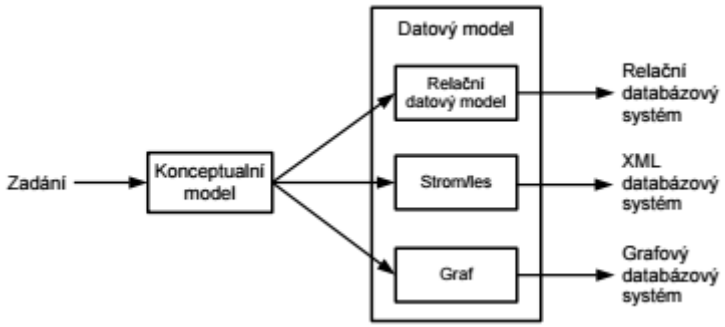
Modelování databázových systémů



Základní pojmy:

- **Entita**: objekt z reálného světa
- **Hodnoty atributů**: vlastnosti objektů z reálného světa
- **Entitní typ**: více entit stejného typu (např. osoba)
- **Atributy**: jsou nimi definované entitní typy, popisují entitní typ

Při vytváření databáze postupujeme ve více krocích (viz obrázek). Jednotlivé kroky využívají různé typy modelů pro popis databáze.

Model – množina matematických konceptů, které mohou být použity pro popis struktury databáze. Slouží jako podklad pro další řešení, je to abstraktní obraz budoucí reality. Vytváří se pro správné pochopení struktury a funkcí systému a pro dorozumění mezi uživatelem, analytikem a realizátorem.

Modelování se skládá z:

1. **Datová analýza** – výsledkem je konceptuální schéma. Návrh struktury databáze (3NF), používá se ERD.
2. **Funkční analýza** – návrh funkcí vykonávaných nad databází. Používají se DFD, minispecifikace, sekvenční diagramy
3. **Dynamická analýza**

Typické modely vytvořené při analýze:

- **Konceptuální model (logický model)** – logický popis struktury databáze, nezohledňuje zatím konkrétní databázový systém
- **Databázové schéma (relační datový model)** – popis DB definované pro konkrétní DB systém

Datová analýza

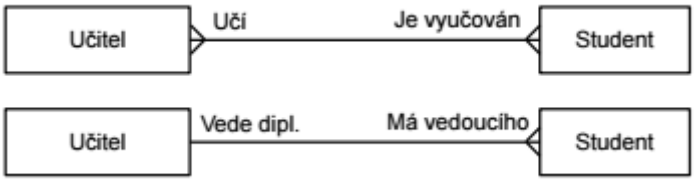
Analýza je studium problému (jeho poznání, popis, modelování) dříve, než se začne s řešením. Úkolem analýzy je zpracování několika typů modelů budoucího systému. Výsledkem datové analýzy je **konceptuální schéma.** Identifikujeme objekty, a vztahy mezi objekty a jednotlivé vlastnosti objektů. Konceptuální modelování definuje omezení kladená na data.

Konceptuální schéma z těchto získaných informací se skládá z:

- ER diagramu
- Lineární zápis entit - Pes (IDPes, jmeno, pohlavi, vek, CRasa, IDUtulek)
- Lineární zápis vztahů - NABIZI (Útulek, Pes) 1:N
- Datový slovník

- Integritní omezení

ERD – Grafické znázornění konceptuálního modelu. Neexistuje standard, takže může být několik verzí.



Datový slovník – podrobný rozpis jednotlivých diagramů, obsahuje typ atributů, velikost atributů a **integritní omezení** (omezení na formát atributů – např. PSČ, login…)

Kardinalita vztahu:

- Kardinalita 1: 1 – zaměstnanec může být vedoucím jen jednoho oddělení
- Kardinalita 1: N - zaměstnanec může být členem jen jednoho oddělení
- Kardinalita M:N - zaměstnanec může být členem několika oddělení

Funkční analýza

Funkční analýza tedy vyhodnocuje manipulaci s daty v systému. Je-li hotov návrh struktury databáze, navrhují se funkce nad ní. Funkční analýza vychází opět ze zadání IS. Je výhodné, když se v zadání vyskytují následující prvky (viz zadání):

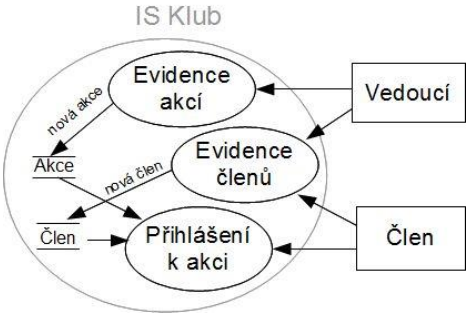
- Seznam funkčních požadavků na vnitřní chování systému nebo seznam událostí a reakcí
- požadované vstupy a výstupy

Z nich vytváří analytik funkční model:

- vnější pohled hrubý (graficky pomocí DFD)
- vnitřní pohled podrobně rozpracovává jednotlivé akce (algoritmy, minispecifikace)

DFD – data flow diagram

- diagram datových toků; DFD je grafický nástroj pro modelování funkcí a vztahů v systému.
- popisuje algoritmy systému; pro modelování vztahů algoritmů
- analyzuje datové toky v IS, definuje hlavní aktéry a jejich omezení nad systémem.
- DFD diagram obsahuje tyto prvky: **aktér** (obdélník mimo systém), **proces** (kruh uvnitř systému), **datové toky** (šipky) a **paměť** (viz. obr. Akce a Člen).



Minispecifikace

- Podrobná analýza algoritmů – procesů nejnižší úrovně DFD

- Pseudokód
- Minispecifikace popisuje postup, jak jsou vstupní data transformována na výstupní datové toky

Nástroje a modely

UML – umožňuje navrhnout konceptuální model IS

- Diagram tříd – popisuje logický náhled na systém (konceptuální model)
- Use-case diagram – kdo se systémem pracuje a jak; případ užití je posloupnost akcí ve vztahu s aktéry
- Stavový diagram – popisuje, do jakých stavů se objekt může dostat (životní cyklus objektu)

ER diagram

- **Oracle data modeler**
- **Toad data modeler**

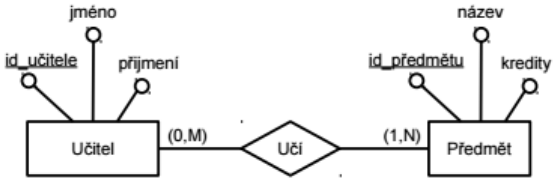


FIGURE : Chenova notace

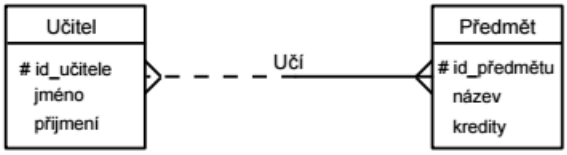


FIGURE : Crow's foot notace - Oracle

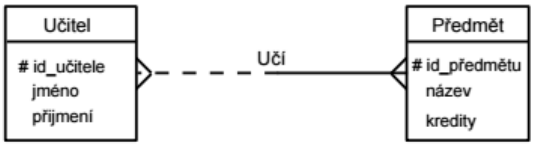


FIGURE : Crow's foot notace - Oracle

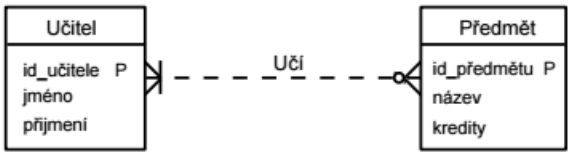


FIGURE : Crow's foot notace - Toad

Povinnost ve vztahu může být určená typem čáry (plná – povinný, přerušovaná – nepovinný vztah) nebo čárkou/kolečkem (čárka – povinný vztah)

Kardinalitu určuje vraní noha.

DFD – diagram datových toků

Modely

- **Konceptuální (logický) model**
- **Relační datový model (databázové schéma) – pro konkrétní DB systém**

Relační datový model

Relační datový model je způsob uchovávání dat v tabulkách. Základem konceptu relačního datové modelu je **relace,** kterou si můžeme představit jako dvourozměrnou tabulku. Relační model je velmi rozšířený a je jednoduchý.

- **Relace (tabulka)** – podmnožina kartézského součinu domén. Je to abstrakce 2D tabulky
- **Doména (typ atributu) –** množina hodnot jednotlivých atributů (takže relace je podmnožina kartézského součinu hodnot různých atributů)
- **N-tice –** jeden řádek tabulky
- **Atribut** – sloupec tabulky neboli atribut entity

Relační schéma

Popisuje strukturu relace – jméno relace, jména atributů a jejich typy. O relaci říkáme, že je instancí relačního schématu.

Databáze (množina relací) VS schéma databáze (množina relačních schémat)

Vlastnosti relačního datového modelu

- Homogenita sloupců
- Každý údaj (hodnota atributu) je atomický
- Na pořadí řádků a sloupců nezáleží (jsou to množiny)
- Každý řádek tabulky je jednoznačně identifikovatelný hodnotou **primárního klíče**

Vazby jsou v relačním datovém modelu řešeny pomocí cizích klíčů. Cizí klič se odvíjí od primárního klíče tabulky, na kterou odkazuje. **Vazební tabulka** je nutná u vazeb typu M:N.

SQL

Pro lepší pochopení SQL je lepší začít **relační algebrou.**

Relační algebra je velmi silný dotazovací jazyk vysoké úrovně. Nepracuje s jednotlivými enticemi relací, ale s celými relacemi. Operátory relační algebry se aplikují na relace, výsledkem jsou opět relace. Protože relace jsou množiny, přirozenými prostředky pro manipulaci s relacemi budou množinové operace.

Operace relační algebry

- **Selekce** $\sigma_{podmínka}$(Relace): výběr některých řádků ze vstupní relace
- **Projekce** $\pi_{seznam\_atributů}$ (Relace): výběr některých sloupců ze vstupní relace
- **Kartézský součin** R x S: každý řádek R se zkombinuje s každým řádkem S
- **Přirozené spojení** – ze součinu R x S se vyberou pouze ty řádky se stejnými hodnotami u stejnojmenných atributů u obou relací
- **Theta spojení**: ze součinu se vyberou pouze řádky odpovídající podmínce
- **Sjednocení, průnik, rozdíl**

SQL (Strucrured Query Language)

- Jazyk pro komunikaci s relačními databázemi
- Podporován prakticky všemi významnými SŘBD
- Deklarativní jazyk založený na relační algebře – příkazy popisují co chceme najít nikoli jak se to má provést. Tato vlastnost je základem pro nezávislost jazyka na fyzickém uložení dat a nepotřebujeme řešit, jak jsou data uložena.
- Existují dva základní standardy: SQL92 a SQL99

Dělení příkazů

1. DML – Data Manipulation Language – příkazy pro získání dat z databáze a pro jejich úpravy (SELECT, INSERT, UPDATE)
2. DDL – Data Definition Language – příkazy pro vytváření struktury DB (CREATE, ALTER, DROP)

SELECT příkaz

- Příkaz pro vyhledávání v datech (SELECT … FROM … WHERE …)
- **Distinct –** výsledek bez duplicit
- **ORDER BY sloupec –** seřazení výsledku podle zadaného sloupce
- **UNION –** sjednocení výsledků
- **INTERSECT –** průnik výsledků
- **EXCEPT –** rozdíl výsledků

Spojení

- **INNER JOIN (JOIN) –** spojení s podmínkou
- **NATURAL JOIN –** přirozené spojení (nemusí se uvádět podmínky spojení, spojení probíhá dle atributů se stejným jménem)
- **LEFT OUTER JOIN –** z výsledku nevypadnou záznamy z levé tabulky, které nejsou propojeny s pravou tabulkou

Agregační funkce

- **AVG, MIN, MAX, COUNT**
- **GROUP BY atribut HAVING podmínka:** shlukování jen za určité podmínky

DML

- **INSERT INTO Tab VALUES (x, y, z)**
- **DELETE FROM Tab WHERE podm**
- **UPDATE Tab SET x = val WHERE podm**

Funkční závislosti

Funkční závislost je v databázi vztah mezi atributy takový, že máme-li atribut Y je funkčně závislý na atributu X píšeme X → Y, pak se nemůže stát, aby dva řádky mající stejnou hodnotu atributu X měly různou hodnotu Y.

Pomocí funkčních závislostí můžeme automaticky navrhnout schéma databáze a předejít problémům jako je redundance, nekonzistence databáze, anomálie při aktualizaci a mazání. Funkční závislosti jsou konceptem, který nám umožní formálně definovat správné schéma databáze.

Armstrongovy axiomy - odvozovací pravidla funkčních závislostí

- Dekompozice (rozklad)
- Sjednocení
- Tranzitivita
- Rozšíření

Uzávěr množiny atributů - Uzávěr X+ je množina všech atributů funkčně závislá na atributech množiny X (Množina všech závislostí, které jsou logicky implikovány množinou F)

Určení klíče lze provést nalezením uzávěru množiny daného atributu

Minimální neredundantní funkční závislosti - neobsahuje redundantní závislosti ani redundantní atributy (to na levé straně)

- Jak určit redundantní funkční závislost? – Nalézt uzávěr potencionálně redundantního atributu A -> B na základě pravidel, pokud uzávěr bude obsahovat B, je pravidlo redundantní

Dekompozice a normální formy

Dekompozice relačního schématu je rozklad relačního schématu na menší relač.sch. (rozloží velkou tabulku na menší) aniž by došlo k narušení redundance databáze. Mezi základní vlastnosti dekompozice patří - zachování informace a zachování funkčních závislostí. Cílem je získání relací v nějaké NF. Výsledné relace musí obsahovat stejná data (atributy) a integritní omezení, jako původní databáze. Jeden krok dekompozice rozděluje původní schéma na dvě, přičemž těchto kroků může být více. Musí být zachováno bezeztrátové spojení (tak, abychom spojením dvou nových tabulek dostali původní relaci).

Normální formy

Normální formy relací (NF) prozrazují jak pěkně je databáze navržena

- **1NF –** Relační schéma obsahuje pouze atomické atributy (příklad adresa – rozdělená na ulici, PSC a město)
- **2NF –** každý atribut je závislý na každém klíči schématu (stačí, když je tranzitivně závislý)
    - **Nebo:** každý sekundární atribut je plně závislý na celém klíči schématu, Tudíž nesmí existovat závislost na podklíči.
- **3NF** - žádný sekundární atribut není tranzitivně závislý na žádném klíči schématu. Pro každou funkční závislost A -> B platí, že A je klíčem NEBO každý atribut v B je obsažen v nějakém kandidátu na klíč schématu.
    - **Nebo:** Všechny neklíčové atributy jsou navzájem nezávislé (tzn. opakující se věci vyhodit do dalších tabulek).
- **BCNF (Boyce-Coddova normální forma) -** Nesmí existovat závislosti ani mezi částmi složeného PK. Pro každou funkční závislost A -> B je A klíčem.
    - **Nebo:** atributy, které jsou součástí primárního klíče, musí být navzájem nezávislé.

Transakce, zotavení, log

Transakce

Transakce je logická (nedělitelná, atomická) jednotka práce s databází, která začíná operací BEGIN TRANSACTION a končí operací COMMIT nebo ROLLBACK.

Obecně transakce neobsahuje jednu databázovou operaci, ale častěji posloupnost několika operací. Úkolem transakce je převést korektní stav databáze na jiný korektní stav, přičemž nemusí zachovávat databázi v korektním stavu po jednotlivých aktualizacích transakce – buď se provede všechno v transakci, nebo nic. Transakce nemůže být uvnitř jiné transakce, tato vlastnost plyne z nedělitelnosti transakcí.

Konzistentní databáze – v databázi neexistují žádné výjimky z daných integritních omezení (integrita databáze). Integritní omezení se mohou týkat jednotlivých hodnot vkládaných do polí databáze (např. známka musí být v rozsahu 1-5) nebo může jít o kombinaci několika atributů (datum narození nesmí být pozdější než datum úmrtí).

Korektní stav databáze – respektujeme výsledek operací, které jsou vykonávány v reálném světě (např. převod z účtu na účet – v reálném světě se nesmí z jednoho účtu odečíst nějaká částka bez toho, aby se na druhý účet stejná částka přičetla)

Manager transakcí – stará se o řízení transakcí

Zotavení

Zotavení (recovery) znamená zotavení databáze z nějaké chyby. Výsledkem zotavení musí být korektní stav databáze a k dosažení tohoto stavu často využíváme nějaké redundantní informace.

Klasifikace chyb

- **Lokální chyby**: chyba v dotazu, přetečení hodnoty atributu
- **Globální chyby**: chyby systémové (výpadek proudu, pád systému), chyby média (hard crash)

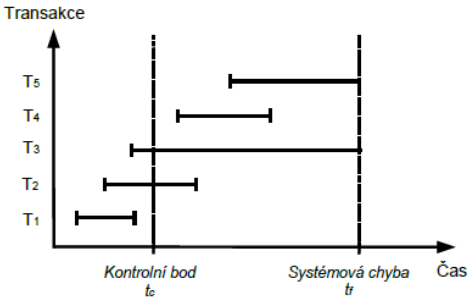
Transakce nejsou jen jednotkou práce ale i jednotkou zotavení – pokud systém zhavaruje, uživatel musí mít k dispozici databázi, která bude obsahovat všechny potvrzené změny.

Zotavení systému

Zotavení není vázáno pouze na jednu transakci, ale na celý databázový systém. Zotavení je nutné provést v případě globálních chyb, které ovlivňují všechny transakce, ve kterých se vyskytla.

Zotavení při systémové chybě – problémem je ztráta obsahu hlavní paměti. Transakce přerušená chybou musí být zrušena (**UNDO**). Transakce, která byla úspěšně dokončena, ovšem změny nejsou přeneseny z vyrovnávací paměti na disk, musí být přepracovaná (**REDO**).

Příklad: po restartu sytému musí být transakce T3 a T5 zrušeny a transakce T2 a T4 přepracovány.



Základní techniky zotavení

1. **Zotavení odloženou aktualizací (NO-UNDO/REDO)**

Neprovádí se aktualizace logu a databáze až do potvrzení transakce: všechny aktualizace jsou zaznamenány do paměti. Po potvrzení transakce (COMMIT) jsou aktualizace nejprve zaznamenány do logu a následně do databáze. Pokud transakce selže, **není nutné provést UNDO**, protože databáze nebyla aktualizovaná. **REDO** bude provedeno v případě, kdy systém zapsal aktualizace do logu, ale k zapsání změn do databáze nedošlo. Do logu jsou zapisovány nové hodnoty. V praxi se tato metoda používá tam, kde se provádí krátké transakce, které mění jen několik málo položek.

2. **Zotavení okamžitou aktualizací (UNDO/NO-REDO)**

Provádí se aktualizace logu a databáze po každé aktualizaci transakce (nejprve do logu, pak do databáze). Pokud transakce selže před dosažením potvrzovacího bodu, pak je nutné provést **UNDO**

(na disk byly zapsány aktualizace, které musí mýt zrušeny). Do logu se proto ukládají původní hodnoty. V tomto případě dochází k velkému počtu zápisů do databáze.

3. **Kombinovaná technika (UNDO/REDO)**

Používá se v praxi. Aktualizace jsou zapisovány do logu po COMMIT a k aktualizace databáze dochází v určitých časových intervalech – **kontrolních bodech.**

Kontrolní bod (check point) – jsou vytvářeny např. po určitém počtu záznamů, které byly zapsány do logu. Kontrolní bod zahrnuje zápis obsahu vyrovnávací paměti do databáze a zápis záznamu o kontrolním bodu do logu.

Log

Programátor nepracuje přímo s datovým souborem databáze. Pro zrušení všech aktualizací provedených v transakci pomocí operace ROLLBACK slouží **log** nebo **journal**, který je uložen na disku a obsahuje detaily o všech provedených transakcích. V případě ROLLBACK operace je systém na základě záznamů v logu schopen vrátit původní hodnoty příslušného atributu.

Implementace databázového systému (2 extrémní případy):

- Všechny aktualizace jsou **uloženy v paměti** – systém je rychlý, po pádu systému přijdeme o data
- Všechny aktualizace jsou uloženy **okamžitě na disk –** pomalý a nepoužitelný systém

Proto se u databázových systémů využívá **vyrovnávací paměť** umístěnou v hlavní paměti a obsahuje aktuální záznamy z databáze. Databáze je pak kvůli perzistenci umístěna na disku. Může se stát, že již potvrzené záznamy nebyly před pádem systému fyzicky zapsány na disk, zůstaly jen v hlavní paměti. I přes to musí být systém schopen zotavení.

Pravidlo dopředného zápisu do logu – všechny změny jsou nejdříve zapsány do logu před samotným zápisem změn do databáze. Před ukončením vykonávání operace COMMIT je do logu zapsán **COMMIT záznam.** Systém je pak schopen na základě informací z logu provést zotavení databáze.

Log je umístěn na disku, ale zapisuje se do něho sekvenčním způsobem, proto je to rychlejší, než aktualizace dat na disku přímo v databázi (náhodné diskové operace).

ACID

Každá transakce musí splňovat vlastnost ACID, což je zkratka pro:

- **A – Atomicity (Atomičnost)** – transakce musí být atomická: jsou provedeny všechny operace nebo žádná
- **C – Correctness (Korektnost)** – transakce převádí korektní stav databáze d jiného korektního stavu databáze, mezi začátkem a koncem nemusí být databáze v korektním stavu.
- **I – Isolation (Izolovanost)** – transakce jsou navzájem izolovány: změny provedené jednou transakcí jsou pro ostatní transakce neviditelné (až po provedení COMMIT operace)
- **D – Durability (Trvalost)** – jakmile je transakce potvrzena, změny v databázi se stávají trvalými i po případném pádu systému

COMMIT, ROLLBACK

Transakce začíná vykonáním operace BEGIN TRANSACTION a končí vykonáním COMMIT nebo ROLLBACK

COMMIT

- Úspěšné ukončení transakce, databáze je nyní v korektním stavu a všechny změny provedené v transakci mohou být trvale uloženy do databáze
- Zavádí **potvrzovací bod (commit point)**
- V okamžiku potvrzení transakce jsou
    1. Všechny změny trvale uloženy do databáze
    2. Všechny adresace a zámky entic jsou uvolněny

ROLLBACK

- Vrací databázi do stavu, ve kterém byla při vykonání BEGIN TRANSACTION – vrací databázi k předchozímu potvrzovacímu bodu.
- Všechny změny provedeny v rámci transakce musí být zrušeny

Problémy souběhu

Pokud k databázovému systému přistupuje více uživatelů současně, mluvíme o **souběhu.** Souběh umožňuje zpřístupnit databázi mnoha transakcím ve stejném čase.

Plán provádění transakcí – posloupnost operací. Pokud jsou plány prováděny souběžně, mluvíme o plánu paralelním.

1. **Ztráta aktualizace**

Jedna transakce aktualizuje entici, druhá transakce hned nato aktualizuje stejnou entici a tím dojde ke ztrátě aktualizace první transakce. (Aktualizace jedné transakce je ztracena, pokud to samé aktualizuje druhá transakce)

2. **Problém nepotvrzené závislosti (špinavé čtení)**

Nastane v případě, kdy jedna transakce načte nebo v horším případě aktualizuje entici, která byla aktualizována dosud nepotvrzenou transakcí. Existuje možnost, že potvrzena ani nebude a naopak bude zrušena (ROLLBACK). První transakce tak pracuje s hodnotami, které nejsou platné – stala se závislou na nepotvrzené změně.

3. **Problém nekonzistentní analýzy**

Transakce A má k dispozici nekonzistentní databázi a proto vykoná nekonzistentní analýzu. Např. počítaní součtů na bankovních účtech během převodu mezi účty druhou transakcí B. (neopakovatelné čtení)

Konflikty čtení/zápisu

- **RW konflikt –**
    1. A čte t a B chce zapsat t. Nastává *problém nekonzistentní analýzy*.
    2. Pokud B vykoná aktualizaci a A načte t znovu, pak A získá odlišné hodnoty – **neopakovatelné čtení**
- **WR konflikt**
    1. A zapíše t a B pak chce číst t. Nastává problém *nepotvrzené závislosti*.
    2. Pokud B přečte, mluvíme o **špinavém čtení**
- **WW konflikt**
    1. A zapíše t a pak B chce zapsat t. Nastává *problém ztráty aktualizace* (pro A) a *problém nepotvrzené závislosti* (pro B)

2. Pokud B zapíše t, mluvíme o **špinavém zápise**

Řízení souběhu – zamykání, úroveň izolace v SQL

Řízení souběhu se snaží řešit problémy souběhu. Mezi nejznámější techniky patří

- **Uzamykání** – pesimistický přístup: předpokládáme, že paralelní transakce se budou pravděpodobně navzájem ovlivňovat.
- Časová razítka
- **Verzování** – optimistický přístup: předpokládáme, že paralelní transakce se ovlivňovat nebudou
- Validace

Zamykání

Pokud transakce A chce provést čtení nebo zápis nějakého objektu v databázi, pak požádá o zámek na tento objekt. Žádná jiná paralelní transakce pak nemůže k objektu, který je již uzamčen, protože nemůže získat zámek, který vlastní již jiná transakce. Rozlišujeme 2 typy zámků:

- **Výlučný zámek (X, pro zápis):** Pokud transakce drží výlučný zámek na entici t, jiné transakce nedostanou ani výlučný ani sdílený zámek na entici t.
- **Sdílený zámek (S, pro čtení):** Pokud transakce drží sdílený zámek na entici t, pak požadavek na X zámek není proveden okamžitě a požadavek na zámek S je proveden okamžitě – transakce bude rovněž držet sdílený zámek; to je možné, protože sdílený zámek umožňuje pouze čtení.

Matice kompatibility

Pravidla můžeme popsat maticí kompatibility. Transakce A drží zámek na entici t uvedený v prvním řádku tabulky. Paralelní transakce požaduje na t zámek uvedený v prvním sloupci tabulky. Symbol N značí konflikt a zámek nebude přiřazen okamžitě. Symbol A značí, že zámek bude přidělen okamžitě.

	X	S	-
X	N	N	A
S	N	A	A
-	A	A	A

Jestliže zámek požadovaný transakcí nemůže být přidělen okamžitě, přejde transakce do stavu čekání.

Zámky jsou většinou přidělovány implicitně – při SELECT se přidělí zámek S, při aktualizaci zámek X.

Uzamykací protokol

- Tento protokol se nazývá **přísné dvou fázové uzamykání**
- Protokol přístupu k datům, který umožní řešení problému souběhu:
    1. Transakce, která chce získat entici z databáze, musí nejprve požadovat sílený 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, pak transakce B přejde do **stavu čekání**
        - Transakce v tomto stavu setrvá nejméně do doby kdy transakce A uvolní zámek

        - Systém se musí postarat o to, aby transakce B nesetrvala v tomto stavu navždy – řadit požadavky do fronty: transakci, která první požádá o zámek, bude zámek přidělen dříve.
    4. Výlučné zámky jsou automaticky uvolněny na konci transakce (COMMIT nebo ROLLBACK). Sdílené zámky jsou nejčastěji také uvolněny na konci transakce.

Uváznutí (deadlock) – dvoufázové uzamykání může způsobit uváznutí, kdy jedna transakce čeká na uvolnění zámků z druhé transakce a ta taky čeká na uvolnění zámků té první transakce. K tomu dochází, když si transakce vzájemně zamknou záznamy, s kterými potřebují a pak čekají, až jim ta druhá ten jejich záznam odemkne.

Řešení uváznutí

- **Detekce uváznutí:**
    1. **pomocí nastavení časových limitů** – transakce může trvat nejdéle nějakou dobu. Jakmile transakce trvá déle, systém předpokládá, že došlo k uváznutí
    2. **pomocí detekce cyklů v grafu Wait-For**, který udává, které transakce na sebe vzájemně čekají. Jedna z transakcí se zruší (ROLLBACK), druhá získá zámek a poté se zrušená transakce znovu spustí.
- **Prevence uváznutí**
    1. **Každé transakci je přiděleno časové razítko** – čas začátku transakce.
        - Pokud transakce A požaduje zámek na entici, která je již uzamčená transakcí B, pak:
            - Wait-Die: Pokud je A starší než B, pak A přejde do stavu čekání; pokud A je mladší než B, transakce B je zrušena operací ROLLBACK a spuštěna znovu
            - Wound-Wait: pokud je A mladší než B, pak A přejde do stavu čekání; pokud A je starší než B, transakce B je zrušena a spuštěna znovu
        - Pokud je transakce spuštěna znovu ponechává si své původní časové razítko
    2. **Wait**-Die: do stavu čekání přejdou transakce starší
    3. **Wound**-Wait: do stavu čekání přejdou transakce mladší
    4. **Nevýhoda**: relativní vysoký počet operací ROLLBACK

Věta o dvoufázovém uzamykání: Pokud transakce dodržující přísné dvoufázové uzamykání, pak všechny možné souběžné plány jsou serializovatelné.

Serializovatelný plán – plán je ekvivalentní s výsledkem libovolného sériového plánu (transakce jsou provedeny za sebou). ((A, B) a (B, A) musí dávat stejné výsledky).

Granularita zámků – zamykat lze i jednotlivé hodnot atributů, záznamů nebo diskových bloků, tabulek nebo databáze, nemusí to být jen zámek na n-tici. Důvodem je zvýšení propustnosti DB systému. Jemná granularita – zamykání malých objektů. Pokud transakce přistupuje k malému počtu záznamů tabulky, je výhodnější použít jemnou granularitu. Pokud přistupuje k mnoha záznamům tabulky, je výhodnější zamknout tabulku.

Plánovaný zamykaní protokol – systém nemůže zkoumat zámky přidělené každému záznamu z R, pokud nějaká transakce chce zámek na relaci R z důvodu náročnosti. Proto se zavádí plánovaný zamykací protokol, dle kterého transakce nemůže získat zámek na záznam, pokud před tím nezískala tzv. plánovaný zámek na celou tabulku, která záznam obsahuje. Systém pak nemusí provádět testování konfliktů na úrovni záznamů.

Verzování

Při správě verzí systém vytváří při aktualizaci kopie dat a systém sleduje, která z verzí má být viditelná pro ostatní transakce. Při uzamykání spravuje systém jednu kopii dat a jednotlivých transakcím přiděluje zámky. Při správě verzí systém vytváří při aktualizaci kopie dat a systém sleduje, která z verzí má být viditelná pro ostatní transakce (v závislosti na úrovni izolace).

Jak může fungovat správa verzí:

t_1	Transakce A čte záznam (48501, 'Osa - ocel', 45). V databázi existuje jedna kopie - označme jako c_1.
t_2	Transakce B aktualizuje záznam, systém vytvoří kopii (48501, 'Osa - ocel', 44), označme jako c_2. Pokud B bude chtít získat tento záznam, získá kopii c_2.
t_3	Transakce C chce získat tento záznam. Transakce B neprovedla COMMIT, proto C získá kopii c_1.
t_4	Transakce B provede COMMIT. Systém zpravuje pouze jednu kopii záznamu.
t_5	Transakce E chce získat tento záznam, získá aktualizovaný záznam.

- Správa verzí umožňuje paralelní zpracování více souběžných transakcí
- Zamykání bude ve většině případů způsobovat, že dvě nebo více transakcí budou čekat na ukončení jiné transakce
- Nevýhoda: velká interní režie – zvýšení požadavek na paměť při zprávě kopii dat
- Pokud bude převažovat čtení, je správa verzí efektivnější než zamykání
- Pokud bude převažovat aktualizace stejných záznamů bude režie správy verzí převyšovat popsané výhody
- Databázové systémy proto používají kombinaci obou přístupů

Protokol řízení souběhu transakcí na bázi **validace** (optimistický plánovač). Na konci transakce se zvaliduje, zda došlo ke konfliktu, pokud ano, jedna transakce se abortuje.

Úrovně izolace v SQL

Za izolovanost transakcí musíme zaplatit menším výkonem souběhu (nižším počtem vykonaných transakcí). Databázové systémy proto umožňují nastavit úroveň izolace, která zvýší propustnost, ale sníží míru izolace transakce:

1. **Read Uncommitted (RU)** – špinavé čtení, neopakovatelné čtení, výskyt fantomů
2. **Read Commited (RC)** – zámek může být uvolněn před ukončením transakce, nopakovatelné čtení, výskyt fantomů
3. **Repeatable Read (RR)** – výskyt fantomů
4. **Serializable (SR)**

Procedurální rozšíření SQL, PL/SQL

PL/SQL (Procedural Language/Structured Query Language) je procedurální rozšíření jazyka SQL od firmy Oracle. Jazyk SQL je sám o sobě neprocedurální, takže uživatel může definovat jen to, jaká data požaduje, ale postup, jakým jsou data získána je ponechán na databázovém systému. Pokud však požaduje při zpracování dat větší flexibilitu, je nutné procedurální rozšíření. Část aplikační logiky se tak přesune přímo do databáze.

Pro MS SQL Server existuje alternativa Transact-SQL.

Vlastnosti PL/SQL

- Efektivní práce s jednotlivými záznamy (pomocí kurzoru)
- Zachytávání výjimek
- Práce s proměnnými, poli, kursory atd
- **Výhody**
    - Optimalizace výkonu uložených procedur a funkcí, protože kód PL/SQL je uložen přímo v databázovém systému
    - Nižší přenášení dat (přenáší se jen konečné výsledky)
    - Sdílení kódu mezi aplikacemi
    - Nezávislost na platformě (ne DB systému)
- **Nevýhody**
    - Horší přenositelnost aplikace mezi databázovými systémy různých výrobců

Základním stavebním kamenem v PL/SQL je **blok**, který může být vnořen do jiného bloku. Blok má následující strukturu:

- **DECLARE** – nepovinná deklarace lokálních proměnných a kurzorů
- **BEGIN** – povinné otevření bloku
- **EXCEPTION** – nepovinné zachytávání výjimek
- **END** – povinné ukončení bloku

```
DECLARE
    v_name VARCHAR2(30) := 'michal.kratky@vsb.cz';
BEGIN
    INSERT INTO Email VALUES (v_name);
END;
```

Proměnné

- **Deklarace v DECLARE:** Jmeno_promenne typ_promenne [NOT NULL := hodnota]
- **Přirazení hodnoty:** SELECT name INTO v_name FROM …
- **Operátor %TYPE:** kopíruje datový typ atributu z tabulky
- **Operátor %ROWTYPE:** strukturovaný datový typ sestávající z datových typů tabulky, instance reprezentuje záznam tabulky

Triggery

Triger je blok, který je spouštěn v závislosti na nějakém DML příkazu (INSERT, UPDATE, DELETE).

- BEFORE | AFTER |INSTEAD OF – specifikace, v jaké chvíli se má triger spouštět, před po nebo místo SQL příkazu.
- INSERT | UPDATE | DELETE – specifikuje SQL příkaz, který spustí trigger
- OF– trigger se spustí jen při aktualizaci daného sloupce
- ON – specifikuje tabulku, na kterou se trigger váže
- FOR EACH ROW – daný trigger se spouští pro každý záznam

```
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
prikazy
```

```
END;
```

Využití triggeru:

- Kontrola zadaných dat
- Zajištění referenční integrity
- Implementace business pravidel
- Logování události
- modifikace dat v tabulce, když DML příkaz používá pohled

Procedury, Funkce

PL/SQL umožňuje vytvářet několik druhů procedur v závislosti, jakým způsobem jsou spouštěny.

Anonymní procedury

Nepojmenované procedury, které nemohou být volány z jiných procedur, spouští se přímo ze souboru nebo příkazové řádky. Mohou být pomalejší než pojmenované procedury, protože nejsou předkompilovány.

Pojmenované procedury

Obsahují hlavičku se jménem a parametry procedury. Je možné ji volat z jiných procedur nebo spouštět příkazem **EXECUTE.** Jsou uloženy v databázi a předkompilovány.

```
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;
```

Funkce

Na rozdíl od procedury funkce vrací hodnotu, takže musí specifikovat návratový typ a vracet hodnotu.

```
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;
```

Kurzory

Kurzory jsou pomocné proměnné vytvořené po provedení nějakého SQL příkazu. Umožňují nám procházet výsledek příkazu. Existuje implicitní kurzor a explicitní kurzor.

Implicitní kurzor

Vytváří se automaticky po provedení příkazů INSERT, DELETE, UPDATE.

Explicitní kurzor

Definuje se již v definiční části procedury podobně jako proměnná. Pomocí kurzoru můžeme procházet jednotlivé záznamy výsledku SELECT příkazu.

Definice: CURSOR jmeno IS select_prikaz

Příkazy pro práci s kurzorem (v LOOP):

- **OPEN jmeno** – otevírá kurzor. Provede SQL příkaz spojený s kurzorem a nastaví kurzor na první záznam výsledku
- **FETCH jmeno INTO promenna_zaznamu** – načítá aktuální záznam kurzoru do proměnné a posune se na další záznam
- **CLOSE jmeno** – zavírá kurzor

Kurzor jde použít i v cyklu FOR, nemusí se tak zavírat pomocí CLOSE: **FOR one_surname IN c_surname LOOP … END LOOP**

Hromadné operace

Vkládání většího počtu záznamů po jednom je neefektivní. Proto využíváme **hromadných operací.** Vkládání většího počtu záznamů najednou snížíme režii zotavení (zápis do logu) a datových struktur (menší počet aktualizací diskových stránek). Výsledkem je rychlejší vkládání záznamů. U Oracle využíváme BULK COLLECT a FORALL

BULK COLLECT

BULK COLLECT říká SQL enginu aby navázal výstupní kolekci s PL/SQL enginem. Používáme zejména s příkazem SELECT INTO.

BULK COLLECT INTO collection_name

FORALL

FORALL říká PL/SQL enginu aby navázal vstupní kolekci před jejím posláním do SQL enginu.

FORALL index IN 0..N sql_prikaz

Fyzická implementace databázových systémů

Fyzická implementace definuje datové struktury pro základní logické objekty, jako jsou tabulky, materializované pohledy a rozdělení dat. Řeší uložení dat na nejnižší úrovni databáze. **Optimální fyzický návrh –** DB systém přistupuje pouze ke stránkám obsahujícím záznamy výsledků.

Stránkování datových struktur

- Všechny datové struktury se skládají ze **stránek**, které jsou umístěné na disku.
- Velikost stránky je nejčastěji 8kB (nebo násobek 2kB – násobek velikosti diskového sektoru (512B))
- **Cache buffer –** oblast v hlavní paměti, kde se ukládají právě používané stránky. Řešení v podobě fronty, kdy načtenou stránku z disku nahraní nejstarší použitou stránku
- **Logický přístup –** přístup ke stránce libovolné datové struktury
- **Fyzický přístup –** načtení nebo zápis na stránky na disk
- **Cache hit –** stránka je nalezena v cache buffer
- **Cache miss –** stránka je až na disku

Ve většině DB systémů najdeme tyto datové struktury pro uložení dat: **tabulka typu halda** (stránkované sekvenční pole) a **index** (B-strom).

Tabulky

- **Tabulka typu halda** – záznamy v tabulce nejsou uspořádány (implicitní volba CREATE TABLE)
- **Shlukování záznamů** – záznamy jsou v datovém souboru seřazeny podle zvoleného klíče
- **Hash tabulka** – záznamy se stejnou hašovanou hodnotou jsou uloženy ve stejném bloku nebo ve velmi blízkém bloku
- **Zhmotněné pohledy** – uložené výsledky dotazů, které bývají často v databázi vyhodnocovány – fragment z tabulky či několika tabulek

Halda

Jedná se o základní typ tabulky, jenž se vytvoří po zadání příkazu CREATE TABLE. Jedná se o stránkované persistentní pole s velikosti 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 nebo na konec pole
- Každý záznam je identifikován pomocí **ROWID**
- Efektivní vkládání a využití místa (s výjimkou častého mazání po hromadném vložení záznamů)
- **Složitost operací:**
    - Vkládání: velmi efektivní: O(1), v případě, že tabulka nemá žádný index.
    - Vyhledávání: sekvenční průchod – O(n); počet načtených stránek je n/C, kde C je průměrný počet záznamů ve stránkách.
- **Jak se udržují prázdné stránky haldy:**
    - Dvojitě spojový seznam s hlavičkou a seznamem volných a prázdných míst
    - Adresář stránek – spoják adresářových stránek, položka ukazuje na datovou stránku a má příznak plnosti

Shlukování záznamů

- Záznamy jsou v datovém souboru seřazeny podle nějakého klíče
- Implementace pomocí B-stromu
- Listové uzly stromu (bloky) obsahují kromě klíče i další záznamy tabulky (nemusí to být nutně všechny atributy tabulky)
- Použití: tam, kde chceme získat kromě klíče i hodnoty ostatních atributů
- **Nevýhodou** je zhoršený výkon ukládání, protože se data musí třídit. Výhodné zejména v případě převahy čtení proti aktualizaci
- Rozlišujeme shlukování pro jednu a více tabulek
- Výhodné všude, kde potřebujeme hodnoty dalších atributů – v operaci SELECT neklíčových atributů, při spojování tabulek, třídění podle klíče.
- Oracle: **indexed organized table (IOT)**, SQL server: **clustered index** (při CREATE TABLE)
- V Oracle existují parametry
    - OVERFLOW – definuje maximální velikost záznamu v listovém uzlu. Část záznamu, která se do listového bloku nevejde, je uložena v odděleném segmentu a data v listu se na ně pouze odkazují
    - INCLUDING – udává atributy, jejichž hodnoty mají být uloženy v listovém uzlu tabulky

Shlukování záznamů pro dvě tabulky

Když chceme eliminovat operaci spojení. V případě shlukování budou záznamy objednávek uloženy do stejného bloku jako záznam zaměstnance. Snižuje se tím ale efektivita dotazů, které nevyužívají tohoto shlukování. Dotazy využívající shlukování musí převažovat.

Hašovaná tabulka

- Záznamy se stejnou hašovací hodnotou jsou uloženy ve stejné nebo sousední stránce
- Hašování poskytuje v některých případech lepší složitost nez B-strom (O(k) vs O(log n)) ale dochází k plýtvání místa (k je počet stránek pro jednu hash hodnotu)
- Vhodné, pokud předem známe velikost dat tak, abychom vytvořili na začátku rozumně velkou hash tabulku
- **Velikost:**
    - **Velikost tabulky je malá:** řetězce stránek jedné hash hodnoty budou dlouhé: O(k), k jde k N, kde N je celkový počet stránek
    - **Velikost je příliš velká:** ke každé hash hodnotě bude existovat jedna stránka, ale bude obsahovat málo položek. Využití stránky bude klesat k 0

Indexy

Index umožňuje rychlé vyhledávání dle klíče. V záznamu je uložen klíč a ROWID, které odkazuje na kompletní záznam v tabulce typu halda, kde nalezneme další atributy.

- **B$^+$-strom** – nejrozšířenější typ indexu
- **Složený index** – index, kde klíčem je více než jeden atribut. Složený index je implementován B-stromem se složeným klíčem (key1, key2) v tomto pořadí. S log(n) složitostí jsou tedy vykonávány dotazy pro key1 nebo (key1, key2). NE pro key2.
- **Pokrývající index** – index obsahující kromě klíče i hodnoty dalších atributů, tak, aby nebyl nutný přístup do tabulky
- **Hašovaná tabulka** – Hašovaná tabulka pro klíč
- **Bitmapový index** – index, který pro každou hodnotu h indexového atributu x a jeden záznam obsahuje jeden bit. Bit je nastaven na 1, pokud má záznam hodnotu h v atributu x, jinak je nula. Nevýhodou je problematická aktualizace indexu, používá se v případě malého či žádného počtu aktualizaci.

B-strom

- Klíčem je indexovaný atribut, ROWID přiřazené ke každému klíči odkazuje na příslušný záznam v tabulce typu halda
- Dotaz na klíč má IO cost h + 1, kde výška h je výška stromu, což je skvělé, ale přístupy jsou provedeny náhodně.
- Využití stránky je v případě B-stromu 50% (50% místa ve stránkách neobsahuje žádná data) – důsledek štěpení stránek B-stromu při vkládání položky a přetečení stránky.
- Tabulka typu halda má 100% (pokud nedochází často k operaci vkládání)
- **Složitosti**
    - **Vkládání:** O(log n) (náhodné přístupy)
    - **Bodový rozsah:** O(log n) (náhodné přístupy
    - **Rozsahový dotaz:** O(N), kde N=n/C (N je počet stránek datové struktury)

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. Nevýhodou je problematická aktualizace indexu, proto se používá v případě malého či žádného počtu aktualizací.

Pokud se dotazujeme více atributů tabulky s malými doménami (7 a méně), je výhodné použít bitmapový index. Vyhledávání bude znamenat sekvenční průchod relativně malým polem bitových řetězců a hledání relevantních záznamů pomocí bitových operací.

Index založený na B-stromu by byl velký. Musíme vytvořit tolik indexů kolik je dotazovaných atributů nebo jeden složený index.

Heap table

fname	lname	gender	rating	department
Jana	Petrová	F	1	2
Petr	Sobota	M	3	5
...				

Bitmap index

gender		rating					department				
0	1	1	0	0	0	0	0	1	0	0	0
1	0	0	0	1	0	0	0	0	0	0	1

Vytvoření indexu v Oracle: CREATE BITMAP INDEX Nazev ON Tabulka(attr1, attr2)

SELECT dotaz bude znamenat sekvenční průchod relativně malým polem bitových řetězců a hledání relevantních záznamů pomocí bitových operací.

Hustý a řídký index – hustý index má ukazatel na každý řádek tabulky a řídký index obsahuje ukazatele na bloky tabulky

TODO DOPLNIT VĚCI PRO ORACLE I PRO SQL SERVER

Materializované pohledy

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. Je to fyzická tabulka definovaná selectem. Výhodou materializovaného pohledu je zrychlení dotazů. Nevýhodou jsou duplicitní data v DB a režie na update tabulky.

Pohled se tváří jako tabulka obsahující sloupce a je dotazován stejným způsobem jako tabulka. Při dotazu na pohled se server dotáže tabulek, na kterých je pohled založen a vrátí data. Pohled se mimo jiné definuje dotazem, který udává vztahy mezi svázanými tabulkami. Ten vrátí sloupce v daném pořadí. Dalo by se říci, že se jedná o SQL dotaz, který je v databázi uložen pod nějakým jménem a lze s ním pracovat jako s tabulkou.

Materializované pohledy se používají pro poskytování kopie vzdálených dat nebo dat v rámci databáze. Tento pohled je založen na dotazu, ale výsledek tohoto dotazu je fyzicky uložen v databázi. Tyto pohledy mohou být jen ke čtení a mohou být pravidelně aktualizovány podle určitého plánu. Materializované pohledy mohou být např. použity v případech, kdy je dotaz, na kterém jsou založeny, časově velmi náročný a data nemusejí být tolik aktuální.

Rozdělení dat

Rozdělení dat se používá pro data s dlouhou historií, fyzické rozdělení datových struktur (a souborů) na více částí. Partitioning je technologie sloužící v relační databázi k fyzickému rozdělení rozsáhlých datových tabulek do menších částí na základě logického členění dat v tabulce, nazývaných partition (český přibližný ekvivalent oddíl se příliš neadaptoval). Pokud databázový server poskytuje takovou možnost, umožňuje tato technologie rychlejší manipulaci s tabulkami, jejichž velikost se pohybuje na hranici možností použitých systémových prostředků. Prakticky (a nejčastěji) bývá realizován rozdělením tabulky na více pevných disků, což představuje rozložení zátěže oproti stavu, kdy všechny požadavky musel obsluhovat pouze jeden disk. Důvodem tedy kromě velikosti dat může být také požadavek na větší rychlost. To, podle jakého klíče k rozdělení dojde, dovoluje většina databázových

strojů, které tuto funkci podporují, určit – většinou se jedná o rozdělení podle určitého sloupce nebo jeho hašovací funkce.

Příklad:

```
CREATE PARTITION FUNCTION myRangePF1 (int)
    AS RANGE LEFT FOR VALUES (1, 100, 1000) ;
GO

CREATE PARTITION SCHEME myRangePS1
    AS PARTITION myRangePF1
    TO (test1fg, test2fg, test3fg, test4fg) ;
GO

CREATE TABLE PartitionTable (col1 int, col2 char(10))
    ON myRangePS1 (col1) ;
GO
```

Výsledkem je rozdělení dat do 4 fyzických částí dle hodnoty atributu `col1` (< 1, < 100, < 1000, ≥ 1000).

Stránkování

Stránkování datových struktur

Záznamy, které uživatel vkládá do databáze, jsou uložena v různých datových strukturách. Tyto datové 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 2kB (nejčastěji je to 8kB), protože stránka musí mít velikost násobku diskového sektoru (512B) a alokační jednotky souborového systému (2kB).

Stránkování záznamů

Zejména u webových aplikací se můžete setkat se stránkováním – aplikace nevypisuje všechny záznamy najednou, ale např. jen prvních deset článků. Další se uživateli zobrazí až po kliknutí na další stránku.

Zpracování dotazů výrazně urychlíme, pokud z databáze získáme jen ty záznamy, které chce uživatel zobrazit. Tabulka obsahuje 100000 záznamů, ale v UI se uživateli zobrazuje jen jedna stránka, která má např. jen 100 záznamů. Dotaz na všechny záznamy vede k

- Vyššímu počtu logických přístupů
- Vyšší režie komunikace s DB systémem – je rozdíl poslat 8MB a 8kB
- Vyšší režie ORM – 100 000 záznamů je v ORM převedeno na 100 000 objektů

Řešením může být

- **Cachování** na úrovni aplikačního serveru – pokud by mělo být 100%, musíme část funkcionality DB systému přenést na aplikační server, např. transakční zpracování. Proč? V cache máme číselník, v DB mezitím někdo smaže záznam z tohoto číselníku, co teď?
- **Stránkování na úrovni DB** systému a jeho podpora v ORM (použití metody např. Student.Select(from, to))

V případě stránkování tedy často neušetříme logické přístupy, jen přeneseme méně dat a zrychlení dotazu nastane kvůli snížení režie komunikace s DB systémem a snížení režie ORM. V takovém případě porovnáváme statistiky např. **Bytes received from server** a **Total execution time.**

Pro stránkování v ORACLE slouží pomocný sloupec **ROWNUM**, který je k dispozici v dotazu. Může mít hodnotu od 1 do N, kde N je počet záznamů vrácených dotazem. Pomocí ROWNUM je možné výsledek dotazu stránkovat, avšak je třeba použít ještě jeden SELECT.

```
select * from (
```

```
  select rownum as cisla_radku, t.* from (
    select * from table_name order by column_name) t
  )
where cisla_radku between 10 and 15;
```

V SQL k tomu slouží **ROW_NUMBER,** což je funkce, která vrací pořadové číslo řádku dle zadaného kritéria v příkazu OVER(). Nebo to lze pomocí OFFSET/FETCH NEXT.

Více: https://sql-vyuka.cz/d/node/13

Řádkové a sloupcové uložení dat

V klasických DB systémech jsou v haldě data uložena po řádcích – mluvíme o **řádkovém uložení.** V blocích haldy jsou tedy uloženy kompletní záznamy za sebou. Tento způsob je výhodný při dotazech typu SELECT *, kde dochází k sekvenčnímu průchodu tabulkou nebo při bodovém dotazu v indexu a načtení záznamu z haldy podle ROWID. Výhodné to není u dotazů, kde se něco počítá nad jedním sloupcem. Např. sekvenční průchod tabulkou a počítání součtů platů, bloky ovšem obsahují i hodnoty ostatních atributů, které se taky procházejí.

Pokud tedy v dotazech pracujeme jen s několika málo atributy, můžeme uvažovat o tzv. **sloupcovém uložení dat**. Typické využití je v datových skladech a dotazy obsahující projekce, agregace a spojení.

Při sloupcovém uložení dat jsou v poli uloženy hodnoty jednotlivých sloupců zvlášť. Záznamy jsou rekonstruovány pořadím hodnoty ve sloupci. Jednotlivé hodnoty tedy neobsahují identifikátor řádku.

Sloupcové uložení pomocí řádkového?

- **Vertikální rozdělení tabulky –** kvůli rekonstrukci záznamu bude mít každá tabulka schéma (key, column), což představuje nárůst prostorové režie.
- **Vytvoření indexu pro každý atribut –** B-strom bude kromě hodnoty obsahovat i RID, což opět představuje nárůst režie
- **V Oracle můžeme u shlukované tabulky (IOT)** určit, které atributy budou uloženy spolu s klíčem ve stránkách stromu. Tato technika zřejmě nebude fungovat pro ad-hoc (libovolné) dotazy.

Atributy ve sloupcovém uložení lze komprimovat (pokud se ve sloupcích opakují hodnoty), komprimace sloupce bude úspěšnější než komprimace celých řádků. Ale třídění sloupce kvůli komprimace sebou nese nevýhodu, protože pak nemůžeme z pořadí hodnoty rekonstruovat řádek. To bychom museli pro setřízené sloupce ukládat ještě klíč záznamu.

Načítání do cache buffer: Když potřebuji přistoupit jen k několika málo sloupcům, načtu pouze jejich data a nemusím se obtěžovat s jinými položkami ze stejného řádku. V řádkových databázích je vždycky nutné načíst celý řádek, protože data se z disků načítají po blocích, které obsahují mnoho řádků najednou. Musím pak přeskakovat nepotřebné atributy, které se načetly do cache buffer a využití cache buffer tak není ideální.

Sloupcové uložení v SQL SERVER

- **Clustered columnsore index** – typ tabulky (sloupcová tabulka). Vytvoří sloupcové uložení pro všechny atributy tabulky, řádková tabulka není nadále potřebná a bude zrušena
- **Nonclustered columnstore index** – typ indexu (sloupcový index). Vytvoří sloupcové uložení pro daný klíč. Kompromis mezi řádkovým a sloupcovým uložením.

Sloupcové uložení v ORACLE

- **Oracle In-Memory Clolumn Store**
- Jedná se o uložení v hlavní paměti, do části paměti vyhrazené systémem Oracle přibude tzv. In_memory Area, která bude obsahovat sloupcově uložené tabulky
- ALTER TABLE Tabulka INMEMORY – tabulka bude převedena na paměťovou sloupcovou tabulku po prvním sekvencím průchodu (nebo při dodatku PRIORITY CRITICAL okamžitě po startu DB)

Plán vykonávání dotazů

Zpracování dotazu zahrnuje parsování, analýzu a přiřazení zdrojům dat. Analýza zahrnuje tvorbu plánů vykonávání a tvorbu nejlepšího. Plány vykonávání jsou sestaveny na základě:

- Způsobu využití tabulek a jejich sloupců (projekce, druhy spojení tabulek, …)
- Existujících indexů
- Přítomnosti „hints“ v dotazu – přikazují, jak se má dotaz provést
- Minulých úspěšných plánu provedení existujících dotazů – plány jsou ukládány v cache
- Statistik nad existujícími daty v databázi i provedenými dotazy

Čas vykonávání dotazu můžeme zlepšit např. parametrizovanými dotazy, hromadnými operaci nebo nastavením transakcí. Na úrovni databáze můžeme ovlivnit výběr efektivnějšího plánu vykonávání dotazu případně dobu vykonávání operací fyzickým návrhem databáze.

S fyzickým návrhem databáze souvisí pojem **vyhodnocení dotazu**, tedy proces SŘBD (konkrétně optimalizátor) hledá cestu jak nejrychleji provést uživatelský dotaz a dotaz poté vykoná. Při zpracování dotazu se nejprve odstraní duplicitní syntaxe jazyka SQL, k čemu slouží model jazyka SQL – relační algebra.

Motivační příklad

Mějme nějaký dotaz, který se skládá z relační operace spojení, selekce a projekce. Při žádné optimalizaci se dotaz může vykonat tak, že se nejprve nějaké dvě tabulky spojí přes všechny záznamy, musí se načíst každý záznam z jedné tabulky a pro každý takový záznam všechny záznamy z druhé tabulky. Až poté se provede selekce a z několika tisíc záznamů nám zůstane jen třeba několik desítek. Poté se provede projekce a vybere se jen třeba jeden atribut.

Při optimalizovaném dotazu se nejdříve provede selekce a až pak spojení. Nejprve se tedy z jedné tabulky vybere z několika tisíc záznamů jen desítka a jen pro tyto záznamy se provede spojení s druhou tabulkou, ne pro všechny, jako to bylo v předešlém případě.

Cena vykonávání dotazu

Abychom byly schopni porovnat různé plány vyhodnocení dotazu, musíme nějakým způsobem měřit cenu vykonání dotazu.

- **IO Cost –** vstupně-výstupní cena, měřená počtem přístupu ke stránkám tabulek
- **CPU Cost –** procesorová cena, měřená operací provedených během vykonávání dotazu

Dále rozlišujeme, zda se přistupuje ke stránkám v cache (**logické přístupy**) nebo ke stránkám na disku (**fyzické přístupy**).

Zpracování dotazu

Zpracování dotazu má na starost **procesor dotazu.** Dotaz je překládán ve třech fázích:

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 SQL jazyka. Interní forma je nejčastěji nějaký druh **dotazovacího stromu**, což je reprezentace výrazů relační algebry. Nebo-li dotazovací strom můžeme chápat jako reprezentaci výrazu relační algebry.

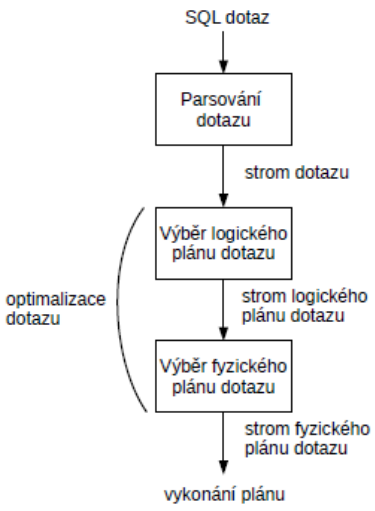
Výběr logického plánu dotazu (převod do kanonické formy)

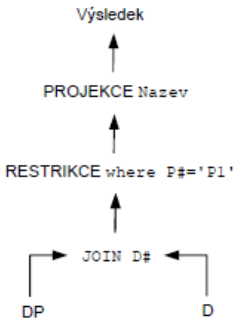
Ve fázi logického plánu hledáme efektivnější tvar dotazu, než nabízel původní dotaz – **přepsání dotazu**. Optimalizátor provádí celou řadu optimalizací, relační algebra umožňuje dotaz vyjádřit mnoha způsoby. Dotaz tedy není ve skutečnosti vykonán přesně, tak jak jej zadá uživatel. To je to, co bylo popsáno v ukázkovém příkladu.

Výběr fyzického plánu dotazu

Zahrnuje výběr algoritmů implementujících operátory logického plánu a výběr pořadí těchto operací. Procesor v této fázi uvažuje velikost tabulek, existenci indexů, distribucí hodnot, shlukování uložených dat a ostatní informace o datech, které načítá ze **systémového katalogu**.

Každému plánu je přiřazena cena, která závisí na výše uvedených statistikách a fyzických operací. Z množiny dotazovaných plánů pak optimalizátor vybere ten nejlepší (nejlevnější).





Obrázek 7.1: Fáze překladu dotazu Obrázek 7.2: Stromu dotazu z předch

Základní typy dotazů

- **Sekvenční průchod tabulkou** – postupně načítá stránky tabulky. Výhodou je maximální výkon čtení z disku. Nevýhodou je, že pro jen jeden záznam musíme přečíst všechny stránky. Složitost operace je O(n)

- **Bodový dotaz** – (v B-stromu) hledá zda ve stromu je či není uložen klíč *k*. Pro každou úroveň stromu je zpracován právě jeden uzel. IO cena je O(h+1) – jednotky přístupů. Nevýhodou je, že stránky stromu nejsou za sebou a jedná se tedy o náhodný přístup.
- **Rozsahový dotaz** – vrací všechny klíče ze stromu v daném rozsahu. Nejprve se bodovým dotazem najde krajní klíč a poté se prochází všechny sousední listové uzly, dokud se nenarazí na pravý krajní klíč. Cena – h + 1 + počet načtených listových uzlů. Vyhodnotí-li optimalizátor, že bude výhodnější provést sekvenční průchod (z důvodu příliš velkého rozsahu klíčů), nepoužije se index.

Operace plánu vyhodnocení dotazu

Sekvenční průchod tabulkou

Operace postupně přistupuje ke všem blokům tabulky a ukládá záznamy do výsledku.

- Oracle: TABLE ACCESS (FULL)
- SQL Server: TABLE SCAN

Přístup k záznamu tabulky typu halda pomocí ROWID získané z indexu

- Oracle: TABLE ACCESS (BY INDEX ROW ID)
- SQL Server: RID Lookup (Heap)

Bodový a rozsahový dotaz v indexu

Bodový dotaz je proveden průchodem od kořene k listu, kde se nachází klíč.

- Oracle: INDEX (UNIQUE SCAN), INDEX (RANGE SCAN)
- SQL Server: INDEX SEEK

Spojení

- **Nested loop join**
- **Merge join**
- **Hash join**

Statistiky databáze

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

- **Pro tabulku ukládá:**
  - Kardinalitu: počet záznamů
  - Počet diskových stránek tabulky
- **Pro sloupce:**
  - Počet totožných hodnot ve sloupci, min a max hodnoty
  - 10 nejčastějších hodnot a počet výskytů
- **Pro index:**
  - Počet listových stránek indexu
  - Výšku indexu

Logické a fyzické operace

- Jelikož propustnost paměti je řádově 10 – 1000x vyšší než propustnost diskových operací, rozlišujeme v databázových systémech dva typy přístupů ke stránkám:
  - **Logický přístup –** vyjadřuje přístup ke stránce libovolné datové struktury
  - **Fyzický přístup –** značí načtení nebo zápis stránky na disk

- **Cache hit –** požadovaná stránka je nalezena v cache buffer, jinak to je **cache miss**
- **Cache hit rate –** míra úspěšnosti použíti cache buffer (cache hits/počet logických čtení) * 100

Vyhodnocení logických a fyzických přístupů

- Pokud počet záznamů výsledku je mnohem nižší než počet logických přístupů, musím uvažovat o optimalizaci dotazu (např. vytvoření indexu)
- Pokud se počet fyzických čtení blíží počtu logických přístupů, můžeme efektivitu zvýšit zvýšením paměti cache, tak aby cca 90% dat byla umístěna v cache.

Náhodné a sekvenční operace

- **Náhodné operace** – např. u indexu (B-strom) – stránky nejsou uloženy za sebou, při dotazování na index se různě skáče na jednotlivé stránky (průchod stromem) a čtení je tak náhodné.
- **Sekvenční operace** – např. u tabulky typu halda se SELECT dotazem. Prochází se celá tabulka, stránky v heap tabulce jsou umístěné za sebou, čtou se postupně. Rychlost je vyšší než v případě náhodného čtení. (pomalejší až o 160x) K tomuto poklesu dochází, přestože disky obsahují vlastní vyrovnávací paměť: úspěšnost cáchování je v případě náhodných operací nízká a klesá k nule.

V případě rozsahového dotazu u indexu se může stát, že cena průchodu indexu bude vyšší než cena sekvenčního průchodu tabulkou (z důvodu velkého počtu náhodných čtení), proto se index nepoužije.

Ladění vykonávání dotazů

V databázových systémech máme možnost zobrazit plán vykonávání dotazu. V plánu vidíme všechny operace provedené DB systémem: průchod tabulkou, přístup k indexu, třídění, spojení atd. Tento plán může sloužit pro odladění dotazu, např. vidíme, že musíme použít index na atribut, který nebyl dosud indexován atd.

Zobrazení plánu

V databázových systémech máme možnost zobrazit plán vykonávání aktuálního dotazu, ve kterém jsou uvedeny jednotlivé fyzické operace a jejich cena. Plán je mocný nástroj, který může sloužit k odladění provádění dotazu. Vhodnou volbou uložení dat (fyzický návrh) ovlivníme výběr operací plánu a tedy i rychlost provedení dotazu. Zobrazení plánu nám slouží k odchycení neefektivních operací pro daný dotaz a můžeme tak zvolit jiný fyzický návrh. V plánu můžeme například vidět, že pro dotaz obsahující selekci došlo k sekvenčnímu průchodu tabulkou a je nutné vytvořit index pro tento atribut atd.

Zobrazení a obsah plánu je závislý na konkrétním systému.

Zobrazení v ORACLE

Explain plan for SELECT …. //plán příkazu pro daný dotaz se uloží do tabulky PLAN_TABLE

Pro zobrazení plánu z tabulky PLAN_TABLE pak musím použít nějaký složitý SELECT příkaz.

Zobrazení v SQL Server

V SQL Server Management Studio v Menu - Show Execution Plan (v menu Query -> Include Actual Execution Plan)

Přesné statistiky provedení dotazu

Výstupem plánu dotazu jsou jednotlivé operace provedené při vykonávání dotazu a jejich cena: **IO Cost** – počet přístupů ke stránkám datových struktur a **CPU Cost** – počet operací. Odhad ceny vykonání dotazu je ovlivněna čtením z cache buffer a dalšími optimalizacemi aplikovanými během provedení dotazu. Pokud v nějaké konkrétní situaci získává systém 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:

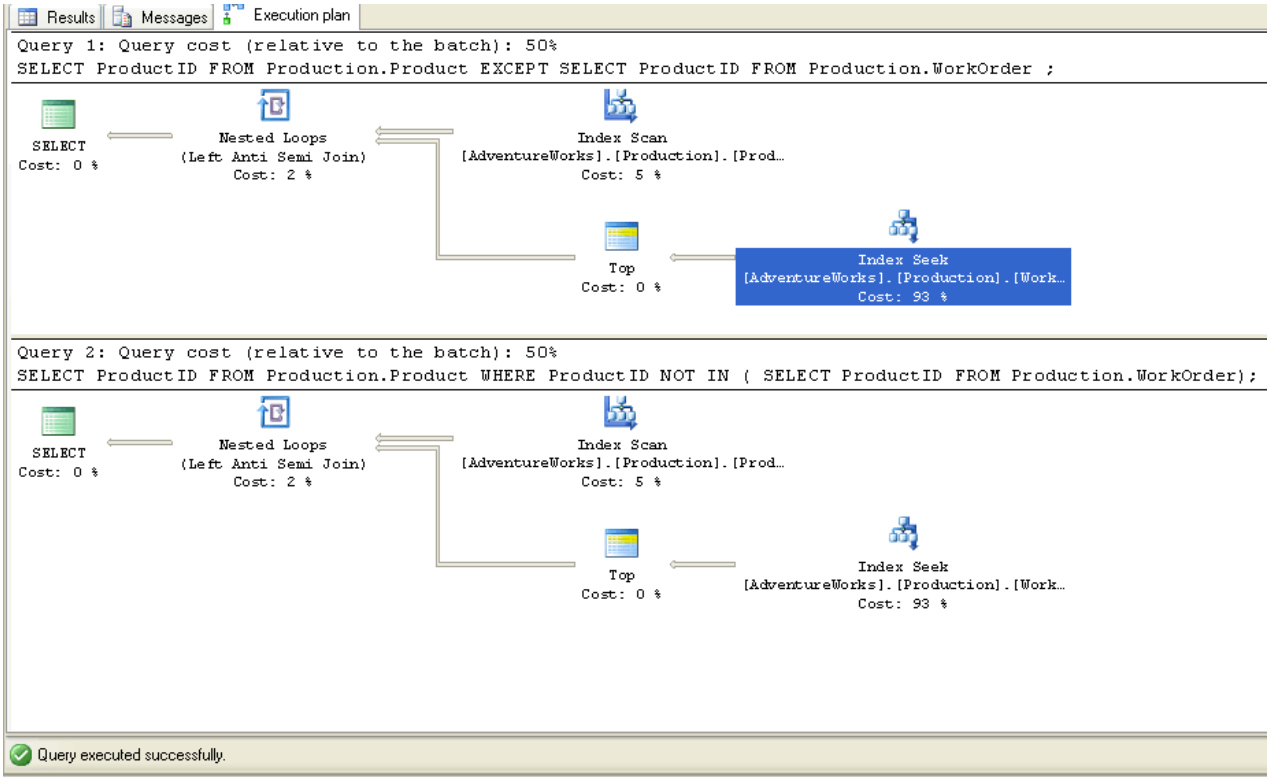
Pro Oracle: SET AUTOTRACE ON, ze kterého získáme:

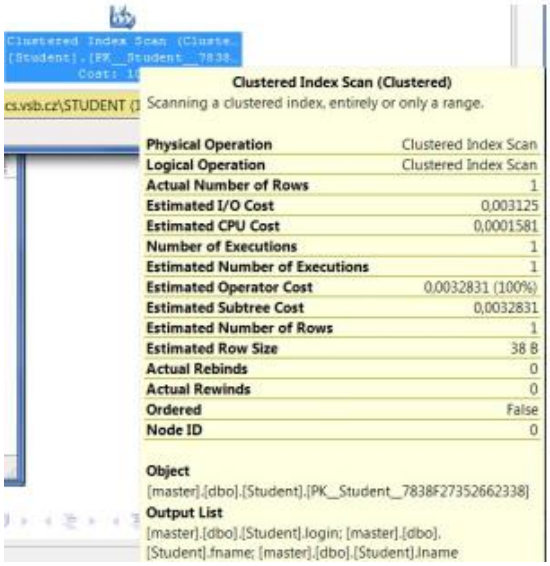
- **Consistent gets** – počet přístupu k blokům – logické přístupy (přesněji počet logických přístupů + počet záznamů výsledku/velikost pole záznamů)
- **Psysical reads** – celkový počet bloků načtených z disku – fyzické přístupy
- **Cache hit** – pokud pro logický přístup neexistuje fyzický přístup, jinak je to **cache miss**
- **Sorts (memory) –** počet operací třídění, které byly vykonány v paměti bez zápisu na disk
- **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

Pro SQL Server: SET STATISTICS IO ON, ze kterého získáme:

- **Logical reads** – počet logických přístupů
- **Psysical reads** – počet fyzických přístupů

Pravidlo pro vytvoření indexu – Vytvoř index pro všechny atributy, které se v dotazech objevují za WHERE.





Algoritmy spojení

- **Nested loop join (hnízděné cykly) –** implementace obsahuje dva zanořené cykly. Pro každý záznam vnější tabulky se hledá odpovídající záznam vnitřní tabulky. Výhodný, pokud je vnější tabulka malá a vnitřní tabulka má index na spojovacím klíči. Tento algoritmus v každé své variantě čte jeden ze záznamů pouze jednou, zatímco další argument bude čten opakovaně. Tento algoritmus může být pro použit pro relace v jakékoliv velikosti, ale je nutné, aby jedna relace se dala uložit do hlavní paměti.
- **Merge join (třídění-slévání)–** Sekvenčně prochází obě vstupní tabulky. Kurzory se postupně posunují a nacházejí se tak odpovídající záznamy. Obě tabulky ale musí být setřízeny dle klíče. Spojovací podmínka obsahuje rovnost. Výhodný, když cena za setřízení není velká a tabulky jsou přibližně stejně velké. Předpokladem pro vykonání toho algoritmu je možnost načtení obou relací do paměti, po načtení do paměti jsou záznamy setříděny podle spojovacího atributu a po té dochází k slévání setříděných záznamů.
- **Hash join –** v první fázi se vytvoří pro menší tabulku hash tabulka. Postupně se prochází větší tabulka a hledají se odpovídající záznamy v hash tabulce. To vyžaduje vytvoření hash tabulky v hlavní paměti a musí jít o podmínku rovnosti. Výhodné, pokud jsou tabulky na vstupu velké. Jestliže nemáme k dispozici indexy na atributech pro spojení relací, je vhodné použít hashování ke spojení. Budeme potřebovat místo v paměti pro načtení relace R, kterou zahashujeme pro spojovací atribut, v dalším kroku čteme sekvenčně relaci S a hashujeme hodnoty v S pro spojovací atribut a přímým přístupem do paměti získáváme odpovídající n-tice R

Operátory plánu vykonávání dotazů

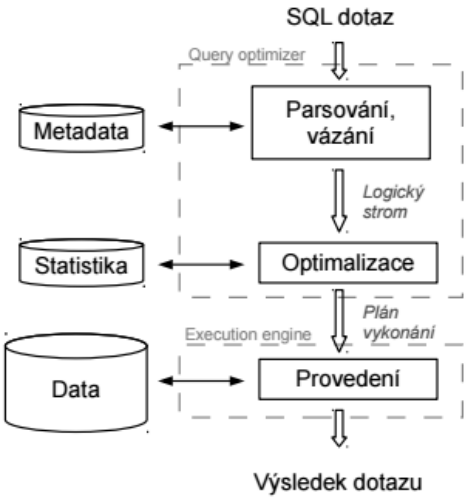
Předmluva

Query processor – část databázového systému, která se stará o zpracování dotazu, skládá se z **Query optimizer** a **Execution engine**.

Kroky vykonání dotazu

- **Parsování a vázaní**

    - **Vstup:** SQL dotaz
    - **Výstup:** Logický strom
    - Ověřuje se syntaktická a sémantická správnost dotazu
    - Ověřuje se existence použitých objektů
    - Logický strom většinou odpovídá syntaxi dotazu
    - Maximálně se provede normalizace
- **Optimalizace**
    - **Vstup:** Logický strom
    - **Výstup**: Plán vykonání
    - Při optimalizaci se cyklicky provádějí dva základní kroky: Generování možných plánů vykonání. Cenové ohodnocení každého plánu
    - V zásadě převádíme logické operátory na fyzické operátory
- **Provedení**
    - **Vstup:** Plán vykonání
    - **Výstup**: Výsledek
    - Po provedení bývá plán vykonání uložen v cache (plan cache)
    - V případě spouštění stejného dotazu pak jsou všechny kroky, které provádí Query optimizer přeskočeny



Plán vykonání – strom, kde uzly jsou jednotlivé operátory, a orientovaná hrana představuje fakt, že výstup jednoho operátoru je vstupem druhého. Operátorů je celá řada, nás budou zajímat především operátory související s vykonáním dotazů:

- Operátory pro přístup k datům
- Agregační operátory
- Spojovací operátory

Search space – množina plánů vykonání, které odpovídají vstupnímu SQL dotazu. Všechny tyto plány musí vracet stejný výsledek odpovídající vstupnímu SQL dotazu. Z těchto plánu vykonání se musí vybrat jeden, který bude použit při provedení. Cenové ohodnocení všech plánů vykonání ze search space není prakticky možné, proto se používá heuristiky pro výběr omezené množiny plánů pro cenové ohodnocení – **kandidáti**. Vybraný plán nemusí být tedy nejoptimálnější, pouze blízký nejoptimálnějšímu.

Cenové ohodnocení

- Je provedeno cenové ohodnocení každého operátoru
- Cenové ohodnocení je dáno matematickým výrazem, kde jednotlivé proměnné představují předpokládané zdroje použité při vykonání (CPU, IO a paměť)
- Velikost využití těchto zdrojů je nejvíc ovlivněna velikostí relace, která se bude v rámci operátoru zpracovávat
- Klíčové je tedy odhadnout velikost relace – k tomu se využívají statistiky

Operátory plánu vykonání

Typy operátorů

- Pro přístup k datům
- Agregace
- Spojovací operátory

Sekvenční průchod

Table Scan pro Heap tabulku, *Clustered Index Scan* pro setřízenou tabulku a *Index Scan* pro index (B-strom).

- Prochází se postupně všechny data v datové struktuře a vyhledává požadované záznamy
- Sekvenční průchod implicitně nemusí vracet data setřízená i když procházíme setřízenou tabulku. Je to ovlivněno organizací dat do stránek, kdy databázový systém jde podle pořadí stránek na disku, které nemusí korespondovat s pořadím záznamů v datové struktuře
- **Výhody**: Jedná se o základní operátor umožňující vyřešit přístup prakticky k libovolným datům v datové struktuře
- **Nevýhody:** Doba průchodu je úměrná velikosti datové struktury

Přímý přístup

Clustered Index Seek v setřízené tabulce a *Index Seek* v indexu (B-strom).

- Prochází stromovou strukturu od kořene k listům s požadovanými daty – navigace ke konkrétním datům
- **Výhody:** Jedná se o velmi efektivní přístup k datům, který zbytečně neprochází data nerelevantní k dotazu
- **Nevýhody:** není možné jej využít, pokud nevyhledáváme podle klíče. Pokud procento vrácených dat je velké, operátor je horší než sekvenční přístup.

Agregace

- Jedná se o operátory, které provádějí sumarizaci dat – sum, avg, count, min, max
- Výstupem agregačního dotazu může být jedna hodnota ale i množina hodnot
- Existují dva základní operátory, které se používají:
    - **Stream aggregate** – vyžaduje setřízení dat před agregaci
        - Pokud data nejsou setřízena, nejprve se provede operátor **Sort**, který provede setřízení
        - Dobré, pokud existuje index – v takovém případě jsou již záznamy setřízeny
    - **Hash aggregate** – vytváří v paměti potencionálně velkou hash tabulku
        - Dobrý pro velmi rozsáhlé tabulky
        - Hash tabulka se používá pro ukládání hodnot, které chceme agregovat a to podle skupiny

Nested loop join

- Pro každý záznam vnější tabulky hledáme odpovídající záznam vnitřní tabulky
- Implementace obsahuje dva zanořené cykly
- Jediný algoritmus pro spojení, které neobsahuje rovnost
- **Výhodný:** Pokud je vnější tabulka (vstup) malá a vnitřní má index na spojovacím klíči

Merge join

- Sekvenčně se prochází obě vstupní tabulky
- Kurzory se postupně posunují a nacházejí se tak odpovídající záznamy
- Obě tabulky ale musí být setřízeny podle klíče a spojovací podmínka musí obsahovat rovnost
- **Výhodný:** Cena za třízení tabulek není velká a tabulky jsou přibližně stejně velké

Hash join

- Build fáze: pro menší tabulku se vytvoří hash tabulka
- Probe fáze – postupně se prochází vnější tabulka a hledají se odpovídající záznamy v hash tabulce
- Vyžaduje se tedy vytvoření tabulky v hlavní paměti a spojovat se musí na rovnost
- **Výhodný:** Pokud tabulky na vstupu jsou velké

Key lookup

- Pokud je v SELECT dotazu požadavek na sloupec, který není součástí indexu, podívá se pomocí operátoru key lookup do setřízené tabulky pro dodatečné sloupce

RID Lookup

- Podobné jako Key lookup, ale RID lookup se odkazuje na tabulku typu halda

Statistiky hodnot v databázových systémech

- Statistiky jsou vytvářeny pro odhad kardinality (velikost výstupu dotazu)
- Odkad kardinality se provádí pro jednotlivé operátory
- Tento odhad pak umožňuje stanovit cenu operátoru
- Statistiky jsou vytvářeny
    - Implicitně – nastavením databáze
    - Explicitně – spuštěním příkazu CREATE STATISTICS nebo při vytvoření indexu

Implicitní vytvoření statistik

- Zapnuta, pokud má databáze nastaveno AUTO_CREATE_STATISTICS na ON
- Statistiky jsou automaticky vytvářeny pro
    - Klíče
    - Atributy, které se nacházejí v příkazech jako **search argument**
- Implicitní statistiky jsou vždy vytvářeny pro jeden sloupec

Search argument

- Podmínky, které jsou za WHERE nebo v JOIN podmínkách
- Argument je „search“, pokud jde vyhledat tak jak je – je to literál, typicky WHERE attr = „name“

- Pokud s atributem v podmínce provádíme nějakou operaci, pak přestává být search argument (DATEADD, násobení atributu, ABS, ČÁST, podmínka na nerovnost atd)
- V mnoha případech lze podmínku přepsat tak, aby k manipulaci nedocházelo a atributy byly search argument

Podrobnosti o existující statistice

- Object Explorer – statistics – properties – Details
- V systémových pohledech sys.stats a sys.stats_columns
- DBCC SHOW_STATISTICS(tabulka, jmeno_statistiky)

Nemožnost použití statistiky

Při použití proměnných v podmínkách výběru nemůže server vyžít statistiky, protože nezná hodnotu proměnné v době sestavování plánu. Pro odhadovaný počet řádků tak server použije průměrnou hodnotu z celé tabulky, která se může diametrálně lišit od hodnoty odpovídající hodnotě proměnné, můžeme tak dostat chybný plán vykonání.

Co statistika obsahuje

```
Name                    Updated             Rows   Rows Sampled Steps
----------------------- ------------------- ------ ------------ ------
_WA_Sys_00000007_2645B050 Feb 24 2010  2:12PM 121317 110678       200

All density Average Length Columns
----------- -------------- -----------
0.003225806 8              UnitPrice

RANGE_HI_KEY RANGE_ROWS EQ_ROWS  DISTINCT_RANGE_ROWS AVG_RANGE_ROWS
------------ ---------- -------- -------------------- ---------------
1.374        0          144.3928 0                    1
2.29         34.27779   2779.8   0                    1
2.994        429.5555   342.3352 3                    1
3.975        34.27779   1        0                    18.33333
3.99         34.27779   2064.53  0                    1
4.611        146.0489   33.46852 3                    1
```

Rows – Počet řádků tabulky, v době kdy byla statistika vytvořená nebo aktualizovaná

Rows Sampled – Počet řádků, ze kterých je statistika

Steps – Počet intervalů histogramu

Hustota (All Density)

- Hustota pro sloupec je definována jako $\frac{1}{počet\ záznamů\ s\ odlišnými\ hodnotami}$
- Bývá používána pro odhad výsledku SQL příkazu s Group By
- Na základě hustoty můžeme rovnou stanovit velikost mezivýsledku generovaného Agregačním operátorem
- Bývá také využívána u dotazů s podmínkou, kde hodnotu podmínky není možné během kompilace stanovit

Histogram

- Nejpoužívanější typ statistik
- Hodnoty v daném sloupci se rozdělí na N intervalů (N typicky 200)
- Pro každý interval pak uložíme pomocné informace jako průměrný počet řádků v intervalu apod.

- Každý řádek (interval) obsahuje
    - **RANGE_HI_KEYS** – nejvyšší hodnota intervalu
    - **RANGE_ROWS** – počet řádků v intervalu, mimo horní mez
    - **EQ_ROWS** – počet řádků odpovídající nejvyšší hodnotě intervalu (RANGE_HI_KEYS)
    - **DISTINCT_RANGE_ROWS** – počet jedinečných hodnot v intervalu mimo horní mez
    - **AVG_RANGE_ROWS** – průměrná hodnota počtu jedinečných hodnot, mimo horní mez – RANGE_ROWS/DISTINCT_RANGE_ROWS
- Histogramy jsou používány pro stanovení selektivity operátoru. **Selektivita** stanovuje, jakou část z celku daný operátor vrací.

Vytváření a aktualizace statistik

- Statistiky jsou vždy vytvářeny a aktualizovány z určitého vzorku dat
- Pokud se statistiky vytvářejí zároveň s indexem, prochází se všechna data
- Při implicitním vytváření se u větších tabulek statistiky vytvářejí jen z 5% dat
- Pokud chceme z většího vzorku, je potřeba použit příkaz CREATE STATISTICS

Cenová optimalizace

- Příklad optimalizace pro agregaci – při group by se sum funkci bylo setřízení záznamů drahé, proto se vytvořil index pro atribut, podle kterého se grupuje, který také obsahuje hodnotu, která má být spočtena. Tím je výsledek již setřízen, protože data pochází z indexu. https://www.simple-talk.com/sql/learn-sql-server/showplan-operator-of-the-week-stream-aggregate/
- Rozdíl mezi Key Lookup a přímím přístupem do setřízené tabulky – podle statistik se optimalizátor rozhodl, že použije raději sekvenční průchod setřízenou tabulkou, protože výsledek bude obsahovat příliš mnoho záznamů a použití indexu + Key Lookup by bylo drahé, protože Key lookup vyžaduje náhodný přístup.

Dvěma či více různými dotazy je možno obdržet stejná data. Rychlost různých dotazů ovšem nemusí být stejná i přesto, že vracejí stejná data. Jedním z hlavních důvodů provádění optimalizace v DB systémech je minimalizace nákladů. Obecná pravidla pro psaní SQL dotazů

- Vyjmenovávat sloupce za SELECT
- Používat co nejméně klauzulí LIKE
- Používat co nejméně IN, NOT IN
- Používat LIMIT
- Za začátek dávat obecnější podmínky (více selektivní)
- Výběr vhodného pořadí spojení
- Používat **hinty** – podnět, kterým optimalizátoru určíme, jaký má použít plán vykonávání dotazu
- Nastavit index

Optimalizace dotazů

V zásadě dochází ke dvěma typům optimalizací:

- Optimalizace plánu vykonání dotazu na základě pravidel relační algebry. Dochází k minimalizaci plánu a k vynechání redundantních částí

- Cenová optimalizace plánu vykonání dotazu na základě statistik relací. Optimalizátor odhadne velikost výsledku, cenu operací a cenu celého plánu vykonání

Hinty

Optimalizační direktivy, které slouží pro ovlivnění optimalizátoru tak, aby vybral jiný plán. Dělí se na:

- **Dotazové hinty** – jsou pro každý dotaz, zadávají se na konec dotazu pomocí klauzule OPTION
- **Join Hinty** – aplikovány na specifické spojení v dotazu, můžeme ovlivnit, jaký algoritmus se použije pro spojení
- Hinty pro tabulky

Můžeme rovněž měnit plány pro agregace. Můžeme optimalizátoru říct, že má použít specifický index. Pomocí hintu USE PLAN dokonce můžeme celý plán specifikovat ručně.

Memo struktura

- Datová struktura, která slouží k ukládání různých alternativ, které generuje optimalizátor.
- Každá operace je vložena do vlastní skupiny
- Při každé nové alternativě se do příslušné skupiny vloží alternativní operace, která produkuje stejný výsledek jako předešlé operace
- Pokud pro danou operaci neexistuje skupina, vytvoří se
- Nová memo struktura je vytvořena pro každou optimalizaci. Nejprve se nakopíruje původní logický strom do memo struktury, každý operátor do vlastní skupiny a následně začne optimalizační proces, při kterém transformační pravidla generují všechny alternativy původního logického stromu.
- Jak se tvoří nové alternativy, přidávají se operátory do příslušných skupin. Transformační pravidla mohou produkovat i nové výrazy, pro které se vytvoří nová skupina
- Memo struktura je navržena tak, aby znemožňovala vytváření stejných výrazů pomocí transformačních pravidel a tak se zabrání redundantní analýze
- Na konci se logické operátory nahradí fyzickými a podle cenového ohodnocení jednotlivých fyzických operátorů se vybere optimální verze různých operátorů ze všech skupin tak, aby se z toho poskládal plán vykonání dotazu

Group 6	Join 3 & 4	Join 1 & 5	Join 5 & 1	Nested Loops 5 & 1	**Hash Join 5 & 1**
Group 5		Join 2 & 4	Join 4 & 2	Nested Loops 2 & 4	**Merge Join 4 & 2**
Group 4	Scan Customer			**Clustered Index Scan**	
Group 3	Join 1 & 2		Join 2 & 1	Nested Loops 1 & 2	
Group 2	Scan Individual			**Clustered Index Scan**	
Group 1	Scan Contact			**Clustered Index Scan**	

Per-row vs per-index plan

- Při updatu tabulky, která má index (B-strom) musí dojít k aktualizaci i toho indexu. Podle toho jak se index aktualizuje, rozlišujeme
    - **Per-row plan –** aktualizace tabulky i indexu probíhá jedním operátorem – jeden řádek za druhým
    - **Per-index plan –** aktualizace tabulky a indexu je provedena zvlášť. Nejdřív tabulka, pak index.
- Výběr záleží hlavně na počtu záznamů, které se aktualizují. Per-row je dobrý, když se aktualizuje malé množství dotazů.

Haloween problém

Update operace se skládají ze čtení a následné aktualizace načtených hodnot. Problém může nastat, pokud procházím index, čtu z něj a aktualizuju. Tím si klíče v indexu posouvám dopředu a může nastat, že již aktualizovaný záznam načtu a změním znovu. Toto se nazývá jako **Haloween problém**. Řešením je důsledné oddělení čtení a aktualizace – nejdřív všechno načíst do paměti a až pak aktualizovat. K tomu slouží v plánu vykonání operátor **Table Spool**, což je blokující operátor, který odděluje čtení a aktualizaci. Říká, že se musí načíst všechny záznamy před tím, než se provede další operace.

Parameter sniffing

Proces, při kterém se při vytváření plánu dotazu během kompilace bere v úvahu i hodnota parametru v dotazu a na základě hodnoty se vytvoří optimální plán. Plán vykonání se ukládá do plan cache a když přijde stejný dotaz, nemusí být optimalizován znovu. Problém je, že plán uložený v cache pro konkrétní hodnotu parametru nemusí vyhovovat pro jinou hodnotu parametru, hlavně když v případě nějakého hodnoty je selektivita dotazu malá ale v cache je uložen plán dotazu pro parametr, se kterým je dotaz selektivní více.

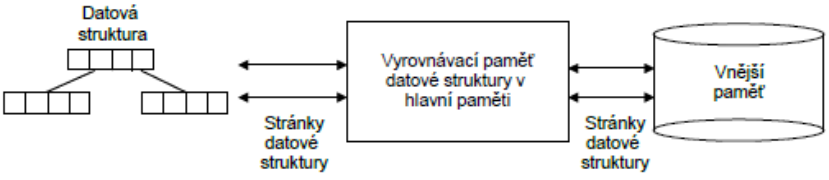
Optimalizace s ohledem na parameter sniffing

- **Optimalizace pro typický parametr**
  - Pomocí hintu OPTIMIZE FOR
  - Pro proceduru se vytvoří plán pro předem zadaný parametr a ten se uloží do cache
- **Optimalizace pro každé spuštění**
  - Pomocí hintu RECOMPILE
  - Plán se po každém spuštění procedury vygeneruje znovu
- **Optimalizace pro neznámý parametr**
  - Využitím lokální proměnné zabráníme optimalizátoru vidět hodnotu při optimalizaci. Není tedy možné ani využít při optimalizaci histogram, ale pouze hustotu

Auto-parametrizace – Optimalizátor se může rozhodnout parametrizovat dotaz a uložit v plan cache parametrizovaný. To se děje, pokud hodnota v parametru nemá ovlivnit plán, např. při dotazu na primární klíč – ten je v tabulce vždy jen jeden, takže počet výsledků dotazu bude vždy 1 a nezávisí na zadané hodnotě parametru. Parametrizaci je možné vynutit nastavením databáze: SET PARAMETERIZATION FORCED.

Fyzická implementace datových struktur a algoritmů vykonávání dotazů

Datové struktury obsahující data musí být **perzistentní** – musí být uloženy v nějaké vnější paměti, kde zůstanou uloženy i po např. restartu počítače. Datové struktury jsou **stránkované** – skládají se ze stránek, kde každá stránka je mapována na různý počet stránek vnější paměti (nejčastěji diskové bloky).



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

Stránkovaný seznam (setřízený)

Seznam, který obsahuje stránky, a samostatné záznamy jsou uloženy v těchto blocích. Každý blok bude mít jedinečné identifikační číslo a bude obsahovat ukazatel na následující blok.

Zatřizováni: záznam se zatřídí do požadovaného bloku a pokud by došlo k tomu, že by v bloku nebylo místo, pak by došlo k rozštěpení bloku. První polovina záznamů by zůstala a pro druhou polovinu by se vytvořil nový blok a ukazatel na následující blok v původním bloku by byl nastaven na blok nový.

Problémem této datové struktury je komplikovanější vyhledávání. Při binárním vyhledávání musíme vybrat nějaký prvek z prostředního bloku a porovnat jej s prvkem vyhledávaným. Poté se přesuneme buď doleva nebo doprava podle toho, zda je hledaný prvek větší nebo menší. Problémem je, že nemáme k dispozici náhodný přístup k blokům, protože uspořádání jedinečných čísel bloků neodpovídá uspořádání klíčů záznamů (poté, když se rozdělí nějaký blok na 2). Proto musíme udržovat pomocné pole, které bude obsahovat jedinečná čísla bloků seřazená dle uspořádání klíčů. Při výběru prostředního prvků bychom ID bloku našli v této pomocné struktuře.

Další nevýhodou je časová složitost vyhledávání je $O(\log_2 n)$.

Binární vyhledávací strom

Stromové datové struktury nabízí logaritmickou složitost pro všechny 4 operace. Kořenový strom je souvislý, acyklický neorientovaný graf, která má kořen. Setřízený strom je kořenový strom, ve kterém jsou potomci každého uzlu seřazeni.

Binární vyhledávací strom je struktura, která obsahuje uzly, které mají levý a pravý podstrom. Jeden uzel je kořen. Každý uzel obsahuje klíč, který se dá seřadit. Levý podstrom obsahuje uzly s ID menší než má aktuální uzel, pravý podstrom obsahuje hodnoty větší, než aktuální uzel.

Vyhledávání hodnoty probíhá rekurzivně, začíná se v kořenu a postupně se navštíví levý nebo pravý podstrom podle porovnání aktuálního klíče uzlu s hledanou hodnotu.

Pro **vkládání a mazání** je základem vyhledávácí algoritmus. Nalezne se nejprve prázdný uzel, do kterého se poté vloží nová hodnota. Z binárního stromu se může stát seznam (pokud vkládáme hodnoty od 1 do 1000), proto je důležité mít strom **vyvážený**. Vyvážený strom je takový strom, jehož hloubka všech listů je totožná.

Složitost základních operací u vyváženého stromu je $O(\log_c n)$.

B-strom

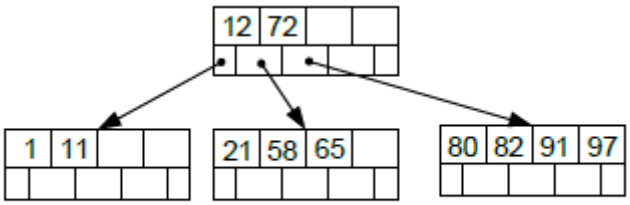
B-strom řádu *c* je (2c+1)-ární strom, který splňuje následující kritéria

- Každá stránka (uzel) obsahuje nejvýše 2c položek (klíčů)
- Každá stránka, s výjimkou kořene, obsahuje alespoň c položek
- Každá stránka je buď listovou (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 $O(\log_c n)$.

Faktor využití paměti – Jak je paměť využita pro ukládání dat, v případě B-stromu to je minimálně 50%.

Fyzická velikost stránky se volí v násobcích velikosti diskového prostoru (většinou 2048B). Dle velikosti položky stromu vybereme vhodnou kapacitu stránky (20-50). B-strom má tedy všechny vlastnosti, které požadujeme po indexovací datové struktuře.: poskytuje dobrou složitost pro základní operace, je přirozeně perzistentní a je relativně snadno implementovatelná.



Obrázek 11.3: B-strom řádu 2

Operace vyhledání

- Nastavme kořenový uzel jako aktuální
- V aktuálním uzlu hledáme pořadí položky *i* takové, že hledaná položka *k* je mezi těmito položkami
- Načteme *i*-té dítě uzlu a nastavíme tento uzel jako aktuální
- Pokud v listech není hledaná položka nalezena, algoritmus končí

Operace vkládání

- Nejprve se algoritmem vyhledávání najde listový uzel, do kterého má být položka vložena, mohou nastat tyto situace
    - Počet položek je menší než 2c, pak položku zatřídíme do uzlu
    - Počet položek je roven 2c. Nastává **štěpení uzlu -** Položku zatřídíme, vyjmeme prostřední prvek, horní polovinu prvků přesuneme do nového uzlu. Rodiče uzlu nastavíme jako uzel aktuální. V nelistovém uzlu mohou opět nastat dvě možnosti
        - Počet položek je menší než 2c, pak vyjmutou položku zařadíme do uzlu. Kromě položky je nutné zatřídit i ukazatel na nově vzniklý uzel. Algoritmus konči.
        - Počet položek je roven 2c. Položku zatřídíme, vyjmeme prostřední prvek, 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 – tu naší vyjmutou, ukazatel na původní kořenový uzel a ukazatel na nový uzel.

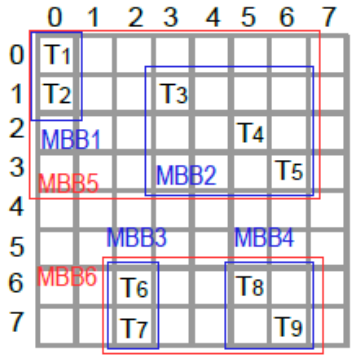
Operace mazání

- Nejprve se nalezne rušený prvek algoritmem pro vyhledávání. Mohou nastat 2 situace:
    - Položka se nachází v listovém uzlu: položka je smazána
    - Položka se nachází ve vnitřním uzlu. Mazanou položku musíme nahradit nejbližší menší nebo větší položkou ze stromu. Takové položky nalezneme úplně vlevo (případně vpravo) v podstromu.

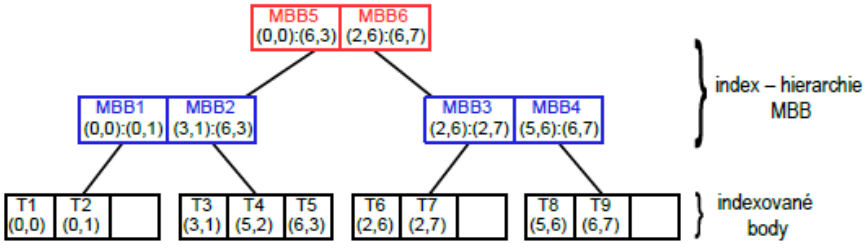
R-strom

Vícerozměrná datová struktura, která dokáže uchovávat např. více klíčů současně. Používá se v případě, kdy potřebujeme dělat vyhledávat na základě více atributů. Data uchovává v jakýsich

prostorových shlucích, které jsou pak uspořádané ve stromové struktuře. R-strom je založen na shlukování v prostoru blízkých bodů na stejné stránky. Pro body jsou konstruovány **minimální ohraničující obdélníky**, které shlukují podobně, rozuměj blízké, body. Body jednoho obdélníku jsou uloženy v listovém uzlu. Vnitřní uzly obsahují hierarchii obdélníků.



Obrázek 11.5: Indexovaný dvourozměrný prostor



Obrázek 11.6: R-strom indexující tento prostor

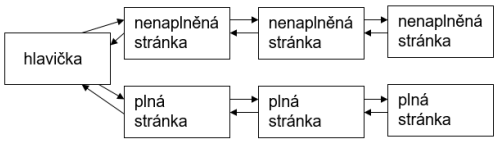
Mezi nejpoužívanější dotazy patří **bodový** a **rozsahový dotaz**. Bodový dotaz je definován bodem a vrací true, pokud se bod v datové struktuře nachází, jinak vrací false. Rozsahový dotaz vrací všechny body, které se nachází v definovaném hyperkvádru – **dotazovacím obdélníku**.

Rozsahový dotaz

- Nastavíme kořenový uzel jako aktuální
- Hledáme, zda datová struktura obsahuje obdélník, který protíná dotazovací obdélník.
- Pokud takový nalezneme, načteme dítě náležející k tomuto obdélníku a nastavíme je jako aktuální
- Znovu hledáme obdélník, který protíná dotazovací obdélník. Takto se postupuje až k listovému uzlu, ve kterém kontrolujeme, zda neobsahuje body ležící v dotazovacím obdélníku
- Nyní se vracíme zpět (aktuální cesta je ukládána na zásobník) a pokračujeme v prohledávání nezkontrolovaných částí stromu.

Halda

Záznamy jsou ukládány na první volnou pozici bloku a pokud je blok plný, najde se nějaká další stránka, které ještě není plná a ukládá se do ní. Pro vkládání tedy existuje ukazatel na první volný blok, do kterého se záznam uloží (složitost O(1)). Při vyhledávání se musí projít všechny záznamy v blocích (složitost O(n)).

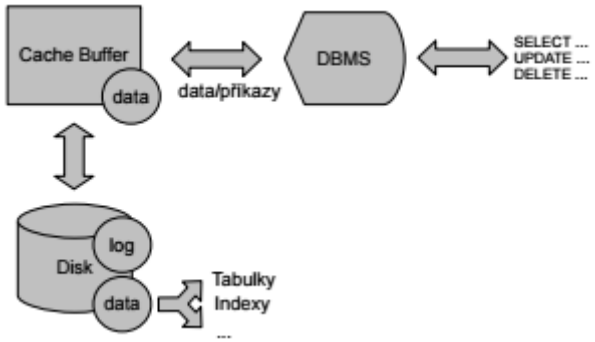


Hash Tabulka

Řádky se stejnou hashovací hodnotou (hash klíče) jsou mapovány do stejného bloku. Bloky náležící jedné hašovací hodnotě jsou propojeny do seznamu. Operace vkládání a vyhledání je O(k), kde k je počet bloků jedné hash hodnoty.

Optimalizace přístupu do hlavní paměti a disku

Každá datová struktura je stránkovaná. **Stránky** jsou umístěny na disku tak, aby databázový systém zaručil trvalost dat. Velikost stránky je volena jako násobek 2 kB (nejčastěji 8kB) – násobek diskového sektoru (512 B) a alokační jednotky souborového systémů (nejčastěji 2 kB). Důvodem je snaha o snížení režie IO operací. Databázový systém pracuje s daty, které jsou umístěné v hlavní paměti (cache buffer), protože hlavní paměť je klidně i 100 násobně rychlejší než disk. Pokud chceme optimálního výkonu databáze, musí být většina dat umístěna v hlavní paměti (v cache buffer). Hranice je 90% - v nejhorším případě bude 10% dat umístěno na disku, zbytek v paměti. Propustnost disku je do 500MB/s, u hlavní paměti to může být až 6GB/s.



Požadujeme-li stránku tabulky nebo indexu a ta je v cache – **cache hit.** Pokud ji musíme načíst z disku, mluvíme o **cache miss**.

Databázové systémy dělíme na dva typy:

- Disk-based – stránkované datové struktury na disku, použití cache buffer
- In-Memory – databáze celé v hlavní paměti

V dnešních databázových systémech se pak oba přístupy kombinují, proto je snahou **optimalizovat běh databázového systému v hlavní paměti**: **algoritmy se především snaží využít L2 cache CPU.**

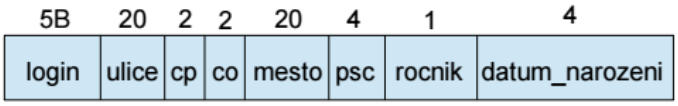
Cache je tvořena rychlou pamětí, která je však dražší. Proto má cache menší velikost než úložný prostor, ke kterému zrychluje přístup. Obecně se používá jako vyrovnávací paměť pro pomalé paměti. V cache se uchovávají často používané data, takže není nutné tak často sahat do pomalejší paměti. L2 cache je cache procesoru a uchovává kopie dat přečtených z adresy v operační paměti. Mezi CPU a cache paměti se přenášejí jednotlivá slova, mezi cache pamětí a operační paměti se přenášejí rámce slovo velikosti několikanásobku velikosti slova procesoru.

Proto je důležité optimalizovat přístup k paměti tak, aby byla cache co nejlíp využita – tak aby když se načtou data do cache, aby obsahovala co nejvíc dat, které budou v nejbližší době použity.

Přístupy do hlavní paměti jsou zrychlovány pomocí L2 cache, která může mít kolem 2 MB s latencí 5ns. Hlavní paměť může mít i TB s latencí 100ns. Optimalizace přístupu do paměti pomocí L2 cache můžeme tedy získat až 20x rychlejší kód.

Asi nejpoužívanější přístup k optimalizaci pro L2 cache je použití **lineární paměti pro ukládání dat**. Objektový přístup ukládá hodnoty záznamů do objektů (např. pro adresu budeme mít jeden objekt, pro datum narození bude objekt date). Nevýhodou je, že potřebujeme více paměti a hlavně jednotlivé hodnoty mohou ležet v hlavní paměti daleko od sebe, což znamená vyšší počet cache miss při přístupu k jednotlivým položkám – jednoduše se načte do cache záznam, ale např. adresa v cache být nemusí, nebo další záznam v cache být nemusí, protože byl od předešlého záznamu v hlavní paměti daleko (objekty nejsou uloženy v paměti za sebou, ale pomocí referencí ukazují na nějaké místo v haldě).

Přístup s lineární pamětí ukládá data bez další prostorové režie, ukládají se do jedné větší alokované části paměti a to za sebou, takže po přístupu k první hodnotě dojde k načtení i ostatních hodnot do L2 cache, k ostatním hodnotám pak přistupujeme vždy jen do L2 cache.

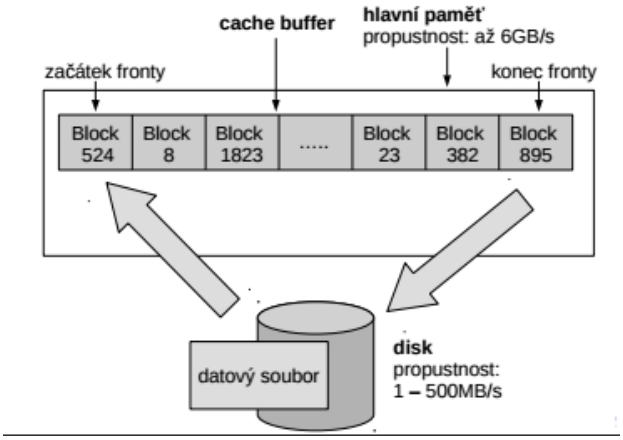


Postup je takový, že se alokuje lineární paměť (pool) pro jednotlivé stránky. Pokud požadujeme novou stránku, z lineární paměti se vrátí ukazatel na první volnou pozici a tento ukazatel se posune o velikost požadovaného bloku. Do těchto stránek následně ukládáme jednotlivé záznamy za sebou. Každý záznam se skládá z několik atributů (obrázek), které se za sebou ukládají do stránky.

Další příklad: Když mám úkol nějakým způsobem zpracovat jeden atribut tabulky, např. porovnávat jednotlivé hodnoty a hledat, které sloupce odpovídají podmínce, můžu si jednotlivé atributy uložit sloupcově. Při klasickém uložení musím načíst celý záznam, do L2 cache se uloží i další nepotřebné atributy a naopak tam nebudou hodnoty požadovaného atributu a bude docházet k velkému množství cache miss. Pokud si to uložím sloupcově, tak se při prvním načtení hodnoty uloží do L2 cache i další hodnoty daného atributu, nic jiného.

Návrh a implementace cache buffer

Cache buffer – slouží k výměně stránek mezi diskem a hlavní pamětí. Je to fronta, která uchovává stránky načtené z disku. Každá stránka je identifikována svým ID. Pokud daná stránka není v cache buffer, musí být načtena z disku do cache. Cache buffer si můžeme představit jako frontu stránek, kde první stránka byla použita jako poslední a poslední stránka jako první. Pokud databázový systém požaduje načtení stránky z disku a cache je plná, pak dojde k uložení poslední stránky na disk a na volné místo je načtena požadovaná stránka. Tento algoritmus je nazýván **LRU (Least Recently Used)** tedy nejméně používaná stránka. K ostatním stránkám přistupoval DB systém až po přístupu k odkládané stránce a můžeme se domnívat, že jsou v daný okamžik důležitější než odkládána stránka.



Obrázek: Naposledy zpracovaná stránka je 524, poslední stránkou je 895 – ta bude odstraněna z cache při dalším požadavku na stránku, která není v hlavní paměti.

Implementace

- Třída představující Cache bude mít lineární paměť pro jednotlivé stránky. Tato paměť bude rozdělena do *n* **slotů**, v jeden okamžik bude jeden slot obsahovat pouze jednu stránku
- Musí udržovat počet stránek v cache, velikost stránky a aktuální číslo stránky, která je na začátku fronty
- Dále obsahuje paměť pro metadata o slotech
- **Metadata** budou pro každý slot udržovat informace o
    - Pořadové číslo slotu
    - Číslo stránky, která je aktuálně uložená ve slotu
    - Příznak, zda někdo čte uzel
    - Příznak, zda někdo zapisuje uzel
    - Příznak, zda byla stránka modifikována
- **Čtení stránky –** parametrem je číslo stránky
    - Hledáme slot, který obsahuje stránku s daným ID
    - Při nalezení nastavíme v metadata příznak, že někdo čte uzel
    - Pokud chceme do stránky zapisovat, nastavíme příznak pro zápis a příznak změny
    - Při nenalezení: Nalezneme nejstarší stránku (hledáme pomocí příznaku timestamp), tuto stránku odložíme (na disk) a případně na uvolněné místo načteme stránku z disku s daným ID.
    - Při fyzickém zápisu stránky na disk musíme zkontrolovat, zda jiné vlákno nepracuje s uzlem (pomocí zkontrolování příznaků). Stránku nemusíme ukládat, pokud nebyla modifikována (opět příznak v metadata)

Varianty

- **LRU (Least-Recently Used)** – stránka, která je nejstarší – před ní byly použity všechny ostatní stránky, se z cache vymaže – implementace pomocí fronty
- **LFU (Least-Frequenty User)** – pryč půjde ta stránka, která byla použita nejméně. Ukládá se počet načtení stránky z cache a ta, která je načítána nejméně je nahrazena novou stránkou

Objektově-relační datový model

Objektový model jsou data uložená v objektové struktuře, umožňují používat uživatelské typy, dědičnost, metody tříd. Fyzická implementace je ale komplikovanější a také pro většinu aplikací je

postačující relační datový model. Proto se přistoupilo ke kompromisu mezi relačním datovým modelem a objektovým modelem.

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. Objektově orientované prvky v relačních datových modelech:

- Uživatelsky definované datové typy s atributy i metodami
- Uložené procedury, triggery – metody uložené v databázi s cílem přenést funkcionalitu na server
- Kolekce (pole proměnné délky, vnořené tabulky)
- Zavedení dědičnosti
- Datový typ ukazatel, reference a dereference

Objektově-relační datový model je klasická tabulková databáze rozšířená o **abstraktní datové typy** – uživatelem definované typy skládající se ze základních datových typu. Nejsou tedy atomické a porušují 1NF.

- Objektové rysy jsou dnes implementovány ve všech velkých databázových systémech
- Objektové typy a jejich metody jsou uloženy spolu s daty v databázi
- Objekty mohou jednoduše reprezentovat vazby, kdy jedna entita se skládá z jiných entit (bez nutnosti použití vazeb)
- Metody jsou spouštěny na serveru – nedochází k neefektivnímu přenosu dat po síti

Objektové datové typy

```
CREATE OR REPLACE TYPE TAddress AS OBJECT (
        Street VARCHAT2(30),
        City VARCHAR2(30),
        PSC NUMBER(5)
);
```

- Mohou obsahovat jak **data (atributy)** a **operace (metody)**. Na data se můžeme dívat jak z relačního, tak i z objektového modelu.
- **Typy metod:**
    - **Členské metody –** jsou volány konkrétním objektem
    - **Statické metody –** jsou volány nad datovým typem
    - **Konstruktor –** pro každý datový typ je definován implicitní konstruktor
- **CREATE OR REPLACE TYPE nazev UNDER TAddress –** dědění
- **NOT FINAL –** označení typu, ze kterého lze dědit
- **NOT INSTANTIABLE –** označení typu, jehož instanci nelze vytvořit
- Objektové datové typy lze používat při definici atributů podobně jako SQL datové typy: **contact TAddress**
- **Vytvoření pomocí INSERT –** Taddres(hodnoty…)
- **Volaní metody –** SELECT c.contact.get_idno() FROM contacts c;

Objekty mohou být uloženy ve dvou typech tabulek:

- **Objektové tabulky**
    - obsahují pouze objekty, kde každý záznam reprezentuje objekt
    - Objekty, které jsou sdílené dalšími objekty by měly být uloženy v objektových tabulkách, protože takto mohou být referencovány

    - Mluvíme o tzv. řádkovém objektu
    - **CREATE TABLE tab OF TAddress**
- **Relační tabulky**
    - Obsahují objekty spolu s ostatními daty
    - Objekt je veden jako atribut – sloupec
    - Mluvíme o tzv. sloupcovém objektu
    - **CRATE TABLE contacts (name VARCHAR(30), contact TAddress);**

REF – ukazatel nebo reference na objekt objektové tabulky, nahrazuje cizí klíč

Datové typy

- **Record –** můžeme vytvořit pomocí %ROWTYPE
- **Kolekce**
    - **Asociativní tabulka –** obdoba hash table; obsahuje dvojice <klíč, hodnota>
    - **Vnořená tabulka –** můžeme uložit v databázové tabulce; pole proměnné délky
    - **Pole (varray) –** pole pevné délky, můžeme uložit v DB tabulce; při vytváření definujeme max. velikost pole

XML datový model

Specifikace XML nám umožní vytvářet a zpracovávat značkovací jazyky pro popis **slabě strukturovaných dat**. Relační data považujeme za data strukturovaná a čistě textové dokumenty za data nestrukturovaná. Slabě strukturovaná data mají nějakou strukturu, nicméně schéma těchto dat může být volnější než v případě pevně nastaveného schématu relační databáze.

XML (eXtensible Markup Langugage)

- Značkovací jazyk specifikovaný organizací W3C v roce 1998
- Jak pevná množina značek (např. HTML), tak i vlastní definovaná struktura
- XML dokument je reprezentován jako **strom**
- Obecně se jedná o jazyk popisující slabě strukturovaná data, která reprezentuje informace pomocí elementů, které mohou obsahovat i další vnořené elementy a atributy

Dobře strukturovaný XML dokument

Dobře strukturovaný XML dokument musí splňovat některá syntaktická pravidla:

- Element je specifikován názvem, který má počáteční značku <nazev> a koncovou značku </nazev>. Mezi počáteční a koncovou značkou je obsah elementu – další elementy nebo text
- Pokud element nemá obsah, lze vynechat koncovou značku a psát <nazev/>
- Element může obsahovat atributy ve formátu název=hodnota; obecně platí, že elementy by měly obsahovat data a hodnoty atributů metadata (data o datech)
- Rozlišujeme pořadí elementů

Schéma XML dokumentu

- Říká, jaké typy elementů se v dokumentu vyskytují, jakého typu jsou podelementy, definují strukturu XML dokumentu
- Původní jazyk pro popis schémat se nazývá DTD, novější je nazýván XML Schema (standard W3C)
- XML Schema umožňuje definovat datové typy hodnot elementů. Tato vlastnost je výhodná zejména u XML databází.

Dotazovací jazyk

XPath

XPath je jazyk používaný pro identifikaci uzlů v XML dokumentu. XPath především umožňuje vyjádřit relativní cestu od nějakého XML uzlu k jinému elementu nebo atributu. Připomíná dotazovací jazyk SQL, hlavně protože vrací jeden nebo množinu záznamů, které odpovídají vstupní podmínce. Pomocí tohoto jazyka lze vybírat jednotlivé elementy a pracovat s jejich hodnotami a atributy.

Základní součástí jazyka je **path expression – výraz popisující cestu** (dotaz na základě cesty v XML dokumentu, podobně jako cesta do nějaké složky). Posloupnost mezi jednotlivými elementy se skládá z názvu těchto elementů oddělených lomítky. Každý přechod je určen pomocí tří složek:

- **Osa** – popisuje směr pohybu po stromové struktuře XML (např. osa child, descendant, parent, ancestor)
- **Test**
- **Predikát** – seznam podmínek v hranatých závorkách

Náležitosti:

- **Lomítko** – pokud je na začátku, výraz není vztažen k aktuálnímu elementu, ale počítá se od kořene
- **Dvě lomítka –** překovává víceúrovňovou strukturu; //moznost – vrací všechny elementy daného jména v jakémkoliv kontextu, nemusí byt daný element přímím potomkem.
- **Test uzlu –** určuje název elementu
- **Podmínky jsou uvedeny v hranatých závorkách –** //moznost[@hlasu="5"] – všechny elementy jejichž atribut hlasu má hodnotu 5

XQuery

- Dotazovací jazyk, který je bohatší než XPath
- Postaven na výrazech
- Může být zapsán uvnitř nějaké HTML stránky stejným způsobem, jakým bývají psány dynamické WWW stránky
- Nejběžnějšími výrazy v XQuery jsou tzv. FLWR výrazy (flower) – obdoba příkazu SELECT, FROM, WHERE

FLWOR

- **FOR –** primární výběr uzlů generující seznam
- **LET –** definice proměnných pro každý prvek seznamu posloupnosti
- **WHERE –** tvorba logické podmínky filtrující prvky seznamu
- **ORDER BY –** seřazení vybraných a odfiltrovaných prvků
- **RETURN** – generování výstupu pro každý vybraný a odfiltrovaný prvek

Příklad:

```
for $x in doc("books.xml")/bookstore/book
where $x/price>30
order by $x/title
return $x/title
```

Více o XQuery na http://www.kosek.cz/xml/2005devcon/

Datová vrstva informačního systému

Informační systém – složitější softwarová aplikace, která ukládá a dotazuje se na data v databázi. Při vývoji IS neřešíme jen problém tvorby samotné databáze, ale i aplikace, která databázi využívá.

Architektura – soustava pojmů, prvků, struktur a interakcí systému z pohledá vnějšího pozorovatele. Návrháři vytvářejí celkový obrázek systému a mohou tak jednodušeji přiřadit jednotlivé části návrhářům a vývojářům. Vhodně zvolená architektura umožní zvládnout další růst systému nebo změny požadavků. Architektura by nám měla umožnit izolovat zbytek systému od změn v nějaké z jeho částí. Umožňuje opakovatelnou použitelnost konceptů, prvků a struktur při rozšiřování systému.

Máme několik typů architektur (jako podle čeho se to dělí):

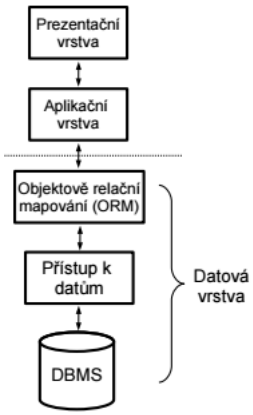
- **Metody a třídy** – logicky oddělují aplikaci na objekty
- **Subsystémy** – balíky těsně svázaných tříd s dobře definovaným rozhraním.
- **Úkoly** – Vertikální členění aplikace podle oblastní (věda, studium, ekonomika). V této architektuře je snížena pravděpodobnost interakce mezi subsystémy.
- **Aplikační 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 architektura** – Logické členění subsystémů aplikace do vrstev, např. prezentační, logika databáze, ukládání dat. Takové členění umožňuje jednodušší pochopení a jednodušší vývoj v rámci vrstvy. Vyšší vrstvy jsou izolovány od změn v implementaci nižších vrstev.
- **Hierarchická architektura** – Členění aplikace mezi více procesů. Nastává uvnitř vrstev, z tohoto důvodu zvyšuje vnitřní složitost vrstev.

Nejčastěji používanou architekturou je architektura **vrstvená.** Komunikace mezi klientem a databázovým systémem pracuje na principu **klient-server**. Klient je webový prohlížeč, který komunikuje se serverem pomocí HTTP (request/response).

Platformy pro vývoj webových informačních systémů – ASP.NET (.NET platforma) a Java2EE

Datová vrstva

Klasickým problémem při implementaci je překonání propasti mezi relačními databázovými systémy a objektově orientovaným jazykem.



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:

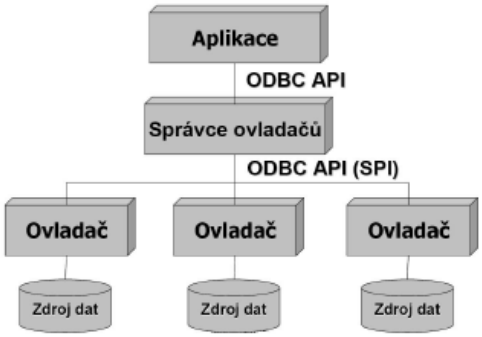
- Otevřít spojení s databází
- Zaslat dotaz
- Získat výsledek
- Uzavřít spojení s databází

Datová vrstva musí mít maximální výkon přístupu k datům.

Existující API

Open DataBase Connectivity (ODBC)

Standard, který umožňuje dotazování dat z aplikace bez ohledu v jaké databázi jsou uložena. Do architektury ODBC je vložena mezivrstva – **ovladač**, který překládá uživatelské dotazy na dotazy databáze. **Správce ovladačů** (driver manager) spravuje ovladače, kterých může aplikace využívat více.



Obrázek 16.1: Architektura ODBC

Provedení dotazu je následující:

- Připojení k datovému zdroji
- Inicializace
- Vytvoření a provedení dotazu
- Získání výsledku
- Ukončení transakce
- Odpojení od datového zdroje

Stejné schéma je i u dalších API:

Java Database Connectivity (JDBC)

ODBC je standard definující funkce pro komunikaci s databází. S nástupem OOP vyvstal požadavek na specifikace objektového rozhraní k databázi. Toto poskytuje pro jazyk Java JDBC. JDBC používá vlastní ovladače, pokud není takový ovladač pro danou databázi k dispozici, můžeme použít bridge mezi JDBC a tímto ODBC ovladačem.

Ovladač je v JDBC představován třídou dodávanou většinou výrobcem konkrétního databázového systému. Príklady: **sun.jdbc.odbc.JdbcOdbcDriver** nebo **com.mysql.jdbc.Driver**

Příklad komunikace: Probíhá to tak, že se nejdříve nadefinuje connection string, který obsahuje přístupové údaje k databázi, následně se připojíme k databázi a provedeme příkaz pomocí metody *execute***.** Výsledek zpracujeme a uzavřeme spojení.

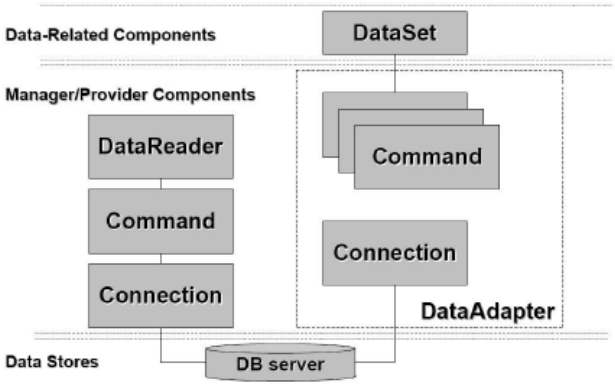
Implicitně je v JDBC každý zaslaný příkaz proveden jako samostatná transakce. Pokud chceme více příkazů provést v jedné transakci, musíme nastavit autoCommit na false.

Připojení k databázi je drahá operace, z těchto všech důvodů se nejčastěji používají tzv. Pooly připojení: při startu aplikace se otevře několik připojení. Transakce pak získá právě volné připojení, které nesdílí s paralelními transakcemi.

ADO.NET

Specifikace přístupů k datovým zdrojům na platformě .NET. ADO.NET (Microsoft ActiveX Data Objects .NET) představuje množinu tříd nabízejících služby pro přístup k datům a tvorbu databázových aplikací. Daty máme nyní na mysli převážně informace uložené v databázích. Ať již se jedná o data v databázích například na Microsoft SQL Serveru či data zpřístupněná přes OLE DB nebo XML. Mezi jeho přednosti patří především jednoduchý způsob použití, rychlost při zpracování a další. Stačí vytvořit spojení se serverem, s kterým budeme chtít pracovat, pomocí zvoleného adaptéru a zadaného dotazu získat z databáze data a ty pak načíst do některé z připravených konstrukcí pro práci s daty z tabulek.

Nástroje ADO.NET byly navrženy tak, aby se oddělil způsob přístupu k datům od manipulace s daty. K první skupině patří .NET Framework data provider obsahující množinu komponent zahrnujících podmnožiny Connection (připojení), Command (množinu příkazů pro vybrání dat), DataReader (načítání dat) a DataAdapter (adaptér pro připojení k databázi). K druhé skupině řadíme mimo jiné objekt DataSet (skládající se z objektů DataTable, DataRow...). Jedná se o objekty uchovávající data načtená z databáze. Tyto objekty mohou s daty pracovat stejně jako s daty v databázi.



Obrázek 16.4: Třídy specifikace ADO.NET

Obsahuje následující třídy:

- **DataReader –** Čte data z databáze. Připraví se příkaz pomocí třídy Command a pomocí DataReader se příkaz provede,
- **DataSet –** uchovává data z databáze v aplikaci. Do toho se načtou data z aplikace a k tomu musíme využít DataAdapter. DataSet může být kolekce z více tabulek. Pro jednu tabulku existuje DataTable
- **DataAdapter –** most mezi DataSet a databází. Načítá data z databáze do DataSetu.

- **SqlConnection -** třída, která vytvoří připojení k databázi, výběr ze jmenného prostoru závisí na použité databázi (např. Systém.Data.SqlClient). Pro připojení se používá connection string, který obsahuje název databáze, login, heslo a další parametry.
- **DataAdapter –** představuje sadu příkazů SQL (insert, update, delete); naplňuje nebo umožňuje uložit změny z DataSet.
- **DataSet –** uchovává data z databáze v aplikaci.

Rámce a implementace

Aplikační rámec (není to Framework?) je sada spolupracujících tříd a rozhraní určených pro řešení specifického problému. Komponenty aplikačního rámce jsou opakovatelně použitelné. Využití aplikačního rámce nám má ušetřit práci. Příklad: GridView v ASP.NET pro zobrazení množiny záznamů).

Aplikační rámce pro Javu

- Barracuda
- Cocoon
- Expesso
- Velocity
- Java Sever Faces
- Jakarta Struts Framework

Jakarta Struts Framework

- Návrhový vzor MVC (Model-View-Controler)

Pro PHP – Nette (taky MVC), obsahuje vrstvu Database pro práci s databází.

Bezpečnost

Co udělat pro bezpečnost v databázových systémech

- Oddělit administrátorské pravomoci od pravomoci aplikace – ponechat aplikaci jen nejnutnější pravomoci, ne vytváření a odstraňování tabulek. Řízení práv obecně
- Silné hesla pro připojení k databázi
- Pravidelné zálohy celé databáze
- Neukládat hesla v plain podobě – hashovat je například SHA algoritmem
- Používat šifrované spojení
- Aplikaci zabezpečit tak, aby nedošlo k tzv. **SQL injection.** To zahrnuje ošetření vstupů od uživatele – escapování potencionálně nebezpečných znaků.

SQL Injection

Technika napadení databáze pomocí vsunutí části kódu pomocí neošetřeného vstupu od uživatele. Kód obsahuje nějaký poškozující SQL příkaz, vykonáním kterého může dojít např. k smazání databáze nebo výpisu uživatelských jmen a jiných citlivých údajů. Stačí, když uživatel zadá např. **a' or 'b'='b** a místo jednoho záznamu se vypíše celý obsah tabulky.

Objektově-relační mapování

Objektově relační mapování je programovací technika zpřístupňující relační data pro objektové prostředí. Některé ORM nástroje (Hibernate, LINQ) se pokouší vytvořit vrstvu mezi aplikací a databází tak, aby aplikace byla nezávislá na databázi. Tím se ale degraduje výkon.

Můžeme na to jít dvěma způsoby

- **Pomocí nástrojů třetích stran** – Hibernate, Entity Framework
    - Musíme definovat mapování konfiguračním souborem nebo anotacemi ve zdrojovém kódu
    - **Výhody:**
        - Rychlejší tvorba aplikace
        - Jednoduchá změna DB systému
    - **Nevýhody**
        - Pouze částečná kontrola nad generovanými SQL příkazy
        - Nižší výkon
- **Pomocí vlastní implementace**
    - **Výhody:**
        - Plná kontrola nad SQL příkazy
        - Můžeme používat rysy specifické pro konkrétní DB systém (datové typy, funkce atd)
    - **Nevýhody:**
        - Doba strávená implementací
        - Problematická změna databázového systému

Mapování

- Pro data: Jedna třída reprezentuje jeden entitní typ (jeden objekt reprezentuje jeden záznam tabulky) (Data transfer objekt – DTO)
- Pro operace: Jedna třída reprezentuje tabulku (a její operace) – Data Access Object (DAO)
- **Doménové objekty** – třídy reprezentující entitní typy (tabulku)
- **Třídy pro práci s tabulkami**
- **Pomocné třídy** – řízení spojení, transakcí apod.
- Vazby mezi entitami jsou reprezentovány referencemi na příslušný objekt
- Pro vazbu 1:N použijeme kolekce
- DAO – obsahuje metody pracující s tabulkou – implementuje všechny funkce z funkční analýzy

Distribuované ŠRBD

Distribuované systémy řízení báze dat – data jsou uložena v různých databázích běžících na libovolném serveru v síti – **uzlech**. Vývojář ale s daty pracuje na logické úrovni, nezajímá se o fyzické uložení v jednotlivých DB systémech.

Příklad použití: informační systém banky. Data jsou fyzicky umístěna blízko zákazníkům tam, kde se budou nejčastěji používat. Účty různých poboček nicméně musí být dostupné i z dalších poboček.

Základní pravidlo – Distribuovaný systém se pro uživatele musí jevit jako systém nedistribuovaný. Všechny problémy distribuovaných systémů se tedy musí řešit na úrovni implementace DB systémů, ne na uživatelské úrovni.

Základní vlastnosti distribuovaných DB systémů

1. **Lokální autonomie**

Uzly musí být autonomní a všechny operace na daném uzlu jsou řízeny tímto uzlem. Uzel řídí vlastní integritu dat, bezpečnost, transakce, fyzické uložení atd. Uzel nesmí být závislý na jiném uzlu, protože pád jednoho uzlu by znamenal pád závislého uzlu. Prakticky ale není úplná autonomie možná.

2. **Nezávislost na centrálním prvku**

Z první vlastnosti plyne, že všechny uzly jsou si rovny. Nemůžeme tedy využívat nějaký centrální uzel pro centrální vykonávání dotazů, centrální řízení souběhu atd. Protože toto místo by bylo úzkým místem systému, což by mělo za následek problémy s výkonem a systém je velmi zranitelný – pád centrálního uzlu znamená pád celého systému. V reálných DB systémech ale často pro některé operace využíváme nadřazenosti některého uzlu.

3. **Nepřetržitý provoz**

Distribuované systémy mohou poskytovat vyšší **spolehlivost** a **dostupnost.**

- **Spolehlivost (reliability)** – pravděpodobnost, že systém je v daném okamžiku funkční. Pro DSŘBD je vyšší, protože mohou pracovat (s nějakou omezenou funkcionalitou) i v případě výpadku nějakého uzlu.
- **Dostupnost (availability)** – je pravděpodobnost, že systém je funkční během definovaného období. Dostupnost u DSŘBD je opět vyšší protože využíváme replikaci.

4. **Nezávislost umístění**

Definuje, že uživatel se nemusí zajímat o to, na kterém uzlu jsou data uložena. Práce s daty ale musí být stejná jako v případě lokálně uložených dat. Tato podmínka umožňuje přesun dat na jiný uzel, aniž by to uživatel poznal.

5. **Nezávislost fragmentace**

Fragmentace dat – znamená, že data jsou uložena na části (fragmenty).

- Stejné fragmenty mohou být uloženy na několika uzlech systému – což se nazývá replikace
- Fragmentace se používá především kvůli výkonu: data jsou umístěna v uzlu, ve kterém jsou nejčastěji používána.
- **Nezávislost fragmentace** říká, že uživatel nesmí poznat, zda jsou data fragmentována nebo nejsou

6. **Nezávislost replikace**

Replikace dat – fragment je uložen v několika kopiích v uzlech systému. Uživatel nesmí poznat, zda jsou data replikovaná nebo ne. Replikace musí být podobně jako fragmentace transparentní k uživateli.

7. **Distribuované vykonávání dotazů**
8. **Distribuované řízení transakcí**

Řízení transakcí zahrnuje jak zotavení, tak souběh. V distribuovaném prostředí transakce zahrnuje provedení operace na několika uzlech. Každá transakce se skládá z **agentů** – proces vykonávaný pro danou transakci a daný uzel. Musíme zajistit, aby byl na všech uzlech proveden COMMIT nebo na všech uzlech proveden ROLLBACK. Toto je zajištěno **dvoufázovým potvrzovacím protokolem**.

Dvoufázový potvrzovací protokol

- Týká se distribuovaného prostředí, kdy transakce musí komunikovat s různými uzly

- Pokud je transakce ukončena úspěšně, pak operace na všech uzlech musí být potvrzeny, pokud je operace zrušena, pak operace na všech uzel musí být zrušeny
- **Koordinátor** se stará o řízení distribuovaných transakci

Koordinátor vykonává následující dvoufázový proces

- **Příprava**
    - koordinátor zasílá všem uzlům zahrnutým v transakci příkaz
    - Lokální manažer transakcí provede zápis do logu pro operace dané transakce. V případě úspěchu vrátí účastník koordinátorovi OK, jinak Not Ok
- **Potvrzení**
    - Pokud koordinátor obdržel odpověď od všech účastníků, zapíše záznam do svého logu i s celkovým výsledkem operace. Pokud koordinátor obdržel od všech Ok, pak je výsledkem COMMIT, jinak ROLLBACK
    - Koordinátor pak musí oznámit všem účastníkům, aby provedli lokální COMMIT nebo ROLLBACK v závislosti na výsledku transakce

Funkce koordinátora je obvykle přeřazena uzlu, na kterém byla transakce inicializována.

9. **Hardwarová nezávislost**
10. **Nezávislost na OS**
11. **Nezávislost na síti**
12. **Nezávislost na ŠRBD**

Problémy distribuovaných systémů

- **Vykonávání dotazů –** Optimalizátor vybírá globální strategii vykonání dotazu na základě velikosti vstupních relací a parametrech sítě mezi uzly (přenosová rychlost atd) – protože např. při spojování tabulek se musí přenášet data
- **Správa systémového katalogu –** Katalog obsahuje další informace – o fragmentech a replikách. Uložení katalogu může být:
    - **Centralizované -** na jednom centrálním uzlu
    - **Plně replikované** – každý uzel obsahuje kompletní katalog
    - **Rozdělené** – každý uzel obsahuje vlastní katalog. Kompletní katalog vzniká spojením katalogů všech uzlů
    - **Kombinace centralizované a rozdělené** – každý uzel má svůj katalog, centrální uzel spravuje jednotnou kopii všech lokálních katalogů
- **Propagace aktualizací** – při replikaci je nutná aktualizace všech kopií, problém nastane, kdy jeden z uzlů není v okamžiku aktualizace dostupný. Okamžitá aktualizace tak není možná. Řešením je **primární kopie**. Aktualizace se považuje za dokončenou, pokud dojde k aktualizaci primární kopie. Uzel s primární kopií je pak zodpovědný za šíření změny ke všem sekundárním kopiím.
    - **Synchronní replikace** – propagace změn musí být provedena před ukončením transakce
    - **Asynchronní replikace** – propagace změn k sekundárním kopiím nemusí být provedena v rámci transakce, ale až potom v nějakém čase – není garantována konzistence databáze
- **Souběh** – založen na uzamykání. Požadavky pro nastavení a uvolnění zámků jsou reprezentovány zprávami, které představují zvýšenou režii.

Fragmentace a replikace

Fragmentace

Fragmentace – data jsou rozdělena na části (fragmenty)

Příklad: Tabulka bankovních účtu může být fragmentována do několika uzlů podle pobočky účtu pomocí příkazu

```
FRAGMENT Account AS
O_ACCOUNT AT SITE 'Ostrava' WHERE pobočka='B1',
M_ACCOUNT AT SITE 'Michigan' WHERE pobočka='B2' OR pobočka='B3';
```

Dělení fragmentace:

- **Horizontální fragmentace**
  - pomocí operace selekce, která musí vytvářet bezeztrátovou ortogonální dekompozici – fragmenty mají nulový průnik
  - Rozdělí se tabulka na řádky
- **Vertikální fragmentace**
  - Pomocí operace projekce, která musí vytvářet bezeztrátovou dekompozici – žádný fragment tedy nemůže být odvozen z ostatních fragmentů
  - Rozdělí se tabulka na sloupce

Rekonstrukce původních dat ze segmentů je možná pomocí:

- Operace **join** v případě vertikálních fragmentů
  - Některé systémy vkládají skrytá jedineční čísla záznamů (TID), které slouží k identifikaci řádků při fragmentaci sloupců – {TID, col1, col2} a {TID, col3, col4}
- Operace **union** v případě horizontálních fragmentů

Operace nad fragmenty

- V případě dotazování fragmentů, optimalizátor z katalogu přečte, na kterém uzlu se daný fragment uložen a nebude přistupovat k ostatním uzlům.
- Aktualizace fragmentů je stejná jako v případě aktualizace pohledů
- Záznam může být např. přemístěn z jednoho uzlu na druhý, pokud po aktualizaci nesplňuje podmínku původního uzlu

Replikace

Replikace dat – fragment je uložen v několika kopiích (replikách) v uzlech systému.

Příklad (z jednoho uzlu se zkopíruje fragment do druhého uzlu a naopak):

```
REPLICATE O_ACCOUNT AS MO_ACCOUNT AT SITE 'Michigan'
REPLICATE M_ACCOUNT AS OM_ACCOUNT AT SITE 'Ostrava'
```

Proč používat replikaci

- **Vyšší výkon** – aplikace může pracovat s lokální kopií dat namísto se vzdáleným uzlem
- **Vyšší dostupnost** – replikovaný objekt je dostupný, dokud je dostupná alespoň jedna kopie
- **Nevýhoda**: v případě aktualizace musí být aktualizovány všechny kopie, což se nazývá **propagace aktualizace**