdí všechny aktualizace předtím než si A vyžádá acc_3.

15.2.4 Podrobný pohled – konflikty čtení/zápisu

Jak je patrno z předchozí kapitoly, z pohledu souběhu identifikujeme dvě databázové operace: čtení (získání entice, angl. **read**) a zápis (aktualizace entice, angl. **write**). Problém souběhu nastává pokud dvě transakce A a B chtějí číst nebo zapisovat stejnou entici t, existují čtyři možnosti:

- **RR** (tedy READ-READ): A a B chtějí číst t. Dvě operace čtení se nemohou negativně ovlivňovat, v tomto případě tedy nenastává žádný problém.

$acc_1 = 30$	$acc_2 = 20$	$acc_1 = 50$
Transakce A	**Čas**	**Transakce B**
READ acc_1 $suma = 30$	t_1	-
READ acc_2 $suma = 50$	t_2	-
-	t_3	READ acc_3
-	t_4	WRITE $acc_3 = 60$
-	t_5	READ acc_1
-	t_6	WRITE $acc_1 = 20$
-	t_7	COMMIT
READ acc_3 $suma = 110$ ne 100	t_8	-

Obrázek 15.4: Transakce A provedla nekonzistentní analýzu.

- **RW**: A čte t a B chce zapsat t. Pokud B vykoná tuto aktualizaci, pak může nastat problém nekonzistentní analýzy prezentovaný na obrázku 15.4. Říkáme tedy, že problém nekonzistentní analýzy je zapříčiněn RW konfliktem. Pokud B vykoná aktualizaci a A načte t znovu, pak A získá odlišné hodnoty. Tomuto jevu říkáme neopakovatelné čtení (angl. **nonrepeatable read**), který je tedy opět zapříčiněn RW konfliktem.

- **WR**: A zapíše t a B pak chce číst t. Pokud B vykoná tuto operaci, pak může nastat problém nepotvrzené závislosti tak jak je prezentován na obrázku 15.2 (pouze byly zaměněny role transakcí A a B). Říkáme tedy, že problém nepotvrzené závislosti je zapříčiněn WR konfliktem. Pokud B přečte t, mluvíme o tzv. **špinavém čtení** (angl. **dirty read**).

- **WW**: A zapíše t a pak B chce zapsat t. Pokud B vykoná tuto aktualizaci, pak může nastat jak problém ztráty aktualizace (viz obrázek 15.1) tak problém nepotvrzené závislosti (viz obrázek 15.3). Říkáme tedy, že problém nepotvrzené ztráty aktualizace je zapříčiněn WW konfliktem. Pokud B zapíše t, mluvíme o tzv. **špinavém zápisu** (angl. **dirty write**).

15.3 Techniky řízení souběhu

V minulosti byla vyvinuta celá řada technik pro řízení souběhu, které se snaží řešit výše zmíněné problémy[2]. Mezi nejznámější techniky patří:

- uzamykání (angl. **locking**) – používá většina DBMS,
- časová razítka (angl. **timestamps**),
- verzování (angl. **multiversion**),
- validace.

Jelikož uzamykání je nasazeno ve většině SŘBD, v následující kapitole bude tato technika podrobně popsána.

15.4 Uzamykání

Uzamykání (angl. **locking**) je jedna z technik řízení souběhu pracující na následujícím principu: pokud transakce A chce provést čtení nebo zápis nějakého objektu v databázi (nejčastěji tedy entice), pak požádá o zámek na tento objekt. Význam zámků spočívá v tom, že žádná jiná paralelní transakce nemůže získat zámek a nemůže tedy provést čtení či aktualizaci (které by způsobilo konflikt) do té doby než A tento zámek uvolní. Rozlišujeme dva typy zámků:

1. **Výlučný zámek** (také zámek pro zápis, angl. **exclusive lock** nebo **write lock**), označujeme X.
2. **Sdílený zámek** (také zámek pro čtení, angl. **shared lock** nebo **read lock**), označujeme S.

Ve skutečnosti existuje více typů zámků (jak uvedeme později), pro zjednodušení dalšího výkladu budeme ale používat pouze tyto dva typy.

Při uzamykání mohou nastat tyto možnosti:

1. Pokud transakce A drží výlučný (X) zámek na entici t, pak požadavek paralelní transakce B na zámek libovolného typu na entici t není proveden okamžitě.

[2] Později budeme říkat, že tyto techniky většinou garantují tzv. serializovatelnost transakcí.

2. Pokud transakce A drží sdílený (S) zámek na entici t, pak:
   (a) požadavek paralelní transakce B na zámek X na entici t není proveden okamžitě,
   (b) požadavek paralelní transakce B na zámek S na entici t je proveden okamžitě. V tomto případě B bude rovněž držet zámek S na t.

	X	S	-
X	N	N	A
S	N	A	A
-	A	A	A

Obrázek 15.5: Matice kompatibility pro typy zámků X a S.

Tato pravidla můžeme popsat **maticí kompatibility typů zámků** (angl. **lock type compatibility matrix**), kterou vidíme na obrázku 15.5. Matice je interpretována takto. Uvažujme entici t a transakci A, která drží zámek uvedený v hlavičce tabulky. Symbol - označuje stav, kdy A nedrží žádný zámek na t. Paralelní transakce B požaduje na t zámek uvedený v prvním sloupci tabulky. Symbol N značí konflikt, tedy že zámek nebude přiřazen okamžitě, symbol A značí kompatibilitu, tedy že zámek bude přiřazen okamžitě. Všimněme si, že matice je symetrická.

Musíme si uvědomit, že zámky jsou přidělovány implicitně: pokud transakce získá entici z databáze (např. příkazem `SELECT`), pak je automaticky požadováno přidělení zámku S; pokud transakce aktualizuje entici, pak je automaticky požadováno přidělení zámku X. Aktualizace záznamu nemusí být provedena pouze příkazem `UPDATE`, ale chápeme ji jako operaci, která mění obsah databáze, může být tedy provedena i příkazy `INSERT` a `DELETE`. Řízení transakcí pro `INSERT` a `DELETE` obsahuje jisté odlišnosti, s určitou mírou zjednodušení, ale můžeme mluvit o jedné operaci aktualizace.

Nyní popišme protokol přístupu k datům (nebo také **uzamykací protokol**, angl. **data access protocol** resp. **locking protocol**), který umožní řešení problémů souběhu uvedených v předchozí kapitole.

1. Transakce, která chce získat entici z databáze, musí nejprve požadovat sdílený zámek na tuto entici.
2. Transakce, která chce aktualizovat entici v databázi, musí nejprve požadovat výlučný zámek na tuto entici. Pokud tato transakce drží S zámek, pak je tento zámek změněn na X.

3. Jestliže zámek požadovaný transakcí B nemůže být přidělen okamžitě, protože požadavek koliduje s dříve přiděleným zámkem transakci A, pak transakce B přejde do **stavu čekání** (angl. **wait state**). Transakce v tomto režimu setrvá dokud požadavek není proveden, což bude trvat nejméně do doby kdy transakce A uvolní zámek (zámek může být požadován dalšími transakcemi). Systém se musí postarat o to, aby transakce B nesetrvala v tomto stavu navždy[3]. Nejjednodušší cesta jak tento požadavek zabezpečit je řadit požadavky do fronty: transakci, která první požádá o zámek, bude zámek přidělen nejdříve.

4. Výlučné zámky jsou automaticky uvolněny na konci transakce (realizovaným operacemi COMMIT nebo ROLLBACK). Sdílené zámky jsou nejčastěji rovněž uvolněny na konci transakce (viz další text).

Tento protokol se nazývá **přísné dvou fázové uzamykání** (nebo jen dvou fázové uzamykání, angl. **strict two-phase locking**) a vrátíme se k němu později.

15.4.1 Vliv uzamykání na problémy souběhu

Nyní se podíváme jak dvou fázové uzamykání ovlivňuje problémy souběhu popsané v kapitole 15.2.

Problém ztráty aktualizace

Obrázek 15.6 ukazuje řešení problému ztráty aktualizace prezentovaného na obrázku 15.1 pomocí dvou fázového uzamykání. Aktualizace transakce A v čase t_3 není akceptována kvůli jejímu implicitnímu požadavku na výlučný zámek, který koliduje se sdíleným zámkem přiděleným transakci B. Transakce A přejde tedy do stavu čekání. Ze stejného důvodu přejde transakce B také do stavu čekání v čase t_4. Obě transakce tedy nepokračují v činnosti. Vidíme, že jsme sice vyřešili původní problém ztráty aktualizace, nicméně objevuje se zde jiný problém zvaný **uváznutí** (angl. **deadlock**) (viz kapitola 15.4.2).

Problém nepotvrzené závislosti

Obrázky 15.7 a 15.8 ukazují řešení problému nepotvrzené závislosti prezentovaného na obrázcích 15.2 a 15.3 pomocí dvou fázového uzamykání. Operace

[3] Tuto situaci nazýváme **livelock** nebo **starvation**.

Transakce A	Čas	Transakce B
`READ` t (získán zámek S na t)	t_1	-
-	t_2	`READ` t (získán zámek S na t)
`WRITE` t (požadavek na zámek X na t)	t_3	-
wait	t_4	`WRITE` t (požadavek na zámek X na t)
wait		wait
wait		wait

Obrázek 15.6: Problém ztráty aktualizace nenastane, ovšem v čase t_4 dojde k uváznutí.

transakce A (čtení na obrázku 15.7 a aktualizace na obrázku 15.8) není v čase t_2 akceptována kvůli jejímu implicitnímu požadavku na zámek, který koliduje s výlučným zámkem přiděleným transakci B. Transakce A přejde tedy do stavu čekání do doby kdy transakce B provedete COMMIT nebo ROLLBACK a uvolní výlučný zámek na t. Po uvolnění zámku pokračuje transakce A v činnosti a dále pracuje s potvrzenou hodnotou entice t: pokud transakce B provede COMMIT, pak A pracuje s hodnotami nastavenými v čase t_1, pokud B provede ROLLBACK, pak A pracuje s hodnotami nastavenými před časem t_1. Jelikož transakce A není závislá na nepotvrzené aktualizaci transakce B, tento problém souběhu je vyřešen.

Transakce A	Čas	Transakce B
-	t_1	`WRITE` t (získán zámek X na t)
`READ` t (požadavek na zámek S na t)	t_2	-
wait	t_3	`COMMIT/ROLLBACK` (uvolnění zámku X na t)
opakuj: `READ` t (získán zámek S na t)	t_4	

Obrázek 15.7: Transakce A nepřečte v čase t_2 změnu nepotvrzenou transakcí B.

Transakce A	Čas	Transakce B
-	t_1	`WRITE` t (získán zámek X na t)
`WRITE` t (požadavek na zámek X na t)	t_2	-
wait	t_3	`COMMIT/ROLLBACK` (uvolnění zámku X na t)
opakuj: `WRITE` t (získán zámek X na t)	t_4	

Obrázek 15.8: Transakce A neaktualizuje v čase t_2 změnu nepotvrzenou transakcí B.

Problém nekonzistentní analýzy

Obrázek 15.9 ukazuje řešení problému nekonzistentní analýzy prezentovaného na obrázku 15.4 pomocí dvou fázového uzamykání. Aktualizace transakce B v čase t_6 není akceptována kvůli jejímu implicitnímu požadavku na výlučný zámek na acc_1, který koliduje se sdíleným zámkem přiděleným transakci A. Transakce B přejde tedy do stavu čekání. Podobně, čtení transakce A v čase t_7 není akceptováno kvůli jejímu implicitnímu požadavku na sdílený zámek na acc_3, který koliduje s výlučným zámkem přiděleným transakci B. Transakce A přejde tedy také do stavu čekání. Vidíme, že jsme sice vyřešili původní problém nekonzistentní analýzy, nicméně nastalo uváznutí.

15.4.2 Uváznutí (deadlock)

V předchozí kapitole jsem viděli jak dvou fázové uzamykání řeší problémy souběhu. Viděli jsme ovšem také, že uzamykání může způsobit uváznutí. Na obrázku 15.10 vidíme obecné schéma uzamykání při kterém dojde k uváznutí. r_1 a r_2 jsou v tomto případě libovolné uzamykatelné zdroje, nemusí se tedy nutně jednat o entice.

Uváznutí je tedy situace kdy dvě nebo více transakcí jsou ve stavu čekání a čekají na uvolnění zámků držených jinou transakcí. Pro řešení problému uváznutí existují tyto strategie:

1. **Detekce uváznutí**:
   (a) nastavení časových limitů,

$acc_1 = 30$	$acc_2 = 20$	$acc_1 = 50$
Transakce A	**Čas**	**Transakce B**
READ acc_1 (získán zámek S na acc_1) $suma = 30$	t_1	-
READ acc_2 (získán zámek S na acc_2) $suma = 50$	t_2	-
-	t_3	READ acc_3 (získán zámek S na acc_3)
-	t_4	WRITE $acc_3 = 60$ (získán zámek X na acc_3)
-	t_5	READ acc_1 (získán zámek S na acc_1)
-	t_6	WRITE $acc_1 = 20$ (požadavek na zámek X na acc_1)
READ acc_3 (požadavek na zámek X na acc_3) wait	t_7	wait wait

Obrázek 15.9: Dvou fázové uzamykání zabránilo nekonzistentní analýze, ale nastalo uváznutí.

(b) detekce cyklu v grafu **Wait-For**.

2. **Prevence uváznutí**.

Nejjednodušší variantou je nastavení **časových limitů**. Systémy používající tuto strategii předpokládají, že transakce může trvat nejdéle nějakou dobu. Jakmile transakce trvá déle, systém předpokládá, že došlo k uváznutí. Efektivnější variantou je detekce cyklu v grafu **Wait-For**, která zaznamenává jaké transakce na sebe vzájemně čekají. Řešení uváznutí spočívá ve výběru jedné z uváznutých transakcí a provedení operace ROLLBACK (po které jsou uvolněny zámky). Ostatní uváznuté transakce tak mohou pokračovat v činnosti. Zrušená transakce je spuštěna znovu nebo je aplikaci spouštějící tuto transakci vygenerována výjimka s informací o zrušení transakce z důvodu vzniku uváznutí. Z pohledu vývojáře je první řešení čistější.

Třetí používaná strategie se snaží **uváznutí předcházet** modifikací uzamykacího protokolu. V článku [38] byly uvedeny dvě verze uzamykacího protokolu zvané Wait-Die a Wound-Wait. Postup je následující:

Transakce A	Čas	Transakce B
získán zámek X na r_1	t_1	
-	t_2	získán zámek X na r_2
požadavek na zámek X na r_2	t_3	-
wait	t_4	požadavek na zámek X na r_1
wait		wait

Obrázek 15.10: Příklad uváznutí.

1. Každé transakci je předěleno časové razítko – čas začátku transakce – které je unikátní.
2. Pokud transakce A požaduje zámek na entici, která je již uzamčena transakcí B, pak:
   (a) Při variantě Wait-Die: pokud A je starší než B, pak A přejde do stavu čekání; pokud A je mladší než B, transakce A je zrušena operací ROLLBACK a spuštěna znovu[4].
   (b) Při variantě Wound-Wait: pokud A je mladší než B, pak A přejde do stavu čekání; pokud A je starší než B, transakce B je zrušena operací ROLLBACK a spuštěna znovu[5].
3. Pokud je transakce spuštěna znovu ponechává si své původní časové razítko.

Čtenář si jistě povšiml, že první část jména popisuje situaci, kdy transakce A je starší než B. V případě Wait-Die, do stavu čekání přejdou transakce starší, v případě Wound-Wait do stavu čekání přejdou transakce mladší. Autoři dokázali, že v případě těchto protokolů nemůže dojít k uváznutí, k nekonečnému čekání transakce (v kapitole 15.4 jsme pro tento jev zavedli pojem **livelock**), ani k nekonečnému znovuspouštění transakce. Nevýhodou obou protokolů je relativně vysoký počet operací ROLLBACK.

Z předchozích kapitol je patrné, že implementace řízení transakcí (především uzamykacího protokolu) velmi ovlivňuje výkon ŠRBD. Z tohoto důvodu se stále vyvíjejí uzamykací protokoly, které umožňují dosáhnout co největší propustnosti (měřeno počtem transakcí za časový úsek, nejčastěji za vteřinu). Na stránkách organizace Transaction Processing Performance Council[6] najdeme žebříčky aktuálně nejvýkonnějších systémů pro jednotlivé testy.

[4] Dle názvosloví autorů transakce A zemře (angl. die).
[5] Dle názvosloví autorů transakce B je poraněna (angl. wound).
[6] `http://www.tpc.org/`

15.5 Plánování transakcí – serializovatelnost

Pokud jsou transakce provedeny za sebou mluvíme o **sériovém plánu**. Sériový plán často zapisujeme jako entici uspořádanou dle pořadí vykonávání jednotlivých transakcí; např. když A je vykonána před B, pak zapisujeme (A, B).

Abychom byly schopni zodpovědět otázku zda souběžně prováděné transakce byly provedeny korektně, musíme zavést nějakou 'míru korektnosti'. Tato míra korektnosti se nazývá **serializovatelnost** (angl. **serializability**): plán vykonávání dvou transakcí je korektní tehdy a jen tehdy pokud je serializovatelný: výsledky takového plánu jsou stejné jako výsledky libovolného sériového plánu (říkáme že plány jsou ekvivalentní).

Pokud se podíváme na plány z obrázků 15.1–15.4 demonstrující problémy souběhu vidíme, že žádný z plánů není serializovatelný; není tedy ekvivalentní s plány (A, B) nebo (B, A). Vezměme sériové plány transakcí z obrázku prezentujícího problém nekonzistentní analýzy 15.4. Plán (A, B) je uveden na obrázku 15.11, plán (B, A) pak na obrázku 15.12. Plán transakcí z obrázku 15.4 tedy není ekvivalentní ani s plánem (A, B) ani s plánem (B, A): výsledek výpočtu sumy částek na účtech je 110, zatímco korektní částka u obou sériových plánů je 100. Vidíme tedy, že serializovatelnost nám skutečně poskytuje míru korektnosti souběžných plánů.

$acc_1 = 30$	$acc_2 = 20$	$acc_1 = 50$
Transakce A	**Čas**	**Transakce B**
`READ` acc_1	t_1	-
$suma = 30$		
`READ` acc_2	t_2	-
$suma = 50$		
`READ` acc_3	t_3	-
$suma = 100$		
-	t_4	`READ` acc_3
-	t_5	`WRITE` $acc_3 = 60$
-	t_6	`READ` acc_1
-	t_7	`WRITE` $acc_1 = 20$
-	t_8	`COMMIT`
		$acc_1 = 20$
		$acc_2 = 20$
		$acc_3 = 60$

Obrázek 15.11: Sériový plán (A, B).

$acc_1 = 30$	$acc_2 = 20$	$acc_1 = 50$
Transakce A	**Čas**	**Transakce B**
-	t_1	`READ` acc_3
-	t_2	`WRITE` $acc_3 = 60$
-	t_3	`READ` acc_1
-	t_4	`WRITE` $acc_1 = 20$
-	t_5	`COMMIT`
		$acc_1 = 20$
		$acc_2 = 20$
		$acc_3 = 60$
`READ` acc_1 $suma = 20$	t_6	-
`READ` acc_2 $suma = 40$	t_7	-
`READ` acc_3 $suma = 100$	t_8	-

Obrázek 15.12: Sériový plán (B, A).

Dvoufázového uzamykání prezentované v kapitole 15.4 zaručuje, že plán bude vždy serializovatelný (podrobnosti podáme později v této kapitole): plány na obrázcích 15.7 a 15.8 jsou ekvivalentní s plánem (B, A). Plány z obrázků 15.6 a 15.9 skončí uváznutím, které bude detekováno a pokud bude zrušena transakce A, pak je plán ekvivalentní s plánem (B, A).

Pozorný čtenář si povšiml, že jsme v definici serializovatelnosti mluvili o tom, že serializovatelný plán je ekvivalentní s libovolným sériovým plánem; tedy nemusí být ekvivalentní se všemi sériovými plány. Zjevně různé sériové provedení transakcí může produkovat různé výsledky, např. transakce A provede $2 \times x$ a transakce B provede $x + 10$. Pokud bude počáteční hodnota $x = 5$ pak výsledkem (A, B) je 20, výsledkem (B, A) je 30. Stejně tak se může lišit výsledek různých souběžných plánů. Je tedy zřejmé proč je plán korektní pokud je ekvivalentní s libovolným sériovým plánem.

V článku [15] autoři dokázali následující větu.

Věta 15.1. *(Věta o dvoufázovém uzamykání) Pokud transakce dodržující striktní dvoufázové uzamykání, pak všechny možné souběžné plány jsou serializovatelné.*

Nyní si uveďme zjednodušenou variantu dvoufázového uzamykacího protokolu uvedeného v kapitole 15.4:

1. Transakce musí požádat o zámek na objekt (např. entici) před tím než chce nad tímto objektem provést nějakou operaci.

2. Po uvolnění nějakého zámku nesmí transakce již požadovat další zámek.

Transakce dodržující tento protokol tak pracují ve dvou fázích: v první fázi požadují zámky, ve druhé fázi zámky uvolňují. Prakticky je druhá fáze realizována jedinou operací COMMIT nebo ROLLBACK, při které jsou uvolněny všechny držené zámky.

15.6 Úrovně izolace

Serializovatelnost garantuje izolaci transakcí ve smyslu podmínky ACID. Pokud jsou tedy plány všech transakcí serializovatelné, pak se programátor nemusí starat o negativní vliv jiné transakce prováděné ve stejném čase. Nic ovšem není zadarmo a tak i za izolovanost transakcí musíme zaplatit – v tomto případě menším výkonem souběhu (tedy nižším počtem vykonaných transakcí – propustností). SŘBD proto umožňují nastavit **úrovně izolace**, které zvýší propustnost, ale sníží úroveň izolace transakce.

V SQL jsou definovány tyto 4 úrovně izolace (seřazeno od nejnižší úrovně po nejvyšší):

- READ UNCOMMITTED (RU)
- READ COMMITTED (RC)
- REPEATABLE READ (RR)
- SERIALIZABLE (SR)

Vyšší úroveň značí vyšší míru izolace, ale nižší propustnost; naopak nižší úroveň značí nižší míru izolace, ale vyšší propustnost. Pokud zvolíme maximální úroveň, SERILIZABLE (ve slovníku DB2 repeatable read), pak jsou transakce maximálně izolovány od vlivů ostatních souběžných transakcí. Uveďme si chování nějaké nižší úrovně izolace. Například úroveň READ COMMITTED (ve slovníku DB2 cursor stability) povoluje následující chování uzamykacího protokolu v transakci (jedná se o dříve popsanou výjimku neopakovatelné čtení):

1. Získáme z databáze entici t.

2. Získáme S zámek na t.

3. Pokud nedojde k modifikaci a tedy k získání X zámku, pak

4. zámek může být uvolněn ještě před ukončením transakce.

Pokud jiná transakce provede modifikace t a COMMIT, pak původní transakce po možném novém čtení t zjistí rozdílnou hodnotu oproti prvnímu čtení – databáze je tedy v nekonzistentním stavu. Pozorný čtenář si jistě povšiml, že tento plán nedodržel dvoufázové uzamykání: zámek byl uvolněn před ukončením transakce. Pokud by úroveň izolace byla SR nebo RR, pak by všechny získané zámky byly uvolněny až na konci transakce a k tomuto problému by nedošlo.

SR je v každém případě jistější volba, kdy díky dvoufázovému uzamykacímu protokolu, máme zajištěnu serializovatelnost transakcí. Na druhou stranu nižší úrovně izolace mohou, za určitých okolností, poskytnout vyšší míru souběhu.

15.6.1 Výjimky souběhu pro různé úrovně izolace

V tabulce **??** vidíme různé výjimky souběhu, které mohou či nemohou nastat pro různé úrovně izolace transakcí. Výjimky špinavé čtení a neopakovatelné čtení jsme prezentovali dříve v kapitole 15.2.4, nyní si navíc ukážeme tzv. výskyt fantomů.

Tabulka 15.1: Úrovně izolace a porušení souběhu definované v SQL.

Úroveň izolace	**Špinavé čtení**	**Neopakovatelné čtení**	**Výskyt fantomů**
READ UNCOMMITTED	Ano	Ano	Ano
READ COMMITTED	Ne	Ano	Ano
REPEATABLE READ	Ne	Ne	Ano
SERIALIZABLE	Ne	Ne	Ne

Mějme následující tabulku `Student`:

`login`	`jmeno`	`rocnik`
jan001	Jan	1
mil002	Milan	3

Pokud je povolena úroveň RR (a nižší) a v transakci se vyskytují dva stejné dotazy, pak může nastat výjimka souběhu zvaná **výskyt fantomů**. Mějme plány těchto dvou transakcí.

Transakce A	Čas	Transakce B
`SELECT * from student`	t_1	-
`WHERE rocnik BETWEEN 1 AND 2`		
-	t_2	`INSERT INTO student`
		`VALUES('mar006',`
		`'Marek',2)`
	t_3	`COMMIT`
`SELECT * from student`	t_4	-
`WHERE rocnik BETWEEN 1 AND 2`		
`COMMIT`	t_5	-

V tomto případě systém vrátí pro stejný dotaz v transakci A v čase t_1 resp. t_4 různé výsledky: SŘBD neprovádí uzamykání rozsahu záznamů z tabulky `Student` (tedy záznamů s hodnotou atributu `rocnik` ≤ 1 a ≥ 2). V případě úrovně SR oba dotazy vrátí stejný počet záznamů.

V případě úrovně RC (a nižší) je umožněno tzv. **neopakovatelné čtení**. V tomto případě příkaz `SELECT` požaduje sdílený zámek na entici, SŘBD ale nedodrží dvoufázový uzamykací protokol a zámky mohou být uvolněny před ukončením transakce. Výlučné zámky jsou ovšem uvolněny až na konci transakce. V případě úrovní SR a RR k této výjimce nedochází, v případě úrovní RC a RU může systém v následujícím plánu vrátit aktualizovaný záznam:

Transakce A	Čas	Transakce B
`SELECT * from student`	t_1	-
`WHERE id='jan001'`		
-	t_2	`UPDATE student SET rocnik=1`
		`WHERE id='jan001'`
		`COMMIT`
`SELECT * from student`	t_3	-
`WHERE id='jan001'`		
`COMMIT`	t_4	

V případě úrovně RU může nastat tzv. **špinavé čtení**, tedy transakce může načíst data změněná a dosud nepotvrzená jinou transakcí, viz:

Transakce A	Čas	Transakce B
`SELECT * from student WHERE id='jan001'`	t_1	-
-	t_2	`UPDATE student SET rocnik=1 WHERE id='jan001'`
`SELECT * from student WHERE id='jan001'`	t_3	-
`COMMIT`	t_4	
	t_5	`COMMIT/ROLLBACK`

Některé SŘBD, zdá se, implementují nekorektní úroveň SR (např. Oracle a PostgreSQL), naproti tomu implementace v DB2 a MS SQL Server odpovídá SQL standardu.

15.6.2 Explicitní uzamykání

Systémy, které podporují úroveň izolace nižší než SR často podporující explicitní uzamykání příkazem `LOCK`. Např. Oracle nebo DB2 podporují příkaz `LOCK TABLE`. Uveďme si nyní syntaxi pro Oracle [33, 32]:

```
LOCK TABLE <názvy tabulek nebo pohledů oddělené čárkami>
IN <lock_type> MODE [NOWAIT];
```

kde `lock_type` může být: `ROW SHARE`, `ROW EXCLUSIVE`, `SHARE UPDATE`, `SHARE`, `SHARE ROW EXCLUSIVE`, `EXCLUSIVE`. Pokud použijeme klíčové slovo `NOWAIT`, systém nečeká s přidělením zámku pokud je tabulka uzamčena jiným uživatelem a vrátí řízení volajícímu programu. Podobně jako v případě implicitního uzamykání, systém uvolní všechny zámky na konci transakce.

Příklad 15.1. (Použití příkazu `LOCK TABLE`)

```
LOCK TABLE Student IN SHARE MODE;
```

Tímto příkazem požadujeme sdílený zámek na tabulku `Student`. Systém bude čekat na přidělení zámku, pokud jiná transakce získala dříve na tuto tabulku zámek.

Jiné SŘBD obsahují další možnosti explicitního uzamykání, např. PostgreSQL umožňuje vynutit si klíčovým slovem `FOR UPDATE` u příkazu `SELECT` výlučný zámek na získaných enticích.

Část VI

Ostatní databázové technologie

Kapitola 16

Datová vrstva informačního systému

Obsah

Cíl kapitoly:
Představit existující specifikace a implementace pro vývoj datové vrstvy informačního systému

16.1 Architektury informačních systémů

Databázový systém není izolovanou aplikací, ale často slouží pro uložení a dotazovaní dat ve složitější softwarové aplikaci. Takovým aplikacím říkáme **informační systémy** (IS). Při vývoji informačního systému tedy neřešíme jen problém

tvorby samotné databáze, ale i aplikace, která databázi využívá. Při vývoji IS musíme nejprve zvolit vhodnou **architekturu**.

Architekturou rozumíme soustavu pojmů, prvků, struktur a interakcí systému z pohledu vnějšího pozorovatele. Účelem architektury je pochopit IS, který budujeme. Návrháři vytváří celkový obrázek systému a mohou tak jednodušeji přiřadit jednotlivé části návrhářům a vývojářům. Architektura tedy slouží nejen pro přiřazení zodpovědnosti vývojářům za jednotlivé části, ale umožňuje vývojářům lépe pochopit jejich úkoly v kontextu celého systému. Vhodně zvolená architektura zachycuje současné potřeby projektu, ale rovněž nám umožní zvládnout další růst systému, nebo dokonce změny požadavků popř. přístupů k implementaci. Při návrhu architektury můžeme identifikovat problematická místa vyvíjeného IS. Architektura by nám měla umožnit izolovat zbytek systému od změn v některé z jeho částí. Umožňuje opakovanou použitelnost konceptů, prvků a struktur při rozšiřování systému.

V současné době rozlišujeme několik typů architektur [21]:

- **Metody a třídy** nám umožní logicky rozdělit aplikaci na objekty.
- **Subsystémy** jsou balíky těsně svázaných tříd s dobře definovaným rozhraním, např. osoby, předměty, zkoušky. Takový IS je většinou spravován jediným týmem, složitost snižujeme oddělením rozhraní od implementace.
- **Úlohy**. Vertikální členění aplikace podle oblastí, např. věda a výzkum, studium, ekonomika. V této architektuře je snížena pravděpodobnost interakce mezi subsystémy, z tohoto důvodu se zjednodušuje komunikace mezi týmy. Mezi jednotlivými úlohami lze definovat formální rozhraní.
- **Aplikační rámce a knihovny**. Aplikační rámec je množina abstraktních a konkrétních tříd tvořících dohromady generický softwarový systém. Aplikační rámec nám umožňuje soustředit se na specifické problémy aplikace, místo toho abychom řešili, jakým způsobem bude aplikace implementována.
- **Vrstvová struktura**. Logické členění subsystémů aplikace do vrstev (**layers**), např. prezentace, ukládání dat a logika aplikace. Každá vrstva představuje abstrakci s vlastní zodpovědností. Takovéto členění aplikace umožní jednodušší pochopení a použití vrstvy a jednodušší vývoj v rámci vrstvy. Vyšší vrstvy jsou izolovány od změn v implementaci nižších vrstev.
- **Hierarchická struktura** (**tiers**). Hierarchické členění aplikace mezi více procesů. Nastává uvnitř vrstev, z tohoto důvodu zvyšuje vnitřní složitost vrstev.

Nejčastěji používanou architekturou je architektura vrstvová nebo kombinovaná. V 90. letech 20. století byla velmi používaná architektura klient-server. Komunikace mezi klientem SŘBD a samotným SŘBD rovněž pracuje na principu **klient-server**. V současné době používáme při vývoji IS z různých důvodů vícevrstvé architektury.

Většina IS musí nějakým způsobem pracovat v síti. Nejčastějším případem je, že klient IS je webový prohlížeč, který komunikuje se serverem pomocí **HTTP** protokolu. HTTP je klient-server protokol. Tj. klient zasílá na server HTTP **požadavky** (angl. **request**), server zpět vrací HTTP **odpověď** (angl. **response**).

16.2 Platformy .NET a Java

Platformy ASP.NET [28] a Java2EE [43] jsou dvě nejrozšířenější platformy pro vývoj webových informačních systémů. Rozdíl mezi oběma platformy je často nepatrný. Pro vývojáře je jistě důležité, že zatímco platforma Java je založena na jediném programovacím jazyce, platforma .NET umožňuje vývoj v libovolném programovacím jazyce implementovaným pro tuto platformu (např. C# nebo C++). Především jazyk C# je velmi podobný jazyku Java. Výrazné podobnosti lze nalézt i mezi knihovnami tříd obou platforem.

16.2.1 .NET

.NET[1] je platforma firmy Microsoft. Pro vývoj webových IS je určena část, zvaná ASP.NET[2]. ASP.NET obsahuje tzv. **Server Controls**, což jsou komponenty, které uživatel může na webové stránce využívat pomocí značek. Uživatel má možnost definovat vlastní **značky**, tzv. **User Control**. Pro přístup k SŘBD je určena knihovna **ADO.NET**[3] [27].

16.2.2 Java2EE

Java2EE[4] (Java2 Enterprise Edition) je platforma firmy Sun pro vývoj webových IS. Pro generování HTML dokumentů je určen tzv. **servlet**. Servlet je kód v jazyce Java umístěny do webového serveru. **Java Server Pages** (JSP) jsou stránky,

[1] `http://msdn.microsoft.com/netframework/`
[2] `http://www.asp.net/`
[3] `http://msdn.microsoft.com/en-us/library/h43ks021(VS.71).aspx`
[4] `http://java.sun.com/javaee/`

které v HTML kódu obsahující JSP značky a kód v jazyce Java. Do HTML stránky můžeme vkládat tzv. **uživatelské značky**, což jsou značky vykonávající definovanou akci. Významnou výhodou pro vývoj webových IS je, že pro platformu Java existuje celá řada aplikačních rámců zjednodušujících vývoj IS, např. **Jakarta Struts**[5] [21], **Java Server Faces** atd. Pro přístup k SŘBD je určeno rozhraní **JDBC**[6] [42].

16.3 Datová vrstva IS

Datová vrstva IS odděluje aplikaci od databáze. Jejím úkolem je transparentní vykonávání databázových operací směrem k aplikaci. Implementace datové vrstvy jsou tvořeny funkcemi nebo třídami, které umožňují: otevřít spojení s databází, zaslat dotaz, získat výsledek a uzavřít spojení s databází. Mezi rozhraní datové vrstvy patří např. ODBC, JDBC a ADO.NET.

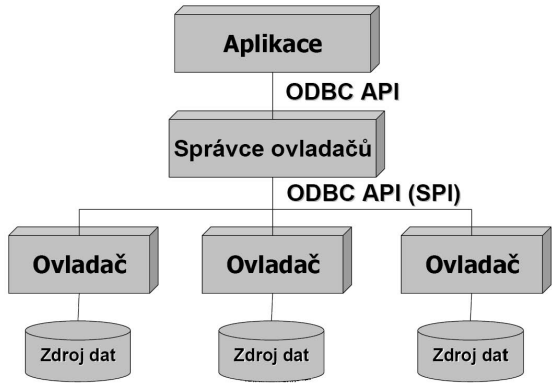
16.3.1 Open DataBase Connectivity (ODBC)

Standard ODBC[7] byl vydán v roce 1992 organizací SQL Access group. Cílem bylo poskytnout specifikaci umožňující dotazování dat z aplikace bez ohledu v jaké databázi jsou uložena. Do architektury ODBC je vložena mezivrstva, **ovladač**, překládající uživatelské dotazy na dotazy databáze. Jelikož aplikace může využívat ovladačů více, je do architektury vložen **správce ovladačů** (**Driver Manager**). Na obrázku 16.1 vidíme popisovanou architekturu ODBC.

Provedení dotazu se realizuje v těchto krocích:

1. připojení k datovému zdroji,
2. inicializace,
3. vytvoření a provedení dotazu,
4. získání výsledku,
5. ukončení transakce,
6. odpojení od datového zdroje.

[5] `http://jakarta.apache.org/struts`
[6] `http://java.sun.com/javase/technologies/database/`
[7] `http://support.microsoft.com/kb/110093`



Obrázek 16.1: Architektura ODBC

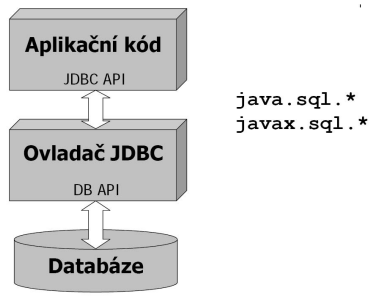
Jelikož se jedná o přirozené kroky, které provádíme při dotazování dat z aplikace, stejné schéma můžeme nalézt u JDBC a ADO.NET popisované v dalších kapitolách.

16.3.2 Java Database Connectivity (JDBC)

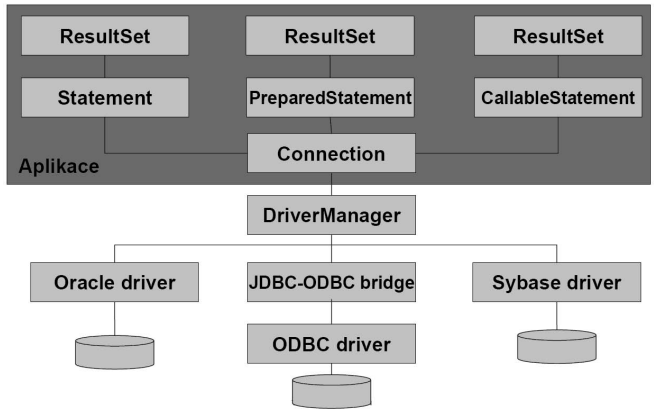
ODBC je standardem definujícím celou řadu funkcí pro komunikaci s databází. S nástupem objektově-orientovaných jazyků vyvstal požadavek na specifikace s objektovým rozhraním k databázi. Takovouto specifikací pro jazyk Java je JDBC [42]. JDBC používá vlastní ovladače, pokud není takový ovladač pro danou databázi k dispozici, můžeme použít bridge mezi JDBC a tímto ODBC ovladačem. Na obrázku 16.2 vidíme obecnou specifikaci JDBC, na obrázku 16.3 pak třídy specifikace.

Ovladač je v JDBC představován třídou (dodávanou většinou výrobcem konkrétního SŘBD), např. `sun.jdbc.odbc.JdbcOdbcDriver` nebo `com.mysql.jdbc.Driver`. Ve výpisu 16.1 vidíme příklad komunikace s databází Oracle pomocí JDBC.

```
DriverManager.registerDriver(new OracleDriver());
String connStr ="jdbc:oracle:thin:@infra.cs.vsb.cz
    :1521:d456";
Connection connection =
  DriverManager.getConnection(connStr,"xxx","yyy");
```



Obrázek 16.2: Obecná architektura ODBC



Obrázek 16.3: Třídy specifikace JDBC

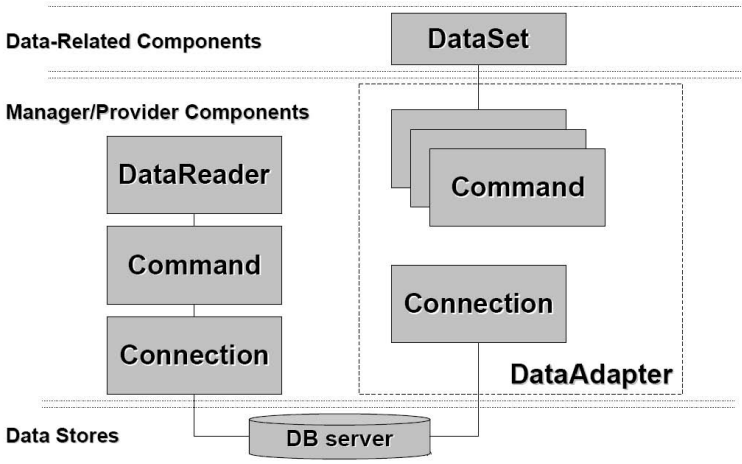
```
Statement statement = connection.createStatement();
boolean resultf = statement.execute(
  "UPDATE PROBE SET VALUE=1 WHERE ID = 2");

ResultSet resultSet = null;
if (resultf)
{
  resultSet = statement.getResultSet();
}
int updateCount = statement.getUpdateCount();
```

```
statement.close();
connection.close();
```

Výpis 16.1: Kód dotazující se SŘBD Oracle pomocí JDBC

16.3.3 ADO.NET

ADO.NET [27] je specifikací přístupu k datovým zdrojům na platformě .NET. Na obrázku 16.4 vidíme třídy specifikace ADO.NET. V kódu 16.2 je uveden příklad komunikace s databází Oracle pomocí ADO.NET.



Obrázek 16.4: Třídy specifikace ADO.NET

```
using System.Data;
using System.Data.OracleClient;
...
OracleConnection connection = new OracleConnection(
   "Data␣Source=info;User␣ID=xxx;Password=yyy");
connection.Open();

OracleCommand command = new OracleCommand("select␣*␣
   from␣student", connection);
```

```
OracleDataAdapter dbAdapter = new OracleDataAdapter(
    command);
DataSet dataSet = new DataSet("student");

try
{
  dbAdapter.Fill(dataSet);
}
catch(Exception exc)
{
  throw new Exception(exc.Message);
}

....

// vypis vysledek
System.Windows.Forms.DataGrid studentDataGrid = new
    System.Windows.Forms.DataGrid();
studentDataGrid.DataSource = dataSet;
studentDataGrid.Update();

...

connection.Close();
```

Výpis 16.2: Kód dotazující se SŘBD Oracle pomocí ADO.NET

16.4 Transakce v datové vrstvě IS

Implicitně je v JDBC každý zaslaný příkaz proveden jako samostatná transakce. Pokud chceme více příkazu provést v jedné transakci musíme nastavit `autoCommit` na `false`. Výpis 16.3 prezentuje práci s transakcemi v JDBC. Obdobně pracujeme s transakcemi v ADO.NET, ve výpise 16.4 vidíme ukázku práce s transakcemi v ADO.NET.

```
// prikazy jsou potvrzeny az zaslanim commit
con.setAutoCommit(false);
try {
  PreparedStatement updateSales = con.prepareStatement
      (
```

```
    "UPDATE␣COFFEES␣SET␣SALES␣=␣?␣WHERE␣COF_NAME␣LIKE␣
       ?");
  // nastaveni parametru
  updateSales.executeUpdate();

  PreparedStatement updateTotal = con.prepareStatement
      (
    "UPDATE␣COFFEES␣SET␣TOTAL␣=␣TOTAL␣+␣?␣WHERE␣NAME␣
       LIKE␣?");
  // nastaveni parametru
  updateTotal.executeUpdate();
  con.commit(); // potvrzeni prikazu
}
catch(SQLExceptione)
{
  con.rollback();
} // obnoveni stavu pred zacatkem transakce
con.setAutoCommit(true);
```

Výpis 16.3: Transakce v JDBC

```
try
{
  Connect(); // nastaveni mConnection
  SqlTransaction SqlTransaction =
    mConnection.BeginTransaction(IsolationLevel.
       Serializable);

  // priprav INSERT prikaz
  SqlCommand command = new SqlCommand(null,
     mConnection,
    sqlTransaction);
  int insertNo = command.ExecuteNonQuery();

  // priprav UPDATE prikaz
  int updateNo = command.ExecuteNonQuery();

  mSqlTransaction.Commit();
  mConnection.Close();
}
catch (Exception e)
```

```
{
  mSqlTransaction.Rollback();
  mConnection.Close();
  throw e;
}
```

Výpis 16.4: Transakce v ADO.NET

Kapitola 17

Rozšiřující databázové technologie

Obsah

Cíl kapitoly:
V této kapitole budou uvedeny některé rozšiřující databázové aplikace: distribuované SŘBD a dolování dat.

17.1 Distribuované báze dat

S rozvojem počítačových sítí vzrostla potřeba distribuce dat mezi jednotlivými počítači sítě. **Distribuované databáze** [16, 36] poskytují některé významné rysy:

1. Zátěž může být rozložena na více počítačů.

2. Data jsou uložena v kopiích na různých počítačích. Tomto případě zůstává databáze přístupná i pokud jeden počítač s databázovým systémem se stane nepřístupným.

Na druhou stranu distribuované zpracování zvyšuje složitost některých činností databázových systémů. V distribuovaných SŘBD jsou data rozdělena **horizontálně** (části relace, fragmenty, jsou rozloženy mezi počítači) nebo **vertikálně** (projekce relace jsou rozděleny mezi počítači). Vykonávání transakcí v distribuovaných SŘBD je komplikovanější, v distribuovaném SŘBD existuje globální plánovač skládající se z lokálních plánovačů.

17.2 Datové sklady a dolování dat

Významnou aplikací databázových systémů jsou **datové sklady** (angl. **data-warehous**) [16]. Datové sklady využívají existující SŘBD pro uložení dat pro následné analýzy a **dolování informací** z dat. Při dolování informací (angl. **data-mining**) se snažíme získat informace, které nejsou na první pohled viditelné. Na rozdíl do SŘBD jsou informace v datových skladech uloženy i s redundantními informacemi, používají se různé předpočítané hodnoty agregačních funkcí apod.

Část VII

Systémy Řízení Báze Dat

Kapitola 18

Oracle

Tato kapitola neslouží jako referenční manuál Oracle, v kapitole pouze chceme popsat některé rysy tohoto SŘBD, podrobnosti pak čtenář může najít v referenčních manuálech[1].

18.1 Datové typy

Příkazy DDL mohou používat jak datové typy Oracle tak datové typy ANSI SQL (viz kapitola 4.3.2). Datové typy ANSI jsou pak převedeny na ekvivalentní datové typy Oracle. V následující tabulce jsou uvedeny ANSI datové typy a ekvivalentní datové typy Oracle.

[1]Seznam jednotlivých referenčních manuálů pro Oracle 11g R2 najdete zde: `http://www.oracle.com/pls/db112/portal.all_books` nebo `http://www.oracle.com/pls/db112/homepage`

Datové typy ANSI SQL	Datové typy Oracle
CHARACTER(n) CHAR(n)	CHAR(n)
CHARACTER VARYING(n) CHAR VARYING(n) VARCHAR(n)	VARCHAR2(n)
NATIONAL CHARACTER(n) NATIONAL CHAR(n) NCHAR(n)	NCHAR(n)
NATIONAL CHARACTER VARYING(n) NATIONAL CHAR VARYING(n) NCHAR VARYING(n)	NVARCHAR2(n)
NUMERIC[(p,s)] DECIMAL[(p,s)]	NUMBER(p,s)
INTEGER INT SMALLINT	NUMBER(38)
FLOAT DOUBLE PRECISION REAL	FLOAT(126) FLOAT(126) FLOAT(63)

18.2 Řízení uživatelských účtů

Vytvoření uživatele:

```
CREATE USER <user_name>
  IDENTIFIED BY <password>
  DEFAULT TABLESPACE <table_space>
  TEMPORARY TABLESPACE <table_space>;
```

Zrušení uživatele:

```
DROP USER <user_name>;
```

18.3 Ochrana dat

V kapitole 4.3.5 jsme popsali ochranu dat v ANSI SQL, Oracle umožňuje definovat další práva, viz následující tabulka:

Právo	Popis
CREATE USER	Právo vytvořit uživatele
CREATE TABLE	Právo zrušit uživatele

Příklad:

```
CREATE USER dec80
  IDENTIFIED BY dec8080
  DEFAULT TABLESPACE USERS
  TEMPORARY TABLESPACE TEMP
  PASSWORD EXPIRE;

GRANT CONNECT TO dec80;
GRANT RESOURCE TO dec80;
```

`PASSWORD EXPIRE` způsobí, že uživatel po přihlášení musí změnit heslo. Změna hesla uživatele se provádí příkazem `ALTER USER`, např.:

```
ALTER USER dec80 IDENTIFIED BY dec8081;
```

`GRANT RESOURCE` je role umožňující vytváření základních typů objektů. Tato role ovšem neumožní vytvářet pohledy a materializované pohledy, musíme tedy použít:

```
GRANT CREATE MATERIALIZED VIEW TO <user_name>;
GRANT CREATE VIEW TO <user_name>;
```

18.4 JDD

18.4.1 Integritní omezení

```
CREATE TABLE course
(
  id INT NOT NULL PRIMARY KEY,
  name VARCHAR2(40) NOT NULL,
  description VARCHAR2(100) NOT NULL
)
```

```
INSERT INTO course (id,name) VALUES (1,'a')

INSERT INTO course VALUES (1,'a','aa')

INSERT INTO course VALUES (1,'b','bb')

COMMIT
```

18.4.2 Definice cizího klíče

```
CREATE TABLE course
(
  id NUMBER(4) PRIMARY KEY,
  name VARCHAR2(50) NOT NULL,
  day VARCHAR2(2) NOT NULL,
  time VARCHAR2(11)
)

CREATE TABLE workgroup
(
  id NUMBER(2) PRIMARY KEY,
  name VARCHAR2(20) NOT NULL UNIQUE,
  course NUMBER(4) REFERENCES course NOT NULL
)
```

insert.sql:

```
INSERT INTO course VALUES (1, 'INS', 'we', '10:45-12:15');
INSERT INTO workgroup VALUES (1, 'test', 1);
COMMIT;

DELETE FROM course WHERE id=1;

CREATE TABLE student
(
  login VARCHAR2(6) NOT NULL PRIMARY KEY,
  name VARCHAR2(20) NOT NULL,
  surname VARCHAR2(40) NOT NULL,
```

```
  email VARCHAR2(70) NOT NULL,
  web VARCHAR2(80) NOT NULL,
  workgroup  NOT NULL REFERENCES TO workgroup
)
```

18.5 PL/SQL

Jazyk PL/SQL[2] představuje rozšíření jazyka SQL o procedurální rysy. Jazyk PL/SQL[3] byl vyvinut firmou Oracle zejména pro zefektivnění běhu a vývoje některých aplikací. Podobná procedurální rozšíření pak byla vyvinuta i pro další relační databáze: Transact-SQL pro Sybase a Microsoft SQL Server, PL/pgSQL pro PostgreSQL a SQL PL pro IBM DB2.

Vlastnosti PL/SQL:

- Efektivní práce s jednotlivými záznamy.
- Zachytávání výjimek.
- Práce s proměnnými, poli, kursory, BFILE atd.
- Hlavní výhodou PL/SQL oproti jazykům jako je Java nebo .NET je optimalizace výkonu uložených procedur a funkcí. Kód PL/SQL je uložen přímo v SŘBD a je tedy možné jej jednoduše spouštět z různých klientských aplikací.

Základní struktura PL/SQL bloku:

- `DECLARE` – nepovinná deklarace lokálních proměnných a kurzorů,
- `BEGIN` – povinné otevření bloku příkazů uložené procedury,
- `EXCEPTION` – nepovinné zachytávání výjimek,
- `END` – povinné ukončení bloku procedury.

[2] `http://www.oracle.com/pls/db112/portal.all_books`, PL/SQL Language Reference
[3] Syntaxe PL/SQL je založena na jazyku ADA.

18.5.1 Proměnné

V sekci `DECLARE` můžeme definovat proměnné, jenž mohou být použity v proceduře. Definice má následující tvar:

```
jmeno_promenne typ_promenne [NOT NULL := hodnota];
```

Kde:

- `jmeno_promenne` je název proměnné, který má obvykle prefix 'v_' [4],
- `typ_promenne` je platný typ proměnné,
- `hodnota` je nepovinná část, která definuje výchozí hodnotu proměnné.

Proměnné můžeme využít při běhu PL/SQL kódu pro uložení dočasných hodnot a pro manipulaci s nimi. Máme k dispozici několik základních příkazů umožňující do proměnné načíst hodnotu (viz tabulka 18.1).

Syntaxe	Příklad
`jmeno_promenne := hodnota`	`v_vek := 20`
`SELECT sloupec INTO jmeno_promenne FROM jmeno_tabulky`	`SELECT vek INTO v_vek FROM student WHERE login LIKE 'bon007'`

Tabulka 18.1: Příkazy, které umožňují do proměnné načíst hodnotu.

Při práci s proměnnou můžeme využít standardní aritmetické operátory v případě čísel. Dále můžeme využít operátor || ke konkatenaci řetězců a také standardní SQL funkce (`TO_CHAR`, `TO_DATE`, `SUBSTR`, `LENGTH`, atd.).

Příklad 18.1. (Proměnné) Následujícím PL/SQL blok načte jméno a příjmení studenta s loginem 'bon007' a vytvoří z něj email.

```
DECLARE
  v_fname VARCHAR2(20);
  v_lname VARCHAR2(20);
 v_email VARCHAR2(60);
BEGIN
  SELECT fname, v_lname INTO v_fname, v_lname
```

[4] Oracle Academy: Oracle Database 10g: Advanced PL/SQL, Student Guide. str. 26.

```
    FROM student WHERE login = 'bon007';
   v_email := fname || '.' || v_lname || '@vsb.cz';
   UPDATE student set email = v_email WHERE login = '
      bon007';
END;
```

18.5.2 Procedury

PL/SQL umožňuje vytvářet několik typů procedur. V zásadě se liší především způsobem jakým jsou spouštěny.

Anonymní procedury

Jsou to nepojmenované procedury, které nemohou být volány z jiné procedury. Tyto procedury tedy mohou být uloženy v souboru, nebo přímo zapsány na příkazový řádek, a spouštěny z klienta jako je SQL*Plus nebo Oracle SQL Developer. Anonymní procedury nejsou předkompilovány a mohou být proto pomalejší než pojmenované procedury a funkce.

Příklad 18.2. (Příklad anonymních procedur) Na následujícím příkladě vidíme anonymní proceduru jenž vkládá email do tabulky `Email` s jedním atributem `email` typu `VARCHAR2(30)`.

```
DECLARE
  v_name VARCHAR2(30) := 'michal.kratky@vsb.cz';
BEGIN
  INSERT INTO Email VALUES (v_name);
END;
```

V druhé proceduře vložíme do tabulky `Email` adresu studenta z tabulky `Student` s emailem bon007. Tabulka `Student` má dva atributy: `email` typu `VARCHAR2(30)` a `login` typu `CHAR(6)`.

```
DECLARE
  v_login CHAR(6) := 'bon007';
BEGIN
  INSERT INTO Email
   SELECT email FROM Student
   WHERE login = v_login;
END;
```

Pojmenované procedury

Pojmenované procedury obsahují hlavičku se jménem a parametry procedury. Takovouto proceduru je možné volat z jiných procedur nebo spouštět příkazem EXECUTE. Na rozdíl od anonymních procedur jsou pojmenované procedury předkompilovány a uloženy v databázi.

```
CREATE [OR REPLACE] PROCEDURE jmeno_procedury
  [(jmeno_parametru [mod] datovy_typ, ... )]
IS|AS
  definice lokálních proměnných
BEGIN
  tělo procedury
END [jmeno_procedury]
```

Kde:

- `jmeno_parametru` je název parametru, který má obvykle prefix 'p_' [5].
- `mod` je mód parametru, který může mít hodnotu IN (vstupní proměnná), OUT (výstupní proměnná) nebo IN OUT (vstupně výstupní proměnná).
- `datovy_typ` je platný datový typ proměnné. Proměnné typu `VARCHAR2`, nebo `NUMBER` se uvádějí bez závorek, které by specifikovaly jejich velikost.

Vstupní proměnné jsou v PL/SQL implicitně předávány odkazem. Proto se v PL/SQL bloku nedá za běžných okolností měnit hodnota těchto proměnných. Za klíčovým slovem AS (nebo IS) pak můžeme specifikovat lokální proměnné jak to bylo popsáno v kapitole 18.5.1.

Příklad 18.3. (Pojmenovaná procedura) Tato pojmenovaná procedura vkládá email studenta jehož login předáváme jako parametr.

```
CREATE OR REPLACE PROCEDURE
InsertEmail(p_login VARCHAR2)
AS
  v_email VARCHAR2(60);
BEGIN
  SELECT email INTO v_email
   FROM ins.student WHERE login=p_login;
  INSERT INTO email VALUES(v_email);
END;
```

[5] Oracle Academy: Oracle Database 10g: Advanced PL/SQL, Student Guide. str. 26.

Uloženou proceduru pak můžeme (pokud máme právo přístupu) spustit příkazem `execute`.

```
EXECUTE InsertEmail('jan440');
```

Uloženou proceduru v klientském prostředí (jako je Oracle SQL Developer) vytvoříme spuštěním kódu začínajícím `CREATE` a končícím `END;`. Takto spuštěná procedura je přeložena a uložena v databázi. Při překladu se mohou objevit chyby, které zobrazíme v Oracle SQL Developeru v okně log.

Funkce

Funkce (přesněji řečeno pojmenované funkce) jsou velmi podobné procedurám. Oproti procedurám specifikují návratový typ a musí vracet hodnotu.

```
CREATE [OR REPLACE] PROCEDURE jmeno_procedury
  [(jmeno_parametru [mod] datovy_typ, ... )]
RETURN navratovy_datovy_typ
IS|AS
  definice lokálních proměnných
BEGIN
  tělo procedury
END [jmeno_procedury]
```

Příklad 18.4. (Funkce) Tato pojmenovaná funkce vrací email studenta jehož login předáváme jako parametr. Datový typ vstupního parametru `p_login` definujeme odkazem na datový typ atributu `login` tabulky `student` (student.login%TYPE). Podobně definujeme i datový typ lokální proměnné `v_email` a návratovou hodnotu celé funkce.

```
CREATE OR REPLACE FUNCTION
GetStudentEmail( p_login IN student.login%TYPE)
RETURN student.email%TYPE AS
 v_email student.email%TYPE;
BEGIN
 SELECT email INTO v_email FROM student
  WHERE login = p_login;
 RETURN v_email;
END GetStudentEmail;
```

Uloženou proceduru pak můžeme (pokud máme právo přístupu) spustit příkazem `execute`.

```
SET SERVEROUTPUT ON;
EXECUTE dbms_output.put_line(GetStudentEmail('sob28'))
    ;
```

Prvním příkazem povolujeme standardní výstup serveru a druhým příkazem voláme uloženou funkci a vypisujeme výsledek.

18.5.3 Základní řídící konstrukce

V PL/SQL můžeme použít několik základních řídících konstrukcí jako je podmínka a cyklus. Jejich syntaxe se výrazně neliší od podobných konstrukcí v jiných programovacích jazycích.

Podmínka má následující syntaxi:

```
IF podminka1 THEN
příkazy
[ELSIF podminka2 THEN příkazy]
[ELSE příkazy]
END IF;
```

V zásadě máme k dispozici tři druhy cyklů. První typ cyklů se ukončuje pomocí klíčového slova `EXIT`. Podmínka ukončení cyklu může být zapsána pomocí `EXIT WHEN podminka`. Takto můžeme zapsat cyklus s podmínkou na konci.

```
LOOP
  příkazy cyklu
  [EXIT; | EXIT WHEN podminka;]
END LOOP;
```

Dalším typem je cyklus s podmínkou na začátku.

```
WHILE podminka LOOP
  příkazy cyklu
END LOOP;
```

Posledním běžně používaným typem cyklu je cyklus `FOR`, kde předem známe počet iterací. Proměnná `value1` představuje výchozí hodnotu proměnné `jmeno_promenne` a `value2` koncovou hodnotu.

```
FOR jmeno_promenne IN [REVERSE] value1..value2 LOOP
  příkazy cyklu
END LOOP;
```

18.5.4 Kurzory

Kurzory jsou pomocné proměnné vytvořené po provedení nějakého SQL příkazu. Tyto pomocné proměnné nám umožní procházet výsledek příkazu. Existují dva typy kurzorů:

- Implicitní kurzor – vytváří se automaticky po provedení příkazů jako INSERT, DELETE, nebo UPDATE.
- Explicitní kurzor – definuje se již v definiční části procedury podobně jako proměnná. Takový kurzor je často spojen s příkazem SELECT, který vrací více než jeden řádek.

Explicitní kurzor

Definice explicitního kurzoru má následující syntaxi:

```
CURSOR jmeno_kursoru IS prikaz_select;
```

Kde `prikaz_select` vrací množinu záznamů. Pomocí kurzoru můžeme postupně procházet jednotlivé záznamy výsledku SELECT příkazu a pracovat s jednotlivými hodnotami výsledku. V každém kroku programu ukazuje kurzor pouze na jeden záznam výsledku.

Práce s kurzorem probíhá pomocí následujících příkazů:

- `OPEN jmeno_kurzoru` – otevírá kurzor. Prakticky provádí SQL příkaz spojený s kurzorem a nastaví kurzor na první záznam výsledku.
- `FETCH jmeno_kurzoru INTO promenna_zaznam` – načítá aktuální záznam kurzoru do proměnné `promenna_zaznam` a posune se na další záznam.
- `CLOSE jmeno_kurzoru` – zavírá kurzor.

Příklad 18.5. (Explicitní kurzor) V následujícím příkladu vytváříme kurzor, který prochází všechna příjmení v tabulce Student.

```
DECLARE
  CURSOR c_surname IS SELECT * FROM Student;
  v_record Student%ROWTYPE;
  v_tmp INTEGER := 0;
BEGIN
  OPEN c_surname;
```

```
  LOOP
    FETCH c_surname INTO v_record;
    EXIT WHEN c_surname%NOTFOUND;
    v_tmp := c_surname%ROWCOUNT;
    DBMS_OUTPUT.PUT_LINE(v_tmp || v_record.surname);
  END LOOP;
  CLOSE c_surname;
END;
```

O něco jednodušší je práce s kurzorem při použití cyklu `FOR`, v tomto případě nemusíme kurzor explicitně uzavírat pomocí `CLOSE`.

Příklad 18.6. (Explicitní kurzor a `FOR` cyklus) V tomto příkladu provádíme stejnou operaci jako v příkladu 18.5, ale s použitím cyklu `FOR`.

```
DECLARE
  CURSOR c_surname IS SELECT surname FROM Student;
  v_surname Student.surname%TYPE;
  v_tmp NUMBER := 0;
BEGIN
  FOR one_surname IN c_surname LOOP
    v_tmp:= c_surname%ROWCOUNT;
    v_surname := one_surname.surname;
    DBMS_OUTPUT.PUT_LINE(v_tmp || ' ' || v_surname);
  END LOOP;
END;
```

V následujícím výpisu vidíme variantu kurzoru s cyklem FOR, kde nedeklarujeme proměnnou typu CURSOR.

```
DECLARE
  v_surname Student.surname%TYPE;
  v_tmp NUMBER := 0;
BEGIN
  FOR one_surname IN (SELECT surname FROM Student)
     LOOP
    v_tmp := v_tmp + 1;
    v_surname := one_surname.surname;
    DBMS_OUTPUT.PUT_LINE(v_tmp || ' ' || v_surname);
  END LOOP;
END;
```

Přestože cyklus FOR má na první pohled jednodušší zápis a rovněž nemusíme myslet na uzavíraní příkazem CLOSE, nebyl dříve doporučen při práci s větším množstvím záznamů; FETCH se totiž prováděl po jednotlivých záznamech. Doporučovalo se využít explicitní kurzor a číst více záznamů současně do pole (FETCH BULK INTO). Od verze Oracle 10g se na pozadí načítají záznamy po 100 do bufferu. Vzhledem k jednoduššímu zápisu, je tedy vhodnější použít u novějších verzích SŘBD Oracle cyklus FOR.

18.5.5 Výjimky

Jazyk PL/SQL nabízí vlastní mechanismus pro zpracování výjimek. Výjimka je chyba, která se vyskytne během provádění PL/SQL kódu. Výjimka může vzniknout jak v samotném Oracle serveru (chyba provádění nějakého SQL dotazu), tak může být vytvořena samotným PL/SQL kódem. Výjimku můžeme v PL/SQL bloku zpracovat, nebo propagovat výše. Dále se budeme zabývat jen zpracováním výjimek v PL/SQL bloku.

K tomuto účelu slouží část EXCEPTION. V případě chyby program automaticky skočí do této části, konkrétně do části, která zpracovává danou výjimku (pokud taková část existuje). V případě úspěšného zpracování se výjimka již dále nepropaguje do částí aplikace, které PL/SQL blok volaly.

```
  ...
BEGIN
  ...
EXCEPTION
  WHEN jmeno_vyjimky THEN
    zpracování vyjímky
END;
```

V případě že chceme zpracovat jakoukoli výjimku (kromě těch co již jsou zpracovány jinými WHEN příkazy), pak namísto `jmeno_vyjimky` dáme klíčové slovo OTHERS. V tabulce 18.2 vidíme některé běžné výjimky z balíku STANDARD.

Příklad 18.7. (Zachycení výjimky)

Procedura 18.1 vypíše zprávu 'Hodnota atributu login musí být unikátní!' v případě výjimky `DUP_VAL_ON_INDEX`. V případě jiné výjimky vypíše chybovou zprávu dané chyby.

Jméno výjimky	Číslo chyby	Popis
ACCESS_INTO_NULL	ORA-06530	Pokus o přiřazení hodnoty do neinicializovaného objektu
DUP_VAL_ON_INDEX	ORA-00001	Pokus vložit duplicitní hodnotu
INVALID_CURSOR	ORA-01001	Neplatná operace s kurzorem
INVALID_NUMBER	ORA-01722	Selhala konverze čísla na řetězec
NO_DATA_FOUND	ORA-01403	Příkaz SELECT nevrátil data
TOO_MANY_ROWS	ORA-01422	Příkaz SELECT INTO vrátil více než jeden řádek
VALUE_ERROR	ORA-06502	Chybná manipulace s hodnotou

Tabulka 18.2: Seznam některých výjimek balíku STANDARD

```
BEGIN
  Insert into student (login, fname, lname)
    values ('bon007', 'James', 'Bond');
EXCEPTION
  WHEN DUP_VAL_ON_INDEX THEN
    DBMS_OUTPUT.put_line ('Hodnota atributu login musí
       být unikátní!');
  WHEN OTHERS THEN
    DBMS_OUTPUT.put_line (DBMS_UTILITY.
       FORMAT_ERROR_STACK);
END;
```

Výpis 18.1: Zachycení vyjímky

PL/SQL umožňuje vyvolat výjimku v případě chyby. K tomuto účelu se používá klíčové slovo RAISE. Je možné tak vyvolat standardní nebo uživatelem definovanou výjimku.

Výjimka definovaná uživatelem

Stejně jako proměnné nebo kurzory je možné v PL/SQL bloku definovat také výjimku. Výjimka se tedy deklaruje spolu s ostatními proměnnými. Syntaxe je následují:

```
jmeno_vyjimky EXCEPTION;
```

```
DECLARE
  too_many_records EXCEPTION;
  v_records INT;
BEGIN
  Select count(*) into v_records from student;
  IF v_records > 20 THEN
    raise too_many_records
  ELSE
  Insert into student (login, fname, lname)
    values ('bon007', 'James', 'Bond');
 END IF;
END;
```

Výpis 18.2: Zachycení vyjímky

Rozsah platnosti výjimky je jen pro danou proceduru. Pokud budeme chtít výjimku deklarovanou v proceduře odchytit mimo tuto proceduru, pak je to možné jen s použitím OTHERS.

Příklad 18.8. Ve výpise 18.2 dojde k vyvolání výjimky `too_many_records`, která není v proceduře ošetřena. Výjimka tedy bude propagována do nadřazeného kódu.

18.5.6 Balíky

Balíky mají podobnou funkci jako knihovny v jiných programovacích jazycích. Shlukují PL/SQL objekty (funkce, procedury, proměnné, výjimky atd.) do jednoho místa. Balíky mají hned několik výhod:

- Umožňují deklarovat proměnné a výjimky která jsou dostupné i mimo proceduru či funkci.
- Vytvoření veřejných a privátních funkcí a procedur. Pomocné (privátní) PL/SQL bloky tedy bude možné volat jen z procedur a funkcí uvnitř balíku.
- Vytvoření vlastního jmenného prostoru, kde je možné mít libovolná jména PL/SQL objektů.
- Snadnější orientace v PL/SQL kódu. Výhodné zejména u větších projektů.

```
CREATE [OR REPLACE ] PACKAGE package_name IS|AS
  veřejné deklarace podprogramů, typů a proměnných
END [package_name];
```

Výpis 18.3: Syntaxe specifikace balíku

```
CREATE [OR REPLACE ] PACKAGE BODY package_name IS|AS
  privátní část balíku
END [package_name];
```

Výpis 18.4: Syntaxe specifikace balíku

Balíky jsou vytvářeny ve dvou částech. Jedna část je **specifikace balíku** a druhá **tělo balíku**. Specifikace balíku představuje veřejnou část, která je dostupná nejen uvnitř balíku a její syntaxi můžeme vidět ve výpise 18.3. Obsahuje definice veřejných proměnných, výjimek a hlaviček procedur a funkcí. Tělo balíku pak obsahuje neveřejnou část balíku a definice procedur a funkcí deklarovaných ve specifikaci balíku. Syntaxi pro definování těla balíku můžeme vidět ve výpise 18.4.

18.5.7 Triggery

Trigger je PL/SQL blok, který je spouštěn v závislosti na nějakém DML příkazu jako je INSERT, UPDATE nebo DELETE. Syntaxe je možné vidět ve výpise 18.5.

- BEFORE | AFTER | INSTEAD OF - povinná část; specifikujeme ve kterou chvíli se má trigger spouštět. Zda-li před provedením spojeného SQL příkazu, po něm nebo dokonce místo něho.
- INSERT [OR] | UPDATE [OR] | DELETE - povinná část; specifikuje SQL příkaz, který spouští trigger.
- OF `jmeno_sloupce` – trigger se spouští jen při aktualizaci atributu `jmeno_sloupce`.
- ON `jmeno_tabulky` – specifikuje tabulku na kterou se trigger váže.
- [REFERENCING OLD AS `stara_hodnota` NEW AS `nova_hodnota`] – volitelný parametr; umožňuje pojmenovat pomocí proměnné staré a nové hodnoty záznamu se kterým manipulujeme. Implicitně jsou tyto proměnné pojmenovány jako `:OLD` a `:NEW`.

```
CREATE [OR REPLACE ] TRIGGER jmeno_triggeru
{BEFORE | AFTER | INSTEAD OF }
{INSERT [OR] | UPDATE [OR] | DELETE}
[OF jmeno_sloupce]
ON jmeno_tabulky
[REFERENCING OLD AS stara_hodnota NEW AS nova_hodnota
   ]
[FOR EACH ROW WHEN (podminka)]
BEGIN
  příkazy
END;
```

Výpis 18.5: Syntaxe triggeru

```
CREATE OR REPLACE TRIGGER del_student
BEFORE DELETE ON student
FOR EACH ROW
BEGIN
  INSERT INTO hist_stud(login, name, surname)
   values(:OLD.login, :OLD.name, :OLD.surname);
END;
```

Výpis 18.6: Příklad jednoduchého triggeru

- [FOR EACH ROW [WHEN (`podminka`)]] – volitelný parametr; specifikuje, že daný trigger se má spouštět pro každý záznam, který je SQL příkazem měněn. Implicitně se trigger spouští pouze jednou pro jeden příkaz (který může měnit více záznamů). Za WHEN můžeme dále specifikovat podmínku, při jejímž splnění se má trigger spustit.

Příklad 18.9. (Trigger)

Ve výpise 18.6 je možné vidět příklad jednoduchého triggeru, který ukládá do tabulky `hist_stud` záznam odstraňovaný z tabulky `student`.

18.5.8 Statické a dynamické PL/SQL

V PL/SQL bloku nemůžeme přímo volat všechny dostupné SQL příkazy. Příkazy, které lze volat v PL/SQL přímo nazýváme **statické příkazy PL/SQL**. Mezi statické příkazy patří:

- SELECT
- INSERT
- UPDATE
- DELETE
- MERGE
- LOCK TABLE
- COMMIT
- ROLLBACK
- SAVEPOINT
- SET TRANSACTION

Je zřejmé, že mezi příkazy které nemůžeme volat přímo jsou všechny příkazy DDL.

18.5.9 Dynamické PL/SQL

Dynamické PL/SQL umožňuje sestavit a volat jakýkoli SQL příkaz (na který má uživatel právo) za běhu aplikace. Tento způsob může být zejména užitečný pokud předem neznáme přesný tvar SQL příkazu, který má být volán. Nevýhodou je, že nelze jednoduše ověřit syntaktickou správnost a sémantické vazby mezi objekty (správné datové typy, počet parametrů atd.).

Ve většině případů spouštíme dynamické PL/SQL pomocí příkazem EXECUTE IMMEDIATE. Použití tohoto příkazu lze vidět na příkladu 18.7.

Příklad 18.10. (Dynamické PL/SQL)

Ve výpise 18.7 je příklad vytvoření a odstranění tabulky s použitím příkazu EXECUTE IMMEDIATE.

18.6 Vytváření indexů

```
CREATE TABLE testi
(
```

```
  DECLARE
    v_command VARCHAR2(50);
  BEGIN
    EXECUTE IMMEDIATE 'Create table book ' ||
    '(id INT UNIQUE, name VARCHAR2(50), ' ||
    'author INT REFERENCES author(author_id))';
    v_command := 'DROP TABLE book';
    EXECUTE IMMEDIATE v_command;
  END;
```

Výpis 18.7: Příklad dynamického PL/SQL

```
  id INT NOT NULL PRIMARY KEY,
  name VARCHAR2(20),
  description VARCHAR2(20)
)

DECLARE
    i NUMBER := 1;
BEGIN
    LOOP
        INSERT INTO testi VALUES(i,'zawb' || i, i || 'zwdv');
        i := i+1;
        EXIT WHEN i>1000000;
    END LOOP;
END;



SELECT * FROM testi WHERE name='zawb456789';

CREATE INDEX index_testi_name ON testi(name)

SELECT * FROM testi WHERE name='zawb456789';

SELECT * FROM testi WHERE name='zawb456789' AND description='123456zwdv



SELECT * FROM user_objects;
```

```
drop index index_testi_name

SELECT * FROM user_objects;
```

18.7 Objektově relační model

18.7.1 Definice typu

```
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,
  birth DATE,
  tel1 NUMBER(20),
  tel2 NUMBER(20),
  www VARCHAR2(50),
  email VARCHAR2(30),
  passwd VARCHAR2(10)
) NOT FINAL NOT INSTANTIABLE;

CREATE OR REPLACE TYPE TTeacher;

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

CREATE OR REPLACE TYPE TTeacher UNDER TPerson (
  department REF TDepartment,
  room REF TRoom,
  enlistment REF TEnlistment,
```

```
  title VARCHAR2(30),
  STATIC FUNCTION authentication (login VARCHAR2, password VARCHAR2, md
);
```

18.7.2 Vytvoření instance

```
CREATE TABLE Department_table OF TDepartment (
  CONSTRAINT IO_DEPARTMENT_1 name NOT NULL,
  CONSTRAINT IO_DEPARTMENT_2 shortcut UNIQUE NOT NULL
);

CREATE TABLE Teacher_table OF TTeacher (
  CONSTRAINT IO_TEACHER_1 login UNIQUE NOT NULL,
  CONSTRAINT IO_TEACHER_2 fname NOT NULL,
  CONSTRAINT IO_TEACHER_3 sname NOT NULL,
  CONSTRAINT IO_TEACHER_4 address NOT NULL,
  CONSTRAINT IO_TEACHER_5 birth NOT NULL,
  CONSTRAINT IO_TEACHER_6 passwd NOT NULL,
  CONSTRAINT IO_TEACHER_7 department NOT NULL REFERENCES Department_tab
  CONSTRAINT IO_TEACHER_8 room REFERENCES Room_table,
  CONSTRAINT IO_TEACHER_9 enlistment NOT NULL REFERENCES Enlistment_tab
);
```

Část VIII

Administrace databázových systémů

Kapitola 19

Základy administrace SŘBD

Obsah

Cíl kapitoly:
Cílem je seznámit se základními principy a technikami při správě SŘBD.

19.1 Úvod

Dnešní nejrozšířenější SŘBD jsou robustními systémy, které vyžadují hlubší znalosti umožňující jejich efektivní spravování. V této kapitole se budeme zabývat hlavními principy a technologiemi, které se ve správě databázových systému vyskytují [23]. Pokusíme se vystihnout společné rysy a také rozdíly mezi jednotlivými SŘBD. Tento text by neměl sloužit jako manuál k těmto produktům ale spíše jako jakýsi náhled, který pak může usnadnit práci na konkrétním problému. Jde o doprovodný text do kurzu Správa databází ve kterém se omezujeme jen na tři největší hráče na databázovém trhu, kterými jsou Oracle, IBM DB2 LUW a také MS SQL Server. Neznamená to, že by například databáze jako MySQL nebo PostgreSQL byly horší, ale bylo nutné vybrat jen omezené množství SŘBD. Většina principů je využívána také v knize nezmíněných SŘBD, snad bude v budoucnu možné se zaměřit ještě na další databáze.

I když v kapitole snažíme podat široký výklad problematiky, pokud chce čtenář získat hlubší informace o jednotlivých SŘBD doporučujeme začít s knihami které připravují na certifikační zkoušky, knihami zaměřené na konkrétní SŘBD, či manuálovými stránkami.

19.1.1 Role administrátora

Role administrátora databáze se mohou být různé v závislosti na jeho schopnostech. Mohou být administrátoři, kteří nejen provádějí nasazení, konfiguraci a údržbu celého SŘBD, ale mohou se částečně podílet také na vývoji databázové aplikace (návrh a implementace procedur v SŘBD) a na ladění databáze. Někdy může administrátor hrát také roli v rozhodovacím procesu, jelikož se od něj očekává znalost produktů a možností dané SŘBD. Dobrý administrátor by měl

být na tyto možnosti připraven jelikož softwarová firma je pak také obvykle připravena: ohodnotit takového administrátora nadstandardním platem.

19.2 Hardware a topologie serveru

Při výběru vhodného hardwaru pro databázový systém hraje roli několik parametrů, které by se měly vzít v úvahu:

- výkon CPU (možnosti paralelizace, vícejádrové procesory),
- serverové klastry,
- operační systém,
- vnější úložiště.

Postupně projdeme každý z nich. Hlavní pozornost je v této kapitole věnována zejména úložišti, které bývá často hlavním parametrem efektivity a spolehlivosti hardwaru v databázových systémech.

19.2.1 CPU

Jako hlavní parametr CPU se často udává **frekvence procesoru** (angl. **CPU clock rate**). Frekvence procesoru je jakousi elementární jednotkou pro provedení instrukce. Tento parametr často nebývá srovnatelný u procesorů používající různé architektury, jelikož ty mohou na provedení několika podobných instrukcí vykonávající stejný kód použít různý počet cyklů. Dalším důležitým parametrem procesoru je velikost jeho **procesorové cache**, která může výrazně zlepšit prodlevy procesoru při práci s RAM pamětí. Vzhledem k tomu, že databázové operace pracují často s poměrně velkými objemy dat (vzhledem k velikosti procesorové cache), může menší procesorová cache znamenat výrazné zpomalení. Dnešní CPU bývají několikanásobně rychlejší než RAM. Procesorová cache je pak jediným způsobem, jak tento rozdíl vyrovnat a využít tak naplno potenciál procesoru.

Paralelizace CPU

Jedním ze způsobů, který umožňují zvýšit procesorový výkon, je využití více procesorů pro zpracování operací. Využití více procesorů ovšem nepřináší vždy

zrychlení, které by bylo úměrné počtu procesorů. Totiž ne každou operaci lze paralelizovat. Tento jev definuje tzv. **Amdahlův zákon** [2]:

Zrychlení procesu na N procesorech *S(N)* je definováno jako podíl doby běhu procesu na jednoprocesorovém počítači a doby běhu na N procesorech. Tento čas je rozdělen na čas sériových operací T_s, které nemohou být prováděny paralelně a operací, které lze paralelizovat T_p.

$$S(N) = \frac{T(1)}{T(N)} = \frac{T_s + T_p}{T_s + T_p/N}$$

U databázových operací pak může být čas sériových operací podstatnou částí celkového času, jelikož paralelizace databázových systémů ovlivňuje mnoho dalších parametrů jako je udržení konzistence databáze, možnost paralelního přístupu k datům atd.

19.2.2 Serverové klastry

Serverový klastr představuje topologii, která využívá pro vykonání databázových operací více serverů. Hlavním rozdílem oproti variantě využívající vícejádrové nebo víceprocesorové zpracování operací jsou oddělené prostředky jako je RAM a vnější paměť. Kvůli tomu bývá tato architektura označována jako **nic nesdílející rozdělení** (angl. **shared-nothing partitioning**). Jednotlivé servery drží svou část databáze (**fragment**), jsou propojeny vysokorychlostními síťovými linkami a navenek se celý klastr jeví jako jeden server.

19.2.3 Operační systém

V dnešní době existuje několik zavedených operačních systémů, které se používají pro běh databázových systémů. Je to AIX, HP UNIX, Linux, Solaris a Windows. Všechny tyto operační systémy již dnes podporují 64-bitovou architekturu, která je nezbytná, pokud nechceme mít omezenu velikost hlavní paměti na necelé 2GB. Linux je poněkud nováčkem na poli databázových řešení, ovšem většina velkých databázových systémů jako je DB2 nebo Oracle poskytuje podporu tomuto OS.

Pokud se rozhodneme používat SQL Server, jsme nuceni také použít Windows, jelikož SQL Server neposkytuje podporu dalším operačním systémům. Obecně se doporučuje pořizovat OS a databázový systém ze stejného zdroje,

kvůli snížení složitosti nastavení a redukci možnosti přehazování „horkého bramboru“ v případě, že nastanou nějaké potíže. V případě SQL Server je to tedy Windows, DB2 je spojeno s AIX a Teradata s NCR.

19.2.4 Vnější úložiště

Nejčastěji bývají jako vnější úložiště využívány magnetické disky. Magnetické disky jsou pomalé a CPU často musí čekat na provedení diskové operace. Jak bylo naznačeno v kapitole 7, počet diskových přístupů je z databázového pohledu kritický. Jakkoli sofistikované algoritmy používáme pro zpracování databázových operací, i jejich optimalizace má své limity a vhodný výběr disků může být nezbytnou součástí návrhu databáze. Nejběžnější technikou jak zvýšit efektivitu I/O operací je využití více disků najednou. Pro databázové servery se pak obvykle používají pevné disky s vyššími otáčkami (10K nebo 15K RPM) než u normálních pracovních stanic. Výhodné může být také použití většího množství menších disků (do 100GB).

Disky obecně mají dva parametry, které určují efektivitu I/O operací:

1. **Sekvenční čtení/zápis** na disk – u magnetických disků bývá až dvacetkrát rychlejší než náhodné přístupy na disk. OS i disk samotný provádí optimalizaci takového přístupu.
2. **Náhodné čtení/zápis** – u náhodných přístupů se projevuje prodleva způsobená vyhledáváním daného bloku na disku. Hlava často musí čekat na správné otočení disku. Parametr vyhledání bloku se u disků udává jako tzv. **seek time**.

V dnešní době se začínají prosazovat také tzv. solid state disky (SSD), které mají výrazně lepší poměr náhodných a sekvenčních přístupů při čtení. I když se dají v tomto ohledu ještě očekávat výrazné změny, sekvenční čtení bude vždy efektivnější než náhodné přístupy, už jen kvůli dopřednému získávání bloků (angl. **prefetching**). Náhodné přístupy se pak negativně projevují u indexů, které často přistupují k datům pomocí náhodných přístupů. V určitých případech (zejména pokud indexujeme malá data) může mít využití indexu negativní dopad, i když provádíme méně I/O operací než při sekvenčním průchodu daty.

RAID

Disková pole RAID pracují na principu, kdy jsou data rovnoměrně rozdělována na svazek RAID disků. Důvody pro využití RAID polí jsou dva:

- Zrychlení přístupu k datům – paralelizace přístupu může mít velmi pozitivní vliv na náhodné přístupy.
- Zvýšení spolehlivosti – RAID disky se používají také k redundanci, takže výpadek jednoho nebo více disků ještě neznamená ztrátu dat.

Standard RAID definuje několik typů RAID. Postupně je v krátkosti shrneme.

RAID 0 Bloky dat jsou rovnoměrně rozloženy na discích **cyklickým způsobem** (angl. **round-robin**). V závislosti na počtu disků se snižuje přístupová doba. Taková konfigurace ovšem není určena ke zvýšení spolehlivosti. Naopak, data jsou rozložena na více discích a možnost selhání některého z nich je vyšší, než pokud by data byla uložena pouze na jednom disku.

RAID 1 V případě RAID 1 je každý blok uložen dvakrát na dvou sousedních discích. Data tak zabírají dvakrát více místa, ale spolehlivost takové konfigurace je vyšší. Operace přístupu na disk mohou být v tomto případě pomalejší, a to zejména operace zápisu.

RAID 2 a 3 U RAID 0 a 1 jsou data rozložena na disky po blocích, avšak v případě RAID 2 jsou data rozložena po bitech. Data jsou dále kódována pomocí Hammingova kódu, který obsahuje možnost opravení chyb v případě výpadku nějakého disku. RAID 3 pracuje podobně jako RAID 2, ale používá rozdělení dat na úrovni bytů. Špatné parametry ve víceuživatelském prostředí vedou k tomu, že tyto varianty RAID jsou využívány jen zřídka.

RAID 4 RAID 4 opět využívá rozdělení dat na disky na úrovni bloků. Jeden disk je vyhrazen pro uložení paritního bloku, který umožňuje doplnit data v případě výpadku jednoho disku. Varianta je podobná RAID 2 a 3, ale má o dost lepší parametry při víceuživatelském přístupu díky rozdělení dat na bloky.

RAID 5 a 6 RAID 5 je variantou RAID 4, kde jsou paritní bloky distribuovány na všechny disky místo toho, aby byly umístěny jen na jednom disku k tomu určenému. Obecně je tato kombinace spolehlivosti paritních bloků a paralelizace přístupu oblíbenou variantou. RAID 6 přidává ještě druhý paritní blok, takže je spolehlivost diskového pole ještě vyšší ovšem za cenu zvýšení přístupové doby.

RAID 6 je schopen zvládnout i výpadek dvou disků zároveň. Tato varianta může být vhodná u většího počtu disků, kde je možnost selhání jednoho disku vyšší.

Kombinace RAID Běžně se používají také kombinace jednotlivých typů. Například je to RAID 1+0 (také označovaný RAID 10), který využívá duplikaci na dvojici disků a bloky jsou zároveň rozloženy na dvě takovéto dvojice. Nebo je to RAID 0+1, kde jsou bloky rozloženy na dva disky a tato dvojice je duplikovaná další dvojicí disků.

19.2.5 Vyvážení jednotlivých parametrů

Při výběru hardwaru je důležité vzájemně sladit jednotlivé komponenty počítače tak, aby spolu mohly spolupracovat bez zbytečných prodlev. Tedy aby například při databázových operacích nebylo využito průměrně pouze 10% procesorového výkonu CPU. V [23] můžeme nalézt tabulku, která přibližně stanovuje poměry některých parametrů:

Parametry	Typický poměr
GB aktivních dat databáze vs. počet CPU	100 : 1 (+/- 200 %)
GB aktivních dat databáze vs. velikost RAM v GB	25 : 1 (+/- 200 %)
počet CPU vs. počet disků	1 : 6 (+/- 400 %)

Tabulka 19.1: Poměr některých parametrů

Parametry udávané v tabulce 19.1 jsou jen velmi přibližné a mohou se zásadně měnit s časem a novými technologiemi, které do databází a světa hardwaru vstupují. Tabulka slouží jen jako orientační.

19.2.6 Minimální požadavky na běh databáze

Každé SŘBD má nějaké minimální požadavky na běh. Tyto požadavky se týkají jak samotného hardware, tak i například operačního systému a jeho konfigurage. Minimální požadavky se pak mohou zásadně měnit v závislosti na verzi SŘBD, typu instalace (enterprise, standard, atd.) nebo architektuře (x86, x64). SŘBD jako Oracle 11g, SQL Server nebo DB2 9 vyžadují obvykle alespoň 512 MB RAM a 1 GHz procesor. Počáteční diskové nároky pak závisí na nastavení databáze. Oracle například vyžaduje 1,5 GB odkládacího prostoru, 400 MB TEMP a 1,5 - 3,5 GB na Oracle_home.

19.2.7 Parametry aplikace

Z pohledu návrhu celé databáze a výběru vhodného hardware jsou důležité také parametry celé aplikace. Databázový administrátor, který se dostal do role, kdy by měl vybrat vhodný hardware pro danou aplikaci, musí mít k dispozici (nebo alespoň musí odhadnout) předpokládaný počet aktivních uživatelů v době největšího zatížení, počet transakcí za minutu, typ transakcí, předpokládaný objem dat a také neméně důležité požadavky na dostupnost databáze. Tyto parametry se pochopitelně mohou u aplikace během životního cyklu často měnit, každopádně alespoň přibližná představa by měla existovat. Vícero různými metrikami aplikace se budeme detailněji zabývat v kapitole 20.1.1.

Aplikace nejčastěji rozdělujeme podle typu transakcí nad daty na:

- OLTP (Online Transaction Processing) - velké procento krátkých a jednoduchý transakcí od velkého množství uživatelů. V takovýchto transakcích se generují převážně náhodné přístupy na disk.
- OLAP (Online Analytical Processing) - menší procento uživatelů, kteří pokládají dotazy nad velkými objemy dat. V důsledku toho v databázi převažuje sekvenční vyhledávání dat.

Jedno ze základních měřítek OLTP aplikace je průměrná velikost výsledku dotazu. U OLTP aplikací je obvyklá hodnota průměrná velikost výsledku dotazu menší než deset [39]. V případě, že máte tuto hodnotu větší než deset a předpokládali jste, že pracujete s OLTP aplikací, pak je nutné podívat se na SQL dotazy. U špatně napsané aplikace může docházet k dodatečnému filtrování dat na úrovni aplikace, což není žádoucí.

19.3 Instance vs. databáze a jejich vytváření

Instancí nazýváme jednu instanci SŘBD, na které může běžet několik databází. Instance je obvykle vázána na jeden počítač a s vypnutím počítače instance zaniká. Instance se skládá z množství procesů a paměťových datových struktur.

Databáze se skládá ze souborů na disku. Databáze tedy existuje i po vypnutí počítače, dokud nedojde k fyzickému vymazání souborů.

I když obvykle takto rozdělujeme SŘBD, tyto dvě části jsou od sebe neoddělitelné a mohou existovat ve vztahu s různou kardinalitou:

- 1:N - jedna instance obsahuje N databází. Nejjednodušší varianta, kterou se budeme zabývat nejvíce. Tuto variantu standardně nabízejí všechny relační databáze.
- M:1 - jedna databáze je rozložena na M instancí SŘBD. Mluvíme o distribuovaných databázích. Nasazují se ze dvou hlavních důvodů: vyšší dostupnost a spolehlivost databáze.
  V případě Oracle: Real Application Cluster (RAC).

Pokud na jednom počítači chceme spravovat více databází, máme dvě možnosti. Můžeme na jedné instanci vytvořit více databází nebo vytvořit pro každou databázi jednu instanci. První možnost může skýtat úskalí v podobě některých sdílených zdrojů. Na druhou stranu více instancí představuje větší úsilí administrátora a také větší nároky na běh.

SŘBD obvykle umožňují vytvořit skripty, jenž umožňují usnadnit vytvoření nové instance či databáze. Tyto skripty obsahují požadované nastavení a umožňují tak administrátorovi vytvořit několik stejných instancí na několika počítačích. Obvykle se tyto skripty dají vytvořit při instalaci první instance, ale je možné je vytvořit také dodatečně. U jednotlivých SŘBD:

- Oracle, **Database creation scripts**
- DB2, **Response file**

19.3.1 Oracle

Režimy instance a databáze při spouštění a zastavování Oracle databáze:

- **Shutdown** - instance i databáze je zastavena,
- **Nomount** - je vytvořena pouze instance, databáze je zcela nepřipojena (nemusí ani existovat),
- **Mount** - došlo k načtení kontrolního souboru databáze,
- **Open** - byly otevřeny všechny soubory databáze a je možné se k databázi připojit.

Parametry

Oracle databáze má zhruba 300 různých parametrů, jenž se dají nastavit. Převážná většina těchto parametrů ale obvykle zůstává ve výchozím nastavení a není

potřeba je měnit. Pro zobrazení parametrů a jejich hodnot je možné použít pohled `v$parameter`:

```
select name,value from v$parameter order by name
```

Druhým pohledem, ve kterém je možné najít hodnoty jednotlivých parametrů je `v$spparemeter`. Zatím co v `v$parameter` jsou hodnoty parametrů, jenž jsou aktuálně používány v instanci, pohled `v$spparemeter` zobrazuje hodnoty parametrů tak jak, jsou uloženy v souboru **spfile** na disku. Obvykle mají oba pohledy stejné hodnoty, pokud ale chceme, aby se změna parametru projevila až po startu pak nastavíme hodnotu v `v$spparemeter`. Naopak pokud chceme, aby se změna parametru projevila okamžitě, ale po restartu se vrátila na původní hodnotu, tak změníme pouze hodnotu v `v$paremeter`.

Rozeznáváme dva druhy parametrů:

- Statické - není možné jejich hodnotu v `v$paremeter` změnit za běhu databáze či instance.
- Dynamické - parametr je možné změnit v `v$paremeter` i za běhu databáze.

Hodnotu v `v$paremeter` je možné změnit vždy.

Mnoho parametrů a nastavení lze také úspěšně nastavovat a monitorovat pomocí nástroje nazvaného **Enterprise manager** (EM). Tento nástroj budeme zmiňovat zejména v souvislosti s monitorováním a laděním Oracle.

Procesy

Oracle instance se skládá z velkého množství procesů. Obecně lze procesy v Oracle rozdělit do několika kategorií:

- Uživatelský proces (user process)
- Procesy v popředí (foreground processes) - procesy běžící na serveru, které ale nejsou přímo součástí instance SŘBD.
- Procesy na pozadí (background processes) - procesy instance SŘBD. Tyto procesy jsou z pohledu běhu a administrace SŘBD nejzajímavější.

19.3.2 SQL Server

SQL Server je tvořen několika základními službami, kde ty dvě nejdůležitější jsou:

- Služba SQL Serveru - služba, která představuje jednu instanci SQL Serveru.
- Služba SQL Agent - služba, která spouští úlohy vytvořené v SQL Serveru.

SQL Server umožňuje nastavit základní parametry instance serveru v okně **SQL Server properties**, kam je možné se dostat v **MS SQL Server management studiu** (SSMS). V tomto okně můžeme nastavit základní parametry instance i databáze. Většina parametrů a jejich nastavení je diskutováno v dalších kapitolách. Nastavení, které je specifické pro SQL Server, je možnost zvýšení priority procesů v OS. To může být problematické v případě, že na serveru běží ještě další aplikace (aplikační server, emailový server, atd.). Pokud se ale jedná o počítač, který je vyhrazen jen pro běh SŘBD, pak je toto nastavení rozumnou volbou. Podobně jako u Oracle i v SQL Serveru vyžaduje nastavení některých parametrů restart.

Nastavení základních parametrů instance se provádí v okně **server properties**, pokud v object exploreru vyberete (po pravém kliknutí myši na instanci) vlastnosti vybrané instance.

Nastavení databáze je možné provést v okně **database properties**, na které se dostaneme opět v SSMS přes vlastnosti konkrétní databáze.

Většinu nastavení databáze a instance je tedy možné provést pomocí těchto dialogových oken. SQL Server ale pochopitelně umožňuje nastavit parametry i pomocí příkazů zaslaných na SQL Server. I před každým nastavením pomocí dialogového okna je možné zobrazit a případně uložit skript stiskem tlačítka.

19.3.3 DB2

Nastavení DB2 je možné provádět několika mechanismy:

- **Registry DB2** (**DB2 profile registry**) - zde je uložena velká část nastavení serveru. Změny parametrů v DB2 registrech se u některých parametrů projeví okamžitě a není potřeba instanci, nebo databázi restartovat. Registry jsou rozděleny na několik druhů, ty nejdůležitější jsou:

  - Registry instance - zde jsou nastaveny hodnoty parametrů pro danou instanci.
  - Globální registry - pokud není hodnota parametru nastavena pro konkrétní instanci, pak se použije globální hodnota parametru.
- **Proměnné prostředí** (**Enviroment variables**) - jedná se o proměnné OS.

Pro administraci DB2 se obvykle využívají dva nástroje: **Command line processor** a **Control centre**.

DB2 Administration Server (DAS) je služba, která je využívána při spouštění jednolivých příkazů nad DB2 instancemi. Tato služba je v systému vždy jedna a je nezbytná při spouštění nástrojů jako je Control centre nebo Task centre.

Databáze obsahuje dvě základní schémata databáze, kde je umístěn systémový katalog:

- SYSCAT - obsahuje pohledy, které jsou pouze pro čtení a které se aktualizují obvykle příkazy DDL.
- SYSSTA - obsahuje pohledy se statistikami objektů databáze. Tyto pohledy jsou využívány optimalizátorem.

19.4 Cache SŘBD

Cache (často také **buffer**) představuje část hlavní paměti počítače, která je SŘBD k dispozici pro odkládání dočasných proměnných, mezivýsledků a nejčastěji také bloků dat z vnějšího úložiště. Typicky existuje v SŘBD několik typů cache, které se dělí o dostupnou hlavní paměť hostitelského prostředí. Představíme si nejčastější typy cache v SŘBD:

- **Data cache** (**buffer cache**, **buffer pool**) – toto bývá nejrozsáhlejší část paměti, která slouží jako vyrovnávací paměť pro vnější úložiště.
- **Log buffer** – paměť používaná při transakčním zpracování. Jde o vyrovnávací paměť při zápisu do redo log souboru. Jelikož data musí být zapsána na disk v co nejkratším čase, je tato paměť malá.
- **Lock memory** – paměť sloužící pro kontrolu zamykání.
- **Communication memory** (**session memory**) – paměť využívaná při komunikaci s klienty.

- **Sort memory, Hash join memory** – paměť využívaná při vykonávání operace spojení SQL dotazu.

Architektura cache je specifická pro každý SŘBD. Správné nastavení cache SŘBD může výrazně zrychlit celý databázový systém. Naopak špatné nastavení může mít fatální následky na výkonnost SŘBD. Moderní SŘBD také často obsahují aplikace, které umožňují automatickou konfiguraci cache. Základním předpokladem je možnost měnit velikost cache za běhu SŘBD. V následujících kapitolách se podíváme blíže na cache konkrétních SŘBD.

19.4.1 Oracle

Oracle [31] rozděluje cache na dva druhy:

- **System Global Area (SGA)** – sdílená paměť pro všechny procesy běžící na dané instanci databázového systému. Tato paměť v sobě zahrnuje buffer cache, log buffer, lock memory a shared pool. Dále mohou být součástí SGA také java pool a streams pool.

- **Program Global Area (PGA)** – paměť vyhrazená pro jeden procesy v popředí (foreground process). Každý klient SŘBD má přiřazen právě jeden takovýto proces. Součásti PGA jsou také communication memory, sort memory a hash join memory.

Shared pool (SP) se skládá s velkého množství menších struktur. Všechny jsou spravovány automaticky. Zmíníme jen některé z nich:

- Library cache - ukládá nedávno spuštěné příkazy ve zkompilované formě. Umožňuje rychlejší opakované spouštění SQL příkazů. Vyhledávání v této cache je založeno na ASCII kódu, tedy i změna velkých a malých písmen znamená, že příkaz nebude nalezen.

- Data dictionary cache - vyrovnávací paměť pro metadata databázových objektů. Umožňuje rychlejší překlad SQL příkazů.

- PL/SQL cache - vyrovnávací paměť pro PL/SQL kód.

- Result cache - ukládá výsledky nedávno spuštěných SQL příkazů nebo PL/SQL kódu. Implicitně je vypnuta.

Stanovení optimální velikosti SP nemusí být jednoduché. Malá velikost povede k opakovanému provádění i u poměrně častých dotazů, na druhou stranu velká velikost může zpomalit vyhledávání v SP.

Oracle má dva parametry, které nastavují automatickou správu paměti:

- **automatická správa paměti** (**automatic memory management**) – pokud je automatická správa paměti zapnuta, nastavujeme pouze maximální velikost paměti, která je dostupná pro celý databázový systém a ten si již vyřeší rozdělení paměti sám.
- **automatická správa sdílené paměti** (**automatic shared memory management**) – pokud je zapnuta, nastavuje velikost SGA a zbývající práce se nechá na SŘBD.

Pokud je automatická správa paměti vypnuta, nastavujeme cache Oracle manuálně. Oracle poskytuje nástroj **memory advisor**, který poradí s nastavením cache SŘBD.

19.4.2 SQL Server

Podobně jako v ostatních databázích má i SQL Server celou řadu různých cache a předalokovaných poolů. Paměť ale zůstává převážně v režii SŘBD a nastavení paměti u SQL Serveru je omezeno především na nastavení maximální paměti alokované celé instanci. Toto nastavení je možné provést v server properties SSMS.

19.4.3 DB2

Každá databáze má svoji oblast paměti, kterou využívá. Tato oblast se nazývá **database heap**. V této paměti jsou umístěny všechny ostatní typy cache, které databáze potřebuje ke svému běhu.

Z těchto cache je důležitý zejména **buffer pool** (BP), jež představuje vyrovnávací paměť uživatelských dat. V databázi několik BP a každý tabulkový prostor (viz. kapitola 19.5) má přiřazen právě jeden. Při vytváření BP můžeme kromě velikosti BP specifikovat také počet stránek, které tvoří nejmenší velikost BP (blocked pages) a také zda-li bude BP dynamicky nastavován nástroji DB2 (doporučuje se nastavit). BP mohou být v relaci s TP mocným nástrojem, který nám například umožní donutit, aby určitá data nahrála celá do paměti, zatímco jiná velká data mají vyhrazenu spíše menší paměť. Výhoda cache se u velkých

dat stejně příliš neprojeví jelikož k zde často přistupujeme náhodně a nepravidelně, bude tedy lepší u nich to s velikostí cache nepřehánět.

19.5 Organizace dat

19.5.1 Úložné struktury (storage structures)

Většinou jde o soubory, jenž jsou uloženy na disku a které dohromady tvoří databázi. Každá databáze má několik druhů souborů, které se používají pro zajištění jejího běhu. Kromě různých logovacích souborů (kde nejdůležitější je redo log) tvoří hlavní část databáze tzv. **datové soubory**, v nichž jsou uloženy všechny logické objekty databáze. SŘBD musí při správě datových souborů řešit problémy s jejich zvětšováním a také hledáním volného místa uvnitř datového souboru. Datové soubory se totiž obvykle rozšiřují skokově, aby se předešlo častým alokacím místa na disku.

SŘBD používají s mírnými rozdíly architekturu, která je na obrázku 19.1.

- **Page** (**block**) - nejmenší jednotka, se kterou manipuluje SŘBD při čtení a zápisu dat z disku. Vždy je násobkem stránky operačního systému. Obvykle se nastavuje na hodnotu 4kB nebo 8kB.

- **Extend** - jde o množinu stránek. Obvykle jde o nejméně 8 stránek (64kB). Extend jsou přidělovány jednotlivým objektům v databázi dle nutnosti. Mluvíme o tzv. **extend management**.

Extend management řeší zejména sledování volných a používaných extend v datovém souboru. Může ale také sledovat naplnění jednotlivých extend. K tomuto účelu se používají obvykle bitová pole nebo bitmapy, které mapují stav jednotlivých extend v souboru.

Redo log vs. log - při popisu jednotlivých souborů SŘBD se používají obvykle termíny **redo log** a **log**, které mají přes podobný název dost odlišný význam. Redo log je soubor změn v databázi, který je používán při zotavení a prakticky nikdy k jej nečteme přímo. Zatímco log soubor obsahuje záznamy o různých událostech a chybách v databázi a často jej v případě neobvyklé chyby musíme přímo prozkoumat. Log souborů bývá mnoho k různým typů událostí či chyb.

Oracle

Fyzická sruktura databáze je tvořena třemi základními prvky:

- **Datové soubory** (**data files**) - fyzický prostor na disku, jenž ukládá data. V Oracle databázi jsou datové soubory spojeny s nějakým tabulkovým prostorem.
- **Redo log soubory** (**online redo log files**) - soubory, jenž se používají k zotavení databáze a k podpoře rollback transakcí.
- **Kontrolními soubory** (**conrol files**) - soubor, jenž obsahuje informace o fyzické struktuře databáze. Je prvním souborem, ke kterému se přistupuje při otevírání databáze.

Extend management může nabývat hodnot **localy managed** a **dictionary managed**. Druhá varianta je k dispozici jen kvůli zpětné kompatibilitě. Vždy používáme první (implicitní) možnost.

Oracle dále umožňuje nastavit velikosti extend (**extend allocation**). Implicitně Oracle nabízí možnost **automatic**, která zvětšuje velikost extend s rostoucí velikostí segmentů. Druhou variantou je použití uniformní velikosti extend (`UNIFORM SIZE`). Tato možnost může být výhodná, pokud víme, že v TP budou například uloženy velké segmenty. V takovém případě nastavíme velikost extend na nějakou vyšší hodnotu, čímž databázi usnadníme růst takového segmentu a režii spojenou s postupným zvětšováním extend. Stanovení uniformní velikosti extend vždy vychází z velikosti segmentu tak, aby nedocházelo ke zbytečnému plýtvání místem. K tomu dojde pokud je příliš velký uniform extend vzhledem k velikosti segmentů v TP. Zároveň by při zvětšování segmentu nemělo docházet k častému hledání volných extend, k čemuž dojde v případě, že máme příliš malé uniform extend u segmentu, který rychle roste.

Obvykle se používají ještě další typy souborů, které mohou být také spíše součástí instance než samotné databáze. Je to například **Instance Parameter File**, který uchovává nastavení SGA a procesů v pozadí. Je to soubor nezbytný pro start celé instance.

Dalším důležitým souborem databáze je tzv. **alert log**. Alert log ukládá důležité události databáze jako je změna parametrů databáze, start a zastavení databáze, chyba databáze (600). Je to obvykle první soubor, který administrátor kontroluje v případě nějaké chyby a při prvním kontaktu s databází. Je zde totiž možné vypozorovat všechny změny (z pohledu administrace), které se v databázi udály během její existence.

Automatic Storage Management (ASM) - umožňuje nahradit souborový systém OS. Administrátor dá k dispozici fyzické partition disku, celé disky či případně nějaké jiné úložné prostory (např. SAN) a vytvoří logické **ASM diskové skupiny**. Jakýkoli soubor vytvořený v rámci ASM diskové skupiny je pak rozložen na jednotlivé disky (partition) takovým způsobem, aby databáze dosáhla co největšího výkonu. Jedná se tedy o variantu řešení jako je RAID.

SQL Server

- **Primární datový soubor** - soubor jenž obsahuje informace o fyzické struktuře databáze a odkazuje se na další soubory databáze (podobně jako kontrolní soubor v Oracle). Může obsahovat také běžná uživatelská data. Přípona souboru bývá .mdf.
- **Sekundární datové soubory** - další datové soubory. Přípona souboru bývá .ndf.
- **Redo log soubory** - redo log soubory, které se používají při zotavení databáze nebo při rollback transakci. Přípona souboru bývá .ldf.

SQL Server má velikost stránky a extend nastavenu pevně. Stránka má velikost 8 kB a extend 64 kB. Oproti Oracle může být v SQL Serveru jeden extend používán více různými objekty v databázi. Mluvíme o tzv. **mixed extend**.

V primární datovém souboru bychom měli nechat pouze systémový katalog a systémové tabulky. Pro uložení objektů vytvořených běžnými uživateli bychom měli použít sekundární datové soubory.

Další log soubory SQL Serveru je možné zobrazit pomocí **Log File Viewer**, který je součástí SSMS. Tento nástroj umožňuje zobrazovat všechny log soubory.

19.5.2 Logické databázové struktury

Logické objekty jsou objekty v databázi jako tabulky, indexy nebo uložené procedury.

Databázové systémy často definují tzv. **tabulkové prostory** (tablespace), které oddělují logické struktury od úložných struktur. V případě SQL Serveru jsou tabulkové prostory nazývány **filegroups**. Mezi jednotlivými SŘBD jsou drobné rozdíly ve způsobu použití tabulkových prostorů, ale hlavním rysem je, že mohou obsahovat několik datových souborů a zároveň v nich mohou být umístěny

různé logické objekty. Celé schéma organizace dat lze vyčíst z obrázku 19.1. Tabulkové prostory tak řeší M:N vztah mezi logickými objekty a úložnými strukturami (datovými soubory).

V zásadě můžeme při vytváření databáze vystačit s těmito tabulkovými prostory, obvykle ale uživatelské segmenty rozdělujeme do různých tabulkových prostorů. Toto rozdělování probíhá na základě několika pravidel:

- Velké a malé objekty by měly být uloženy v různých TP.
- Tabulky a jim odpovídající indexy by měly být v různých TP (souvisí s předchozím bodem).
- V různých TP by měly být segmenty, jenž rostou rychle a pomalu (statické segmenty - read only).
- Ve odděleném TP od výchozí tabulky ukládáme také materializované pohledy.

Běžný uživatel, který se připojuje k databázi, pracuje jen s logickými datovými strukturami. Z principu SŘBD uživatel nemůže přistupovat k úložným strukturám (nejčastěji souborům na disku) přímo.

Datový slovník (systémový katalog) - popisuje logické a úložné struktury databáze. V podstatě se jedná o soubor tabulek a pohledů.

Oracle

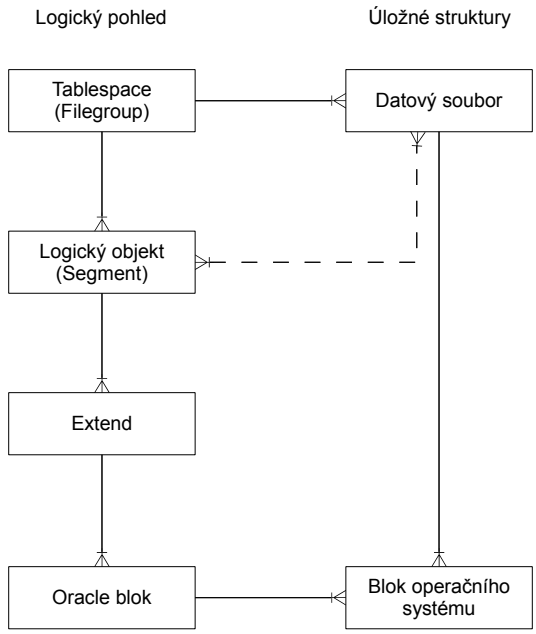
V Oracle se logický objekt nazývá také segment.

Pohledy systémového katalogu mají tři druhy prefixů:

- `USER_` - obsahuje záznamy objektů, které jsou vlastněny uživatelem pokládajícím dotaz.
- `ALL_` - obsahuje záznamy objektů, ke kterým má uživatel pokládající dotaz přístup.
- `DBA_` - obsahuje všechny záznamy objektů.

Tabulkové prostory jsou definovány několika základními parametry:

- **Jméno** - jednoznačné jméno TP.



Obrázek 19.1: Úložný model

- **Datové soubory** - udává počet datových souborů TP.
- **Typ** - udává typ objektů, jenž jsou uloženy v TP.
- **Extend management** - technika alokování extend v segmentu (v podstatě se jedná o způsob rozšiřování segmentu). Implicitní hodnota je LOCAL.
- **Segment management** - způsob vybírání bloků, do kterých se vkládají nové záznamy. Implicitní hodnota je AUTO.

TP může být z pohledu datových souborů buď SMALLFILE nebo BIGFILE. V případě, že se jedná o SMALLFILE, pak může mít tabulkový soubor více datových souborů. TP typu BIGFILE mohou být obecně větší (až 128 TB) a jejich správa může být jednodušší, jelikož jsou vždy spojeny jen s jedním datovým souborem.

Typ tabulkového prostoru může být **permanent**, **temporary** a **undo**. DBA obvykle vytváří pouze první typ TP, který slouží pro ukládání běžných objektů

jako jsou indexy a tabulky. Undo TP může být v databázi aktivní pouze jeden. Jedná se o návratový TP, který v podstatě představuje log soubor databáze. Používá se při volání rollback u transakcí nebo při použití technologie flashback.

Oracle databáze po vytvoření obsahuje několik základních TP:

- SYSTEM - jedná se o systémový TP.
- SYSAUX - jedná se o systémový TP.
- TEMP - dočasný TP sloužící převážně pro odkládání dočasných dat.
- UNDOTBS1 - návratový TP.
- USER - slouží pro uložení segmentů vytvořených uživateli.

SQL Server

Organizace logických objektů není v SQL Serveru tak striktně definována jako u Oracle. Logický objekt může být uložen ve více různých filegroup (synonymum pro tablespace v SQL Serveru, dále označujeme jen jako FG). Obvykle je jeden objekt umístěn do jednoho filegroup.

SQL Server obsahuje dva typy FG:

- **Primary** - obsahuje primární datový soubor a všechny ostatní datové soubory, které nejsou přiřazeny nějakému jinému FG.
- **User-defined** - ostatní FG v databázi.

Log soubory nejsou přiřazeny žádnému FG a jejich prostor je spravován odděleně od ostatních dat.

Jedna instance SQL Serveru obsahuje tzv. **TempDB**, což je databáze sdílená všemi databázemi běžícími na dané instanci. Jedná se o databázi pro odkládání dočasných dat, jenž se nevejdou do hlavní paměti (např. mezi výsledky dotazů, data při znovu vytváření indexů). TempDB může představovat slabé místo pokud na instanci běží několik databází, které TempDB zatěžují.

DB2

Zásobník (**Container**) úložný prostor definovaný souborem, adresářem nebo diskovým zařízením. Každý zásobník náleží právě jednomu TP a TP může mít mnoho zásobníků. Jde v podstatě o fyzické umístění datových souborů.

Základní typy TP:

- **Regular** - velikost TP je omezena na 4×10^9 řádků. Například u 8kB stránky budeme mít velikost TP omezenu na 128GB.
- **Large** - velikost TP je omezena na 10^12 řádků. U 8kB stránky bude velikost TP omezena na 4TB.
- **System temporary** - využívané SŘBD pro odkládání dočasných dat.
- **User temprary** - využívané aplikacemi pro odkládání dat.

Dále můžeme běžné TP (regular, large) rozdělit podle toho jak provádíme správu úložného prostoru:

- **Správa prostoru systémem** (System managed space - SMS) - správa úložného prostoru se přenechává na souborovém systému operačního systému. Zásobníkem může být pouze adresář a navíc nemůžeme ukládat odděleně indexy a tabulky.
- **Správa prostoru databází** (Database managed space - DMS) - správa úložného prostoru databází. Jedná se v podstatě o druh souborového systému spravovaného SŘBD. Zásobníkem může být buď soubor nebo diskové zařízení. Vyžaduje zásah DBA při zvětšování datových souborů.
- **Automatická správa** (**Automatic storage**) - správa TP je plně v režii DB2. Nevyžaduje žádný zásah DBA. Zásobník je vytvořen bez možnosti administrace.

19.6 Další nastavení databáze

19.6.1 Zabezpečení databáze

V každé databázi je nutné vytvořit také uživatele, jenž budou k dané databázi přistupovat. Uživatele mohou mít při práci s databází různá oprávnění, čímž zamezíme nechtěným chybám či dokonce pokusům o zneužitím přístupu k databázi.

V zásadě se pro přístup používají standardní řešení, která spadají spíše do oblasti identity managementu. V databázi můžeme definovat:

- **uživatele** definované jménem, heslem a případně dalšími atributy,
- **zdroje**, což mohou být v podstatě jakékoli objekty, operace či procesy v databázi,
- **oprávnění** uživatele pracovat se zdroji, které mohou být definovány nějakými dalšími atributy (můžeme například definovat způsob práce se zdroji),
- **role**, což je množina oprávnění.

Při definování zabezpečení databáze hraje do jisté míry roli velikost celé databáze a celkový počet uživatelů. Při velkém množství uživatelů můžeme identity management přenést mimo databázi - například na aplikační server nebo na samotnou aplikaci, která k databázi přistupuje. V takovém případě necháme aplikaci přistupovat přes jeden společný účet (nebo menší množství účtů), který má práva přístupu odpovídající všem uživatelům aplikace.

Oracle

Při vytvoření databáze jsou standardně vytvořeny čtyři uživatelské účty. Všechny tyto účty mají oprávnění SYSDBA:

- SYS - vlastník všech systémových tabulek a pohledů.
- SYSTEM - používá se k vytváření dalších administrativních informací.
- DBSNMP - používaný EM pro přístup ke statistikám databáze.
- **SYSMAN** - admin. Může vytvářet další administrátorské účty.

SQL Server

SQL Server umožňuje přenechat autentifikaci a správu účtů plně operačnímu systému. Autorizaci a práva již pak pochopitelně musíme řešit v rámci SŘBD. Autentifikace windows je tedy implicitní volba, která nemůže být vypnuta. Navíc můžeme zapnout autentifikaci oproti SQL Serveru, která je nutná zejména při vzdáleném přístupu k instanci.

Při vytvoření instance je vytvořen administrátorský účet, který má přiřazenu roli sysadmin.

DB2

DB2 podporuje celou řadu způsobů autentifikace. Uživatele je možné autentifikovat s pomocí OS systému klienta, DB2 serveru, Kerberos serveru, nebo GSS-API. Uživatele není možné autentifikovat přímo na úrovni databáze, DB2 vždy používá k autentifikaci nějaký externí prvek (OS, autentifikační servery).

Při autorizaci DB2 zavádí koncept autorit, což je v podstatě obecnější role. Existují autority dvojího typu: autority instance a autority databáze:

- Autority instance:
  - SYSADM - administrátor instance,
  - SYSCTRL - administrátor bez přístupu k datům.
- Autority databáze:
  - DBADM - administrátor databáze,
  - LOAD - autorita na nahrávání dat do databáze (LOAD TABLE).

19.6.2 Zotavení

Zotavení může v databázi mít mnoho podob. Obecně se jedná o uvedení databáze do konzistentního stavu v případě nějaké chyby. Chyby jednotlivých SQL příkazů či procedur jsou obvykle řešeny automaticky v rámci běhu databáze. Více o zotavení a chybách lze nalézt v kapitole 14.4. Zde se budeme zabývat především zotavením z globálních chyb, které mají za následek nekonzistenci databáze a ukončení instance. Rozlišujeme:

- Zotavení instance - nastává po systémové chybě, provádí se od posledního kontrolního bodu.
- Zotavení databáze - nastává po chybě média, provádí se od poslední zálohy datového souboru.

Při systémové chybě zpravidla nedochází ke ztrátě dat a databáze je schopna zotavení i bez přístupu k záloze. Zotavení provede ukončené transakce od posledního kontrolního bodu s využitím redo logu a případně zruší změny provedené v datovém souboru nedokončenými transakcemi. Ve většině dnešních SŘBD můžeme nastavit požadovanou **dobu zotavení** databáze v případě systémové chyby. Doba zotavení je dána především frekvencí kontrolních bodů, SŘBD se

tedy snaží provádět kontrolní body tak, aby doba zotavení nepřekročila požadovanou dobu.

Větší problém ale nastává v případě, že došlo k poškození datového souboru. V tom případě je nutné zotavení provést ze zálohy, která není aktuální. Zotavení proběhne opět s využitím redo logu, který se tentokrát musí číst od poslední zálohy (tedy ne od konrolního bodu). Záloha v kombinaci s redo logem při správném použití představuje poměrně spolehlivé zabezpečení před ztrátou dat.

Redo log soubor - je velmi vhodné mít datové soubory a redo log soubory na různých místech v databázi. Obvykle se log soubor a datový soubor umisťují na různé disky, ale na stejný počítač. Pokud by redo log byl umístěn zcela odděleně mohlo by to vést ke zpomalení databáze. Přístup k redo log souborům je velmi častý a proto by tyto soubory měly být umístěny na discích s nejrychlejším přístupem.

Archivní redo log soubor - redo log soubory mají omezenou velikost. Z toho důvodu dojde dříve nebo později k přepsání starých log záznamů, záznamy novými. To pochopitelně není problém pokud nedojde k poškození datového souboru jelikož databáze pravidelně provádí kontrolní body. Pokud bychom měli provést zotavení databáze, mohl by nastat problém s tím, že potřebné redo záznamy již byly přepsány. Z toho důvodu se provádí archivace redo log souborů. Archivace redo log souborů je obvykle výrazně rychlejší než archivace celého datového souboru a umožní nám zotavit databázi i ze starší zálohy.

Multiplexování souborů - další běžnou technikou používanou pro co největší bezpečnost dat je multiplexování kritických souborů. Soubory jsou drženy v několika identických kopiích (pozor neplést se zálohováním). Typicky multiplexujeme kontrolní soubor a redo log soubor. Jednotlivé kopie by měly být umístěny na různých discích avšak na jednom serveru.

19.6.3 Záloha dat

Záloha je nezbytnou součástí každé databáze, kde chceme předejít ztrátě dat v případě poškození datového či kontrolního souboru. Rozlišujeme dva druhy zálohy podle toho v jakém stavu se nachází databáze:

- **Offline** (**konzistentní**) - databáze je zavřena a všechna data z cache jsou uložena v databázi. Žádný uživatel se nemůže k databázi připojit.
- **Online** (**nekonzistentní**) - záloha probíhá za plného (či jen limitovaného) provozu databáze.

Zálohujeme především datové soubory a kontrolní soubory. Záloha dalších souborů pak může být zaležitost jednotlivých SŘBD. U datových souborů nemusíme vždy zálohovat celý datový soubor. Obvykle je možné zálohovat jen některé tabulkové prostory. Další běžně podporované typy zálohy jsou:

- **Plná záloha** (**full**) - je kompletní záloha, která může být použita sama o sobě ke zotavení databáze.
- **Inkrementální, rozdílová** (**incremental, differential**) - záloha při které ukládáme pouze bloky souborů změněné od poslední zálohy. Inkrementální záloha potřebuje předchozí zálohy, aby byla kompletní.

19.6.4 Záloha a zotavení jednotlivých SŘBD

Oracle

Zotavení instance probíhá v Oracle automaticky při zavolání příkazu `startup`. Proces SMON detekuje nekonzistenci hlaviček jednotlivých souborů a automaticky spouští zotavení. Zotavení se nemůžeme vyhnout.

Při nastavení doby zotavení po systémové chybě můžeme nechat implicitní hodnotu (0) nebo vyžadovat nějakou konkrétní dobu. V případě nenulové hodnoty se navíc ještě provádí optimalizace kontrolních bodů tak, aby se doba zotavení co nejvíce zkrátila. Při optimalizaci se Oracle snaží **posunout** checkpoint tím, že uloží stránky jenž obsahují změněná data.

Redo log soubory jsou v Oracle představovány tzv. **redo log skupinami**. V jeden časový okamžik Oracle pracuje pouze s jednou skupinou. Tato skupina je aktivní. Jakmile je skupina plná, dojde k přepnutí a aktivní roli převezme další redo log skupina. Každá skupina může být reprezentována jedním nebo více fyzickými soubory. Pokud je ve skupině více něž jeden soubor jde o multiplexované redo log soubory.

The data recovery advisor - nástroj, který umožní detekovat a pomůže vyřešit problémy se zotavením. Například umožní zjistit, které soubory byly poškozeny a jak je potřeba je nahradit.

Zálohu databáze je v Oracle možné provádět s využitím nástrojů operačního systému (kopírování souborů) nebo s pomocí Oracle nástroje `RMAN`. Záloha prováděná pomocí RMAN umožňuje větší množství nastavení (zejména inkrementální záloha) a měla by být z toho důvodu upřednostňována. Záloha se provádí s využitím RMAN skriptů, které specifikují co, jak a kam se má zálohovat. Tyto skripty je možné vyvářet a spouštět i pomocí EM.

Flash recovery area je oblast na disku, kde se implicitně ukládají následující soubory:

- RMAN záloha
- Archivní redo log soubory
- Flashback log databáze

SQL Server

SQL Server má definovány tři základních modely zotavení databáze:

- **Simple** - neprovádí se záloha redo logu. Záznamy se do redo logu navíc zapisují cyklicky, takže po určité době dojde k přepisování starých log záznamů. Z toho důvodu může v případě chyby média dojít ke ztrátě dat. V případě chyby média je obnova možná jen k poslední záloze systému.
- **Full** - systém pravidelně zálohuje log, takže je schopen v kterékoli chvíli provést zotavení k určitému bodu databáze (point-in-time recovery).
- **Bulk-logged** - je to varianta full modelu zotavení, kdy vyžadujeme větší efektivitu bulk operací. Pro většinu bulk operací se log vůbec nepoužívá.

Zálohu a zotavení je v SQL Serveru možné provádět buď přímo pomocí T-SQL příkazů (skriptů) a nebo je možné využít nastavení s pomocí SSMS. Modely definují způsob manipulace s redo log souborem a také jaké způsoby zotavení jsou podporovány v databázi. Toto nastavení tedy samo o sobě nepředstavuje žádné zabezpečení databáze. Další nastavení zálohy je možné provést pomocí plánů na údržbu databáze (maintenance plans)

DB2

Modely zotavení:

- **Crash recovery** - zotavení po systémové chybě.
- **Version recovery** - zotavení po chybě média, kdy dojde pouze k obnovení z poslední zálohy.
- **Roll-forward** - zotavení po chybě média, které kromě obnovení ze zálohy aplikuje log soubory.

Způsoby zapisování do log souborů:

- Cyklické zapisování (circular loging) - DB2 má množinu log souborů, které mohou být automaticky znovu použity, jakmile přestanou být aktivní. Podporuje crash recovery a version recovery. Při cyklickém zapisování máme dva typy log souborů:
  - Primární - redo log soubory jenž jsou využívány při normálním běhu log souborů
  - Sekundární - redo log soubory jenž jsou využity jen pokud všechny dostupné
- Archivace log souborů (archival logging) - archivace neaktivních log souborů. Log soubory se nikdy nepřepisují, vytvářejí se stále nové jakmile je to potřeba. Mažou se pouze archivované log soubory.

Multiplexování redo logů se v DB2 nazývá mirroring a je možné logy multiplexovat jen do dvou různých souborů.

19.6.5 Údržba databáze

Během života databáze je nutné ji neustále udržovat a sledovat. Je nutné pravidelně provádět celou řadu úkonů, které zajistí optimální běh databáze. Mezi tyto úkony patří:

- **Předělání indexů** (**index rebuild**) - efektivita indexů se může u některých dotazů zásadně měnit v závislosti na tom, zda-li jsou stránky indexu v souboru uspořádány podle klíče. Častým vkládáním a mazaním se toto uspořádání nevyhnutelně poruší, což degraduje výkon indexu. Předělání indexu pak opět setřídí stránky v souboru podle klíče. Obvykle lze předělání indexu provádět za plného provozu databáze.
- **Kontrola integrity databáze** (**checking database integrity**) - kontroluje integritu systémových i uživatelských dat databáze.
- **Aktualizace statistik databáze** (**updating index statistics**) - statistiky tabulek a indexů se díky vysoké režii neupravují s každým vložením a mazáním. Statistiky každého dynamického objektu tedy postupně zastarají a je nutné je pravidelně aktualizovat, aby se optimalizátor mohl rozhodnout na základě aktuálních dat.

- **Záloha dat** (**database backup**) - umožní obnovení dat i při poškození datového souboru. Více v kapitole 19.6.2.

Většina úkonů představuje znatelnou zátěž databáze a serveru. Například záloha, či předělání indexu znamená postupné načtení a opětovného zapsání velkého množství dat. Je potřeba tedy tyto úkony provádět mimo hlavní zátěž serveru. Moderní SŘBD poskytují nástroje, které umožňují provádění těchto úkonů alespoň částečně automatizovat. V rámci SŘBD jsou vytvářeny **úlohy** (**jobs**), jejichž spuštění je naplánovano na nějakou určitou dobu a často bývá naplánováno pravidelné spouštění.

Oracle

Nástroj nazvaný **Oracle scheduler** umožňuje definovat úlohy Oracle. Tyto úkohy jsou spouštěny přímo EM, takže v případě výpadku EM nemusí být spuštěny. Scheduler používá tzv. okna (window), které specifikují určitý interval ve kterém se má daná úloha spouštět. Úkoly jsou pak spouštěny jen v těchto oknech. Úloha může být spuštěna i konkrétní čas.

V Oracle jsou definovány automatické úkoly, které aktualizují statistiky, provádějí předělání indexů, ale i analyzují informace o provozu databáze. Tyto úlohy bývají automaticky vytvořeny při vytvoření databáze.

SQL Server

V SQL Serveru lze údržbu databáze automatizovat pomocí nástoje **Maintanance Plans**, který lze nalézt v SSMS. Tento nástroj umožňuje jednoduše naplánovat úlohy, které se mají automaticky spouštět. Lze naplánovat i sekvenci úkolů, jenž se budou spouštět za sebou. Aby se úlohy mohly spouštět je potřeba mít spuštěn také službu SQL Server Agent!

DB2

Nástroj **Task center** umožňuje spouštět úlohy ve stanovenou dobu. Pro běh úloh je nezbytné mít spuštěný DAS a také je nutné mít vytvořen **Tools catalog**. Tools catalog bývá obvykle vytvořen v samostatné databázi, která se vytvoří při instalaci instance. Je možné jej umístit, ale i jinam.

DB2 pak poskytuje několik nástrojů pro údržbu databáze:

- REORGCHK - analýza tabulek a indexů.
- REORG - rebuild indexů a tabulek
- RUNSTAT - aktualizace statistik databáze

Úkoly jenž mají naplánované spuštění na určitou dobu jsou v DB2 spravovány pomocí Task center. Všechny tyto úlohy je možné naplánovta s využitím nástroje Control center.

Kapitola 20

Monitorování a ladění SŘBD

Obsah

Cíl kapitoly:
Cílem je seznámit se pokročilejšími technikami při správě SŘBD.

20.1 Monitorování databáze

20.1.1 Monitorování databáze

Při monitorování databáze sledujeme výkon databáze z pohledu určitých parametrů, které mohou být klíčové pro ladění databáze a pro detekování a nalezení

zdroje nějakého problému. Monitorování obvykle udává hodnotu nějakého parametru za nějakou dobu (čas, transakce). Obvykle monitorujeme parametry z následujících kategorií:

- Disk
- CPU
- Paměť

Disk U disku sledujeme čas v procentech, který je u disků stráven čtením a zapisováním. Tato hodnota by neměla překročit 80 - 90 %. Dalším sledovaným parametrem je **průměrná délka diskové fronty**. Ta by neměla být větší než dvojnásobek počtu disků. Průměrná délka diskové fronty může být sledována i u jednotlivých disků.

Paměť nejčastěji uváděným parametrem v souvislosti s pamětí je **buffer cache (pool) hit ratio**. Jedná se o poměr mezi počtem stránek načtených z paměti a celkovým počtem stránek požadovaných operacemi. Určité procento stránek vždy bude načítáno z vnějšího úložiště, ale snažíme se tento počet minimalizovat dostatečnou velikostí hlavní paměti, ale především dobrým návrhem aplikace. Databáze by měla mít buffer cache hit ratio vyšší než 90%. Důležité jsou ale i další parametry aplikace jelikož sekvenční procházení tabulek a třídění je problém i když probíhá jen v hlavní paměti.

Pokud detekujeme vysoká čísla u některého z monitorovaných parametrů (zvýšená disková aktivita, zvýšená zátěž CPU), pak je nutné pokusit se najít hlavní příčinu než se například rozhodneme koupit nový hardware. V zásadě bychom tyto parametry měli sledovat neustále. Doporučuje se spočítat hodnoty uvedené v tabulce 20.1 za určité období [39]. Kritická hodnota udává jakousi mezní hodnotu, kterou by OLTP aplikace neměla překročit.

Jak je z tabulky 20.1 vidět, mnoho parametrů nemá typické hodnoty a odhad vysoké hodnoty může být otázka praxe. Záleží na mnoha parametrech a někdy jsou zvýšené hodnoty nevyhnutelné. Každý administrátor by měl zpozornět zejména při zvýšených hodnotách parametru BPLRTX. Vysoký počet bloků čtených během transakce nenaznačuje nic dobrého o aplikaci. U jednoduchých transakcí, které neprovácí žádné bulk operace ani sestavování náročných reportů, by tato hodnota měla být v rámci jednotek maximálně několik desítek. U databáze se zvýšenou hodnotou BPLRTX by administrátor měl předejít řešení problémů pouze pomocí zvětšování hlavní paměti. Hlubší analýza je nutná.

Označení	Popis hodnoty	Rozumná hodnota
IREF	Poměr mezi celkovým počtem řádků které jsou čteny a počtem řádků, které jsou nakonec použity	< 10
SELX	Průměrný počet SELECT operací během transakce	< 10
DMLTX	Průměrný počet DML operací během transakce	< 4
SRTTX	Průměrný počet sort operací během transakce	
FETTX	Průměrný počet řádků vrácených transakcí	
RRTX	Průměrný počet řádků čtených během transakce	
BPLRTX	Průměrný počet bloků čtených během transakce. Počítají se všechny přístupy do data cache včetně indexů.	

Tabulka 20.1: Parametry aplikace

Oracle

Oracle obsahuje tzv. **úložiště vytížení** (**automatic workload repository - AWR**), kde se zaznamenávají změny v databázi. Toto úložiště je součástí tabulkového prostoru SYSAUX. Informace z AWR jsou pravidelně ukládány na disk v tzv. snímcích (snapshot). Tyto snímky se pak dodatečně analyzují nástrojem nazvaným Automatic database diagnostic monitor (ADDM) (více v kapitole 20.4). Ve snapshotu je tedy uložen veškerý provoz v databázi za určitý časový interval. Z každého snapshotu je možné vytvořit report, který obsahuje celou řadu podrobných informací o provozu v databázi. Každý administrátor by se měl naučit tyto reporty číst, jelikož mohou velmi pomoci při hledání příčiny jednotlivých problémů.

Varování oracle (alerts) představují velmi užitečný nástroj při monitorování databáze. S jejich pomocí můžeme nastavit celou řadu prahových hodnot různých parametrů. Po překročení těchto hodnot je v vyvolána událost (varování). Implicitně se varování pouze zaznamenávají v databázi a jsou zobrazeny při přihlášení do EM. DBA může ale nastavit i notifikace, jenž se mají v reakci na varování spouštět. Je potřeba nastavit způsob notifikace (např. email)

SQL Server

Nástrojů pro monitorování SQL Serveru je hned několik. Kromě standardních nástrojů obsažených ve windows je to **SQL Server profiler** a **Data collector**.

SQL Server profiler je nástroj, který je k dispozici již v předchozích verzích SQL Serveru. Umožňuje zachytit provoz na databázi a pak jej opětovně spustit. Zachycený provoz je uložen v tzv. **trace souborech**, které obsahují jednotlivé SQL příkazy spolu s jejich statistikami. Tyto trace soubory je možné následně spustit nebo analyzovat nástroji **ReadTrace** a **Reporter**. ReadTrace provede analýzu zachyceného trace souboru a výsledek je následně zobrazen pomocí nástroje Reporter. V zásadě jsou dvě možnosti jak vytvářet trace soubor: na straně klienta a na straně serveru. Na straně klienta se trace soubor nedoporučuje vytvářet jelikož může významně zpomalit běh a nemusí zachytit všechny příkazy. Na straně serveru jsou do trace souboru vždy zaznamenány všechny SQL příkazy a dopad na běh serveru je výrazně nižší. I tak se kvůli efektivitě doporučuje při velké zátěži pro trace soubor vyhradit disk.

Novým násrojem jenž usnadňuje monitorování provozu je Data collector. Ten umožňuje vytvářet tzv. datové kolekce (**data collection sets**) jenž mohou být tří ruzných druhů:

- Využití disku - například průměrný růst diskových souborů atd.
- Statistiky dotazu
- Aktivita serveru

Datové kolekce jsou ukládány v **datovém skladu pro správu** (management data warehouse - MDW).

DB2

Nástroje pro monitorování v DB2:

- **Snapshot monitor** - zachycuje snapshoty v pravidelných intervalech.
- **Event monitor** - zachycuje události (activity events) v databázi.
- **Activity monitor** - slouží k rychlémů odhalení problému v databázi. Provádí analýzu monitorovacích dat.

20.1.2 Problémy aplikace

Za běhu aplikace můžeme detekovat celou řadu problémů, které její běh zpomalují. Administrátor by měl být schopen detekovat zdroj a pokud je to možné

sjednat nápravu. V této části budeme mluvit o několika problémech, které se u databází vyskytují a o možnostech řešení.

Jedním z problémů jenž se často vyskytují až v produkční verzi SŘBD jsou časté **kolize zamykání** (**lock contention**). Jde o problém kdy transakce často přistupují ke stejným datům, takže dochází ke kolizím. K těmto problémům dochází zejména pokud máme dlouhé transakce, které pracují s mnoha daty. Tento problém vyplývá z vlastnotí ACID a je možné jej řešit buď snížením izolace transakcí nebo zkrácením problematických transakcí.

Oracle

Pro detekování problému s častými kolizemi zamykání poskytuje Oracle nástroj nazvaný **Database control lock manager**. Ten je možné najít po otevření EM v záložce **performance** v odkazu **Instance link**. V této záložce můžeme zobrazit zámky jenž blokují jiné transakce v běhu.

20.2 Vysoká dostupnost

Prvky vysoké dostupnosti (high-availability) jsou v podstatě obsaženy ve všech částech správného nastavení instance a databáze. Správně nastavené pravidelné zálohy, RAID, ale i rozumně nastavená přístupová práva, to vše tvoří základ dobré dostupnosti databáze. Další používaným prvkem vysoké dostupnosti je využití více serverů. Mezi nevýhody tohoto přístupu patří zejména zvýšené nároky na hardware a tedy vyšší finanční nároky. Při rozhodování zda-li využít techniky vysoké dostupnosti jenž implementují jednotlivé SŘBD vycházíme z dohodnutých hodnot v SLA (Service Level Agreement). V Tabulce 20.2 můžeme vidět příklad maximální doby nečinnosti pro různé hodnoty požadované dostupnosti. U databáze, která běží na jednom serveru budeme jen stěží dosahovat dostupnosti převyšující 98%. Měli bychom počítat s plánovanými odstávkami serveru (záloha, upgrade hardware nebo software), ale i s neplánoványmi výpadky kvůli chybě systému, či hardware.

Nyní se podíváme na možnosti, jenž nabízejí jednotlivé SŘBD.

20.2.1 SQL Server

Mezi techniky, jenž využívají pro zvýšení dostupnosti SQL Serveru patří:

Požadovaná dostupnost [%]	Očekávaná doba nečinnosti
99.995	0.5 h
99.97	2.5 h
99.8	17.5
99.5	43 h
99	88 h
98	7.3 dní
96	14.6 dní
6 * 16	Všechny neděle a 8 hodin přes noc

Tabulka 20.2: Dostupnost a odpovídající maximální doba nečinnosti

- Failover clustering
- Log shipping
- Zrcadlení (mirroring)

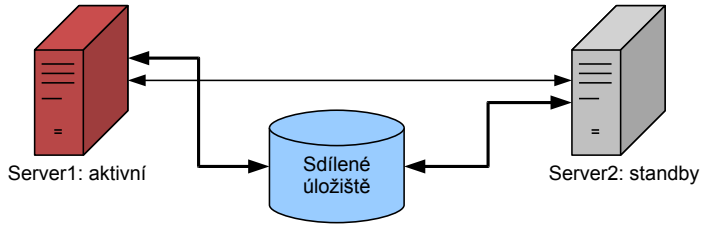
Jednotlivé techniky je možné mezi sebou kombinovat, abychom dosáhli řešení, jenž se velmi přibližuje očekávání zákazníka. Každá z technik má svá úskalí se kterými je potřeba při rozhodování počítat.

Failover clustering

Failover clustering instance - instance, která se skládá z několika nezávislých serverů (architektura shared-nothing). Jednotlivé servery mají sdílené úložiště, kde jsou datové soubory databáze. V každé chvíli je aktivní pouze jeden server. Tento server odpovídá na požadavky na databáze v instanci. Náhled na celou architekturu řešení můžeme vidět na obrázku 20.1. V případě výpadku serveru může jeho roli převzít jiný server v klastru. Tato technika nezvyšuje výkon databáze, spíše naopak.

Výhody:

- Výpadek netrvá obvykle déle než 90 sekund a k převzetí dojde zcela automaticky v rámci failover clustering instance.
- Dá se využít při plánovaných upgrade (například přidání RAM).
- Jednoduše takto zvyšujeme dostupnost všech databází v instanci. Není potřeba žádného zvláštního nastavení pro jednu databázi.



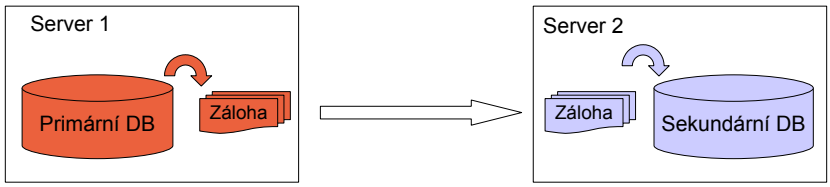
Obrázek 20.1: Failover cluster architektura

Nevýhody:

- Sdílené úložiště představuje slabé místo z pohledu ochrany před výpadkem.
- Kvůli nutnosti rychlého přístupu ke sdílenému úložišti musí být také jednotlivé servery umístěny ve stejné lokalitě.

Transaction log shipping

Tato technika slouží spíše k validaci zálohy. Není neobvyklé, že záloha vytvářená po celé měsíce bez povšimnutí selhává a k tomuto odhalení dojde teprve až dojde k výpadku. Pochopitelně pozdě. Transaction log shipping je založen na pravidelném odesílání transakčních logů, které jsou následně obnoveny na vzdálené databázi, jenž je v podstatě zpožděným obrazem té stávající. Celý postup je naznačen na obrázku 20.2.



Obrázek 20.2: Transaction log shipping process

V prvním kroku dojde k vytvoření zálohy transakčního logu primární databáze. Ten je následně odeslán na sekundární databázi, kde je po té provedeno zotavení z transakčního logu. Sekundární databáze se takto vlastně dostane do stavu v jakém byla primární databáze v době odeslání log souboru. Tento postup se opakuje v pravidelných intervalech.

Výhody:

- Máme záruku, že zálohy databáze půjde využít v případě výpadku.
- Cílovou instanci lze mezi jednotlivými přenosy log souboru využít jako read-only databázi na vytváření reportů.
- Narozdíl od Failover custering není sdílené úložiště, které by mohlo být zdrojem výpadku celého řešení.
- Sekundárních databází může být i více a mohou být teoreticky velmi daleko od primární databáze.

Nevýhody:

- Prodleva mezi zálohami log souboru.
- V případě výpadku primárního serveru je potřeba manuálně přesměrovat požadavky na kopii.

Zrcadlení databáze

Architektura je velmi podobná Log shipping. Jde tedy o nezávislé servery, jenž nemají žádný sdílený prvek jako úložiště u failover clutering řešení. Databáze jsou vždy dvě: jedna primární a druhá sekundární. Mezi databázemi existuje zakódovaný a komprimovaný stream, kde jsou záznamy log souboru zasílány v reálném čase.

Zrcadlení podporuje dva módy přenosu:

- **Vysoký výkon** - záznamy jsou přenášeny asynchronně. Sekundární databáze pak může být trochu 'pozadu'. Je k dipozici jen v Enterprise verzi.
- **Bezpečnost** - transakce není ukončena dokud není i sekundární databáze synchronní.

Servery je možné nakonfigurovat tak, že sekundární server převezme kontrolu v případě výpadku primárního. Sekundární databázi není možné přímo využít k vytváření reportů. Je to možné jen pokud vytvoříme snapshot sekundární databáze, ze kterého se pak report vytváří. To ale může představovat nepřijatelné zpoždění sekundárního serveru. Narozdíl od log shipping řešení musí zrcadlená databáze používat full recovery model. SQL Server obsahuje nástroj **Database mirroring monitor**, jenž nám může usnadnit správu zrcadlených databází.

Můžeme zde sledovat informace jako je průměrná delka fronty streamu, frekvence zasílaných transakčních záznamů, průměrné zpoždění atd.

Výhody:

- Není sdílené úložiště.
- V případě výpadku primárního serveru převezme řízení sekundární server do tří sekund.

Nevýhody:

- Servery musí být identické, aby bylo možné zrcadlit operace.
- Databáze jsou vždy maximálně dvě.

20.2.2 Oracle

V Oracle je momentálně primární technologii **Real Application Cluster** (RAC), jenž vychází z Oracle Parallel Server (OPS). RAC také používá shared-nothing architekturu se sdíleným úložištěm podobně jako failover clustering SQL Serveru. Tím ale jejich vzájemná podobnost končí. Oracle RAC podporuje také paralelizaci operací a na vykonávání jednotlivých požadavků se podílejí všechny servery v RAC. Toto řešení má řadu výhod, ale v Oracle serveru se musí řešit řada problémů, které z takového přístupu vyplývají. Hlavním problémem je synchronizace bloků v cache jednotlivých serverů. Jelikož jednotlivé servery pracují paralelně může dojít k načtení stejného bloku různými servery. V Oracle RAC se používá technika nazvaná **Cache fusion**.

U cache fusion se jedná o sdílení bloků cache mezi servery přes rychlé síťové propojení. V případě potřeby je blok přenesen z jednoho serveru na druhý místo načítání bloku z diskového uložiště. Oracle RAC pak musí mít přehled o takovýchto blocích. K tomu mu slouží charakteristiky bloku:

- **Mód** - v podstatě se jedná o přístupová práva k bloku v rámci klastru. (N - none, S - shared, X - exclusive)
- **Role** - zda-li je blok již umístěn na dalších serverech v klastru (Local, Global).

V RAC existuje adresář, kde jsou uloženy informace o sdílených blocích, který je spravován všemi servery.

20.3 Fyzický návrh databází

SŘBD poskytují obvykle několik nástrojů pro fyzickou implementaci databáze. Do fyzického návrhu zařazujeme techniky, které se týkají:

- Typů tabulek
- Typů indexů
- Materializovaných pohledů
- Partitioning

Fyzický návrh databáze v podstatě znamená, že navrhujeme způsob jakým budou data uložena na disku. Tento návrh ve většině případů nevyžaduje žádné změny v aplikaci a nemá žádný dopad na konceptuální model databáze.

Začneme tedy nejprve jednotlivými typy tabulek, jenž můžeme najít v SŘBD. U některých druhů tabulek, jde spíše o jakýsi přechod mezi tabulkou a indexem.

- **Heap tabulka** - jde o základní typ tabulky, jenž se vytvoří po zadání příkazu CREATE TABLE. Záznamy v tabulce nejsou nijak uspořádány a při vkládání je záznam vždy umístěn na první nalezenou volnou pozici v tabulce. Tento typ tabulky je velmi efektivní z pohledu DML operací a využití místa.
- **Shlukování záznamů** (**Data clustering**) – n-tice jsou v datovém souboru seřazeny podle nějakého klíče. Pro efektivní vyhledávání jsou data organizována do jakési obdoby B*stromu. Listové uzly obsahují setřízené záznamy tabulky a vnitřní uzly obsahují klíče a odkazy na potomky. Výhoda těchto indexů se projeví u dotazů, jenž provádějí třídění záznamů při zpracování tabulky (může nastat i u spojování tabulek!), ale i u běžného dotazu s BETWEEN operací. Pokud je tabulka setřízená dle hledaného atributu, pak mohou být hledané záznamy uloženy v jednom bloku (nebo v několika málo sousedních), což zrychlí provádění dotazu. V Oracle je tento typ uložení nazývá **indexově organizované tabulky** (index organized tables).
- **Hašované tabulky** (**Hash tables**, **Hash clustered tables**) - jde o způsob uložení záznamů, kde jsou záznamy se stejnou hash hodnotou uloženy ve stejném bloku, nebo ve velmi blízkém bloku. V Oracle můžeme navíc v těchto tabulkách ukládat záznamy z více různých tabulek, čímž můžeme snížit počet přístupů na disk při jejich spojování.

- **Zhmotněné dotazy** (**Materialized views**) – výsledky dotazů, které bývají často v databázi vyhodnocovány. V tomto případě už se nejedná přímo o tabulku, ale spíše o fragment z tabulky, či několika tabulek. Jde o kopii dat z jedné nebo více tabulek uspořádaných tak, aby se co nejvíce zrychlil přístup k těmto datům.

Dále uvedeme několik pojmů, které se často vyskytují v souvislosti s indexy databáze:

- **B^+-strom** – nerozšířenější typ indexu popsaný v kapitole **??**. Tento index bývá implicitním indexem při vyvolání příkazu CREATE INDEX.
- **Kompozitní index** (**composite index**) – index, kde je klíčem více než jeden atribut.
- **Pokrývající index** (**covering index**) – index, který obsahuje dostatek informací, aby pokryl dotaz bez nutnosti přistupovat do datového souboru relace, které se index týká.
- **Hustý a řídký index** (**dense and sparse index**) – hustý index má ukazatel na každý řádek relace a řídký index obsahuje ukazatele pouze na bloky (stránky) relace.
- **Bitmapový index** - index, který pro každou hodnotu h indexovaného atributu x obsahuje řádků bitů, jenž je stejně dlouhá jako je počet záznamů v tabulce. Tedy každý bit odpovídá jednomu záznamu. Bit je nastaven na jedna pokud má záznam hodnotu h v atributu x, jinak je nula. Umožňuje zásadně urychlit zejména agregační dotazy nad celou tabulkou.

Existuje několik pravidel pro vytváření indexů v databázi. Uvedeme ty nejdůležitější z nich:

1. Index se vytváří obvykle pro klíče a cizí klíče, pokud není relace výjimečně malá.
2. Atributy, které se často vyskytují v klausuli `where`, jsou potenciálními kandidáty na index.
3. Každý index znamená zvýšení počtu operací při změnách v databázi. Vytvoření nového indexu tedy musíme vždy pečlivě zvažovat.
4. Vybírejte pečlivě atribut, podle kterého se provádí shlukování záznamů v relaci. Měl by to být atribut (atributy), který odpovídá nejdůležitějšímu dotazu na relaci.

5. Pokrývající index může být užitečný, pokud pokrývá více důležitých dotazů (alespoň částečně), jinak bychom se mu měli vyhnout. Může vést až ke zpomalení některých dotazů.

6. Použití hustého indexu může být dobré v kombinaci s kompozitním indexem nebo dokonce s pokrývajícím indexem. Tedy přístup do relace u některých důležitých dotazů pak nemusí být nutný. Řídký index je pak vhodný, když má relace krátké záznamy, výška takového indexu může být o úroveň nižší. Celková velikost řídkého indexu může být tak malá, že podstatnou část indexu nebo index celý jsme schopni držet v hlavní paměti. Tímto způsobem optimalizujeme výkon indexu.

Při prvotním návrhu databáze můžeme vytvořit pouze nezbytné minimum indexů a další indexy můžeme vytvořit dalším ladění databáze. Toto pravidlo nemusí platit vždy. Někdy jsme schopni na základě zkušeností fyzickou implementaci řešit již ve fázi návrhu. Každopádně je dobré po vložení nového indexu ověřit, zda nový index nenarušil efektivitu databáze. Dnešní SŘBD obsahují nástroje na sledování efektivity databází, které sbírají nezbytné statistiky. DB2 používá tzv. **DB2 instrumentation facility**, SQL Server používá **performance monitor a PSSDIAG** a u Oracle je k dispozici **automatic workload repository (AWR)**. V případě velkých databází s množstvím operací za sekundu se takové sledování vždy vyplatí.

Každé SŘBD má své specifické datové struktury a specifické způsoby jejich nastavení. V následující kapitolách se pokusíme nahlédnout na nejdůležitější z nich.

20.3.1 Fyzická implementace v Oracle

Oracle při vytváření tabulek umožňuje nastavit celou řadu parametrů, které mohou ovlivnit zejména způsob plnění bloků a souběžný přístup. Oracle nabízí dva způsoby **správy prostoru segmentu** (**space segment management**):

- **Manuální** (**manual space segment management**) - správa prostoru segmentu se pak provádí na základě několika parametrů jako je FREELIST, PCTUSED, PCTFREE, INITTRANS.

- **Automatická** (**automatic space segment management**) - je součástí Oracle od verze 9i. Odpadá nutnost konfigurovat některé parametry. Dále již bude počítat jen s touto variantou.

I při automatické správě ale existuje několik parametrů, které mohou zásadním způsobem ovlivnit práci se segmentem:

- PCTFREE - udává hodnotu (v procentech), kdy je blok považován za plný. Každý blok může být z pohledu správy prostoru segmentu buď plný, nebo je v něm místo pro další inserty. Jakmile je překročena hranice zaplnění bloku určená parametrem PCTFREE, nedochází již dále ke vkládání do bloku. Volný prostor je vyhrazen pro případné UPDATE operace na záznamech v bloku. Pokud necháme nízkou hodnotu PCTFREE pak může dojít při UPDATE operacích k přesunu záznamů mimo blok (tzv. migrování) a ve stávajícím bloku zůstane pouze odkaz na novou pozici. K tomu dojde, pokud po UPDATE operaci není pro záznam v bloku místo. Naopak vysoká hodnota PCTFREE u záznamů, které se téměř nemění znamená, že budeme plýtvat místem.

- LOGGING / NOLOGGING - nastavuje generování redo log záznamů pro data v daném segmentu. Implicitní hodnota je LOGGING. V případě NOLOGGING můžeme generování redo log potlačit, ale pouze jen pro některé operace, jako je počáteční vytvoření tabulky, bulk-load operace s použitím SQL*Load nebo znovuvytvoření segmentu (rebuild).

- INITTRANS - každý blok obsahuje v hlavičce pole záznamů o tom, které záznamy v bloku jsou zamknuty. Nastavení INITTRANS udává jaká je počáteční velikost tohoto pole. Implicitně je hodnota nastavena na 2. Tato velikost se pak dynamicky mění podle potřeby, ale vyšší počáteční hodnota může být výhodná u dynamických segmentů.

Shlukování záznamů - indexově orientované tabulky

Indexově orientované tabulky (**index organized tables**, IOT), vynucují uspořádání záznamu na disku dle primárního klíče. Pro jejich vytvoření je potřeba přidat na konec příkazu CREATE TABLE také klíčové slovo ORGANIZATION INDEX. Oracle umožňuje nastavit dva parametry specifické pro IOT: OVERFLOW a INCLUDING. Oba paramtery souvisí se způsobem jakým jsou v listových uzlech ukládány záznamy.

Parametr OVERFLOW stanovuje maximální velikost záznamu tabulky uložený v listovém uzlu. Maximální velikost je stanovena parametrem PCTTRESHOLD xx, kde hodnota xx udává maximální velikost záznamu v procentech celkové velikosti bloku. Tedy pokud je PCTTRESHOLD 5 a blok má velikost 8kB, pak maximální velikost záznamu je 410B. Část záznamu, která se do listového bloku

nevešla, je uložena v odděleném segmentu a data v listu se na ně pouze odkazují.

Parametr INCLUDING $attr_1, attr_2, ...$ udává atributy, jenž mají být uloženy v listovém uzlu tabulky. Zbývající atributy jsou opět uloženy v odděleném segmentu.

Parametr PCTFREE je u IOT využit jen při vytváření tabulky. Udává tedy místo v blocích, které je ponecháno volné pro další vkládání.

Indexově klastrované tabulky

Tyto tabulky jsou občas zaměňovány za shlukování záznamů. Jedná se ale o dost odlišnou datovou strukturu, která má s IOT jen málo společného. U indexově orientovaných tabulek ukládáme záznamy se stejnou hodnotou atributu do stejného bloku, přičemž záznamy mohou být z různých tabulek. Hlavní výhodou je uložení dat ke kterým často přistupujeme dohromady v jednom bloku. Logicky jsou tedy tabulky odděleny, ale fyzicky jsou jejich data smíchána v jednom segmentu.

20.3.2 Partitioning

Partitioning je možnost fyzického návrhu, která umožňuje rozdělit tabulku do menších celků na základě nějakého **partitioning klíče**. Data jsou rozdělena horizontálně, tedy skupiny řádků jsou mapovány do jednotlivých partition. Základní výhodou je že partition mohou být umístěny v různých tabulkových prostorech a tedy na různých discích, či svazcích. Spolu s rozdělením tabulky jsou rozděleny i indexy na dané tabulce. Jedná se pouze o fyzické rozdělení, z logického pohledu jde stále o jeden celek. Není tedy třeba partitioning nějak zohledňovat v aplikaci. Stejně jako předchozí změny ve fyzickém návrhu také partitioning je možné provést skrytě za rozhraním databáze.

Nyní se pokusíme rozebrat jednotlivé výhody partitioning technologie:

- **Dostupnost dat** - Výpadek jedné partition ještě neznamená nedostupnost celé tabulky. SQL příkaz tedy může proběhnout pokud z chybějící partition zrovna nepracuje. Pokud optimizer může odstranit nějaké partition z plánu, tak to udělá. To se může přihodit v případě, že vyhledáváme podle partitioning klíče. Další vlastností partitioning je, že v případě poškození jedné partition může trvat zotavení této partition podstatně kratší dobu,

než by trvalo zotavení celé tabulky. Tzn. že doba výpadku je pak více uměrná poškozené části dat.

- **Administrace** - Administrativní úkoly jsou vždy jednodušeji a rychleji provedeny na menších celcích. Partitioning může být důležitý u operací jako je rebuild, či offline záloha. V případě že partitioning klíč je doba vložení, máme v jedné partition nejnovější data. V této části tabulky bude také probíhat nejvíce změn. V důsledku toho bude spíše potřebovat rebuild (nebo aktualizaci statistik) ve srovnání s ostatními partition, které jsou již relativně statické. Tyto úkony pak trvají úměrně kratší dobu. Zjednodušit se může také přesun tabulky na nové disky, či další manipulace se segmenty, která může probíhat paralelně.
- Zvýšení výkonu určitých operací.
- Může snížit kolize při vkládání a modifikacích u velmi dynamických dat.

Zvýšení výkonu může být namířeno jak na DML operace, tak na dotazování. Pokud budeme používat scénář, kde data v dané partition odpovídají nějakému intervalu (například měsíc) pak při importu dat za měsíc může být předpřipravena celá partition a ta je pak jednoduše připojena k tabulce. Takový scénář může být výhodný u datových skladů, které provádějí analýzu jednou za měsíc. Jakmile data zastarají tak jsou z databáze následně vymazána odstraněním celé partition. U menších DML operací může docházet k paralelizaci pokud jsou dotčená data napříč více partition. Co týká výkonu dotazů, ten může být ovlivněn dvěma již zmíněnými aspekty:

- Eliminace některých partition během vykonávání.
- Paralelizace vykonávání dotazů.

V zásadě bychom u OLTP databází neměli příliš spoléhat na paralelizaci výkonu při vykonávání dotazů. V OLTP aplikaci jsou data se kterými pracujeme obvykle velmi malá, takže v důsledku nutnosti procházet více menších indexů může dojít spíše ke zpomalení. Výrazně jiná situace je u datových skladů. Datové sklady pracují s velkými objemy dat a zde se může velmi pozitivně projevit inteligentní rozdělení dat do jednotlivých partition. Partitioning může naplno využít obě výhody: eliminaci a paralelizaci.

V Oracle můžeme najít tyto pruhy Partitioning:

- **Range partitioning** - definujeme intervaly dat které by měly být uloženy společně. Nejčastěji to bývá posle datumu, či časového razítka.

- **Hash partitioning** - ze sloupce (nebo sloupců) záznamu se spočítá hash funkce, která pak určuje partition. Slouží k rovnoměrnému rozložení dat.
- **List partitioning** - definujeme seznam dikrétních hodnot určitého sloupce, které pak definují jednotlivé partition. Například zkratka státu, pobočka atd.
- **Kompozitní partitioning** - kombinace před druhů partitioning.

SQL Server podporuje pouze range partitioning. Dá se ovšem říct, že je to nejčastější způsob využití partitioning technologie.

20.4 Ladění databází

Ladění databáze je postupný proces, kterému předchází plánování hardware a databáze, fyzická implementace databáze a její monitorování. Je potřeba předejít vyvozování závěru z malého množství dat a zejména monitorování je dobré provádět delší dobu.

Několik pojmů, se kterými se při ladění setkáme:

- Čas provádění (service time, CPU time) - doba strávená vykonáváním příkazů,
- Čas čekání (wait time) - doba strávená čekáním na prostředky jenž jsou potřeba k vykonání příkazu.
- Doba odezvy = Čas provádění + Čas čekání.

Automatické ladění databáze je pojem, který se v některých SŘBD objevil v posledních letech. Návrh a ladění databáze je náročný proces, který vyžaduje hlubokou znalost všech aspektů databázového stroje, jeho částí a především jistou dávku zkušeností získanou praxí. Nástroje pro automatické ladění navrhují změny v databázi na základě statistik a v synchronizaci s optimalizátorem dotazu. Nástrojů pro automatické ladění může být celá řada a každý z nich se obvykle zaměřuje na určitý typ problému. Takové nástroje mohou velmi pomoci při optimalizaci určitého problému, ale návrhář databáze by měl vidět vždy dál než nástroje, která mají k dispozici jen omezená data. Proto je dobré vždy všechna doporučení zvážit a je třeba počítat i se změnami v databázi a se změnami ve vytížení, které takový nástroj nemohl vzít do úvahy.

20.4.1 Oracle

Obecně by se daly rozdělit na nástroje, které usnadňují nastavení a údržbu celé databáze a na nástroje,které slouží k ladění SQL příkazů. Nástroje, které souvisí spíše s údržbou databáze:

- Automatic database diagnostic monitor (ADDM) - provádí analýzu AWR snapshotu po jeho uložení. Výsledkem analýzy je report, který popisuje případné obecné problémy v databázi a navrhuje možná řešení. Často odkazuje na další nástroje pro automatické ladění, které by měly problém analyzovat podrobněji.
- Automatic undo advisor - sleduje frekvenci UNDO dat a navrhuje velikost undo tabulkového prostoru.
- Mean time to recovery advisor - je schopen odhadnout dobu nutnou po zotavení po systémové chybě.
- Data recovery advisor - navrhuje postup při zotavení po chybě média.
- Segment advisor - navrhuje reorganizaci (předělání) indexů či tabulek.

Další kategorii nástrojů jsou ty zaměřené na ladění SQL:

- Automatic database diagnostic monitor (ADDM) - jedná se o poměrně obecný nástroj, takže slouží i k identifikaci problémů spojených s výkonem SQL.
- SQL access advisor - podrobná analýza všech SQL příkazů ve vytížení databáze.
- SQL tunning advisor - slouží k podrobnému ladění jednotlivých SQL příkazů.
- Database replay - umožňuje zachytit provoz na databázi a opětovně jej spustit.

SQL access advisor

Pomocí tohoto nástroje můžeme analyzovat provoz na databázi velmi komplexně. Analýza se snaží zohlednit vliv změn na všechny SQL příkazy ve vstupním vytížení. Vytížení může mít několik forem:

- Jeden SQL příkaz,
- SQL tuning set,
- obsah SQL cache.

Jelikož se jedná o poměrně náročný úkol, často je analýza časově omezena a naplánována na nějakou dobu mimo hlavní vytížení. Výsledkem běhu je report, který doporučuje změny ve fyzické implementaci databáze.

Database Replay

Pomocí tohoto nástroje lze zachytit provoz na databázi a opět jej spustit. Navíc tento nástroj umožňuje porovnat výkon jednotlivých spuštění provozu. Některý provoz není zachycen, Jedná se zejména o bulk-load operace spuštěné s pomocí SQL*Loaderu a o stream operace. Database replay má několik fází:

- Zachycení provozu - doporučuje se před začátkem restartovat Oracle.
- Předzpracování zachyceného provozu
- Znovu spuštění provozu - je možné nastavit provádění jednotlivých zachycených požadavků. Je možné spustit provoz pouze jednoho klienta, či je možné simulovat provoz více klientů.
- Analýza a sestavení reportů. Je možné využít také AWR pro porovnání výkonu.

Literatura

[1] A. Aho, J. Ullman, and J. Hopcroft. *Data Structures and Algorithms*. Addison-Wesley Series in Computer Science, 1983.

[2] G. Amdahl. *Validity of the single processor approach to achieving large scale computing capabilities*. Morgan Kaufmann Publishers Inc., 2000.

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

[4] R. Bayer and E. McCreight. Organization and Maintenance of Large Ordered Indices. *Acta Informatica*, 1(3):173–189, 1972.

[5] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*, pages 322–331. ACM Press New York, NY, USA, 1990.

[6] A. B. Chaudhri, A. Rashid, and R. Zicari. *XML Data Management: Native XML and XML-Enabled Database Systems*. Addison Wesley Professional, 2003.

[7] P. Chen. The entity-relationship model—toward a unified view of data. *ACM Transactions on Database Systems (TODS)*, 1(1):9–36, 1976.

[8] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. In *Proceedings of 23rd International Conference on Very Large Databases (VLDB'97)*, pages 426–435. IEEE, 1997.

[9] E. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[10] Databased Inteligence. dbase, `http://www.dbase.com/`, 1997.

[11] C. Date. *An Introduction to Database Systems*. Addison Wesley, 2003.

[12] S. Dietrich and S. Urban. *An Advanced Course in Database Systems: Beyond Relational Databases*. Prentice-Hall, 2004.

[13] J. Dvorský, E. Ochodková, and D. Ďuráková. *Základy algoritmizace*. Technická univerzita Ostrava, `http://www.cs.vsb.cz/kratky/courses/za/za.pdf`, 2004.

[14] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems, 5th Edition*. Addison Wesley, 2006.

[15] K. P. Eswaran, J. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19(11):624–633, 1976.

[16] H. Garcia-Molina, J. Ullman, and J. Widom. *Database systems: the complete book*. Prentice Hall, 2002.

[17] A. Guttman. R-trees: a dynamic index structure for spatial searching. *Proceedings of 1984 ACM SIGMOD International Conference on Management of Data*, 14(2):47–57, 1984.

[18] T. Härder and A. Reuter. Principles of Transaction-Oriented Database Recovery. *ACM Comp. Sur.*, 15(4), 1983.

[19] IBM. Db2, `http://www.ibm.com/db2`, 2002.

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

[21] M. Krátký and M. Beneš. Sylaby předmětu tvorba informačních systémů, `http://www.cs.vsb.cz/kratky/courses/tis/`, 2008.

[22] T. Lahdenmaki and M. Leach. *Relational Database Index Design and the Optimizers*. Wiley-Interscience; 1 edition (July 7, 2005), 2005.

[23] S. Lightstone, T. Teorey, and T. Nadeau. *Physical Database Design*. Morgan Kaufmann Publishers Inc., 2007.

[24] Y. Manolopoulos, Y. Theodoridis, and T. V.J. *Advanced Database Indexing (Advances in Database Systems)*. Springer-Verlag, 1999.

[25] J. Martin. *Information engineering, planning & analysis: book 2*. Prentice-Hall, Inc., 1990.

[26] Microsoft. Microsoft SQL Server 2005, 2002, `http://www.microsoft.com/sql/`.

[27] Microsoft. Microsoft ADO.NET – An Overview, 2008, http://msdn.microsoft.com/en-us/library/h43ks021(VS.71).aspx.

[28] Microsoft. Microsoft .NET Framework, 2008, http://msdn.microsoft.com/netframework/.

[29] I. Mlýnková, M. Nečaský, J. Pokorný, K. Richta, K. Toman, and V. Toman. *XML technologie - Principy a aplikace v praxi*. Grada, 2008.

[30] Object management group, OMG. Unified Modelling Language (UML) 2.0 , 2005, http://www.uml.org/.

[31] Oracle. Oracle Databáze 10g, 2004, http://www.oracle.com/database.

[32] Oracle. *Oracle Database PL/SQL Language Reference 11g Release 1 (11.1)*. Oracle, http://www.oracle.com/pls/db111/portal.all_books, 2008.

[33] Oracle. *Oracle Database SQL Language Reference 11g Release 1 (11.1)*. Oracle, http://www.oracle.com/pls/db111/portal.all_books, 2008.

[34] D. A. Patterson, G. Gibson, and R. H. Katz. A case for redundant arrays of inexpensive disks (raid). In *Proceedings of the 1988 ACM SIGMOD international conference on Management of data*, SIGMOD '88, pages 109–116, New York, NY, USA, 1988. ACM.

[35] J. Pokorný. *Základy implementace souborů a databází*. Karolinum, Praha, 1997.

[36] J. Pokorný. *Databázová abeceda*. Veletiny: SCIENCE, 1998.

[37] J. Pokorný. *Dotazovací jazyky*. Nakladatelství Univerzity Karlovy, Praha, 2002.

[38] D. J. Rosenkrantz, R. E. Stearns, and P. M. L. II. System level concurrency control for distributed database systems. *ACM Trans. Database Syst.*, 3(2):178–198, 1978.

[39] Scott Hayes. DB2 LUW Performance, 2009, http://www.dbisoftware.com/blog/db2_performance.php.

[40] D. Shasha and P. Bonnet. *Database Tuning*. Morgan Kaufmann Publishers, 2003.

[41] I.-Y. Song, M. Evans, and E. K. Park. A comparative analysis of entity-relationship diagrams. *Journal of Computer and Software Engineering*, 3(4):427–459, 1995.

[42] Sun Microsystem. Java DataBase Connectivity – An Overview, 2006, http://java.sun.com/javase/technologies/database/.

[43] Sun Microsystem. Java2 Enterprise Edition, 2008, http://java.sun.com/javaee/.

[44] W3 Consortium. Extensible Markup Language (XML) 1.0, W3C Recommendation, 10 February 1998, http://www.w3.org /TR/REC-xml.

[45] W3 Consortium. XQuery 1.0: An XML Query Language, W3C Working Draft, 12 November 2003, http://www.w3.org/TR/xquery/.

[46] W3 Consortium. XML Path Language (XPath) Version 2.0, W3C Working Draft, 15 November 2002, http://www.w3.org/TR/xpath20/.

[47] W3 Consortium. XML Schema, W3C Recommendation, 2 May 2001, http://www.w3.org/XML/Schema.

[48] N. Wirth. *Algorithms + Data Structures= Programs*. Prentice Hall, 1978.

[49] C. Yu. *High-dimensional indexing*. Springer-Verlag, 2002.

Příloha A

Aukční systém – analýza

Ukázkový projekt do předmětu DAIS

Informační systém aukcí

Radim Bača, Peter Chovanec, Michal Krátký

Katedra informatiky, FEI, VŠB – Technická univerzita Ostrava

http://dbedu.cs.vsb.cz/

Obsah

A.1 Obecné zásady

- Analýza neslouží k popsání co největšího množství papíru, ale má jasně a výstižně popisovat systém, který bude implementován.
- Cílem analýzy není použít co největší počet diagramů, ale úplně popsat systém, tak aby dle analýzy jej bylo možné implementovat.
- V analýze ze zásadně vyhýbáme popisu triviálních aspektů (např. triviálních operací nad číselníky) a soustřeďujeme se na komplikované části systému (např. složité funkce).
- Stejným problém jako neúplná analýza je analýza, která popisuje úplně vše (tzv. **analysis-paralysis**). V prvním případě nejsme dle analýzy schopni systém naimplementovat, ve druhém případě nejsem schopni v moři zbytečného popisu najít podstatné věci.
- Funkční analýza je stejně důležitá pro implementaci jako datový model, bez funkční analýzy nejsem schopni systém naimplementovat. Počet funkcí není roven počtu tabulek * 4 (databázové operace), v systému se může vyskytovat celá řada dalších funkcí.
- Pro popis složitých funkcí vybíráme pro danou situaci nejvhodnější nástroj (minispecifikaci, sekvenční diagram, DFD). Není možné obecně říct, že nějaký nástroj je vždy nejlepší.
- Pokud je součástí analýzy komplikovaný SQL příkaz, definujeme jej v analýze, pomůže nám doladit datový i fyzický model systému.

A.2 Specifikace zadání

PROČ? Potřebujeme informační systém pro realizaci elektronických aukcí.

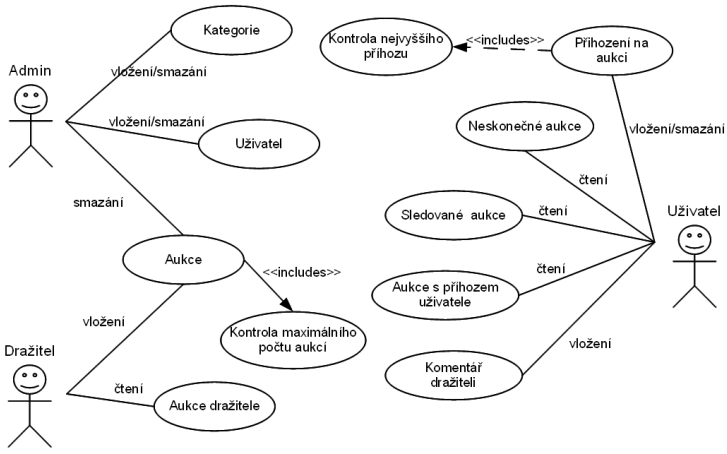
KDO? Systém budou moci používat pouze uživatelé vytvoření administrátorem. Uživatelé budou dvou kategorií. Jedna kategorie jsou uživatelé, kteří mohou pouze přihazovat (dále jen uživatel) a druhá kategorie uživatelé, kteří mohou

pouze aukce vytvářet (dále jen dražitel). Administrátor bude mít speciální práva, pro správu uživatelů, kategorií a také jako jediný bude moci rušit aukce.

VSTUPY: Údaje o jednotlivých aukcích a příhozech na ně. Dále budou moci uživatelé zadat sledování jednotlivých aukcí a okomentovat zadavatele aukce, ve které zvítězili.

VÝSTUPY: Informace o stavu jednotlivých aukcí, komentáře o zadavatelích aukcí a možnost výpisu sledovaných aukcí.

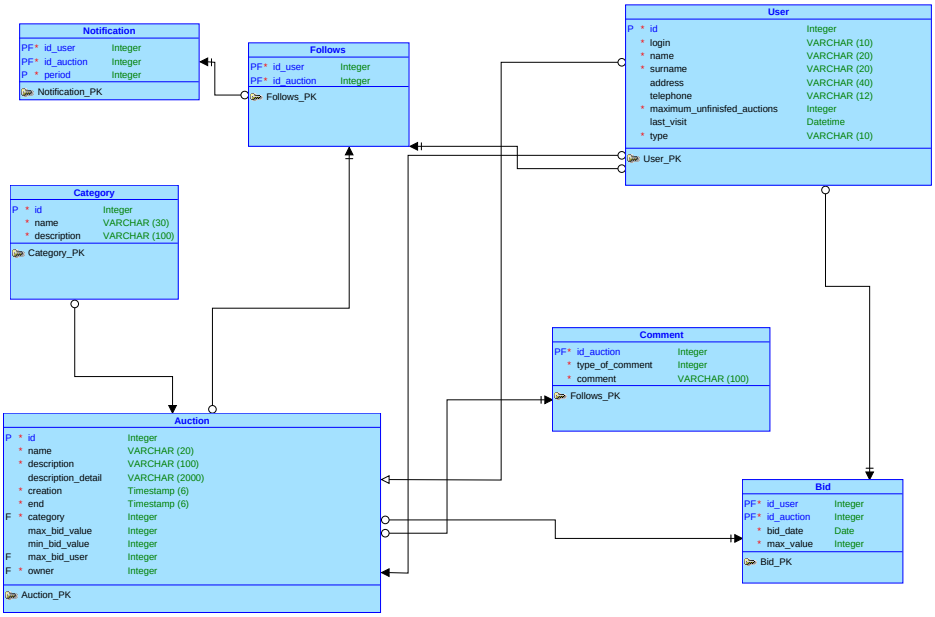
FUNKCE: Systém bude zejména řešit přihazování na aukce, kde je nutné porovnávat aktuální příhoz s maximálními částkami ostatních příhozů a příslušným způsobem nastavovat u aukce aktuální informaci o nejvyšším příhozu. Dále bude systém v pravidelných intervalech provádět kontrolu sledovaných aukcí a pokud se sledovaná aukce přiblíží ke konci, bude posílat upozornění pozorovatelům dané aukce. Systém si bude pamatovat datum poslední návštěvy uživatele a po opětovném přihlášení zvýrazní nové aukce.



Obrázek A.1: Interakce uživatele a systému

A.3 Konceptuální model

ER Diagram



Lineární zápis

Legenda: **Tabulka**, primární klíč, **cizí klíč**, atribut

User(id, login, name, surname, address, telephone, type, maximum_unfinished_auctions, last_visit)

Auction(id, name, description, description_detail, creation, end, **owner**, **category**, **id_user_max_bid**, max_bid_value, min_bid_value)

Bid(**id_user**, **id_auction**, bid_date, max_value)

Comment(**id_user**, **id_auction**, type_of_comment, comment)

Follows(**id_user**, **id_auction**)

Category(**id**, name, description)

Notification(**id_user**, **id_auction**, period)

A.4 Datový model

Popis jednotlivých tabulek je uveden v následujících tabulkách.

Tabulka **User**

	Dat. typ	Délka	Klíč	Null	Index	IO	Význam
id	Int		Primární	N	A		
login	Varchar	10		N			Login uživatele používaný při přihlašování
name	Varchar	20		N			Jméno uživatele
surname	Varchar	20		N			Příjmení uživatele
address	Varchar	40		A			Ulice a město uživatele
telephone	Number	12		A			Telefonní číslo
type	Varchar	10		N		1	Kategorie uživatele
maximum_unfinished_auctions	Integer			N		2	Maximum aukcí, které může dražitel najednou vlastnit
last_visit	Timestamp			A			Datum poslední návštěvy IS

Tabulka **Auction**

	Dat. typ	Délka	Klíč	Null	Index	IO	Význam
id	Int		Primární	N	A		
name	Varchar	20		N			Jméno aukce
description	Varchar	100		N			Krátký popis
description_detail	Varchar	2000		A			Dlouhý popis
creation	Timestamp			N		3, 4	Datum vytvoření
end	Timestamp			N		3, 4	Konec aukce
owner	Int		Cizí (User)	N			Dražitel
category	Int		Cizí (Category)	N			Kategorie do které aukce patří
max_bid_user	Int		Cizí (User)	N			Kdo má na aukci nejvyšší příhoz?
max_bid_value	Int			N			Hodnota nejvyššího příhozu na aukci
min_bid_value	Int			N			Minimální hodnota příhozu na aukci

Tabulka **Bid**

	Dat. typ	Délka	Klíč	Null	Index	IO	Význam
id_user	Int		Primární, Cizí (User)	N	A		
id_auction	Int		Primární, Cizí (Auction)	N	A		
bid_date	Timestamp					5	Čas přihození
max_value	Int						Maximální hodnota příhozu

Tabulka **Notification**

	Dat. typ	Délka	Klíč	Null	Index	IO	Význam
id_user	Int		Primární, Cizí (User)	N	A		
id_auction	Int		Primární, Cizí (Auction)	N	A		
period	Int	Primární	N	A			Počet celých hodin před koncem aukce kdy bylo uživateli zasláno upozornění

Tabulka **Comment**

	Dat. Typ	Délka	Klíč	Null	Index	IO	Význam
id_auction	Int		Primární, Cizí (Auction)	N	A		
type_of_comment	Int						
comment	Varchar	100					Obsah komentáře

Tabulka **Follows**

	Dat. typ	Délka	Klíč	Null	Index	IO	Význam
id_user	Int		Primární, Cizí (User)	N	A		
id_auction	Int		Primární, Cizí (Auction)	N	A		

Tabulka **Category**

	Dat. typ	Délka	Klíč	Null	Index	IO	Význam
id	Int		Primární	N	A		
name	Varchar	30		N			
description	Varchar	100		N			Slovní popis aukcí dané kategorie

Integritní omezení:

1. `Type` musí mít hodnotu admin, dražitel, nebo uživatel.
2. `maximum_unfinished_auctions` $<$ 100.
3. `creation` $<$ `end`.
4. `end` - `creation` $<$ 21 dní.
5. `bid_date` $<$ `#auction.end`, kde `#auction` je aukce definovaná hodnotou `id_auction` (tedy aukce na kterou přihazujeme).

A.5 Stavová analýza

Definujeme tyto stavy aukce:

- **Vytvořená** – vložená aukce.
- **Běžící** – vytvořená aukce kde datum zahájení aukce `Auction.creation` je mladší aktuálního data a zároveň je datum ukončení `Auction.end` starší aktuálního data.
- **Ukončená** – běžící aukce kde `Auction.end` <= aktuální datum. S tímto stavem je spojen typ aukce **Vítězná bez komentáře** a **Vítězná**.

Z pohledu dražitele a uživatele definujeme tyto typy aukcí:

- **Vlastní** – aukce vytvořená dražitelem (`Auction.owner`).
- **Sledovaná** – aukce je sledovaná uživatelem pokud pro uživatele a aukci existuje záznam ve `Follows`.
- **S příhozem** – aukce je s příhozem pokud pro uživatele a aukci existuje záznam v `Bid`.
- **Vítězná bez komentáře** – ukončená aukce kde `max_bid_user` je id uživatele a pro kterou chybí záznam s `id_auction` v `Comment`.
- **Vítězná** – stejné jako předchozí, jen existuje záznam s `id_auction` v `Comment`.

A.6 Funkční analýza

A.6.1 Seznam funkcí

1. **Evidence uživatelů**
   Tabulka: User, **Zodpovědnost**: Admin

   - **1.1 Vložení uživatele**
   - **1.2 Aktualizace uživatele**
   - **1.3 Zrušení uživatele**
   - **1.4 Seznam uživatelů** s možností vyhledání dle různých atributů
   - **1.5 Detail uživatele**
     Zodpovědnost: Admin; Uživatel a Dražitel pouze svůj záznam

     Pro uživatele budou zobrazeny hodnoty záznamu a počet aukcí kde:
     - je dražitelem,
     - které sleduje,
     - kde přihodil,
     - je vítězem bez komentáře,
     - je vítězem s komentářem.

     Tyto hodnoty budou vracet jednoduché funkce obsahující SQL dotazy dle definovaných stavů (viz Kapitola A.5).

2. **Evidence aukcí**
   Tabulka: Aukce, Category, User

   - **2.1 Nová aukce**
     Zodpovědnost: Dražitel
   - **2.2 Aktualizace aukce** – aktualizovat je možné jen některé aukce
     Zodpovědnost: Dražitel
   - **2.3 Smazání aukce**
     Zodpovědnost: Admin – aktualizovat je možné je některé aukce, bude ale možné použít CASCADE pro zrušení aukce a všech podřízených záznamů.
   - **2.4 Detail aukce**
     Tabulky: všechny, **Zodpovědnost**: Admin, Uživatel, Dražitel
     Využívané funkce: **3. Seznam aukcí** pro výběr aukce a funkce vracející detaily k aukci z ostatních tabulek. Detail se bude lišit v závislosti na stavu aukce. Návrh formuláře je uveden v kapitole A.7.2.

3. **Seznam aukcí**

   Tabulky: všechny

   - **3.1 Neskončené aukce** – výpis všech neskončených aukcí pro vybrané kategorie. V reálném systému bychom chtěli vyhledat aukci i podle dalších atributů, např. `description`.
   - **3.2 Sledované aukce**
   - **3.3 Aukce s příhozem**
   - **3.4 Seznam vítězných aukcí bez komentáře**
   - **3.5 Seznam vítězných aukcí**
   - **3.6 Aukce dražitele**

   Jednotlivé SQL příkazy vracející seznamy jsou definovány v kapitole A.6.2.

4. **Přihazování**

   Tabulka: Bid, **Zodpovědnost**: Uživatel

   - **4.1 Nové přihození** – nové přihození na aukci se provede jen pokud je `max_value` > `min_bid_value` u dané aukce.
   - **4.2 Aktualizace přihození** – při aktualizaci je možné provádět pouze navýšení hodnoty `max_value`, `bid_date` se bude aktualizovat automaticky.
   - **4.3 Automatické přihození** – funkce je spouštěna funkcemi 4.1 a 4.2. Nastavuje se hodnota `max_bid_value`, pokud dojde k přehození maximální hodnoty na aukci. Funkce je popsána v kapitole A.6.2.
   - **4.4 Zrušení přihození** – jedná se o běžnou funkci, která ale v tomto ukázkovém projektu nebude navržena a implementována. Možnost zrušit příhoz by měl mít pouze vlastník (atribut `owner`) dané aukce.
   - **4.5 Historie příhození k aukci** – funkce bude vyvolána při zobrazení detailu aukce a bude vracet seznam přihození na danou aukci seřazených od nejaktuálnější po nejstarší přihození.

5. **Evidence kategorií**

   V reálném systému bychom mohli očekávat možnost vytvářet hierarchie kategorií, v tomto ukázkovém projektu nebudou ale hierarchie kategorií podporovány.

   V případě této funkce se jedná o práci s číselníkem, jednotlivé funkce jsou implementovány SQL příkazy nad jednou tabulkou a nebudou tady popisovány.

   Tabulka: Category, **Zodpovědnost**: Admin

- **5.1 Nová kategorie**
- **5.2 Aktualizace kategorie**
- **5.3 Zrušení kategorie**
- **5.4 Seznam kategorií**

6. **Evidence komentářů**

   Tabulka: Comment, **Zodpovědnost**: Uživatel, který zvítězil v aukci

   - **6.1 Nový komentář**
     Využívané funkce: 3.4 Seznam vítězných aukcí bez komentáře pro výběr aukce
   - **6.2 Aktualizace komentáře**
     Využívané funkce: 3.5 Seznam vítězných aukcí pro výběr aukce
   - **6.3 Zrušení komentáře**
     Využívané funkce: 3.5 Seznam vítězných aukcí pro výběr aukce
   - **6.4 Komentáře k aukci** – selekce komentáře k aukci.
     Zodpovědnost: Dražitel

7. **Ostatní funkce**

   - **7.1 Upozornění pozorovatelům** – funkce, která je spouštěna v daném intervalu, hledá aukce, které se blíží ke konci a zasílá mail pozorovatelům těchto aukcí. V ukázkovém projektu nebude řešeno posílání mailů, pouze dojde k zápisu do tabulky `Notification`. Funkce je popsána v kapitole A.6.2.

A.6.2 Detailní popis funkcí

Funkce 3. Seznam aukcí

Vstup: id aktuálního uživatele **$idBidder** pro funkce 3.2 – 3.5

- **3.1 Neskončené aukce**

```
select a.* from auction a, subcategory sb
where
  a.end < CURRENT_TIMESTAMP and sb.id=idCategory
  and sb.id=a.subcategory
order by (CURRENT_TIMESTAMP-end) asc;
```

- **3.2 Sledované aukce**

```
select a.* from auction a,follows f,subcategory sb
where a.id=f.ID_auction and f.ID_user=$idBidder
  and sb.id=idCategory and sb.id=a.subcategory
  order by (CURRENT_TIMESTAMP-end) asc;
```

- **3.3 Aukce s příhozem**

```
select a.* from auction a,bid b,subcategory sb
where a.id=b.ID_auction and b.ID_user=$idBidder
  and sb.id=idCategory and sb.id=a.subcategory
  order by (CURRENT_TIMESTAMP-end) asc;
```

- **3.4 Seznam vítězných aukcí bez komentáře**

```
select a.* from auction a, subcategory sb
where a.ID_user_max_bid=$idBidder
  and a.id not in
    (select ID_auction from Comment where ID_buyer = $idBidder)
  and sb.id=idCategory and sb.id=a.subcategory
  order by creation asc;
```

- **3.5 Seznam vítězných aukcí**

```
select a.* from auction a, subcategory sb
where a.ID_user_max_bid=$idBidder
  and a.id  in
    (select ID_auction from CommentUser
      where ID_buyer = $idBidder)
  and sb.id=idCategory and sb.id=a.subcategory
  order by creation asc;
```

- **3.6 Aukce dražitele**

  Vstup: $idOwner reprezentuje id dražitele

```
select a.* from auction a, subcategory sb
where a.owner = $idOwner
  and sb.id=idCategory
  and sb.id=a.subcategory
  order by creation asc;
```

Funkce 4.3 Automatické přihození na aukci

Při vytváření nového záznamu v tabulce `Bid` nebo při navyšování `max_value` se bude automaticky spouštět procedura provádějící korekci `max_bid_value` a případně i `max_bid_user`. Procedura bude transakce.

Vstup: `$id_user`, `$id_auction`, `$max_value`

1. Načteme `max_bid_value` aukce s `$id_auction`.
2. Pokud je `max_bid_value` této aukce `null` tak `max_bid_value` = `$max_value`.
3. Pokud je `$max_value` < `max_bid_value` tak proceduru ukončíme.
4. Nalezneme a nastavíme novou hodnotu `max_bid_value` dané aukce s pomocí SQL dotazu:

```
SELECT MAX(max_value) FROM bid
  WHERE bid.id_auction = $id_auction and
    max_value NOT IN
      (SELECT MAX(max_value) FROM bid
         WHERE bid.id_auction = $id_auction)
```

   Jedná se o nalezení druhé nejvyšší hodnoty mezi příhozy.

5. Nastavíme také `max_bid_user` dané aukce s pomocí cursoru:

```
SELECT id_user, bid_date FROM bid
  WHERE bid.id_auction = $id_auction and
    max_value IN
    (SELECT MAX(max_value) FROM bid
       WHERE bid.id_auction = $id_auction)
```

   Procedura bude procházet jednotlivé příhozy s nejvyšší nabídkou a nastaví hodnotu `max_bid_user` na uživatele jehož příhoz má nejmenší `bid_date`.

Funkce 7.1 Upozornění pozorovatelům

1. Funkce bude v SŘBD spouštěna v pravidelných intervalech, např. jednou za 30min.

2. Aktuální čas se bude během vykonání funkce měnit, což by mohlo dělat problémy. Proto před začátkem funkce uložíme aktuální čas a datum do proměnné **$p_current_datetime**, např. **$p_current_datetime** = CURRENT_TIMESTAMP.

3. Vytvoříme kurzor, který bude procházet neukončené aukce, které budou brzy končit a v tuto hodinu nebylo pozorovatelům zasláno upozornění. Vstupní proměnná **$period** obsahuje počet hodin od kterého má pozorovatel každou hodinu dostávat upozornění, např. 12h.

   Funkce **GetPeriod** vrací počet celých hodin mezi dvěma časy, např. pokud je rozdíl 45min, funkce vrátí 0, pokud 68min, funkce vrátí 1 atd.

```
SELECT * FROM Auction a, Follows f WHERE
  -- vezmeme v úvahu jen aukce, které budou brzy končit
  a.end >= $p_current_datetime AND
  GetPeriod(a.end, $p_current_datetime) > $period AND
  -- projdi všechny pozorovatele těchto aukcí
  a.id_auction = f.id_auction AND
  -- a zjisti jestli už pro tuto hodinu nebylo
  --   posláno upozornění
  AND (
    SELECT Count(*) FROM Notification
      WHERE n.id_auction = f.id_auction AND
        n.id_user = f.id_user AND n.period =
        GetPeriod(a.end, $p_current_datetime)) = 0
```

   Poznámka: Je nutné otestovat výkon dotazu, počet porovnaných záznamů části dotazu `end >= CURRENT_TIMESTAMP AND GetPeriod(a.end,` $p\_current\_datetime$) > `period` je závislá na konkrétním SŘBD. V případě nízkého výkonu (např. při sekvenčním průchodu tabulkami), je nutné změnit fyzický návrh. Musíme si uvědomit, že uzamknutí tabulek `Auction` a `Follows` na delší dobu, by mělo negativní vliv na propustnost systému, přestože tato funkce bude spouštěna v určitých intervalech.

4. Pro každý záznam vložíme záznam `(f.id_auction, f.id_user, GetPeriod(a.end, $p_current_datetime))` do tabulky `Notification` (dotaz i vložení můžeme řešit v rámci jednoho příkazu INSERT). V reálném systému by při vložení záznamu došlo k zaslání mailu.

5. Tato funkce bude rovněž automaticky mazat upozornění pro již ukončené akce (tabulka by mohla mít ohromné množství záznamů, historii upozornění není nutné evidovat).

```
DELETE FROM Notification WHERE id_auction, id_user IN (
  SELECT n.id_auction, n.id_user
    FROM Auction a, Follows f, Notification n
    WHERE
    -- vezměme jen ukončené aukce
    a.end < CURRENT_TIMESTAMP AND
    -- vezměme jen pozorovatele těchto aukcí
    a.id_auction = f.id_auction
    -- vezměme jen pozorovatele, kteří dostali upozornění
    f.id_auction = n.id_auction AND f.id_user = n.id_user)
```

A.7 Návrh uživatelského rozhraní

A.7.1 Menu

1. **Přehled aukcí** (**zodpovědnost**: Admin, Dražitel, Uživatel)

   (a) **Neskončené aukce** – akce: 3.1 Neskončené aukce

   (b) **Moje aukce** – akce: 3.6 Aukce dražitele

   (c) **Sledované aukce** – akce: 3.2 Sledované aukce

   (d) **Aukce s mým příhozem** – akce: 3.3 Aukce s příhozem

   (e) **Seznam vítězných aukcí bez komentáře** – akce: 3.4 Seznam vítězných aukcí bez komentáře

   (f) **Seznam vítězných aukcí** – akce: 3.5 Seznam vítězných aukcí

   Pro každou aukci bude nabízena akce: 2.4 Detail aukce (viz kapitola A.7.2).

   Pro aukce dražitele bude nabízena akce: 2.2 Aktualizace aukce

   Pro admina bude nabízena akce: 2.3 Smazání aukce

2. **Nová aukce** (**zodpovědnost**: Uživatel) – akce: 2.1 Nová aukce

3. **Můj profil** (**zodpovědnost**: Uživatel) – akce: 1.5 Detail uživatele

4. **Administrace** (**zodpovědnost**: Admin)

   (a) **Správa uživatelů**

      i. **Vložení uživatele** – akce: 1.1 Vložení uživatele

      ii. **Přehled uživatelů** – akce: 1.4 Seznam uživatelů
      Pro každého uživatele budou nabízeny akce: 1.2 Aktualizace uživatele, 1.3 Zrušení uživatele, 1.5 Detail uživatele

   (b) **Správa kategorií** – podobné jako správa uživatelů.

A.7.2 Detail aukce