04 Databáze

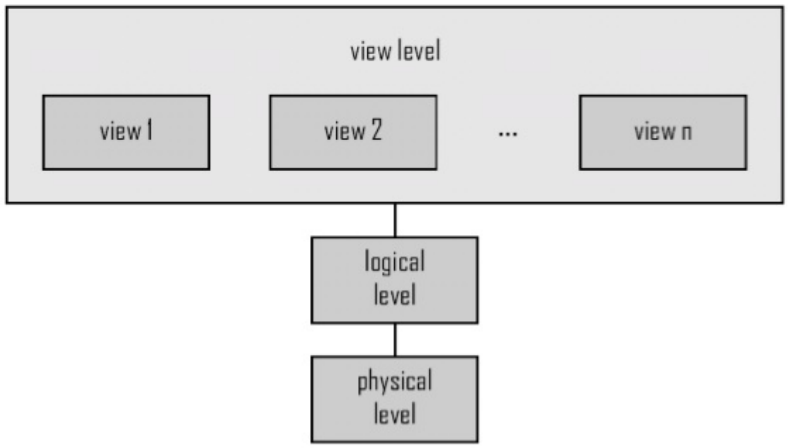
tags: `#sss-základ`

> Databáze. Řízení relačních databázových systémů, terminologie, principy. Reprezentace a ukládání dat. Vyhodnocení dotazů a optimalizace, statistiky, rozdělování tabulek. Indexování a hašování, indexování pro více atributů. Transakční zpracování, zotavení. Zabezpečení, přístupová práva, SQL útoky. (PA152)

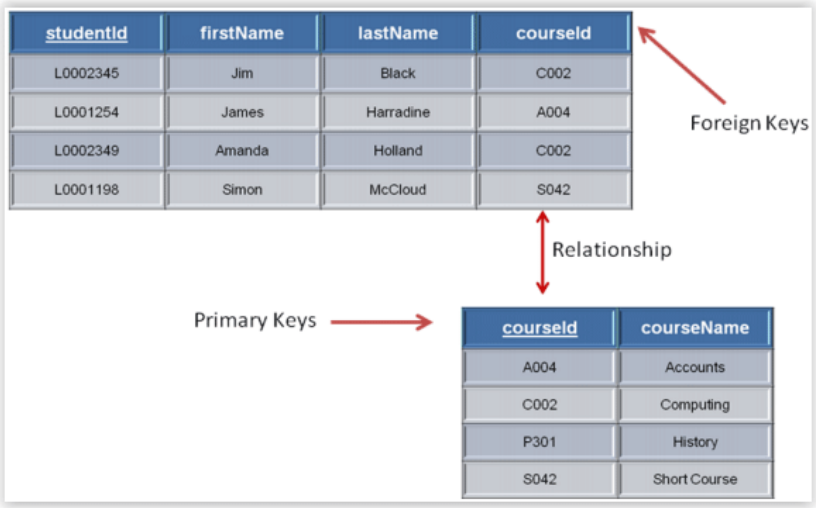
> **Na co se ptají:** > **Batko**: Databáze - základ (tabulky, atributy, záznamy), klíče (kandidátní, primární, superklíč) a relační algebra (kartézský součin, selekce, projekce), jak proběhně vyhodnocení dotazu (syntaktická, sémantická kontrola, převedení do relační algebry). Optimalizace dotazů - jak se vybere dotaz s nejnižší cenou? = na základě statistik. SQL jazyk (co tam patří). Hashování a indexování. > > **Matyska**: Relační databáze > > **Ošlejšek**: Indexování (hustý, řídký index), B-stromy, B+stromy > > **Pitner**: Databáze, jak funguje, optimalizace dotazů, table partitioning a indexing > [color=blue]

Řízení relačních databázových systémů, terminologie, principy

Základní pojmy: * **Databáze** * Databáze (DB) je uspořádaná množina dat, se kterými můžeme dále pracovat. Správa databáze je realizována prostřednictvím Systému pro správu databáze (Database Management System, DBMS). **DB + DBMS tvoří dohromady databázový systém.** * "Programuje" je většina programátorů * Potřebuje je každá firma * Součást většiny aplikací * Obsahuje data, která nás zajímají, která zpracováváme * Jde o kolekci relací, integritních omezení, indexů, triggerů, … * Potřeba rozlišovat schéma databáze vs. instance databáze * Schéma/struktura databáze - odkazuje na organizaci dat, jak je databáze navrhnuta - tj. atributy (sloupce). * **Relační model** * Relační model (RM) pro správu databáz je způsob správy dat pomocí struktury a jazyka, kde jsou všechna data reprezentovaná v podobě n-tic seskupených do relací. Databáze organizovaná z hlediska relačního modelu je relační databáze. * Účelem relačního modelu je poskytnout deklarativní metodu na specifikaci dat a dotazů: uživatelé přímo uvedou, jaké informace databáze obsahuje a jaké informace od ní chtějí, a nechají Database Management System, aby se postaral o popis datových struktur na ukládání dat a procedur na získávání dat (queries answering) * **Struktura - data v relacích (tabulkách)** * **Operace - dotazování, modifikace** * pomocí SQL, relační algebry * **Databázový systém** * **Kolekce nástrojů pro ukládání a zpracování dat.** * Poskytuje datovou abstrakci (viz. obrázek)



Další pojmy: - **Primární klíč** (PK - primary key) - nezávislá entita, která jednoznačně identifikuje instanci relace z databázové tabulky - **Cizí klíč** (FK - foreign key) - klíč, který úlohu jednoznačné identifikace zajišťuje v jiné tabulce, než v kontextu které je na něj nahlíženo -



- **Superklíč** (Super Key) - kombinace všech možných atributů, které mohou unikátně identifikovat řádek v tabulce. - V následující tabulce může být superklíčem např.: 1. {Roll_no} 2. {Registration_no} 3. {Roll_no,Registration_no} 4. {Roll_no, Name} 5. {Name, Registration_no} 6. {Roll_no, Name, Registration_no} - - **Kandidátní klíč** (Candidate key) - Kandidátní klíč je minimální superklíč bez redundance. Nazývá se minimální superklíč, protože ho vybíráme ze seznamu superklíčů tak, aby bylo potřebné minimum atributů k identifikaci řádku v tabulce. Tím, že je vybírán ze seznamu superklíčů, tak všechny kandidátní klíče jsou superklíče. 1. {Roll_no} - tento klíč může být kandidátní, protože nemá žádný redundantní nebo opakující se atribut. 2. {Registration_no} - stejný, jako předchozí klíč. 3. {Roll_no,Registration_no} - Tento klíč nemůže být kandidátním klíčem, protože jeho subset už je obsažen v klíči 1. a 2. - **Arita** (arity) *refers to the number of columns in a table. If a table has five columns, we say it is of arity 5.* - **Relační algebra** - Selekce: σ - Mějme relaci (tabulku) *Person*, operace $\sigma_{Age} \geq 34(Person)$ nám vrátí všechny osoby starší nebo rovno 34 let. Je to to stejné, jako `SELECT * FROM Person WHERE Age >= 34` -

Person

Name	Age	Weight
Harry	34	80
Sally	28	64
George	29	70
Helena	54	54
Peter	34	80

$\sigma_{\text{Age}\geq 34}(\text{Person})$

Name	Age	Weight
Harry	34	80
Helena	54	54
Peter	34	80

$\sigma_{\text{Age}=\text{Weight}}(\text{Person})$

Name	Age	Weight
Helena	54	54

- Projekce: Π - Operace $\Pi_{Age, Weight}(Person)$ nám z relace *Person* vrátí atributy *Age*, *Weight*. Je to to samé, jako `SELECT Age, Weight FROM Person`. -

Person

Name	Age	Weight
Harry	34	180
Sally	28	164
George	28	170
Helena	54	154
Peter	34	180

$\Pi_{\text{Age},\text{Weight}}(\text{Person})$

Age	Weight
34	180
28	164
28	170
54	154

-

Union: ∪ - Spojení dvou relací *r* a *s*. - Pro $r \cup s$ musí platit: 1. *r* a *s* musí mít stejnou aritu (stejný počet sloupců/atributů) 2. Domény atributů musí být kompatibilní (např. 2. sloupec relace *r* musí obsahovat hodnoty stejného typu, jako 2. sloupec relace *s*). -

- Relations *r*, *s*:

A	B
α	1
α	2
β	1

r

A	B
α	2
β	3

s

- r ∪ s:

A	B
α	1
α	2
β	1
β	3

-

Příklad: Najdi všechny kurzy, které se vyučovaly na Podzim 2009, nebo na Jaře 2010, nebo obojí: - $\Pi_{course\_id}(\sigma_{semester} = \text{Podzim} \wedge year = 2009(section)) \cup \Pi_{course\_id}(\sigma_{semester} = \text{Jaro} \wedge year = 2010(section))$ - Set difference: − - Rozdíl dvou relací *r* a *s*. - Pro $r \cup s$ musí platit: 1. *r* a *s* musí mít stejnou aritu 2. Domény atributů musí být kompatibilní -

- Relations *r*, *s*:

A	B
α	1
α	2
β	1

r

A	B
α	2
β	3

s

- $r - s$:

A	B
α	1
β	1

-

Příklad: Najdi všechny kurzy, které se vyučovaly na Podzim 2009, ale ne na Jaře 2010: - $\Pi_{course\_id}(\sigma_{semester} = \text{Podzim} \wedge year = 2009(section)) - \Pi_{course\_id}(\sigma_{semester} = \text{Jaro} \wedge year = 2010(section))$ - Kartézský součin: × - Prakticky spojení dvou tabulek s rozdílnými sloupci -

- Relations *r*, *s*:

A	B
α	1
β	2

r

C	D	E
α	10	a
β	10	a
β	20	b
γ	10	b

s

- $r \times s$:

A	B	C	D	E
α	1	α	10	a
α	1	β	10	a
α	1	β	20	b
α	1	γ	10	b
β	2	α	10	a
β	2	β	10	a
β	2	β	20	b
β	2	γ	10	b

-

Rename: ρ - Dávají nám možnost pojmenovat výsledky relačně-algebraických operací a tím tak i možnost se na ně odkazovat

- Hlavní součásti databázového systému
  - **Storage Manager** - spravuje bloky na disku, spravuje vyrovnávací paměť, ukládání dat
  - **Query Processor** - překládá a optimalizuje dotazy, vyhodnocuje dotazy
  - **Transaction Manager** - zajišťuje atomičnost, konzistenci, izolovanost a trvalost transakcí (ACID)
- Hierarchie pamětí - dolů stoupá kapacita a klesá rychlost - primární paměti jsou nejrychlejší s nejmenší kapacitou
  - **Primární**
    - **vyrovnávací** (cache)
      - procesor
    - **hlavní** (operační paměť)
      - RAM
  - **Sekundární**
    - disk, flash
  - **Terciární**
    - záložní
      - pásky, optické disky

Mooreův zákon = „*počet tranzistorů, které mohou být umístěny na integrovaný obvod, se při zachování stejné ceny zhruba každých 18 až 24 měsíců zdvojnásobí.*“ * stoupá i rychlost procesorů a pamětí * obdobně platí i pro **kapacitu disků** * **Kryder's Law** - 1000x více místa co 15 let * neplatí pro rychlost disků, ale **pro velikost**

Väčšina relačných databáz používa SQL data definition and query language. **Structured Query Language** (SQL) je special-purpose domain-specific jazyk používaný v programování a je navrhnutý na správu dát uchovávaných v Relational Database Management systémech (RDBMS).

Pôvodne založený na relačnej algebre a na 'tuple relational calculus', sa SQL skladá z **Data**

Definition Language (DDL), **Data Manipulation Language** (DML), **Data Control Language** (DCL) a **Transaction Control Language** (TCL). Scope SQL zahŕňa insert, query, update, delete; vytváranie a modifikáciu schém a riadenie prístupu k dátam. Aj keď sa SQL často označuje ako deklaratívny jazyk (programovací jazyk štvrtej generácie), obsahuje aj procedurálne prvky.

DDL - Definujeme data:

```
DROP TABLE employees;
CREATE TABLE employees (
    id            INTEGER       PRIMARY KEY,
    first_name    VARCHAR(50)   not null,
    last_name     VARCHAR(75)   not null,
    fname         VARCHAR(50)   not null,
    dateofbirth   DATE          not null
);
```

DML - Manipulujeme s daty: SELECT … FROM … WHERE … INSERT INTO … VALUES … UPDATE … SET … WHERE … DELETE … FROM … WHERE …

DCL - Řídíme přístup k datům: GRANT SELECT, UPDATE ON example TO some_user, another_user; REVOKE SELECT, UPDATE ON example FROM some_user, another_user;

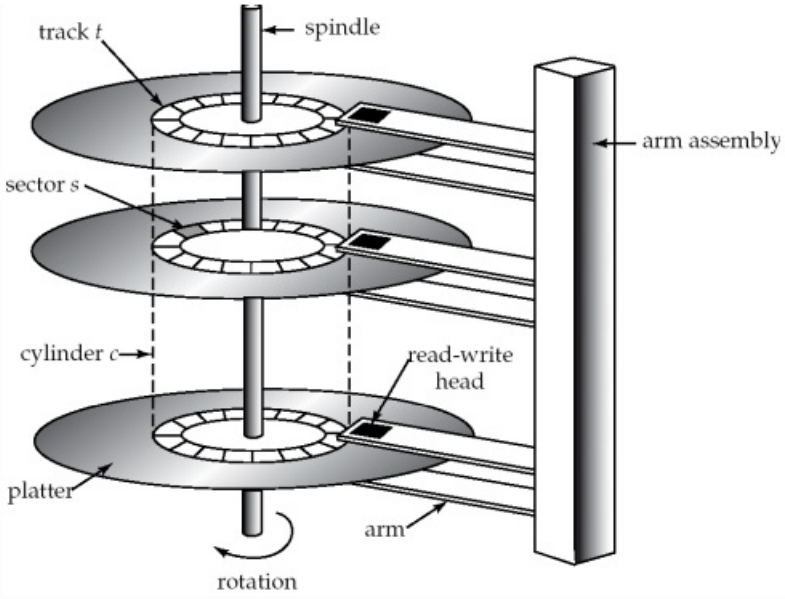
TCL: COMMIT, SAVEPOINT, ROLLBACK

RDBMS examples: * Oracle * MySQL * Microsoft SQL Server * PostgreSQL * IBM DB2 * Microsoft Access

Reprezentace a ukládání dat

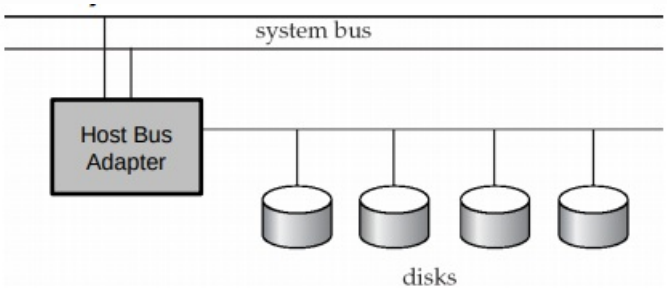
Paměťová úložiště

- **Cache**
  - nejrychlejší a nejdražší, závislá na napájení
- **RAM**
  - rychlé = 10-100 ns (nanosekund)
  - příliš malé nebo drahé pro uložení celé databáze
  - závislé na napájení (volatilní = po odpojení napájení se data smažou)
- **Flash**
  - rychlé čtení
  - nezávislé na napájení
  - pomalý zápis - nejdříve smazat, pak zápis (navíc se zapisuje celá oblast neboli banka)
  - omezený počet zapisovacích cyklů
- **Rotační disk**
  - velká kapacita
  - nezávislá na napájení
  - čtení a zápis téměř stejně rychlé
  - vhodné místo pro většinu databází



Rotační disk vlastnosti a pojmy

- **Přístupová doba** (access time)
  - Čas **mezi požadavkem** na čtení/zápis a **počátkem přenosu** dat
  - Obsahuje několik složek
    - **vystavení hlaviček** (seek time) - 4-10ms (milisekund)
      - přesun hlaviček na správnou stopu disku
      - průměrná doba vystavení hlaviček = ½ nejhorší doby pro vystavení hlaviček
    - **rotační zpoždění** (latency) - 4-11ms (závisí na otáčkách disku, ty jsou 5400-15000 otáček za minutu)
      - čas pro otočení disku na správný sektor
      - průměrná latence = ½ nejhoršího případu latence
    - **přenosová rychlost**
      - rychlost čtení/zápisu dat z/na disk
      - 50-200 MB/s, závisí na vzdálenosti od středu – u středu je nižší. Na okrajích stopy obsahují více sektorů a jedna rotace přečte více bloků dat
- Data jsou blokována = rozdělena na bloky
  - atomickou jednotkou čtení je **sektor/diskový blok**
- Rychlost řadiče
  - Řeší se v případě více disků zapojených na jeden řadič
  - SATA 3.0 (6Gb/s, 600 MB/s)
  - SAS-2 (6Gb/s, 600 MB/s)



- Vlastnosti
  - náhodné čtení je pomalé
    - access time může být až 20 ms
  - sekvenční čtení je rychlé
- Optimalizace přístupu v HW
  - Cache

        - Buffer pro zápis, zálohovány baterií nebo flash
    - Algoritmy pro minimalizaci pohybu hlavičky
        - Algoritmus “výtah” (Když si představíme, že po disku jezdí výtah od středu směrem k okraji a zpátky, tak při velkém počtu požadavku jsou seřazeny tak, jak jede výtah, jede k okraji a sbírá všechny požadavky jedním směrem, i když přišly v různou dobu a potom jede dolů a dělá to stejné. Jde o to, aby čtecí hlavička byla co nejvíce zaměstnaná čtením a ne přesouváním na pozice podle pořadí.)
        - Fungují pouze při velkém počtu požadavků současně.

Diskové operace v DBMS

- Přístup po blocích (atomická jednotka)
    - Blok je skupina sousedních sektorů. Typicky 4-16KB
- Čtení bloku
- Zápis bloku = zápis a ověření (Ověření = otočení disku + čtení)
- Modifikace bloku
    - čtení -> změna v paměti -> zápis a ověření

Algoritmy v DBMS

- Pracují s bloky (velikost bloku DB, FS, zařízení)
- Minimalizují počet náhodně čtených bloků
- Náklady na čtení a zápis jsou stejné (časté zjednodušení)

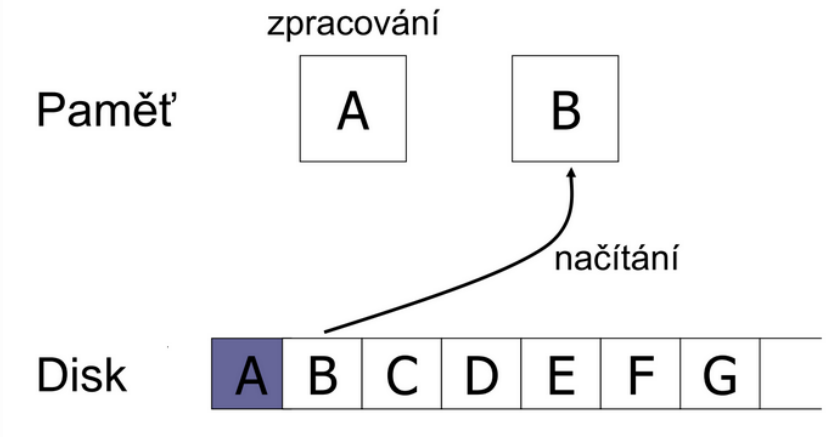
SSD flash memory

- Diskové úložiště na „flash“ pamětech
- Žádné pohyblivé části
- Odolnější proti poškození
- Tiché, nižší přístupová doba a zpoždění
- 4x dražší než HDD (za GB)
- Používají se NAND čipy (ukládají se 1 až 4 bitové informace)
- Organizace do bloků
    - Stránky 4KiB organizované do bloků (64 / 128 stránek)
    - Mnohem rychlejší čtení než zápis
    - Zápis: pouze do „nového bloku“
        1. write a new copy of the changed data to a fresh block
        2. remap the file pointers
        3. then erase the old block later when it has time
    - Omezení zápisu – 1k-100k přepsání
- Samotný NAND čip je relativně pomalý
- RAID 0 pro zvýšení výkonu

Optimalizace přístupů na disk - tři typy

1. Omezení náhodných přístupů (např. algoritmus Výtah)

Defragmentací - uspořádání bloků do pořadí jejich zpracování. **Plánování přístupů** - obdobně jako HW řešení zmíněné výše. Jde o přeuspořádání požadavků na disk, aby se hlavička pohybovala jedním směrem jako výtah. **Využití bufferů** - využívá se toho, že zpracování přečteného bloku z disku zabere nějaký čas, po který může disk už číst data další. Mezi zmíněné metody patří Single Buffer a Double Buffering * **Single Buffer** - moc se nepoužívá, neboť je Double Buffering výrazně lepší řešení. Nejdříve se data načtou do bufferu a pak se zpracují. Single buffer time = n(R + P) * **Double Buffering** - v paměti existují 2 buffery, které se používají střídavě. Do jednoho se načítají data a z druhého se zpracovávají.



Vzorec pro výpočet času zpracování = **R + nP**, za předpokladu, že **P >= R**. R = čas k přečtení 1 bloku. P = čas ke zpracování 1 bloku. n = počet bloků.

2. Velikost bloku

Pokud jsou data uložena ve velkých blocích snižují se náklady na I/O (input/output operace). Velký blok, ale zároveň může znamenat čtení více "nepotřebných" dat a čtení jen části dat může trvat déle. Hledá se kompromis mezi těmito dvěma problémy. Aktuální trend - cena pamětí klesá, data rostou a bloky se zvětšují. Typická velikost bloků v současné době jsou 4 KB.

3. Disková pole

Jde o spojení více fyzických disků do jednoho logického seskupení. * Zvětšení kapacity * Paralelní čtení/zápis * Průměrná doba vystavení hlaviček typicky zachována

Existují 2 metody pro dosažení zmíněných cílů: * **Zrcadlení dat (mirroring)** - Zápis stejných dat je prováděn na více disků současně. Čtení lze provádět z libovolného disku. * **Zvyšuje spolehlivost** (při výpadku některého disku jsou data stále konzistentní) a **rychlost čtení** (lze číst související bloky z různých disků zároveň).

- **Rozdělování dat (data striping)** - Umožňuje zvýšení přenosové rychlosti rozdělením dat na více disků => **rychlejší čtení i zápis**. Snížení spolehlivosti. Existují 2 metody rozdělování dat:
  - **Bit-level striping** = na disky jsou rozdělovány jednotlivé bity dat. Málo používané, neboť přístupová doba je horší než u jednoho disku.
  - **Block-level striping** = postupné rozdělování bloků dat mezi jednotlivé disky. **Čtení i zápis lze paralelizovat.** Čtení různých bloků lze paralelizovat. Velké čtení může využít všechny disky.

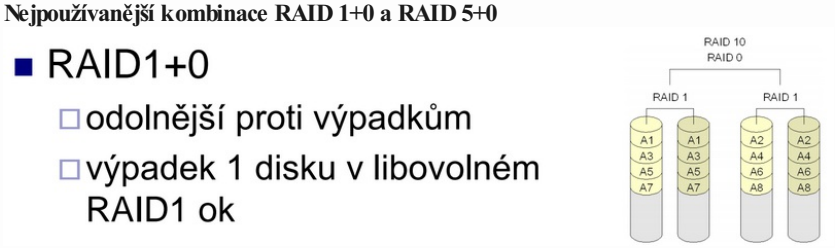
Mezi nejznámější disková pole patří **RAID** (Redundant Arrays of Independent Disks). Existuje 7 základních typů, ze kterých se dále skládají kombinace: * **RAID 0 - block striping**. Neredundantní. Velmi zvýšený výkon, spolehlivost spíše snížena, nesnížená kapacita. * **RAID 1 - zrcadlení**. Kapacita 1/n, rychlé čtení, zápis jako na 1 disk. * RAID 2 (bit-striping), RAID 3 a RAID 4 (block-striping) používají paritní disky, jsou neefektivní a jsou nahrazeny RAID 5, který je často používaným řešením. **RAID 5 data i paritu rovnoměrně rozděluje mezi všechny disky.**

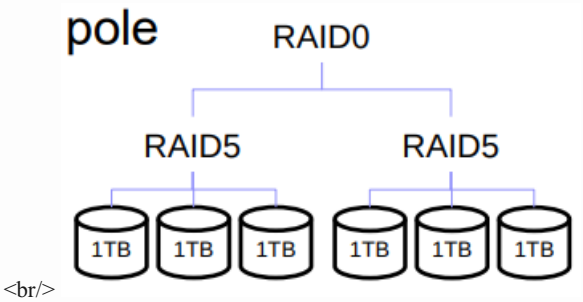
- Příklad (5 disků)
  - Parita pro blok *i* je na: $\lfloor i/4 \rfloor \bmod 5$

P0	0	1	2	3
4	P1	5	6	7
8	9	P2	10	11
12	13	14	P3	15
16	17	18	19	P4

- **RAID 5** má rychlejší čtení i zápis díky paralelizaci a je schopný obnovit data při výpadku jednoho disku.
- https://www.sqlpassion.at/archive/2017/05/08/how-to-calculate-raid-5-parity-information/
- **RAID 6** - podobné RAID 5, ale má 2 paritní disky => obnoví výpadek dvou disků.

Nejpoužívanější kombinace RAID 1+0 a RAID 5+0





Ukládání dat

Typy datových elementů * **Celá čísla** - podle rozsahu na 2, 4, 8 bajtů. Obvykle se používá přímý nebo inverzní kód. Př. číslo 35 na 16 bitech: 0000 0000 0010 0011 * **Reálná čísla** - 2 typy uložení: * Plovoucí čárka - paměť pro uložení čísla je rozdělena na mantisu a **exponent** * Pevná čárka - kódování každých 9 cifer do 4 bajtů a následné uložení jako řetězec. * **Znaky** - mezi nejznámější patří ASCII kódování = 1 bajt, či UTF-8. UTF-8 představuje kódování variabilní délky. Různé znaky mohou být uloženy na různém počtu bytů. UTF-8 znak může zabírat 1 až 4 bajty. * **Pravdivostní hodnota** (boolean) - obvykle jako celé číslo. TRUE = 1111 1111, FALSE = 0000 0000. Méně jak **1 bajt** nemá význam kvůli práci s daty. * **Bitové pole** - Délka pole + samotné bity. Celé zaokrouhleno na celé bajty. * **Datum** - typicky uloženo jako **počet dní** od definovaného počátku (často se používá 1.1.1970) nebo jako řetězec ve formátu YYYYMMDD. * **Čas** - uložen jako **počet sekund** od půlnoci. Často obsahuje časovou zónu a je tak uložen v UTC. * **Výčtový typ** - očíslování hodnot. * **Řetězec**: * Pevná délka - omezení velikosti. Kratší řetězce doplněny mezerami, delší řetězce jsou oříznuty. * Proměnlivá délka - buď se před samotný řetězec uloží jeho délka, nebo je řetězec ukončen nulou (nelze ji pak použít v textu).

Každý datový element má svůj typ, který určuje interpretaci bitů, velikost, speciální hodnoty. Související **datové elementy jsou uloženy v seznamu, který se označuje jako záznam**. Příklad záznamu:

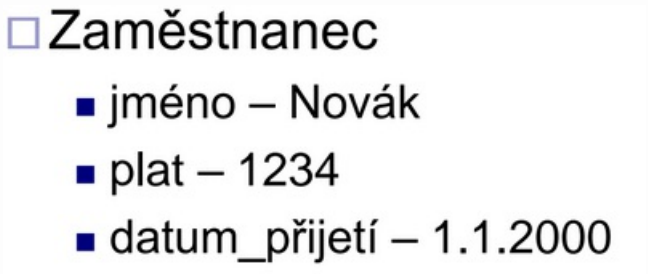
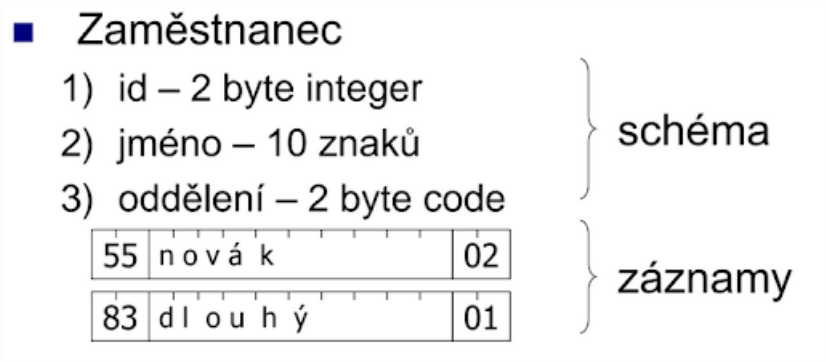
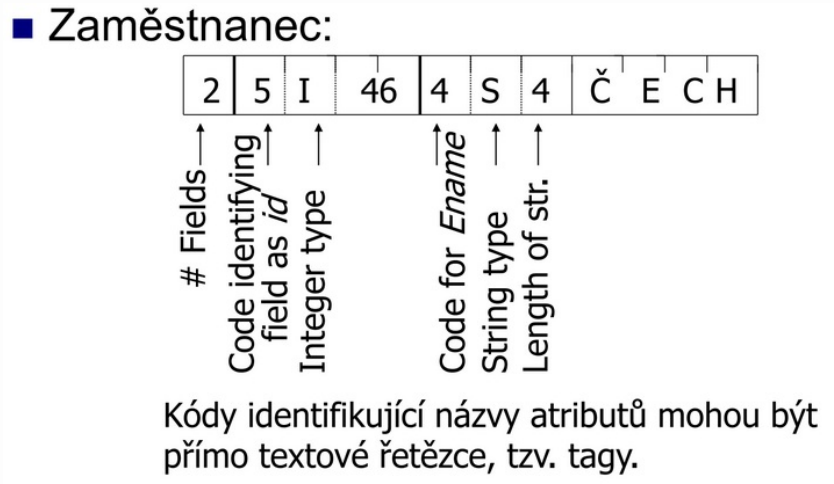


Schéma záznamu pak popisuje strukturu záznamů, počet a pořadí jeho atributů, typ/název každého atributu.

Záznamy lze dělit podle typu schématu a délky na pevné a proměnlivé: * **Pevné schéma i délka** - schéma je společné pro všechny záznamy a je uloženo mimo (v Data Dictionary) záznamy. Každý záznam má stejnou délku, př.:



- **Proměnlivé schéma i délka** - Každý záznam má vlastní schéma. Složitější implementace, ale možnost uložení velkých dat, př.:



- **Mezi-varianta mezi pevným a variabilním schématem spočívá v přidání schéma záznamu do hlavičky záznamu.** Hlavička záznamu obsahuje informace o záznamu (fixní délka), které nesouvisí s hodnotami atributů:
  - Schéma záznamu (odkaz)
  - Délka záznamu
  - Čas vytvoření/změny/čtení
  - OID - "ID" záznamu
  - Pole pro NULL hodnoty

Záznamy se dále skládají/ukládají do bloků. Popsáno níže.

Databázové relace/záznamy lze ukládat buď po **sloupcích** (hodnoty stejného atributu pohromadě) nebo po **řádcích**. Sloupcové uložení je kompaktnější a má lepší efektivitu čtení např. při data miningu. Řádkové ukládání má výhodu v rychlejších aktualizacích a vkládání a také při přístupu k celým záznamům. V praxi se používá nejčastěji sloupcové ukládání.

Uložení záznamů do bloků

Bloky mají pevnou velikost a obsahují **záznamy**, které mohou být **pevné** nebo **proměnlivé** délky. Při ukládání záznamů se řeší tyto problémy:

1. Oddělování záznamů

- **Záznamy s pevnou délkou** - není potřeba oddělovač. Stačí si pamatovat počet a

ukazatel na první záznam.

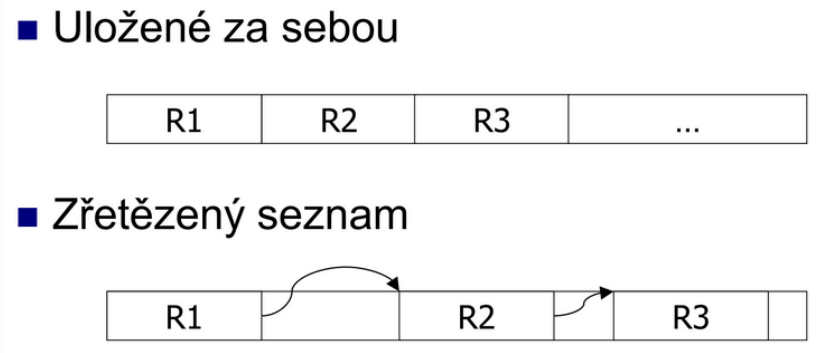
- **Záznamy s proměnnou délkou** - používá se oddělovač a vzniká potřeba ukládat délky záznamů, nebo jejich počátky (a to v rámci záznamu nebo v hlavičce bloku).
- **Blok** typicky **obsahuje hlavičku** a za ní **záznamy**. Hlavička pak obsahuje:
  - Odkaz na další bloky
  - Typ bloku
  - Příslušnost relaci
  - Časová razítka (vytvoření/modifikace/čtení)
  - (Adresář offsetů na záznamy)

2. Rozdělování / nerozdělování záznamů

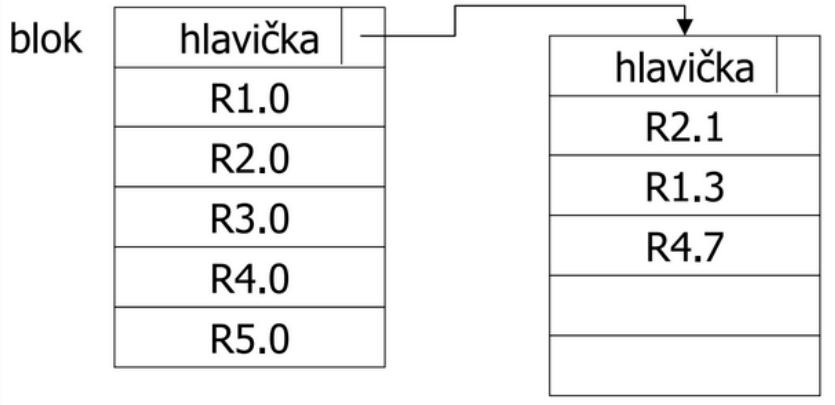
- **Nerozdělování** - každý **záznam** je pak **součástí jednoho bloku**, nemůže být přes více bloků. Toto řešení je jednoduché, ale plýtvá místem, neboť málo kdy vyjdou záznamy tak, aby vyplnili blok celý. V nejhorším možném případě (př: blok = 4096 bajtů, záznam = 2049 bajtů) může být skoro polovina paměti prázdné místo. Nelze použít při podpoře záznamů větších než je blok.
- **Rozdělování** - záznamy mohou přetékat mezi bloky => potřeba udržovat pořadí bloků. Záznam je rozdělen na **fragmenty** - je potřeba přidat **bitový příznak**, zda byl záznam fragmentován a také **ukazatel na další/předchozí fragment**.

3. Uspořádání záznamů

Záznamy jsou v souboru/bloku uspořádány podle hodnoty nějakého klíče z důvodu efektivnějšího čtení v daném pořadí, např. pro merge-join, order by a další. Existují 2 varianty uspořádání:



Pří uložení za sebou je potřeba myslet na situace při aktualizaci záznamů vedoucích ke změně jejich délky. Buď je potřeba záznamy **reorganizovat** (pro každou změnu velmi nákladné) a nebo využít **přetokových oblastí/bloků**, kdy v případě zvětšení záznamu se jeho přebývající část zapíše do jiného bloku a nastaví se na něj odkaz.



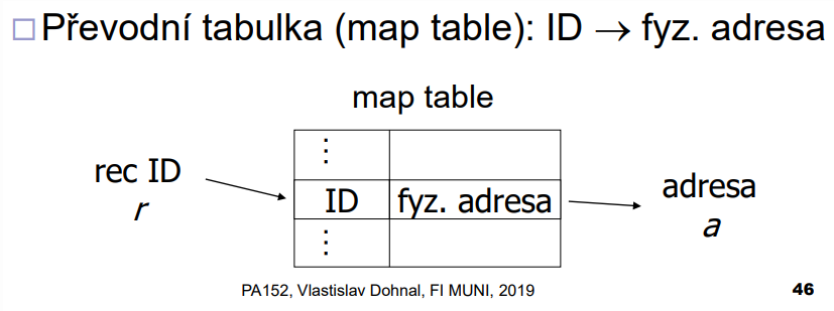
4. Odkazy na záznamy

Pří různých situacích (rozdělování záznamů, odkazování bloků/záznamů, zřetězení bloků,

…) je potřeba odkaz na jiný záznam. Záznam může být uložen: * **V paměti** - přímá adresace * 4/8 bajtový ukazatel ve virtuální paměti procesu * **V úložišti** - sekvence bajtů popisující umístění * **přímá vs. nepřímá adresace**

Přímá adresace * Jde o **fyzickou adresu záznamu** = adresa v úložišti = ID disku + stopa + povrch + blok + offset v bloku. * Nepraktické kvůli realokaci bloku či záznamu.

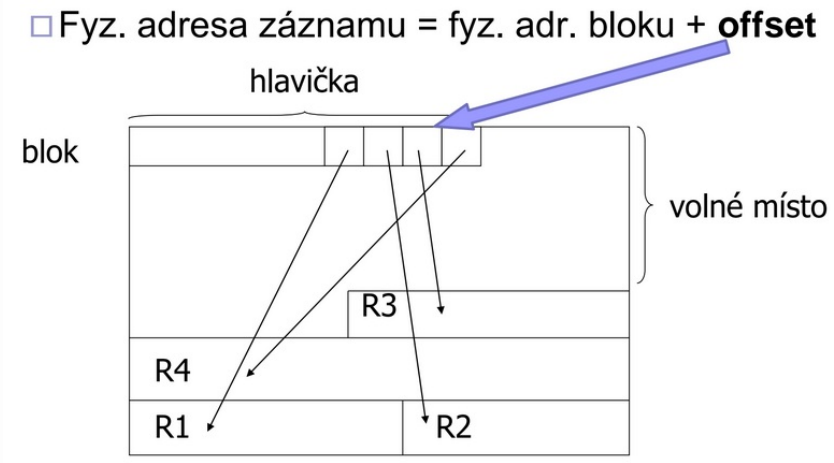
Nepřímá adresace * Záznam (i blok) je identifikován svým ID = **logickou adresou záznamu** * Musí existovat převodní tabulka z ID na fyzickou adresu = Map Table. * Nevýhody: zvýšené náklady kvůli průchodu Map Table a vytváření a ukládání Map Table. * Výhody: Velká flexibilita - jednoduché mazání a vkládání nových záznamů. Optimalizace uložení bloků. *



Kombinace * Fyzická adresa záznamu = **fyzická adresa bloku + offset záznamu**. (Offset představuje pořadí záznamu v bloku). * Výhody * lze přesouvat záznamy v bloku, beze změny fyz. adresy * lze přesunout záznam do jiného bloku - v původním místě se jen udělá odkaz na nový blok+offset * lze zrušit map table * Nevýhody * nízká flexibilita při přesouvání bloků (defragmentace)

Používaná varianta * **Adresa záznamu = ID souboru + číslo bloku + offset záznamu** * Uložení bloku určuje systém souborů

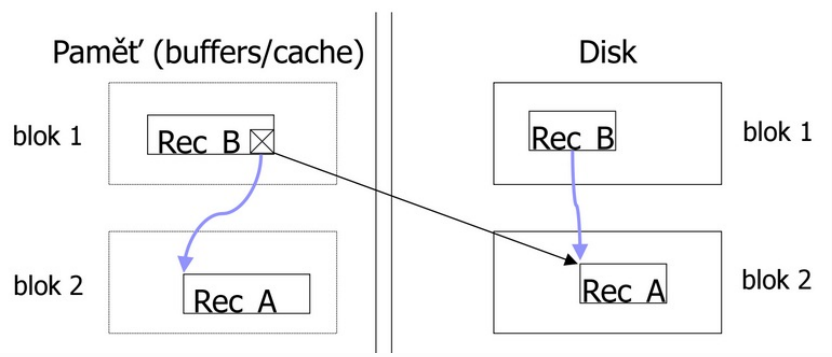
Nepřímý odkaz v bloku



Hlavička bloku obsahuje: * File ID * Typ bloku (Př: záznam typu X, přetoková oblast, TOAST záznamy, …) * ID bloku * Adresář záznamů s odkazy * Ukazatel na volné místo (začátek a konec) * Ukazatel na další blok (dobré např. pro indexy) * Čas modifikace

Modifikace záznamů v bloku může být problematická * Při **mazání záznamů** je potřeba spravovat vzniklé volné místo. Navíc je potřeba zajistit, aby odkazy vedoucí na smazané záznamy byly neplatné a neodkazovaly tak na nová místa. K označení smazaného záznamu se používá **tombstone** (náhrobek). * **Vkládání** nových **záznamů** většinou v pohodě. U neuspořádaných souborů se záznam prostě mrskne do volného místa nebo na konec. V uspořádaných souborech musí dojít k reorganizaci či uložení do přetokových oblastí. * Při **změně záznamů**, pokud se liší velikost, vznikají stejné problémy jako při vkládání/mazání

Ještě zmínka o **pointer swizzling**. Jde o prohazování odkazů ve vyrovnávací paměti.



Poté co jsou bloky B a A načteny do paměti, je potřeba aby B v paměti odkazovalo na A v paměti, tedy došlo k prohození odkazů. To se dělá: * Automaticky - hned * Na žádost - při prvním použití * Nikdy -> vždy se používá překladová tabulka.

Implementace probíhá tak, že DB adresa je změněna na paměťovou. Za tímto účelem se buduje překladová tabulka.

Vyhodnocení dotazů a optimalizace, statistiky, rozdělování tabulek.

Vyhodnocení dotazů

1. **Dotaz**
2. **Syntaktická a sémantická kontrola** (sestavení stromu)
3. **Logický plán** (+ úpravy)
4. **Fyzický plán**
5. **Vyhodnocení**

1. Dotaz

Běžné zadání dotazu do SQL - např. SELECT * FROM courses WHERE …

2. Syntaktická a sémantická kontrola

Parser přetvoří dotaz do interní reprezentace ve formě stromu a provede normalizaci dotazu (viz. CNF - Konjunktivní normální forma). V téhle fázi se kontroluje cache, pokud záznam existuje, přeskočí se na vyhodnocení dotazu.

Syntax - skladba dotazu - správné uspořádání příkazů **Sémantika** - význam dotazu - kontroluje se např. jestli odkazované sloupce existují

3. Logický plán

Vezme strom dotazu, převede ho do relační algebry, vyhodnotí používané datové typy a operátory a pokusí se optimalizovat strom pro větší efektivitu logických operátorů (AND, OR, IN, JOIN…). Jedná se o vyšší úroveň zpracování (z pohledu abstrakce). IN operátor se nahrazuje součinem (přirozeným).

4. Fyzický plán

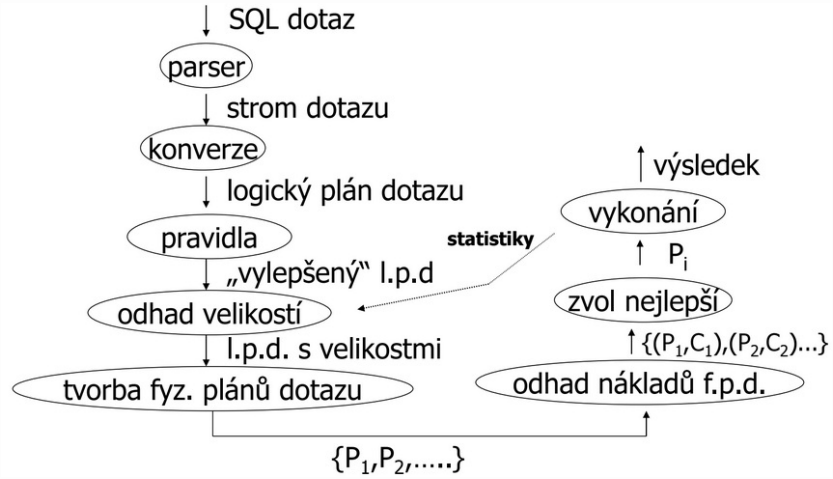
Vezme logický plán a nahradí jednotlivá data a prvky konkrétní implementací databázového systému na nižší úrovni. Při generování fyzického plánu se bere v potaz cena jednotlivých operací (čtení z disku, ukládání do RAM), počítá se množství postihnutých záznamů, čas provedení apod. Na základě tohoto vyhodnocení vznikne tzv. “cena” dotazu a pokud se zvažovalo více variant, vybere se ta s nejnižší cenou. Správný odhad ceny dotazu je závislý

na správných **statistikách** ohledně tabulek a jejich dat.

5. Vyhodnocení

Připraví se a vyhodnotí zdrojový kód dotazu, který vrací konkrétní set hodnot.

Schéma vyhodnocení a optimalizace dotazů



Optimalizace dotazů

Lokální změna - přepsání dotazu - ovlivní pouze daný dotaz * Použití indexu - nemusí být použit kvůli špatně specifikovanému SELECTu * Rušení nadbytečných DISTINCT - u primárních klíčů, pokud SELECT vrací mezi atributy PK, pokud ve WHERE porovnáváme PK a FK. * Přepsání korelovaných dotazů - nahradit pomocí WHERE nebo atributů z nevnořeného SELECTu v podmínce ve vnořeném SELECTu * Dočasné tabulky * Používání HAVING (ne u HAVING x = 100, lze například u HAVING x > 100), zkrácení dotazů, které filtrují podle agregační funkce, nelze použít ve WHERE, nepoužívat HAVING, když lze použít WHERE * Používání pohledů (VIEW) * Uložené pohledy (MATERIALIZED VIEW)

Globální změna - ovlivní všechny dotazy * Přidání nových indexů (viz. sekce Indexing, hashing…) * Změna schéma (vertical partitioning) * Rozdělení relací (horizontal partitioning)

Statistiky

Správný odhad ceny dotazu je závislý na správných statistikách ohledně tabulek a jejich dat. Databáze si udržuje statistiky ohledně počtu řádků, velikosti dat na disku atd. Některé statistiky se počítají v průběhu vyhodnocení dotazu, jiné se počítají např. při vytváření indexů, nebo zavoláním příkazu pro update. Mezi typické statistiky patří:

- T(R) – počet záznamů
- S(R) – velikost záznamu v bajtech
- S(R,A) – velikost atributu (hodnoty) v bajtech
- B(R) – počet obsazených bloků
- V(R, A) – počet unikátních hodnot atributu A
- Pro správné odhady jsou aktuální statistiky nutné!
- Kartézský součin W = R1 x R2 - T(W) = T(R1) * T(R2), S(W) = S(R1) + S(R2)
- Selekce W = omegaA=val(R) S(W) = S(R), T(W) = T(R) / V(R,A)
- Selekce W = omegaA!=val(R) T(W) = T(R) - (T(R) / V(R,A))
- Přirozené spojení T(W) = (T(R1) * T(R2)) / max {V(R1,A), V(R2,A)}

Dělení tabulek

+ Normální formy

Normální formy jsou pravidla, která by data v relaci měla splňovat. Čím vyšší norma, tím lepší a jednodušší by práce s daty, jejich vybíráním a aktualizacemi měla být (správný návrh databáze by měl být alespoň v **3NF**). **Požaduje se bezztrátovost spojení, žádné redundance, uchování závislostí.**

První normální forma (1NF)

Relační schéma je v 1NF pokud obsahuje pouze **atomické** hodnoty (=dále nedělitelné) - Původní: -

Osoba

Jméno	Přijmení	Adresa	Telefony
Jan	Novák	Havlíčkova 2 Praha 3	125789654;601258987;789456123
Petr	Kovář	Svatoplukova 15 Brno	369852147;357951456;963852741
Pavel	Pavel	Papalášova 25 Kocourkov	546789123;123456789;987456123

- Po převodu do 1NF - telefony jsou rozdělené a adresa je rozdělená na ulici, číslo popisné a okres/město: -

Osoba

ID	Jméno	Příjmení	ulica_id
1	Jan	Novák	1
2	Petr	Kovář	2
3	Pavel	Pavel	3

Adresa

ulica_id	Ulica	Popisné č.	Okres
1	Havlíčkova	2	Praha 3
2	Svatoplukova	15	Brno
3	Papalášova	25	Kocourkov

Telefon

ID_osoby	Cislo
1	125789654
1	601258987
1	789456123
2	369852147
2	357951456
2	963852741
3	546789123
3	123456789
3	987456123

Druhá normální forma (2NF)

Relační schéma je ve 2NF, pokud je v 1NF a každý neklíčový atribut je plně závislý na primárním klíči, a to na celém klíči a nejen na nějaké jeho podmožině. Tzn. řešíme "jen" v případě, máme-li vícehodnotový primární klíč. Pro 1 primární klíč máme splněno triviálně. Nesplnění 2NF doprovází redundance (opakování) - Původní: -

Sklad

Název	Výrobce	Telefon Výrobce	Cena	Množství
Mléčná čokoláda	Milka	+420123456789	30Kč	2500
Oříšková čokoláda	Milka	+420123456789	30Kč	2800
Tyčinka milkyway	Milka	+420123456789	10Kč	7000
Mléčná čokoláda	Orion	+420987654321	25Kč	5800
Oříšková horalka	Horalka	+420897654321	7Kč	4560

- Po převodu do 2NF: - Klíčem relace je *Název výrobce* a *Výrobce*. *Telefon Výrobce* je závislý pouze na *Výrobci* (čili na podmnožině celého klíče). Pokud by se vymazaly výrobky jednoho výrobce, ztratilo by se na něj i číslo, což je nežádoucí. -

Výrobek

Název	Výrobce_ID	Cena	Množství
Mléčná čokoláda	1	30Kč	2500
Oříšková čokoláda	1	30Kč	2800
Tyčinka milkyway	1	10Kč	7000
Mléčná čokoláda	2	25Kč	5800
Oříšková horalka	3	7Kč	4560

Výrobce

Vyrobce_ID	Vyrobce	Telefon
1	Milka	+420123456789
2	Orion	+420987654321
3	Horalka	+420897654321

Třetí normální forma (3NF)

Relační schéma je v 3NF, pokud je ve 2NF a žádný z jeho atributů není tranzitivně závislý na klíči (všechny neklíčové atributy jsou navzájem nezávislé). - Původní: -

Zaměstnanec

r.č	Jméno	Příjmení	Město	PSČ	Funkce	Plat
1	Jack	Smith	Jihlava	58601	CEO	150000
2	Franta	Vomáčka	Praha10	10000	Senior Software Architect	80000
3	Pepa	František	Plzeň	12345	Senior Software Architect	80000
4	Pavel	Novák	Kocourkov	99999	Junior Developer	30000
5	Petr	Koukal	Praha10	10000	Database Designer	75000
6	Honza	Novák	Plzeň	12345	Junior Developer	30000

-
Klíčem relace je r.č.(rodné číslo). Všechny atributy závisí na klíči, ale i další závisloti: *PSČ* na *Městu*; *Plat* na *Funkce*. Máme tedy tranzitivní závislost. Řešením je rozpad na více relací. - Po převodu do 3NF: -

Zaměstnanec

r.č	Jméno	Příjmení	PSČ	Funkce_ID
1	Jack	Smith	58601	1
2	Franta	Vomáčka	10000	2
3	Pepa	František	12345	2
4	Pavel	Novák	99999	4
5	Petr	Koukal	10000	3
6	Honza	Novák	12345	4

Funkce

Funkce_ID	Funkce	Plat
1	CEO	150000
2	Senior Software Architect	80000
3	Database Designer	75000
4	Junior Developer	30000

Město

PSČ	Město
58601	Jihlava
10000	Praha10
99999	Kocourkov
12345	Plzeň

Boyce-Coddova normální forma (BCNF)

Variace 3NF. Relační schéma je v BCNF, pokud pro každou netriviální závislost $X \rightarrow Y$ platí, že X je nadmnožinou nějakého klíčes schémetu R (tzn. mezi kandidátními klíči nesmí být žádná funkční závislost; všechna data (i klíčová) jsou závislá jen na klíči a ne mezi sebou) Aby byla BCNF porušena (stává se ve specifických případech): - relace musí mít více kandidátních klíčů (minimálně 2) - minimálně 2 kandidátní klíče musí být složené z více atributů - některé složené kandidátní klíče musí mít společný atribut

Mámě dvě netriviální funkční závislosti: - {Město, Ulice} → PSČ a PSČ → Město

Protože neplatí Ulice → PSČ ani Město → PSČ, tvoří dvojice {Město,Ulice} klíč schématu. Klíčem je ale i {Ulice, PSČ}, platí totiž PSČ → Město, nikoliv však PSČ → Ulice. Tudíž je i {PSČ, Ulice} kandidátním klíčem schématu. Schéma má všechny atributy

atomické a nemá žádný neklíčový atribut a tudíž je v 3.NF, ale není v BCNF. Řešením je rozpad na dvě tabulky. Vzhledem k tomu, že neplatí PSČ → Ulice, musíme spojit PSČ a Ulice. Výsledkem tudíž budou relace Město(PSČ, město) a Adresa(PSČ, Ulice). -

Adresa

Město	Ulice	PSČ
Praha 10	Černokostelecká	100 00
Jihlava	Žižkova	58601
Praha 10	Vrátkovská	100 00
Brno	Dvořákova	589 74
Praha 6	Chaloupeckého	160 00

Město

PSČ	Město
100 00	Praha 10
160 00	Praha 6
586 01	Jihlava
Brno	589 74

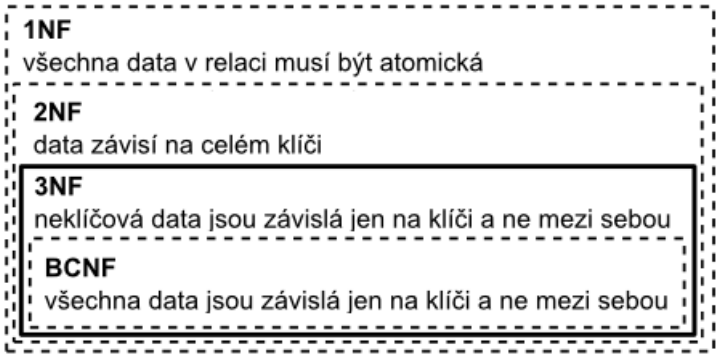
Adresa

Ulice	PSČ
Černokostelecká	100 00
Vrátkovská	100 00
Dvořákova	586 01
Chaloupeckého	160 00
Dvořákova	589 74

Vztahy mezi normálními formami

Třída schéma v BCNF je vlastní podtřídou tříd schémat 3NF, ta je podtřídou třídy relací v 2NF, …: - $BCNF \subset 3NF \subset 2NF \subset 1NF$.

Existuje ale i 4NF a 5NF.



Vertikální dělení (partitioning)

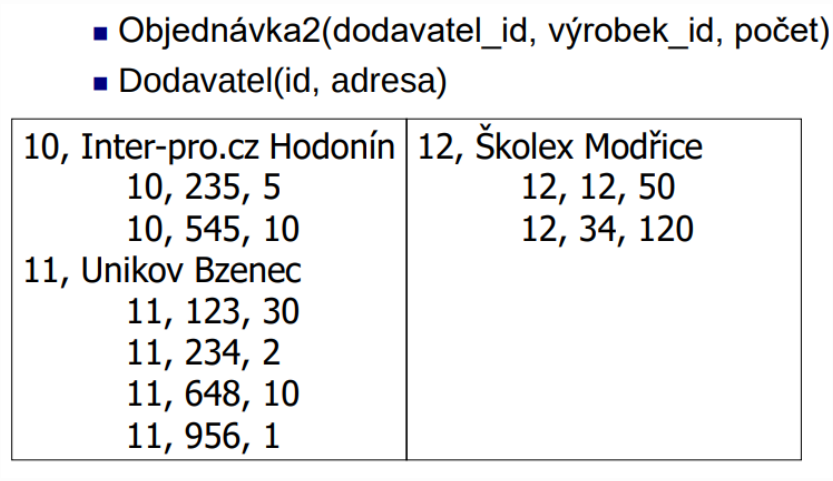
Vertikální partitioning se rozděluje podle sloupců tabulky, tj. sloupce tabulky mohou být uloženy na jiných discích. Co je výhodnější? Zákazník(id, adresa, kredit) nebo ZákazníkAdresa(id, adresa), ZákazníkKredit(id, kredit)? Výpisy z účtu jsou posílány jednou měsíčně. Kredit je měněn po každém telefonním hovoru. Pak je výhodnější druhé schéma. Dvě relace jsou vhodnější, pokud jsou atributy: * přistupovány samostatně (nebo některé řádově častěji) * velké * některé častěji aktualizované

Vertikální spojování (anti-partitioning)

- Začínáme s normalizovaným schématem
- Přidáváme atributy k jedné z relací
- Příklad Akciový trh
  - Historie cen akcií za posledních 3 000 dní
  - Makléř se rozhoduje hlavně podle posledních 10 dnů
- Schéma: AkcieDetail(akcie_id, datum_vydání, firma), AkcieCena(akcie_id, datum, cena)
- Dotazy na 10ti denní historii jsou náročné
  - I když je index na akcie_id, datum
  - Navíc pro další informace je třeba spojení s AkcieDetail
- Proveďme replikaci dat:
  - AkcieDetail(akcie_id, datum_vydání, firma, cena_dnes, cena_včera, …, cena_před_10_dny), AkcieCena(akcie_id, datum, cena)
- Dotazování na 10ti denní historii je 1x prohledání indexu, není třeba spojení
- Nevýhoda je replikace dat - nepříliš velká, lze odstranit neukládáním v AkcieCena (dotazy na průměrné ceny se ale komplikují).

Prokládání relací

Alternativa k denormalizaci. Není vždy podporováno DB systémem. Příklad: U záznamu dodavatele jsou uloženy jeho objednávky.



Horizontální dělení

V případě velkého množství dat lze rozdělit tabulky do partitions (horizontální dělení), které jsou z hlediska dotazů transparentní (nic není potřeba měnit, databáze s nimi pracuje jako s jednou tabulkou), ale na disku mají svá specifická označení.

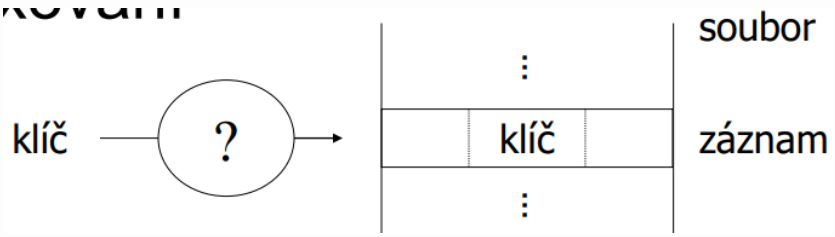
Při rozdělení tabulky se určí tzv. partition key, který pro každý záznam v tabulce určí, do které partition patří. Podle podoby klíče se rozeznávají 4 typy partitioningu * **List partitioning** - dělení podle výčtu hodnot (např. sloupec město patří do [Praha, Brno, Ostrava]) * **Range partitioning** - dělení podle intervalu (např. sloupec věk je z intervalu <10, 20>) * **Hash partitioning** - podobné jako List, ale počítá se hash nějakých hodnot * **Composite partitioning** - kombinace předchozích

Indexování a hašování, indexování pro více atributů

Indexování

Pro detailnější vysvětlení:
https://www.fi.muni.cz/~xdohnal/lectures/PB154/czech/zezula11.pdf

Důvod indexování: rychlejší přístup k datům, než kdybychom je sekvenčně procházeli Sekvenční soubor: soubor s daty z tabulky **Index: soubor ukazatelů na jednotlivé záznamy (<klíč, odkaz na záznam>)** Varianty indexování: * **Řazené indexy** * **B^+-stromy** * Existuje víc variant B-strom, B^*-strom - Obvykle se při vyslovení B-strom myslí B^+-strom * **Hašování**

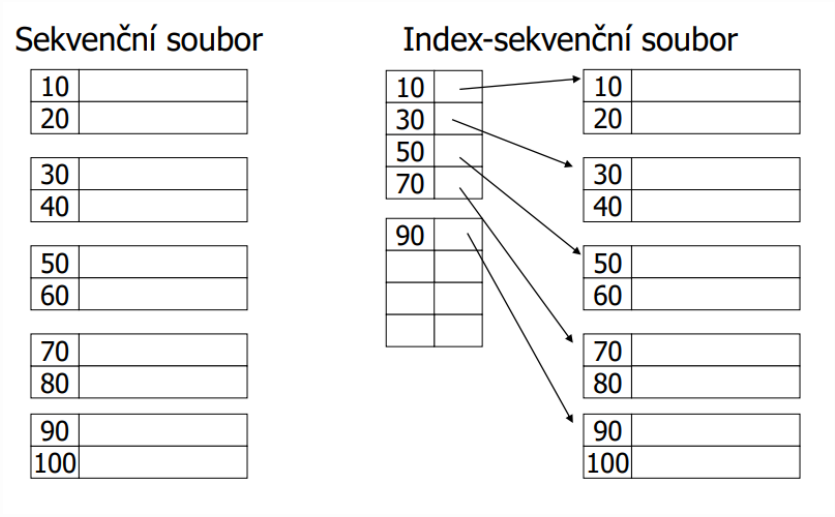


1. Řazené indexy

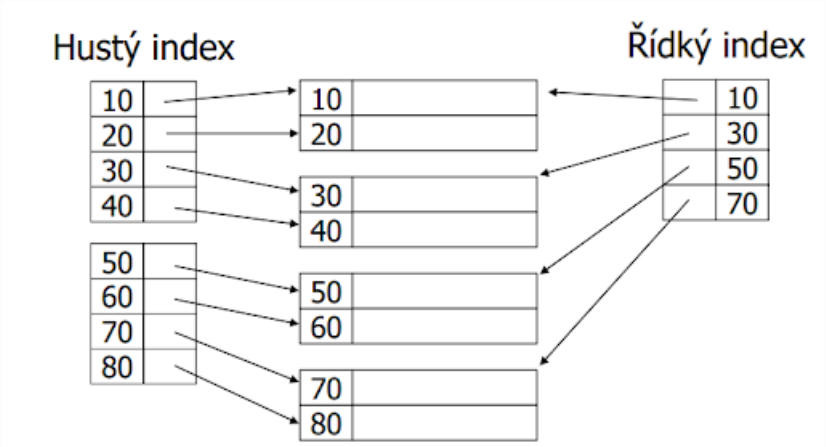
Index je sekvenční soubor. Index-sekvenční soubor: soubor s indexem a daty tabulky

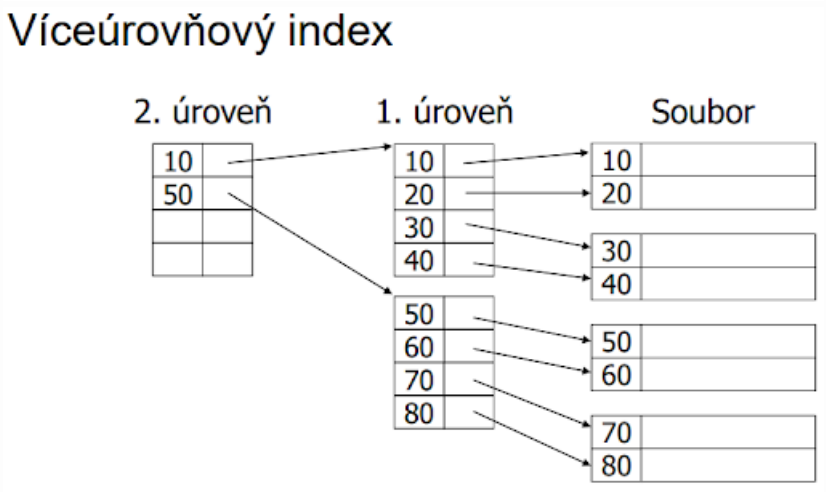
Výhody: * Jednoduché * Index je sekvenční soubor -> vhodné pro “full scan”

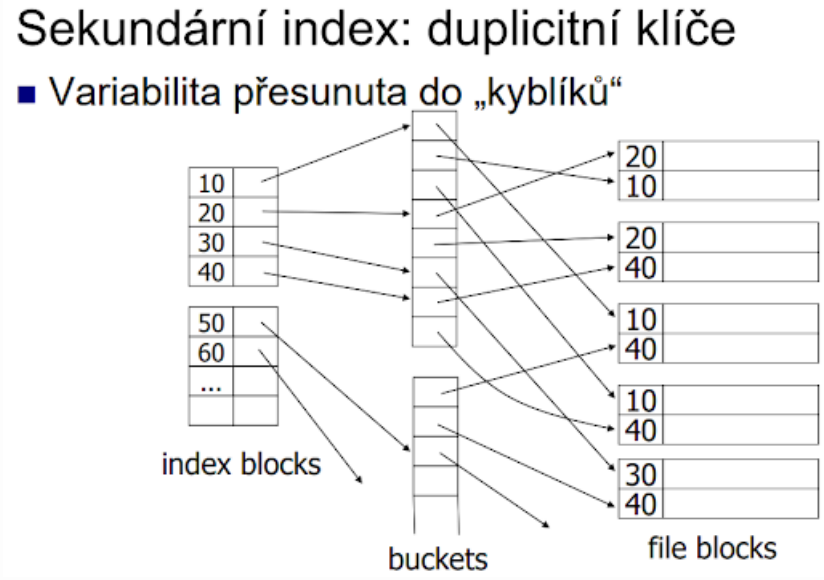
Nevýhody: * Aktualizace a inserty jsou drahé * Ztráta “sekvenčnosti” a vyváženosti kvůli přetokovým oblastem

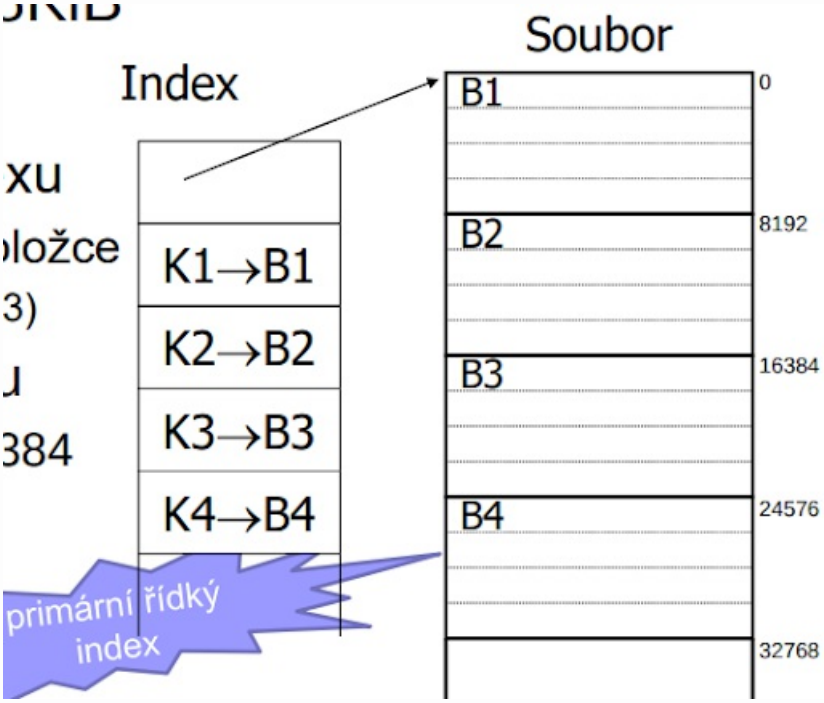


Hustý index: obsahuje indexové záznamy pro všechny záznamy v tabulce **Řídký index:** obsahuje indexové záznamy jen pro některé záznamy v tabulce **Víceúrovňový index:** rekurze indexů (index z indexu z indexu z indexu…) **Sekundární index:** soubor index záznamů uspořádaných jinak než primární index, soubor je uspořádaný podle jiného klíče.









Pokud máme duplicitní klíče, lze to vyřešit v sekundárním indexu pomocí kyblíků.

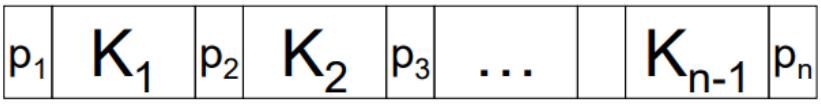
Index a ukazatele Ukazatele jsou v indexech, **ukazatel na záznam = adresa bloku + číslo (offset) záznamu. Adresa bloku = identifikace souboru + číslo bloku.** Pokud je soubor spojitý a uspořádaný, tak ukazatele na bloky se nemusí ukládat a stačí spočítat implicitní odkazy (číslo bloku * velikost bloku)

2. B+ stromy

Typ indexu, kde sekvenční uspořádání není nutné. Strom, kde každý prvek má dva ukazatele a hodnotu (případně více hodnot a příslušný počet ukazatelů). Ukazatel vlevo odkazuje na záznamy menší než hodnota, ukazatel vpravo odkazuje na záznamy větší než hodnota.

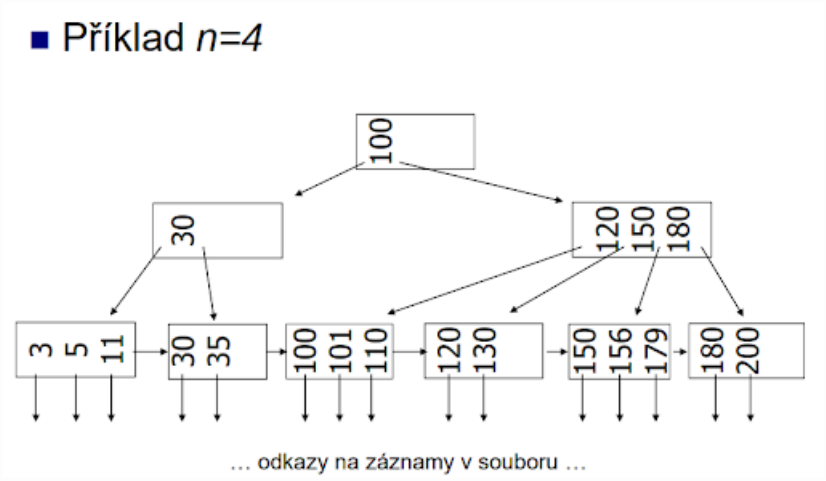
Vhodnější jako běžné indexy, protože rychleji vyhledávají (přestože mají větší režii a potřebují víc místa) a garantuje I/O pro přístup (tedy vyváženost). Navíc jsou B+ stromy plně **dynamické** (řeší vkládání/mazání), statický index vyžaduje reorganizace, B+ provádí

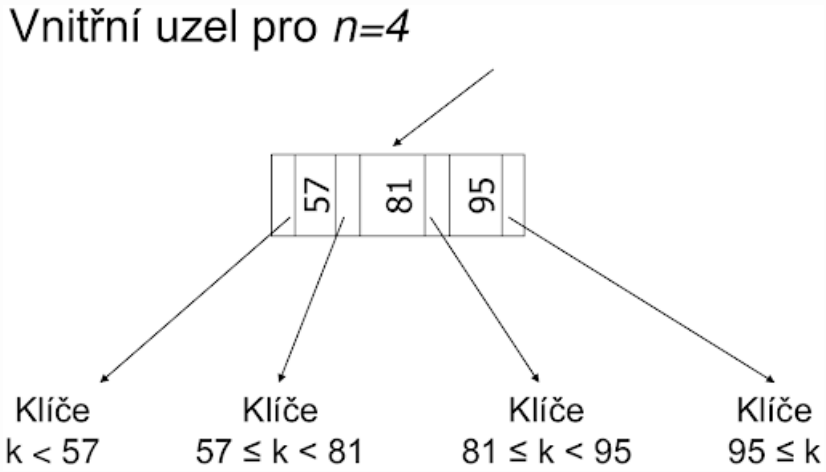
malé, statický velké. Nevýhodou je, že mají **větší nároky na paměť**. B+ je vhodnější organizace. Existuje spousta variant, ale obvykle se používá B+ strom. U B+ stromů se definuje "**n**", což je *arita* stromu, která ovlivňuje (PA152-S4/41): - tvar uzlu -



- minimální naplnění - listový uzel - Vždy na stejné úrovni - p_i odkazuje na záznam s klíčem K_i (data) - p_n odkazuje na další list (zřetězení listů) - nelistový uzel - p_i odkazuje na uzel, kde jsou klíče K: $K_{i-1} \leq K < K_i$

Více variant: * B-strom, B + -strom, B * -strom, … * Obvykle se při vyslovení „B-strom" myslí „ B + -strom"!





Parametr *n* (arita stromu) ovlivňuje:

- Tvar uzlu:

p_1	K_1	p_2	K_2	p_3	...		K_{n-1}	p_n

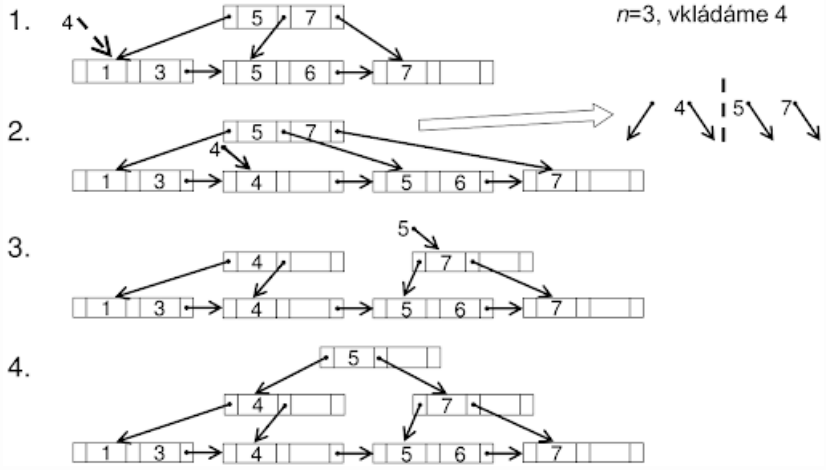
- Minimální naplnění
- Listový uzel
  - Vždy na stejné úrovni
  - p_i odkazuje na záznam s klíčem K_i (data)
  - p_n odkazuje na další list (zřetězení listů)
- Nelistový uzel
  - p_i odkazuje na uzel, kde jsou klíče K: $K_{i-1} \leq K < K_i$

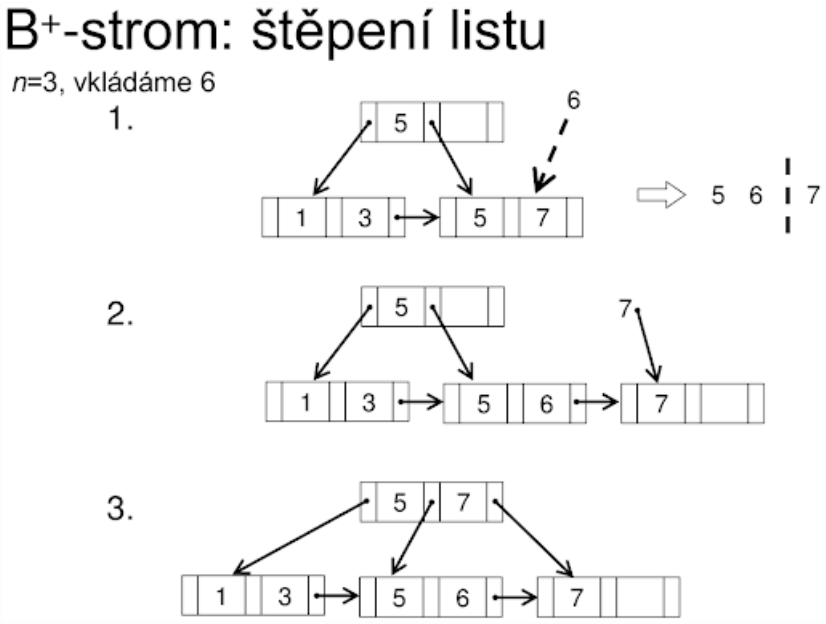
Podmínky naplnění

	Max ukazatelů	Min ukazatelů	Max klíčů	Min klíčů
Vnitřní uzel (ne kořen)	n (potomků)	$\lceil n/2 \rceil$ (potomků)	n-1	$\lceil n/2 \rceil$ -1
Vnitřní uzel (kořen)	n (potomků)	2 (potomků)	n-1	1
List (ne kořen)	n-1 (záznamů)	$\lceil (n-1)/2 \rceil$ (záznamů)	n-1 (záznamů)	$\lceil (n-1)/2 \rceil$ (záznamů)
List (kořen)	n-1 (záznamů)	0 (záznamů)	n-1 (záznamů)	0 (záznamů)

B+ strom: vkládání * Princip: Růst od listu ke kořenu * Postup: Nalézt listový uzel a vložit klíč * Včetně odkazu na záznam * Popř. aktualizovat rodiče * Případy: * a) Bez reorganizace (snadné) * V listu je volné místo * b) Štěpení listu * c) Štěpení vnitřního uzlu * d) Nový kořen

B+-strom: štěpení vnitřního uzlu





B+ strom vs. index-sekvenční soubor * B+-strom má větší režii (potřebuje více místa) * Je plně dynamický * B+-strom má složitější zamykání * Statický index vyžaduje reorganizace * DB neví kdy reorganizovat! * B+ -strom provádí pouze malé lokální reorganizace * Statický index provádí velké reorganizace * Buffer manager * B+-strom má „fixní“ nároky (log hloubka) * Konvenční index nikoli! (lineární složitost) - kvůli přetokovým oblastem * LRU (Least recently used) není pro B+ -strom vhodný!

B+ strom je vhodnější organizace.

3. Bitmapový (rastrový) index

Pokud je počet unikátních hodnot odkazovaného atributu malý, může se sestavit **bitový vektor** pro každou hodnotu a postavit index z těchto vektorů > matice bitů. Bitmapový index se typicky komprimuje například pomocí RLE.

- Relace $R(F,G)$

F	*G*
30	foo
30	bar
40	baz
50	foo
40	bar
30	baz
34	foo
80	baz

- Bitmapový index pro *G*

Hodnota	*Vektor*
foo	10010010
bar	01001000
baz	00100101

Výhody: * rychlé operace na bitech. * použitelné i pro rozsáhlé dotazy * snadná kombinace více indexů dohromady (bitové operace snadné)

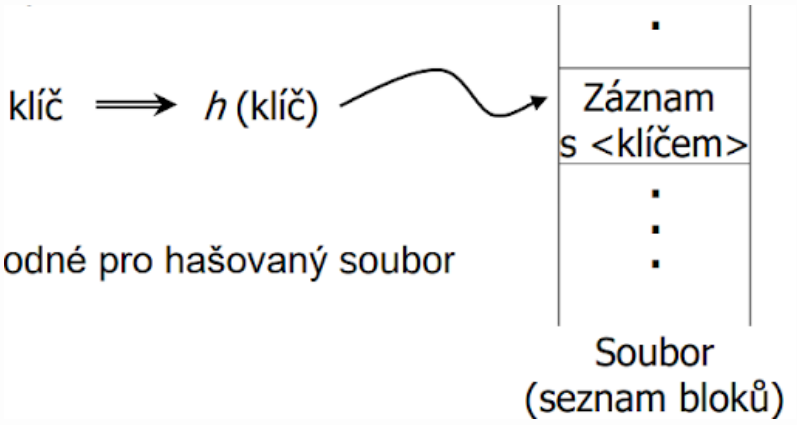
Nevýhody: * paměťová náročnost * aktualizace záznamů je náročná

Hashování

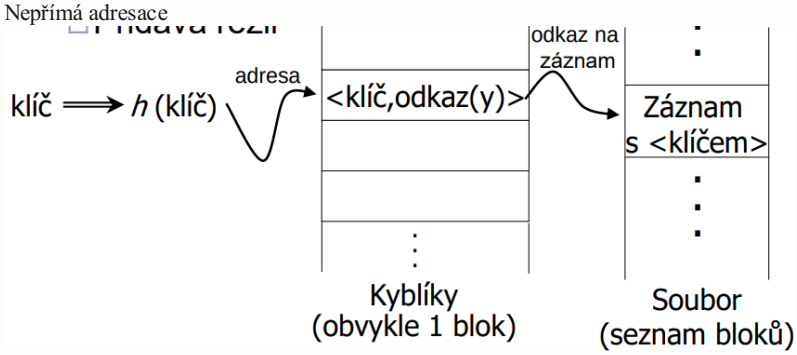
Transformace klíče na adresu za pomoci hashovací funkce. Ta by měla být rovnoměrná (rovnoměrně odkazovat na data) a náhodná (žádná korelace mezi vstupem a výstupem). Hashovací funkce je jednosměrná matematická funkce, která vstup přetvoří na výstup (Např. z "Jan Novák" udělá "a5x5dg4fg2", který slouží jako klíč.).

Existují dva druhy adresací pomocí hashe: * **Přímá adresace** - odkazuje přímo na záznam (vhodné pro hašovaný soubor) * **Nepřímá adresace** - odkazuje např. na další index (vhodné pro sekundární indexy), ale přidává režii

- Přímá adresace



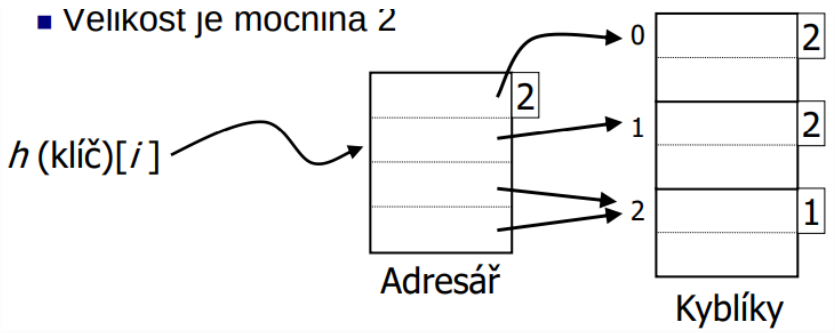
- Nepřímá adresace



Statické hashování - hashovací funkce generuje jen omezený počet hodnot, kolize se řeší přetokovými oblastmi. Např. mod 5 generuje pouze 5 klíčů (0, 1, 2, 3, 4) **Dynamické hashování** - hashovací funkce "roste" s množstvím dat, takže stále generuje větší počet hodnot. Přidáváme kyblíky, využíváme více horních bitů adresy pro hashování.

Když je na vypočtené adrese už něco uloženo, vzniká kolize. Existují dva druhy řešení kolize: * **Uzavřené hashování (otevřené adresování)** - Vytvořená adresa je fixní. Při přetečení se vytvoří přetoková oblast dat, a data se zřetězují. * **Otevřené hashování (uzavřené adresování)** - použije se další (tzv. kolizní) hashovací funkce a hledá se pro vkládaná data předem daným způsobem další volné místo tak dlouho, dokud se nenalezne nebo dokud se neprojde celá tabulka a místo se nenajde.

Rozšiřitelné hašování - Využití pouze několika prvních (horních) bitů adresy, Přidání další tabulky – adresář, Velikost je mocnina 2.

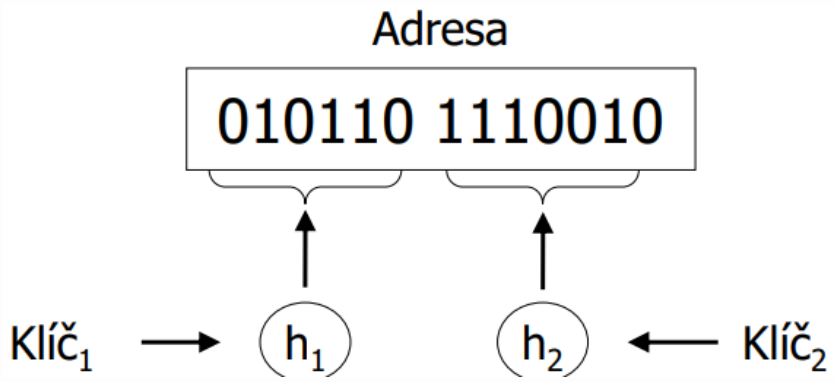


Výhody - měnící se data, méně plýtvá místem (než statické hašování), lokální reorganizace. Nevýhody - další úroveň nepřímých odkazů (OK, pokud je adresář v paměti), Adresář se zdvojnásobuje (nemusí se vejít do paměti, ale kyblíky rostou lineárně). **Hašování** (vhodné pro dotazy WHERE a=5) vs. **Indexování** (a>5)

Indexování pro více atributů

Index pro více atributů (sloupců DB) zároveň.

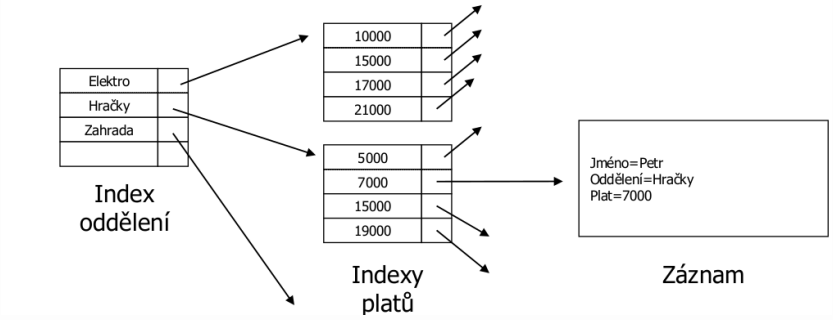
Existují 4 možná řešení * **Index pro jeden atribut** + filtrování * **Nezávislé indexy pro atributy + průnik vyhovujících** = vyfiltrují se záznamy z databáze pro každý zvlášť, a potom se udělá průnik. * **Index v Indexu** - v listech jednoho indexu jsou odkazy na druhý index. * **Spojení klíčů v jeden** (snadné u bitmapového indexu) - v indexování se nepoužívá. Lze využít u hašování = dělená hashovací funkce - N atributů je spojeno v jeden klíč a používá se N funkcí pro získání jednotlivých kombinací hodnot. * 2 klíče, 2 hašovací funkce, 1 adresa



Index v indexu:

SELECT jméno, plat FROM zam WHERE

a) oddělení = ‘Hračky’ AND plat ≥ 10000
b) oddělení = ‘Hračky’
c) plat = 10000



Transakční zpracování, zotavení

Integrita dat - chceme, aby byla data pořád bezesporná a správná.

Integrity docílíme pomocí **integritních omezení** = predikáty, které musí data splňovat. Například: číslo musí být kladné, sloupec “grade” musí být jedno z [A,B,C,D,E,F]…

Určité výrazy nelze implementovat pomocí integritních omezení (obvykle složitější podmínky - *záznam bankovního účtu lze smazat pouze pokud je balance = 0*)

Podle pozorování nemůže být DB konzistentní stále. Snaha je o maximální možnou konzistenci, k tomu se využívají transakce (Soubor akcí měnících data, ale udržujících konzistenci).

Transakce

Pokud provádím nad daty sekvenci operací, může se v jejich průběhu DB dostat do “průběžně nekonzistentního stavu”. K eliminaci tohoto stavu se používají **transakce**. **Databázová transakce je skupina příkazů, které převedou databázi z jednoho konzistentního stavu do druhého**. Transakce musí splňovat tyto podmínky ACID * **Atomicita** (angl. atomicity, A) - dále nedělitelná operace, buď se provede celá transakce nebo vůbec * **Konzistence** (angl. consistency, C) - při provádění transakce není porušeno žádné integritní omezení * **Izolovanost** (angl. isolation, I) - operace uvnitř transakce jsou skryty před vnějšími operacemi (pokud si transakce vytvoří pohled, ostatní paralelně běžící transakce ten pohled nevidí) * **Trvalost** (angl. durability, D) - změny při úspěšné transakci jsou skutečně uloženy v databázi, a již nemohou být ztraceny

Principy transakce * pokud je DB konzistentní před transakcí, bude konzistentní po transakci * pokud přerušíme transakci, DB zůstane konzistentní a neuloží se změny provedené v transakci * transakci můžeme zrušit během jejího průběhu pomocí příkazu REVERT * transakce se potvrzuje pomocí příkazu COMMIT

DB se může dostat do **nekonzistentního stavu**, pokud * nastane **chyba v transakci** * nastane **chyba v DB systému** * nastane **výpadek HW**

Kategorizace událostí * **žádoucí** - viz. manuál DB * **nežádoucí** * *očekávané* - ztráta obsahu RAM, zastavení CPU, vypnutí PC * *neočekávané* - ztráta dat na disku, živelná pohroma

Nežádoucí neočekávané události se řeší mimo DB systém - RAID pole, kontrola CPU, lepší RAM…

Atomičnost transakce a recovery

Atomičnost = provedení všech operací v transakci, nebo žádné Implementace obvykle pomocí **žurnálu - souboru o změnách**. Při chybě se čte žurnál a operace se buďto zruší (**UNDO**), nebo se provedou znovu (**REDO**). * **UNDO** - operace se ihned zapisují na disk, pokud není jistota (100%) uložení změn dokončené transakce, pak se při chybě podle žurnálu obnoví na původní hodnoty. Záznamy o operacích se do žurnálu ukládají dopředu, ale potvrzovací záznam "commit" se do žurnálu uloží až po uložení dat na disk. **Pokud je COMMIT nebo ABORT v žurnálu pro START, nedělej nic, jinak proveď UNDO.** * **REDO** - operace se zapisují až po skončení transakce, v případě chyby se jen spustí znovu. Záznamy v žurnálu, včetně "commit" musí být zapsány před samotným zápisem dat na disk. **Pokud transakce má v žurnálu COMMIT ale ne END, proveď REDO od nejstarších. Jinak nedělej nic.**

Nevýhody * UNDO - Ze zálohy DB nelze vytvořit aktuální stav DB * REDO - Všechny modifikované bloky musíme držet v paměti až do potvrzení (commit) transakce * Zápisy na disk jsou vynuceny pravidly žurnálu a ne přístupem k datům

Řešení: Undo/Redo logging, záznam v žurnálu obsahuje starou i novou hodnotu <T, x, nová, stará>

Undo/Redo logging * hodnota X může být uložena před i po potvrzení Ti * před zapsáním hodnoty X na disk, musí být na disk zapsán odpovídající záznam žurnálu (WAL) * ulož žurnál při potvrzení transakce * při obnově ukončené transakce zopakujeme od začátku a nedokončené transakce vrátíme od konce

DB systémy si v žurnálu dělají **checkpointy** s konzistentním stavem **žurnálu**. Pokud nastane incident, při kterém je porušený žurnál, vrátí se k poslednímu checkpointu. Pak se provede obnova (UNDO nedokončených a REDO dokončených). Žurnálování poskytuje systém, DB ho jen využívá.

Zabezpečení, přístupová práva, SQL útoky

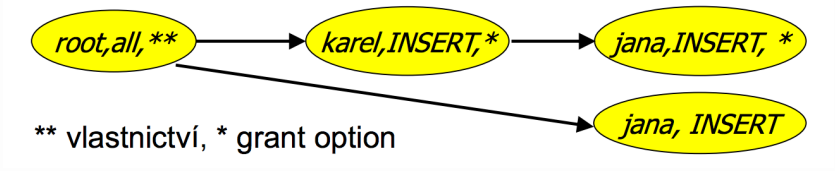
Zabezpečení DB

- **Přístupová práva v DB** Lze nahlížet na práva **podobně jako u souborového systému**. Přístupová práva v DB systému, **ale nabízí větší granularitu**. Práva lze definovat specificky pro tabulky, pohledy, sekvence, schéma, databáze, procedury, uživatele, skupiny uživatelů … Máme sadu práv pro různé subjekty, ty subjekty, které jsou mimo námi definované spadají do skupiny PUBLIC (pokud povolíme přístup pro PUBLIC znamená to, že přístup mají všichni). Přístupová práva pro relace
- **SELECT** - čtení obsahu, výběr řádků
- **INSERT** - vkládání řádků (lze omezit i na jednotlivé atributy)
- **DELETE** - mazání řádků
- **UPDATE** - aktualizace (lze omezit i na jednotlivé atributy)
- **REFERENCES** - vytvoření cizího klíče

VIEW Přístupová práva můžeme omezit i použitím vhodného **pohledu**. Např. pokud chceme na relaci Zamestnanci(id, jmeno, adresa, plat) skrýt výši platu, tak vytvoříme pohled, který nebude obsahovat plat … **CREATE VIEW** ZamestnanciAdresa **AS** SELECT id, jmeno, adresa FROM Zamestnanci. Potom už jenom stačí odebrat práva SELECT na relaci Zamestnanci a přidání práva SELECT na vytvořeném pohledu.

GRANT Práva se udělují pomocí GRANT <seznam práv> ON <relace nebo objekt> TO <seznam rolí, authorization id's>. Můžeme povolit i přidělování práv WITH GRANT OPTION Příklad: GRANT SELECT, UPDATE (price) ON sells TO sally; - uživatel sally může zobrazovat obsah relace sells a měnit obsah atributu price.

REVOKE Odebírání práv REVOKE <seznam práv> ON <relace nebo objekt> TO <seznam rolí, authorization id's>. Po odebrání práv určitému uživateli může nastat situace, kdy má stále přístup povolený - práva mu byla udělena ještě někým jiným případně spadá do skupiny uživatelů, která má práva nastavená. Toto se vyřeší přidáním **CASCADE za REVOKE**, zruší i oprávnění povolené uživatelem, kterému právě odebírám oprávnění. Přístupová práva se dají zakreslovat do diagramů. Diagramy reprezentují práva udělená kým a komu.



Uložené procedury

Uložená procedura je kód provádějící nějakou činnost (výpočet vzdálenosti GPS souřadnic, průměrného platu etc…)

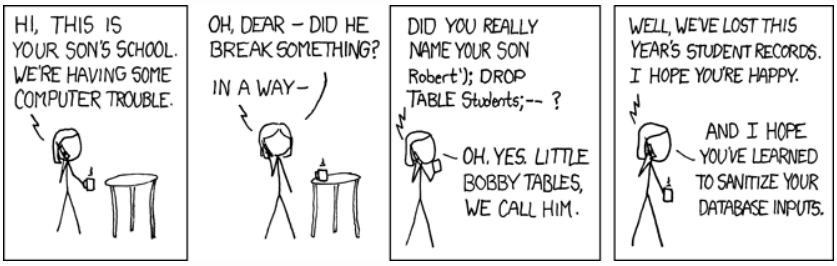
CREATE FUNCTION název (parametry) [návratová hodnota] kód Příklad: funkce pro výpočet průměrného platu na relaci Zamestnanci(id, jmeno, adresa, plat). `CREATE FUNCTION avgsal() RETURNS real AS 'SELECT avg(plat) FROM Zamestnanci'.` Uživatel pak jen zavolá `SELECT avgsal()`. Problém nastane v tom, že nemáme ošetřený přístup k platům - mohou je vidět všichni uživatelé. Je nutné *odstranit práva* na SELECT a přidat `GRANT EXECUTE ON FUNCTION avgsal() TO` těm uživatelům, kteří mají mít přístup pro výpočet průměrné mzdy. Provádění procedury probíhá pod aktuálním uživatelem - musí mít povolený SELECT na Zamestnanci… To ale neřeší problém, takže proceduře upravíme kontext provádění – pod kterým uživatelem se procedura zavolá. Zde to můžeme vyjádřit např. speciálním uživatelem pro tento účel. Normální uživatel tak zavolá tuto proceduru, ke které má práva, ta se spustí pod vlastníkem procedury a normální uživatel zjistí průměrnou mzdu, ale nebude mít právo na SELECT relace Zamestnanci.

Procedurám lze nastavit kontext provádění * **volající** - SECURITY INVOKER - provede se v kontextu uživatele, který volá * **vlastník** - SECURITY DEFINER - provede se v kontextu vlastníka procedury

Útoky na DB

Security problémy DB systémů * **nezabezpečené připojení** - vyřeší se šifrovaným SSL/TLS * **používání jednoho účtu k celému DB systému** - definovat skupiny uživatelů * **slabé heslo** *(hlavně u admina)* - admin-admin - nutit uživatele mít dostatečně silná hesla * **povolení přihlášení odkudkoliv** - omezit na specifické IP, případně na skupiny uživatelů

SQL Injection * SQL injection is a technique where malicious users can **inject SQL commands into an SQL statement, via web page input**. * Injected SQL commands can alter SQL statement and compromise the security of a web application. [w3schools] * uživatel zadá systému nějaké SQL příkazy místo platných vstupů ve formulářích a systém je potom chybně vyhodnotí. Například zadáním do poznámky uživatel může dropnout celou tabulku: * Vader'; DROP TABLE zakaznik; – * Nebo přeskočí podmínku a neoprávněně se autentizuje při vstupu: * a' or 'b'='b'; * príklad: SELECT * FROM uzivatelia WHERE meno = '$zadaneMeno' AND heslo='' OR 1=1–';



Obranou pro tento typ útoku je správně ošetřovat vstupy tak, aby se "escapovaly" v řetězcích znaky, které mají v SQL speciální význam.

SQL Injection countermeasures * používání uživatelských účtů a ne jen admina * kontrola vstupních hodnot, escapování stringů - dnes už to umí snad všichni * funkce v DB quote_literal(str) - vrátí string, který je vhodný pro uložení v DB a nenapáchá žádnou škodu - prostě se uloží jako string * funkce programovacího jazyka - PHP: mysql_real_escape_string() * prepared statements: * používají se na opakované vykonávání

stejných nebo podobných databázových výrazů (s vysokou efektivitou) * naparsované šablony připravené na použití * hodnoty sa nahradí, eliminuje sql injection

SQL Denial of Service (DoS) * použití vyhledávacích formulářů nebo vzdáleného přístupu (JSON/SOAP web service request) na vykonávání náročných a dlouhotrvajících dotazů * stránka sa stane pro uživatele nepoužitelnou

Krádež fyzických záloh (HDD, pásky) * triviální ale účinné * zálohy by měly být šifrované