Hlody relační algebry (pouze chytáky a špeky)

→ Vhodný materiál ke slidům ze cvičení, spolu tvoří vše co je potřeba znát ke zkoušce.

→ Budou tu nějaké chyby z nepozornosti (spíš pomálu)! V případě pochyb koukni do slidů!

- Výsledky operací nad relací v ní MAŽOU DUPLICITY
    - pozor na příklady typu kolik řádků má… s projekcí - smaže to co je tam vícekrát (např jména), hlavně když nezahrne atributy superklíče
- Pozor na POŘADÍ ATRIBUTŮ V RELACI při ověřování podobností výsledků v úkolu
- Operátory v podmínce = <= >= < > ^ (and) V (or) ¬(negace)
- Přejmenování musí mít vždy název tabulky (i když se nemění, a měním název atributů)
- Přejmenování se musí použít pokud
    - chci odkazovat na něco co nemá jméno, jako je výsledek agregační funkce
        - PÍ[maximum](G_max(A) as maximum(...))
    - je více sloupců stejného jména a vadí to (tzn. není to sloupec s hodnotami stejných domén
    - **více sloupců může vzniknout při kartézském součinu, musí se použít přejmenování**
        - **ve slidech ale používá při kartézském součinu sama se sebou jen přejmenování tabulky**
        - **pravděpodobně stačí přejmenovat tabulku, i v SQL to tak je**
- Konstantní relace – určení schématu přejmenováním (tedy definice přejmenováním)
    - ρ nova_relace('sloupec1', 'sloupec2', 'sloupec3')({('d', 'd', 8)}) je definice nové relace i s hodnotami
- Pro sloučení / rozdíl, relace musí mít kompatibilní počet atributů se stejnými doménami
- Insert a <- {výraz} U a musí sedět schéma výraz, a
- Delete a <- a \ {výraz} musí sedět schéma výraz, a
- Update a <- PÍ_attr(a) (tedy v projekci upravím pomocí oparátorů hodnoty atributů a)
- AVG, SUM, MIN, MAX na všech NULL vrací NULL
    - COUNT vrací 0
- GROUP BY(atr) vrátí i **NULL jako samostatnou skupinu třídy** (pokud je NULL v hodnotách přes který dělám třídu rozkladu)
- a,c ρ min(a), max(b) as nejvic vrátí relace(**a, c, min, nejvic)** - povoleno přejmenovávání
    - btw min(a) je celkem hloupost, protože a je stejné když podle něj skupím :)
- **neexistuje průnik, to je vyjádřeno pomocí sjednocení a rozdílu** a - (a - b)
    - tzn. neměl by se používat v příkladech rel algebry

SQL - Data Definition Language

DDL typy – char(n), varchar(n), int, smallint, numeric(a,b), real, double precision, float(n), date = 2010-5-22, time = 12:05:64, timestamp = date time, interval např 1 day (lze přičíst, je výsledek rozdílu), blob, clob

Vlastní typy: CREATE TYPE t AS <type> FINAL

CREATE DOMAIN d AS <type> <constrains>

Indices

CREATE INDEX <name> ON table(attribute)

Constraints

NOT NULL / ,PRIMARY KEY(a,b,c..), / ..,attribute domain PRIMARY KEY,.. /

Pokud je primárním klíčem, má i not null.

,FOREIGN KEY (a,b,c) REFERENCES s, / ..,attribute domain REFERENCES s,...

ON [DELETE/UPDATE] [CASCADE/SET NULL/SET DEFAULT] - možnost přidat za cizí klíč

UNIQUE (a,b,c..) - kombinace atributů v závorce musí být vždy unikátní - tedy superklíč

CHECK(predicate) např: CHECK (A in (<SQL query> spíše nepodporováno takže {'Pepa', 'Franta'}))

Tvorba a modifikace

CREATE TABLE t(attribute1 domain1, ⋯ attributen domainn, constraint1⋯)

lze až za: *foreign key (id_o) references osoby*

CREATE TABLE t(attribute1 domain1 constraint1,..)

i u definice - pokud není víceatributový: *id_o int references osoby*

ALTER TABLE t ADD attribute domain / ALTER TABLE t DROP attribute

DELETE FROM t vymaže tabulku, maže data, ponechá schéma

DELETE FROM t WHERE ⋯ smaže údaje vyhovující predikátu ve WHERE

bacha na změnu podmínky ve WHERE při změně údajů způsobenou smazáním

DELETE FROM auto WHERE cena < (SELECT AVG(cena) FROM auto) NOK

→ najít instruktory a pak teprve mazat (asi dva dotazy pod sebou, moc nejde jedním)

DROP TABLE t zničí tabulku - smaže schéma i data

UPDATE table SET A=val1, B=val2⋯ WHERE

UPDATE table SET A=CASE WHEN P() THEN val1 ELSE val2 END

UPDATE⋯SET A = (SELECT⋯) opět korektní pokud jen jedna hodnota s jedním atributem

INSERT INTO table(a,b,c⋯) VALUES (1, 'ahoj', 1531⋯)

INSERT INTO table VALUES (1, 'ahoj', 1531⋯)

musí odpovídat všem sloupcům table, null pokud nechceme specifikovat nějakou hodnotu

INSERT INTO table SELECT ⋯ FROM ⋯

vložení relace do relace, musí odpovídat sloupcům table

sql dotaz je vyhodnocen před vložením jako celek

problém pokud jsou duplicitní hodnoty prim. klíče

INSERT INTO table SELECT * FROM table → ERROR pokud je v table primární klíč a není prázdná, každý prim. klíč bude kolidovat

Triggers - struktura

CREATE TRIGGER name [AFTER/BEFORE] [UPDATE OF table ON attribute/DELETE ON table/INSERT ON table]

REFERENCING OLD [ROW/TABLE] AS col_name_old

REFERENCING NEW [ROW/TABLE] AS col_name_new

FOR EACH [ROW/TABLE] WHEN <predicate> BEGIN <SQL> END;

DML - data manipulation language

pořadí: **SELECT FROM WHERE GROUP BY HAVING ORDER BY LIMIT**

SELECT	operátory	* / + -	povinné uvést
WHERE	operátory	and or not	povinné uvést

- LIKE je case sensitive!
- || je concat na řetězci (asi v selectu by šlo SELECT name || '_' || surname,... → Pepa_Novák)
- ORDER BY má u každého specifikaci ASC/DESC řazením podle abecedy je ASC, default je ASC
- Algebra nelze na multimnožinu – lze jen SQL na množiny pomocí DISTINCT, výsledek v SQL je vždy nadmnožina výsledku relační algebry
- TEČKA v desetinných číslech (ne čárka)!
- PŘEJMENOVÁVÁNÍ nemusí být pokud dávám anonymní vnořený SQL příkaz, ale implementace to často vyžadují - i když bych to nevyužil
- **PŘEJMENOVÁVÁNÍ musí být pokud mám stejnou tabulku vícekrát v joinu/součinu**
- kartézský součin (a JOIN⋯ ON) NEELIMINUJE duplicitní sloupce

AS

⋯. SELECT neco FROM table AS t(a, b, neco)

Join

INNER JOIN / [FULL/LEFT/RIGHT] OUTER JOIN

+ jakým způsobem: NATURAL / USING(a,b,c) / ON r.a = s.a

neuvedení jednoho z nich dělá kartézský součin

ON neeliminuje duplicitní sloupce, USING a NATURAL eliminuje

INNER JOIN je asociativní, OUTER nikoliv!

BETWEEN val1 AND val2 (použito ve where)

Agregace

GROUP BY dělá skupiny, takže dá dohromady EKVIVALENTNÍ hodnoty, tedy pro každou skupinu GROUP BY se udělá funkce**, pokud chybí tak skupina == celá tabulka**

Atributy použité v SELECT/HAVING při agregaci MUSÍ být uvedeny v GROUP BY / vevnitř funkce !!!! např. *SELECT * ⋯ GROUP BY attribute* je špatně

SELECT COUNT(B) ⋯ HAVING COUNT (B) ok SELECT COUNT(B) ⋯ WHERE A ok

SELECT **A** ⋯. HAVING AVG(B) nok SELECT COUNT(B) ⋯ HAVING **A** nok obě chybí group

SELECT A ⋯. GROUP BY A ok SELECT COUNT(B) ⋯. GROUP BY A HAVING A ok

- dává smysl, WHERE je před, tedy ještě nebylo seskupování a ty atributy tam jsou, having je po, tedy už mám co řádek to skupina, pokud atribut nebyl uveden v group by a je použit v having, tak nemám zaručeno že všechny řádky co spadly do stejné skupiny mají stejnou hodnotu, nevím kterou bych měl vybrat

Pozor na úlohy typu COUNT(*) < 1 ⟶ toto pravděpodobně nefunguje, často je v tabulce primary key(=>not null) takže tohle je vždy false, musí se použít COUNT(to_co_chci_zjistit_zda_ma_null) < 1

Pozor na úlohy typu WHERE attribute <> NULL a podobné, NULL funguje pouze s IS/IS NOT, jinak vrací undefined == false

Vnořené sql (v implementaci musí být pojmenované, zde nemusíme pokud nechceme jméno použít)

⋯attr IN r / NOT IN r zda je atribut v relaci r - často výsledek vnořeného SQL dotazu

⋯attr <op> SOME r zda platí: atribut <op> některý z r (op:: <, >, <=, >=, =)

⋯attr <op> ALL r zda platí: atribut <op> všechny z r (op:: <, >, <=, >=, =)

EXISTS R / NOT EXISTS R zda je R neprázdná/prázdná, R často výsledek SQL dotazu

UNIQUE R zda je R bez duplikátů, kouzlo je použít ve vnořeným dotazu to co je z vnějšího, - tedy v závislosti na hodnotě jedné tabulky udělám dotaz nad jinou a zajímá mě zda jsou tak unikátní hodnoty

WHERE atr = (SELECT...) jako IN, ale validní je pouze jeden sloupec s jednou hodnotou jako výsledkem

FROM může mít vnořený SQL!!! SELECT * FROM table, (SELECT ⋯) AS table2...

SELECT může mít vnořený SQL !!! SELECT name, (SELECT sum(...)) AS salary,... FROM employees

– ale pouze s jedním sloupcem a jedním výsledkem jinak error

např: SELECT nazev, (SELECT SUM(kapacita) FROM skupina WHERE skupina.kod=predmet.kod) AS kapacita FROM predmet

INSERT INTO table (SELECT * FROM ⋯) -už tu bylo ale je to důležitý tak proč ne znovu :)

lze vkládat výsledek selectu, ! pouze **pokud sedí atributy** - tedy pokud jsme uvedli table(a,b,c) a výsledek selectu má kompatibilní a,b,c a nebo jsme neuvedli nic a výsledek selectu je všechny atributy

problém pokud vkládám i primární klíč, pak **duplicitní vložení hodnoty prim. klíče ohlásí chybu**

OPERACE s množinami

UNION /EXCEPT / INTERSECT

- Možno kombinovat s ALL - ponechává duplicity

VIEW

CREATE VIEW v AS <SQL>

dotazy mají následující podmínky - jinak se těžko zachovává integrita dat

pouze jedna tabulka může být specifikována v údaji FROM

pouze atributy této tabulky v SELECT, bez agregací či aritmetických operací

všechny "neviditelné" atributy (nespecifikované ve CREATE VIEW) musí jít nastavit na null

žádné agregace, bez having

ERD

Entitní množina – typ (osoby, auta)

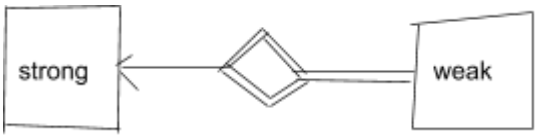
Entita – jeden prvek (pepa, bmw)

Vztah je relace mezi dvěma a více entitami E: {(e1, e2...en) | e1 z E1, e2 z E2 ⋯. en z En}

Typy atributů - jedno / více hodnotové jméno / adresa
ulice
město

- jednoduché nebo složené příjmení / {děti}
- odvozené datum_narozeni()

weak entity set - musí mít TOTÁLNÍ one-to-many relationship k identifikující skupině (tedy ta slabá má právě jednu identifikující entitu)



discriminator - atributy slabé entitní množiny které spolu s primárním klíčem silné jednoznačně identifikují objekt slabé (tedy skupina předmětu bude mít číslo jako diskriminátor, předměty budou mít více skupin stejných čísel, pokud přidám identifikátor předmětu mím jednoznačně určenou skupinu)

Identifikující vztah je dvojitý diamant (ten vztah se pak VŽDY modeluje přidáním primárního klíče do slabé entitní množiny, oddělené je to jen v tom diagramu)

!!! Do RELACE VZTAHU (tedy tabulky vzniklé z kosočtverce) lze uložit vztah POUZE JEDNOU, **nelze mít tedy např. více výpůjček** mezi stejné auto-zákazník pokud je návrh entita-*relace*-entita: auto(id_auta, ..), zákazník (id, ⋯), *výpůjčka(id_auta, id)*

Naopak, lze pokud návrh entita-*relace*-entita-*relace*-entita: auto(id_auta, ..), zákazník (id_zak, ⋯), výpůjčka(id_výpůjčky, ...), *půjčka_a(id_auta, id_výpůjčky)*, *půjčka_z(id_zak, id_výpůjčky)*

Modelování vztahů

M-N - dvě tabulky pro entity, jedna tabulka pro vztah z prim. klíčů obou entit (e1klíč, e2klíč)

- možnost mít v tabulce vztahu i další atributy (datum vytvoření vztahu atd⋯)

1-M - dvě tabulky pro entity, přidej cizí klíč pro e1 do entity e2 (na straně M)

- každý z těch M hodnot si pamatuje která jedna hodnota mu patří, těžko obráceně, to by si jeden musel pamatovat všechny hodnoty co k němu patří - porušení 1NF

multivalued attribute - maps to (only if the parent entity has more attributes) new entity (id, val1), (id, val2), (id, val3)...

INHERITANCE / DĚDIČNOST

- disjoint /výlučné - jen jeden nebo druhý (spojená šipka generalizace, pokud je TOTÁLNÍ tak modelováno tak že rodičovská entita není a její atributy se kopírují do dětí)
- overlapping/překrývající - může být entita ve více narozdíl od disjoinu (a **tedy nelze modelovat tak že do děděných tříd nakopíruju společné atributy**, by duplikovalo data)
- total - instance pouze potomků (tedy rodič je abstraktní - jako abstract class)
- partial - instance jak potomků tak i rodiče, **nelze opět modelovat kopírováním do potomků**

Modelování:

1. schéma pro rodiče, schéma pro potomky s klíčem na na rodiče(nutnost číst více tabulek)

2. schéma pro rodiče, schéma pro potomky s atributy rodiče (pokud total tak jen potomci instance rodiče nemůže být), problém - více dat duplicitně pokud není disjoint (společná data rodiče, potomek v relaci 2x)

AGREGACE - dívám se na entity s jejich vztah jako na entitu, reprezentace - id agreagace (= id všech entit -ne vztahů- v ní), id entity a další atributy

Závislost **PLATÍ** nad tabulkou pokud je splněna pro všechny sémanticky správná naplnění id->jméno

Závislost **JE SPLŇOVÁNA** tabulkou pokud v ní závislost platí, i když obecně nemusí jméno->adresa

Návrh databáze

FD - Funkční závislost A->B: shodují-li se dva řádky na A, pak i na B (t,s t[A] = s[A] ⟶ t[B] = s[B])

Dekompozice R na R1...Rn ⇔ Sjednocení R1...Rn dává dohromady R

relace splňuje FD pokud všechny řádky splňují FD (někdy platí jmeno->ID_osoby)

FD platí v relaci pokud všechny možná správná naplnění relace splňují FD (vždy ID_osoby->jmeno)

1NF

Žádný atribut neobsahuje logicky více informací, tedy nějaký další program neextrahuje data

2NF

Každý atribut, který sám není v **kandidátském** klíči, není parciálně závislý na **kandidátském** klíči (ne superklíči, superklíčem jsou často i všechny atributy) - tedy A není v kandidátském klíči, (B,C) tvoří kandidátský klíč, pak neplatí B -> A ani C -> A (ale samozřejmě platí B,C -> A)

Výpočet uzávěru F+ (výsledkem je MNOŽINA závislostí)

lze dokola aplikovat armstrongovy axiomy

lze pro každou podmnožinu atributů N vypočítat uzávěr N+ a přidat N->N+ do F+ (správně se

má přidat pro každou podmožinu N+, ale pokud počítám F+ pro svoje potřeby ne jako výsledek, pak je lepší si tam dát pouze to jedno pravidlo, a pak vědět že pokud A->BC pak A->B a A->C)

Uzávěr atributů a1...an (výsledkem je MNOŽINA atributů)

přidám do resultu atributy a1...an pak pokud ai->bi; ai je v resultu, přidej bi do resultu

Přebytečný atribut E v A -> B

E vlevo: vypočítat uzávěr A bez E, když vyjde B - E je zbytečný

E vpravo: vypočítat uzávěr A s použtím pravidla A -> (B bez E), když vyjde B - E je zbytečný

Kanonický uzávěr F

Dokola: 1. spoj takové závislosti co mají levou stranu stejnou

2. odstraň přebytečné atributy; if (F changed) goto Dokola

Ověřování funkční závislosti v relacích R1...Rn vzniklých z R

Pro každé pravidlo A -> B dej do výsledku A a dokud se ve výsledku něco změnilo od posledního cyklu:

1. V každé Ri uvažuj z výseldku jen atributy z Ri, udělej nad tím atributový uzávěr
2. uzávěr ořež na to co je jen v Ri a přidej do výsledku

Ověř že výsledek obsahuje vše z B

BCNF - pro všechna FD tvaru A->B platí: je triviální nebo A je **superklíč**

BCNF vytvoření

- každou A->B co ruší BCNF dát do nové tabulky, a ve staré nechat všechno bez (b-a) tedy odstranit všechny atributy z A, B a pak tam vrátit A (zachová lossless join)

- **nutnost provést kontrolu zachování BCNF po dekompozici**

- nezapomenout na převod do 1NF - tedy zkontrolovat atomicitu

Ověření BCNF - hard

Vypočítej atributový uzávěr všech pravidel A -> B a musí vždy vyjít result == R(celá relace)
uzávěr počítej z: 1 tabulka - F, více tabulek - F+

Ověření BCNF - easy (pro dekompozici na R1...Rn bez použití F+)

Vypočítej atributový uzávěr **všech S-podmnožin atributů** v Ri a musí vždy vyjít S+
obsahuje Ri nebo result **neobsahuje nic z (Ri - A)** - v S+ může být to co je v A nebo není v Ri

3NF - BCNF + nebo každý atribut v (B-A) v A->B musí být v **kandidátském** klíči (ne superklíči, superklíčem je mj. celá relace, to by pak platilo vždy)

3NF Vytvoření

Vypočítej kanonickej uzávěr, každou FD z něj narvi do samostatné relace, pokud žádná z relací neobsahuje kandidátskej klíč původní relace, tak ho přidej v další nové relaci, smaž všechny relace co jsou podmnožinou jiných (obsahují stejné atributy)

MVD pokud mám platící A ->-> B1 a řádky **s, t** se shodují na A, pak je v relaci následující:

s(A, B1, X1) t(A, B2, X2) u(A, B1, X2) v(A, B2, X1) (může B1, X1 != B2, X2)

X⋯ hodnoty atributů v ostatních sloupcích relace, B - hodnoty v práve straně závislostí

např: osoba(jméno, rok, město, nápoj) platí jméno ->-> rok, město

s(A: pepa, B: 1998, B: Brno, čaj) t(A: pepa, B: 2005, B: Praha, kafe)

u(A: pepa, B: 2005, B: Praha, čaj) v(A: pepa, B: 1998, B: Brno, kafe)

Uzávěr MVD:

platí pokud a->b pak a->->b (tzn. používej FDs)

vypočítej uzávěr pomocí pravidel MVD a FD podobně jako předtím

Test 4NF

vypočítej uzávěr MVD a zkontroluj že všechna pravidla ve výsledku mají vlevo superklíč

Tvorba 4NF

pro každé A->->B (a tedy i A->B) které porušuje 4NF v tabulce T dej do nové tabulky (podobné jako tvoření BCNF) tak, že T1 = (T - (B-A)) T2 = B + A, opakuj dokud některá tabulka porušuje 4NF

Dočasná data mají určený interval po která jsou validní

Snapshot - hodnota v určitém momentu

Nelze přidat do schématu, rozbíjí FD (mění data, tedy neplatí že jsou jednoznačně určeny)

Dočasná FD X->Y platí pokud FD tvaru X->Y platí pro všechny snapshoty

Kdy normalizovat? - někdy je lepší naopak normalizaci nedělat, pokud bych kvůli tomu musel často dělat join → nechci pořád zbytečně počítat

1) nedělat normalizaci, rychlejší ale paměťově mám duplicitní data, a možnost že tam bude chyba -> nutnost dělat extra programátorské výpočty, prostor možných chyb
2) udělat normalizaci a join definovat jako materialized VIEW (tedy uložím se join), podobné jako předtím akorát není třeba dělat extra výpočty v uživatelském kódu, bez možných chyb

Paměti (odsud dál je to prakticky výtah ze slidů)

Klasifikace podle: rychlosti čtení/zápisu, ceny za jednotku úložného prostoru, spolehlivosti (MTTF, schopnost zotavení - žurnály)

Energeticky závislá (volatile) VS Nezávislá (non-volatile)

Flash - má 10-1M limit, NOR vs NAND (levnější), 25MB/s

musí vymazat než znova zapíše - pomalé, radši přemapuje adresy logické - to co poskytuje ven

na fyzické - to co je uvnitř - nabídne prázdné místo aby mazal zbytečně požadované adresy,

toto vykonává flash translation layer

SSD disky - více flash pamětí, paralelizovaný přístup - až 100MB/s

Magnetický disk - Čas přenosu = Doba přístupu (Doba vystavení 4-10ms + Doba rotačního zpoždění 4-11ms) + Doba přenosu (transfer rate, omezena i podporou řadiče) 25-100 MB/s

má přímý přístup na rozdíl od pásky

Sektory (500-2000) po 512b ve **stopách (track)** (50k-200k) na **disku (platter)** (1-5 na zařízení)

Posuvné rameno - čtecí a zápisová hlava, **Cylindr** - stopy číslo i na všech discích

Diskový řadič (disk controller) - rozhraní, vykonává zápisy, čtení: výroba a kontrola kontrolních součtů sektorů, mapování vadných sektorů (ví které jsou vadné - přemapovává na nové) a ověřování správnosti zápisu (často to dělají disky pokud víc připojeno přes jeden řadič on to řídí)

- ATA, SATA, SCSI, SAS

Připojení disků

- Typicky přes sběrnici a řadič - osobní počítače
- **Storage Area Networks (SAN)** - připojeny disky vysokorychlostní sítí k serveru (ne sběrnicí)
- **Network Attached Storage (NAS)** - připojení k úložnímu serveru přes FTP (tedy rozhraní sítě) místo rozhraní řadiče

MTTF - mean time to failure, průměrná doba běhu bez poškození, klesá se stářím, cca 3-5 let u nových

Optiky JukeBoxy na automatickou výměnu

CD - 640MB, doba přístupu 100ms, přenos 3-6 MB/s

DVD- 4/9.5/17GB BLUE-RAY 27GB

-R zápis jen jednou, nesmazatelné, velká životnost (další +R, -RW, +RW, RAM, ROM)

Magnetické pásky - pouze sekvenční přístup, Jukeboxy na automatickou výměnu (petabajty dat), drahej je drive ne

medium (vyměňují se)

pár GB Digital Audio Tape 10-40GB Digital Linear Tape 100+ Ultrium 330 Ampex

záloha na delší dobu, omezeno na sekvenční přístup, pomalé čtení

← HIGHER PRICE | HIGHER STORAGE →

registry, cache, main memory, flash memory, magnetic disc, optical disc, magnetic tapes

Primární paměť - energ. závislá, málo paměti ale rychlá (registry, cache, RAM)

Sekundární paměť - nezávislá, středně rychlá, větší kapacity, stále “online” (HDD, flash - SSD)

Ternární paměti - nezávislá, jen na uložení a občasné použití, offline storage (pásky, optiky, dnes i HDD disky)

Blok - několik sektorů fyzicky vedle sebe - typicky 4-16kB (sektor má 512, tedy 8-24 sektorů)

Přísup k pamětem

Blokový přístup - několik sektorů dohromady jako optimalizace - typicky 4-16 kb

menší bloky - více přístupů na disk pro čtení

větší bloky - více nevyužitého místa, větší fragmentace, časem defragmentovat

Algoritmy přístupu - výtah (jezdí od kraje ke kraji disku a dělá vždy nejbližší požadavek v tom směru kam jede, šetří pohyb hlavy - ne zigzag ale vždy jedním směrem), SSTF (shortest seek time first - vždy nejbližší), FCFS (first come first served, podle toho jak přišly)

Uspořádání na disku - file organization

řeší souborové systémy, např uložit soubor vedle sebe na stopu - ok rychlý přístup, časem problém fragmentace, další přístupy pomocí zřetězených bloků (např FAT) či tabulky adres bloků k souboru (i uzly)

Zápisové buffery - NON VOLATILE RAM, NV-RAM - urychlení zápisu místo na disk do bufferu - RAM paměť se záložním zdrojem napájení, zapisuje až má čas - když vypne náhle disk, tak se data neztratí, zapíše až se znovu nahodí, **přeuspořádává požadavky aby minimalizoval pohyb hlav** (viz algoritmy)

Log disk - princip zápisového bufferu ale jako celý samostatný disk, zapisuje rychleji protože se v něm nevyhledává a pak když je čas tak přepíše na hlavní

ŽURNÁLY, - log disk nebo NV-RAM, má historii změn na disku (drží jakýsi "*diff*"), tedy náhle odpojení nepoškodí, lze (většinou) data obnovit (drží i indexy, může se stát že se poškodí indexy - tedy nevíme kde je co uloženo a ja potřeba disk přeindexovat, může trvat i hodiny → žurnály to urychlí)

RAID

Proužkování na úrovni bitů - každý bit z byte na jeden disk (tedy 8 disků), čte teoreticky 8x rychleji ale mnohem horší access time (tedy nejhorší čas vyhledání na všech discích), problém nelze serializovat přístup k různým datům,

Proužkování na úrovni bloků - blok i ze souboru jde z n disků na disk **i mod n + 1** (pokud čísluji od 1), pokud chci různé bloky lze serializovat přístup, jak pro jeden dlouhý záznam (který je na více discích) tak i pro vůbec nesouvisející záznamy

RAID 0 - blokové stripy (blok i jde na disk (i mod n) + 1 (+1 čísluje od jedničky)) , více disků, rychlé, rizikové, používá se u dat co nevadí když se ztratí

RAID 1 - duplikuje všechny zápisy, drahé ale bezpečné, rychlý zápis, např 4 disky dat + 4 disky kopie, rychlejší zápis než další varianty RAID (kontrolní součty: musí jak číst data tak zapisovat), tady zapisuje jen 2x a to ještě lze paralelně s HW podporou - ALE moc disků, drahé - používá se pro kritické věci (např log disky, ty nesmí selhat)

RAID 2 - ECC kontrolní kódy (přidává X redundancy bitů k bajtu), bit striping - každý bit na jiný disk, např 8 disků na data a 3 disky na 3 ECC bity, lepší je použít RAID 3

RAID 3 - Bit interleaved parity, jeden bit oprava na disk - XOR přes všechny bity bajtu, pokud failne jeden bit dat, tak ho získá jako XOR přes všechny bity z disků ale místo toho vadnýho bitu dá bit opravy, např 8 disků na data a 1 na opravu (levnější než 2, ale pomalé - všechny disky se musí používat), nelze paralelizovat, příliš mnoho seek time (pohyb hlavy),

RAID 4 - jako 3 ale s použitím bloků - rychlejší, různé bloky paralelní čtění, pořád problém musí používat ten paritní disk vždy (omezuje paralelismus), lepší je kvůli úzkému hrdlu toho paritního disku

RAID 5 - paritní blok n je uložen vždy na disku (n mod [počet disků]) + 1, jako 4 ale vyhýbá se závislosti na jednom disku, rychlejší než 4, volba pokud aplikace s málo častou aktualizací dat a hodně daty

RAID 6 - jako 5 ale ukládá DVA kontrolní součty vedle (n mod | n+1 mod), stabilnější, dražší, většinou se nepoužívá stačí 5

SOFTWARE VS HARDWARE RAID (hardware problematickej při selhání napájení, když se nepodaří například zapsat kopii na miror disk), používá log disky, NV-RAM aby se dokázal zotavit

Problémy HARDWARE RAID a jejich HW řešení

Latent failures - poškození již správně zapsaných dat (tedy časem dojde k chybě na médiu)

Data scrubbing - stále procházení disku, periodická kontrola a obnova ze zálohy

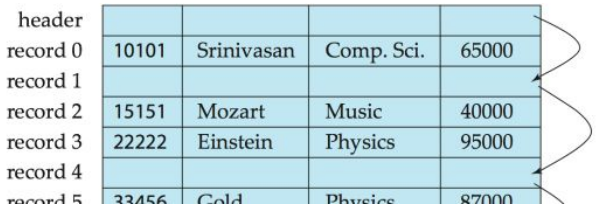
Hot swapping - prohození RAID disku za běhu (např. drží jeden disk stranou který je online a připraven na prohození při poruše)

Problémy "single point of failure" = jeden řadič, jeden zdroj napájení, snaha o to mít záložní zdroje a případně více řadičů, pokud něco selže

Organizace v souborech

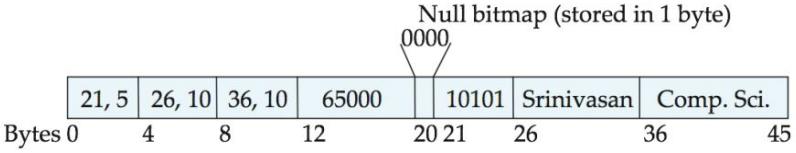
Záznamy pevné délky - jednoduše záznam i délky n uložen na n * (i-1) pozici: jako pole, omezení - pevná délka neumožňuje přetíkat bloky, těžko se pracuje se stringy ad.

mazání - buď musím přesunout všechny následující, nebo ho nahradit posledním, nebo mapa volných pozic (linkedlist volných bloků, tzv **free list, volný seznam**)



Záznamy proměnné délky

např varchar, atributy řádku tabulky jsou stále ukládány v pořadí sloupců, ale tam kde je atribut proměnné délky, je uložen offset + length a vlastní data ukládána za všechna pole fixní délky



null hodnoty reprezentovány jedním bytem nul (0000)

Slotová stránka (slotted page) - hlavička s

počtem záznamů, pozice konce volné části, pozice a délky pro každý záznam (1|2|3)

|HEADER|1|2|3| FREE SPACE | BLOCK1 | BL2 | BLOCK3 |

odkazy zvenku do hlavičky, ne do záznamu! (1 a ne na BLOCK1, v 1 nalezne odkaz i délku)

Uložení dat v souborech

heap / hromada -> kdekoliv, mapa adres a pozic dat

sequential / sekvenční -> seřazené podle vyhledávacího klíče, používá pointery aby nemusel přesouvat data při modifikaci

hashing / hashování -> podle hodnoty funkce na klíči skok na adresu

multitable - uložení dvou souvisejících relací v jednom souboru pro ušetření joinu

SEKVENČNÍ ULOŽENÍ

mazání - pomocí pointerů na další záznam - přeskočí

přidávání - pokud není místo tam, kam chci záznam dát (ve free-listu) tak musí do přetokové oblasti, přeuspořádávat po čase (nevhodné pro 24/7 systémy)

MULTITABLE CLUSTERING FILE ORGANIZATION

dobré na joiny, horší na dotazy nad jednou tabulkou, naházené údaje více tabulek v jednom souboru tak že je seřazen pod cizím klíčem tabulky celý záznam s tímto klíčem jiné tabulky (šetří místo), pointery mezi takto oddělenými záznamy jedné relace (osoba X telefon, telefony pod osobami)

super pro joiny - kde chceme vždy uživatele a jeho telefon, špatný pokud chceme jen data o uživatelích atd.., pro lepší použitelnost má pointery mezi uživateli

```
|pavel| novák| 5| ------
|5 | 544 240 640 |      |
|5 | 775 156 321|       |
|jitka |stará | 6 |  ←----
|6 | 777 651 333 |      |
```

Data dictionary storage (oblast datového slovníku) - úložiště metadat, tedy schémata, názvy a definice views, omezení, definice omezení (nut null, prim key) a práv přístupu uživatelů k databázi, záznamy o uživatelích, statistická data (počet záznamů..), jak(hash/sekvenčně) a kde(adresy) jsou data uložena..

Přístup na disk - data tabulek jsou rozdělena do bloků, v RAM je alokován buffer (spravuje buffer manager) kde se bloky nahrávají aby se s nimi dalo pracovat, - pokud je hledaný blok v bufferu, vrací se adresu v bufferu, jinak

- vyhradí se místo v bufferu (pokud není - vyhodit starý blok, nutnost zápisu na disk pokud vyhazovaný byl modifikován - používán pro identifikaci tzv dirty bit)
- načíst blok z paměti a vrátit adresu

Výběr oběti - LRU (last recently used)

- oběť nejdéle nepoužitý blok
- může být hloupý pokud děláme join 2x for each (bude vždy tahat data přes celou relaci které se nevleze do bufferu a bude ji n krát nahrávat znovu)

Pinned block - blok který je zakázán vracet na disk

Toss-immediate strategy - uvolňuje blok hned poté co byla použita poslední entice bloku (předpoklad - přečtena = použito, nepotřebné)

MRU - most recently used strategy - systém poznačí pin na blok který je zpracováván, poté co je zpracován tak je poznačen za most recently used a oddělán pin

Indexování

- klíč a pointer na data
- Každý prim.klíč má by default index
- efektivní podle: podpora dotazů (range VS exact), doba přístupu, doba vkládání/mazání a paměťová náročnost
- ulehčují vyhledávání ale ztěžují insert/delete
- exact VS range (rychlé jen na primárním indexu) dotazy

Seřazené indexy

primární index (clustering index, seskupující) - podle něj je uspořádán celý soubor, nemusí být založen na primárním klíči-často je, často řídký (sparse)

sekundární index (non-clustering index, neseskupující) - definuje uspořádání jiné než jak je soubor uložen, nesmí být řídký ale hustý (dense), tedy vyhledávání podle různých dalších atributů
není vhodný pro sekvenční procházení souboru, ukazuje na různé pozice, seek pro každý řádek

index-sekvenční soubor - soubor který je sekvečně seřazený a je k němu dostupný primární index

řídký index - ne index všech 1,2,3,4 ale pouze 1,3; chci 2 -> jdu do 1 a pak sekvenčně hledám následující), úspornější ale pomalejší, dobré po blocích (na každý blok jeden index na začátek)

hustý index - adresy na všechny údaje

víceúrovňový - multilevel index - řídký index do bloků hustého indexu a ty do záznamů (řídký index je použitelný protože hustý je VŽDY SEŘAZEN), urychluje vyhledávání i nad sekundárními idexy

MAZÁNÍ

hustý index buď přesunout všechny údaje / mít seznam volných bloků a prázdná pole přeskakovat (jako záznam v sekvenčním souboru)

řídký index - pokud mažu údaj který má klíč v indexu, klíč v indexu je nahrazen klíčem následujícího údaje (pokud tam již není), pokud už tam byl(tak jen smažu ze seznamu indexů

VKLÁDÁNÍ - hustý index vždy vložit nový index, řídký - pouze pokud vložením vytvářím nový blok, pak přidat index na první údaj do nového bloku, multilevel index - kombinace obou přístupů

Problémy -> moc údajů podobných klíčů - husté jen někde, hodně přetokových oblastí při insert/delete, nutnost reorganizace čas od času ⇒ řešení B+Stromy

B+Strom arity n (BALANCED, ne binary)

list má alespoň "větší polovinu maxima" (POKUD NENÍ KOŘENEM) - ceil((n-1)/2) a max n-1 hodnot, - jednoduše arita 6, hodnot max 5, větší polovina z 5 je 3;

má maximálně n-1 hodnot

ukazatel VLEVO od klíče ukazuje na záznam, poslední vpravo na další list

vnitřní uzel má alespoň ceil(n/2) (min 2 POKUD JE KOŘENEM)

max n ukazatelů

ukazatel nalevo od klíče ukazuje na podstrom s hodnotami ostře menšími než klíč

ukazatel vpravo ukazuje na podstrom s hodnotami stejnými nebo vyššími

vysoká arita -> malá hloubka - uvažujeme-li nejhorší případ všude jen polovina pointerů(**h = log_(ceil(n/2))(počet klíčů)**), málo přístupů na disk

typicky uzel 4kb, arita 100

vkládání - **vždy** přidat novou hodnotu tam kam ji chci, pak pokud je tam víc hodnot než povoleno, tak rozpůlit aby pokud lichý víc hodnot vlevo, půlení listu - z pravého nového nodu **kopíruju** první hodnotu nahoru, půlení nelistového uzlu - vzít prostřední hodnotu tak aby (pokud sudý problém, nekopíruju ale posouvám hodnotu výš, tedy sudý-1 = lichý počet hodnot co rozdělit do 2 uzlů) bylo víc hodnot vlevo, ty vlevo dát do starého uzlu, v pravo do nového, tu prostřední dát o úroveň výš (tedy **neduplikovat**)

mazání - pokud mám málo hodnot, půjčím si vlevo z uzlu/ v pravo pokud mohu (od sourozenců) - vyžaduje změnu klíče v rodiči, pokud nemohu, tak spojuji uzly (je zaručeno že můžu, sourozenci mají taky málo klíčů, protože si nemohu půjčit) a řeším problém rekurzivně

B+Strom souborová organizace - soubor je vlastně B+stromem (podobně jako z indexů jsme udělali b+stromy, index je taky soubor), v listech ukládáme záznamy místo pointerů akorát do uzlu se vleze mnohem méně hodnot

Aktualizace indexů po modifikaci

problémem je že aktualizace dat vynutí změny na všech sekundárních indexech, musí být opravovány náročné štěpení mnoha stromů, lepší je v sekundárních indexech mít odkazy na primární ne do souboru, tedy jedna úroveň procházení navíc ale pak stačí opravit jen primární index, nutnost přidat record-id pokud primární index není unikátní

Indexace řetězcem

problém délky, lepší dělit na základě velikosti klíče a ne počtu klíčů v uzlu

hodnoty ve vnitřních uzlech stačí prefixy (prase, praděd -> stačí v rodiči klíč prad)

Vkládání mnoha hodnot naráz (bulk loading)

po jednom pomalé, lepší buď seřadit a vkládat setřízená data po jednom, nebo setřízená data rovnou nahrát do stromu - pokud tvořím úplně nový (tedy do listů a pak vrstvu po vrstvě dogenerovat vnitřní uzly)

B stromy - jako B+ akorát nemají duplicitní data ve vnitřních uzlech, tedy hodnoty jsou kdekoliv ve stromě, menší velikost a někdy najde klíč dřív - nemusí jít až do listu, ale nemá zřetězené listy (nelze sekvenčně procházet), typicky hodnoty větší než samotné klíče (viz řetězce)- míň místa ve vnitřních uzlech = větší hloubka, ve vnitřním uzlu musí být u klíče dva pointery (jeden na potomka a jeden adresa na záznam s hodnotou toho klíče)

Vyhledávání podle víceatributových klíčů

probíhá lexikograficky, tedy pokud se rovná první porovnává se podle dalšího

STATICKÉ HASHOVÁNÍ - organizace souboru

získání adresy přihrádky (bucket - kolekce více záznamů) se záznamem (typicky bucket size = disk block size) pomocí hashovací funkce (zobrazuje doménu argumentů na menší doménu výsledků -> není prostá, více záznamů padne na stejnou adresu)

Ideální hashovací funkce je : **uniformní** - každý vstup mapován na výstup tak že distribuce je rovnoměrná nad **všemi možnými hodnotami** (tedy modulo je uniformní, všechna čísla rovnoměrně rozdělí)

Ideální hashovací funkce je : **náhodná** - přiřazení probíhá nezávisle na množině hodnot ze kterých vybírám a na tom jak jsou přihrádky již zaplněny

- nejčastěji se dělá modulo prvočíslem, + nějak se upravuje vstupní klíč na číslo (řetězec - součet bitů / násobení bitů * 2^i kde i je pozice bitu), či transformuje číslo aby rozbil podobnost sousedních hodnot (násobení vhodným polynomem atd..)

Uzavřené hashování

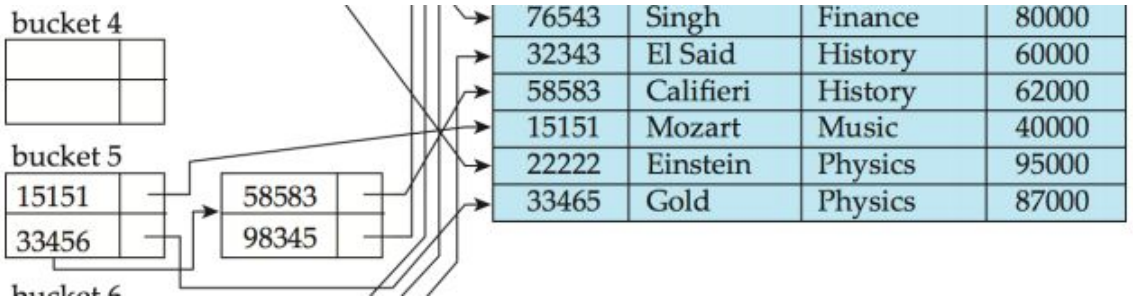
- hledaná hodnota vždy leží na adrese výsledku nezávisle na počtu hodnot které se tam mapují, nutnost na té adrese dělat zřetězené seznamy bucketů (**overflow chaining**) aby se tam vše vlezlo, ty se prochází sekvenčně

Otevřené hashování

- hledaná hodnota může být na vypočítané adrese, pokud není vypočítá se další hodnota a tak dál dokud se nenarazí na prázdné pole / na první hodnotu co se objevila (vrácení na začátek)
- problematické je že pokud tam ta hodnota není trvá dlouho než se na to přijde
- např. hashovací funkce f(h,i) = ((h+i) * 3^2 + (h+i) * 7 + i) mod 13 (násobení klíče polynomem)
    - první krok i=0 klíče h=3, f(3,0) = 9 druhý krok f(3, 1) = 0 třetí f(3,2) = 4
- vkládání - počítej tolik kroků až narazíš na prázdnou adresu
- mazání - musím na pole **poznačit prázdné ALE pokračuj v hledání** (když vkládal a nenašel tam místo, počítal dál, taky zde nesmím zastavit)

Hashové indexy

lze jak struktura souboru, tak pro tvorbu indexů, tedy mám množinu přihrádek do kterých rozděluji adresy podle hashovací funkce, pokud se nevlezou - jdou do overflow



tyto jsou vždy “sekundární - i když se použijí jako primární - a tedy soubor je podle hashování uspořádán”, protože hashování nemá žádné uspořádání, a tedy různé klíče ukazují různě do paměti, nelze mít řídký index - tedy mají vlastnosti sekundárních indexů nezávisle na použití

Nevýhody hashování

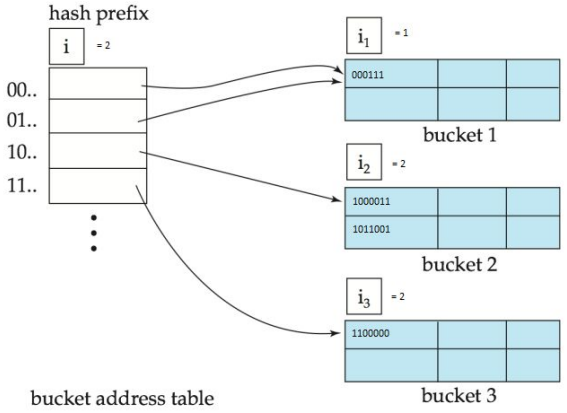
nutnost mít přetokové oblasti pokud mnoho záznamů / mnoho míst nevyužitých (funkce mod 251 - při 20 záznamech 231 prázdných) - řešení dynamický počet “buckets”, horší pro ranged queries

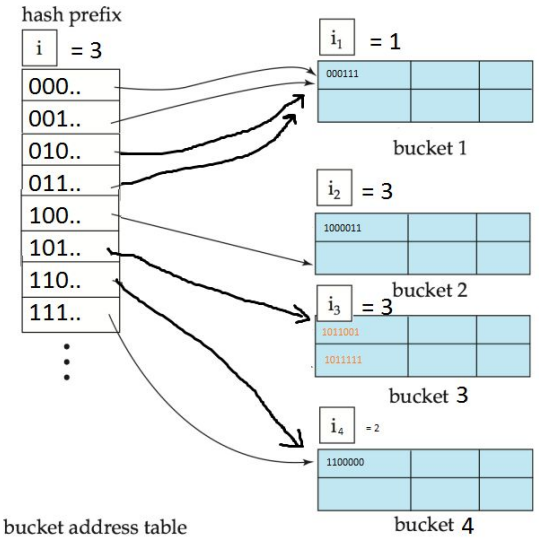
Výhody - rychlejší (ALE ne vždy), neklesá výkon s velikostí souboru, lepší pro exact queries

Dynamické hashování

Rozšiřitelné hashování (extendable hashing)

- dělím podle nejvyšších bitů hodnot, i = max (i1, i2, i3..) určuje velikost adresové tabulky jako 2^i i1, i2⋯ lokální velikosti bucketů, pokud x-tý přeteče a platí i < ix, zvedne se i na hodnotu ix a zdvojnásobí počet adres
- zde i1 = 1, tedy bere prefix délky 1 , tedy adresy 0*
- i2 = i3 = i = 2
- vlož klíč 1011111 ⟶ vezmu i hodnot klíče - 10, jdu do bucket 2, přeplněn, zvednu i2 = 3, i = 3
- nově i = 3, adresy 000, 001, 010, 011 ukazují na bucket 1, 100 na bucket 2, 101 na bucket 3 s naší hodnotou + pokud bylo něco v bucket 2 tvaru 101... , 110, 111 na bucket 4





někdy se split nemusí vykonat (pokud i dosáhne nějaké hranice, pak se přidává do přetokových oblastí = nevýhoda že i roste exponenciálně, počká se až se zaplní pár přetokových oblastní, nereorganizovat kvůli jednomu záznamu - co kdybych ho zase za chvíli smazal), nevýhoda nutnost semtam přeorganizovat při zvyšování i (drahá operace), adresová tabulka může přerůst velikost paměti (řešení použití B+ stromů místo lineární tabulky)

při mazání lze znovu spojovat oblasti, symetricky jako při tvoření

celkově hashování nevhodné na range dotazy, taky se může dít to že některá data se chovají pitomě a všechno padá na stejné adresy - narozdíl B+stromy jsou vždy "stejně výkonné", - spíše se používají setřízené indexy (implicitní volba)

Lineární hashování

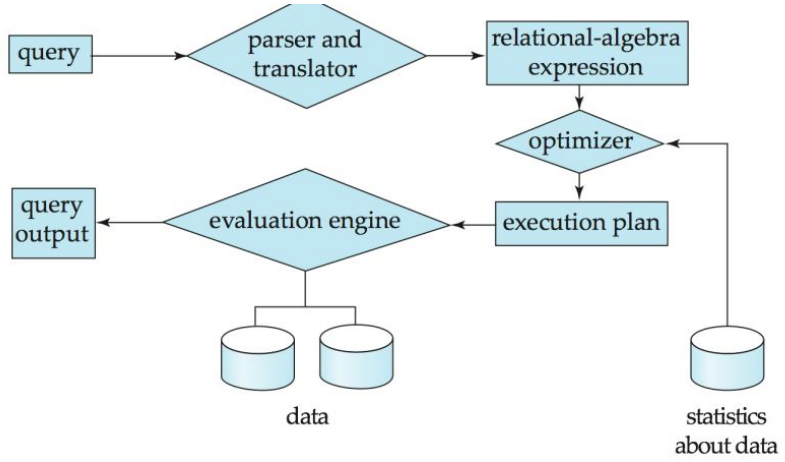
buduje od konce adres(poslední bity), drží si ukazatel na příště štěpený blok, při přetoku nezávisle na tom který přetekl štěpí ten blok, na kterém je právě ukazatel a ten posune o jedno dál

SQL: CREATE INDEX name ON table(a1, a2⋯)

CREATE UNIQUE INDEX name ON table(a1, a2⋯) - atributy a1,a2⋯ jsou superklíč, není nutné lze nahradit ve schéma podmínkou UNIQUE → CREATE TABLE ⋯ ... login VARCHAR(50) UNIQUE, ...

DROP INDEX name

Query processing - vyhodnocování dotazů



1) Parsování SQL do relační algebry, kontrola syntaxe a relací (tedy že sedí názvy)
2) Optimalizace dotazu pomocí metadat (počet tabulek, počet záznamů..) - přeskládání operací..

3) Vyhodnocení podle navrženého plánu z optimalizace a vrácení výsledku

Evaluační plán - způsob získání výsledku - prohozením operací (např SELECT před součinem, sníží počet takto vytvořených hodnot), použitím indexů - snaha o urychlení

Cena dotazu je ovlivněna dobou přístupu na disk, procesorem, popř. sítí (SAN/NAS), potřebou číst / zapisovat(dražší, zápis se zpětně čte aby se zkontroloval) bloky (a zápis se může dít i při čtení - chci načíst do bufferu blok ale musím udělat místo - někdy i vyhodit něco co bylo modifikováno - zapsat na disk), snaha používat větší buffery - ovlivněno daným systémem, často známé až za běhu, vždy uvažujeme nejhorší možný případ

Skenování souborů **A - algoritmy asi není nutné se učit ty vzorečky**

TB - čas na přenos bloku TS - doba přístupu BR - počet bloků s vyhovujícími záznamy

A1 - linear search - použitelný vždy

br * TB + TS	br=počet bloků obsahující záznamy
br/2 * TB + TS	pokud vyhledáváme klíčem - lze zastavit po nalezení (je jen jednou) průměrně tedy vždy projdu průměrně polovinu záznamů

A2 - primární index, hledání podle klíče - nalezni záznam podle rovnosti na klíči

(hi + 1) * (TB + TS)

A3 - primární index, hledání podle neklíčového atributu (více záznamů)

hi * (TB + TS) + TS + TB * BR	první část jako předtím - nalezení prvního záznamu
	druhá část - přenos dalších vyhovujících záznamů, jsou u sebe

A4 - sekundární index, podle neklíčového atributu

(hi + 1) * (TB + TS)	pokud vyhledávám podle atributů které jsou kandidátským klíčem
(hi + n) * (TB + TS)	pokud je to atribut k jehož nalezení je nutnost projít vše, drahé
	n - počet vyhovujících hodnot dotazu

A5 - primární index hledání podle porovnání (hledám klíč v indexu tak že je menší/větší..)

pokud je větší, jdu na záznam v indexu a pak od tohoto záznamu v souboru sekvenčně až do konce relace

jinak jdu od začátku relace bez použití indexu, zastavím se až narazím na první větší hodnotu

A6 - sekundární index, hledání podle porovnávání

pokud větší, skenuj index od první vyhovující hodnoty (ne relaci, není podle něj uspořádaná)

jinak skenuj index od začátku dokud nenarazíš na první větší

IO operace pro každý index - možná lepší lineární procházení souboru

A7 - konjunkce podmínek nad jedním indexem ...WHERE kod!=1 AND kod < 80

vyber jednu podmínku a k ní algoritmus (A1-A7) tak aby výsledek byl co nejmenší, na zbytku vytřiď podle ostatních podmínek

A8 - konjunkce podmínek nad složeným indexem (více klíčů v indexu)

použij více-klíčový index pokud existuje

A9 - konjunkce podmínek průnikem identifikátorů (asi superklíčů) (více "superklíčů" s vlastním indexem v podmínce)

pro každou podmínku použij její index a vyber průnik získaných pointerů

pokud nemá nějaký z klíčů vlastní index, proveď kontrolu až v paměti PC

A10 - disjunkce podmínek sjednocením identifikátorů

pro každou podmínku použij její index a vyber sjednocení získaných pointerů

dělej pouze pokud VŠECHNY podmínky mají svůj index, jinak projdi lineárně

NEGACE - ⋯ WHERE attribute != value

použij lineární sken souboru, pokud jen málo výsledků vyhovujících podmínce, a atribut má svůj index, použij index k získání dat ze souboru

Třídění

pokud se vlezou data do paměti, stačí použít quick sort / merge sort či další⋯

jinak je potřeba použít **external sort-merge** (externí merge sort)

External merge sort M - paměť k dispozici, počet stránek

→ Dokud je něco nezpracovaného v relaci Načti M bloků relace do paměti

Setřiď je v paměti

Ulož do kontejneru Ri, i++

→ Dokud jsou data, předpoklad M < i (počet Ri)Přečti x bloků z každého Ri do stránky Mi (M < i)

Nejmenší ze všech Mi zapiš do output bufferu

Pokud je output buffer plnej, zapiš do souboru

Smaž tento blok z Mi, pokud Mi prázdný načti z Ri

Pokud mám více Ri než stránek dostupné paměti, vyber M prvních Ri a spoj je výše uvedeným postupem, -> sníží počet Ri až #Ri < M

M = 4; #Ri = 15; → Vezmi R1...R4, spoj setřízeně do R21, R5...R8 doR22 ⋯ vznikne R21-R24

M = 4; #R2i = 4 → lze spojit do jednoho

Asi netřeba vědět...

Počet přesunutých bloků je nejhůře BR * (1 + 2 * ceil(log_M-1(BR / M)))

Počet seeků je nejhůře 2 * ceil(BR/M) + ceil(BR/buffer_size) * (-1 + 2 * ceil(log_M-1(BR / M)))

Joiny

B1 počet bloků relace table1 B2 počet bloků relace table2

n1 počet řádků table1 n2 počet řádků table2

TB - čas na přenos bloku TS - doba přístupu

Nested loop join drahý, hloupej každý s každým ale použitelný vždy

for each row1 in table1 do ← outer relation / vnější relace

for each row2 in table2 do ← inner relation / vnitřní relace

if (row1, row2 splňují podmínky joinu) přidej nový řádek z row1, row2 do výsledku

cena: n1 * B2 + B1 přenosů, n1 + B1 seeků nr - num of rows

Block nested loop join levnější, čteme vnitřní blok X krát pro každý vnější (ne každý řádek)

for each blok1 in table1 do ← outer block / vnější blok, zde se čte IO

for each blok2 in table2 do ← inner block / vnitřní blok, zde se čte IO

for each row1 in blok1 do

for each row2 in blok2 do

if (row1, row2 splňují podmínky joinu)

přidej nový řádek z row1, row2 do výsledku

cena: B1 * B2 + B1 přenosů, 2 * B1 seeků

možnost jít odpředu dozadu (zigzag - poslední blok je již v bufferu, jdi od konce ať ho můžeš použít znova, první tam už asi nebude) → šetří IO

lze ukončit inner loop pokaždé když najde hodnotu a join se děje na klíči (jinou už stejně nenajde)

drž outer block v paměti(o velikosti M) v M-2 blocích, 1 blok si vyhraď na vnitřní relaci a 1 na output buffer - cena B1/(M-2) * B2 + B1 → tedy ušetřím M-2 IO operací

Index nested loop join použitelný pokud natural/equi(=USING) join a máme index pro vnitřní relaci

pro každý řádek table1 vyhledej v indexu table2 vyhovující prvky

pokud mám oba indexy, zvol jako vnější tu relaci s méně prvky

cena: B1 * (TB + TS) + n1 * c c je cena vyhledání řádků table2 vyhovujících řádku table1 pomocí indexu

Merge join použitelný jen pro equi/natural join

setřiď sort-merge obě relace na atributech podle kterých se dělá join

lineárně (jen v případě že mám duplicitní hodnoty v první relaci, pak se musím v druhé vracet na začátek seznamu těch odpovídajících hodnot) pro každý prvek první relace přidej všechny odpovídající řádky druhé relace

Hybridní merge join - join setřízené relace a relace která má B+ strom na sekundárním indexu

Spoj setřízenou relaci s každým listem indexu relace druhé

Setřiď tyto podle adres sekundárního indexu

Lineárně projdi adresy v pořadí a nahraď je hodnotami - rychlejší než random vyhledávání adres

Hash Join jen pro equi/natural join

pokud hashovací funkce zobrazuje join atributy do 1...N hodnot, pak v každé relaci každý řádek dáme do chlívečku číslo i, kde i je hash(join attrs) - tedy stejné atributy nutně spadnou k sobě (+ i jiné) ale je jich tam podstatně méně, máme chlívečky zvlášť pro obě relace, každý chlíveček i jedné relace musí být porovnán pouze s i-tým relace druhé

pro každý chlíveček číslo i table1 nahraj do paměti a postav hash index nad atributem s joinem s použitím jiné hashovací funkce, pak ber řádky druhé tabulky symetrického chlívečku a jeden po jednom je hledej v indexu

relace1 se nazývá **build input** a druhá **probe input** (sonda - dívá se do té první)

N je vybráno jako BR / M * f (fudge factor typicky 1.2)

Rekurzivní dělení (recursive partitioning) - potřeba pokud máme v jedné relaci více chlívečků než stránek paměti M, sniž n na počet stránek paměti (N = N mod M), použij M-1 bloků na uložení build input - tyto vnitřní chlívečky dále rozděl podle jiné hashovací funkce → s druhou relací jako sondou se musí udělat ta samá posloupnost hashovacích funkcí

Hash table overflow (Přetoky hashovací tabulky)- nemusí se vlézt všechny údaje do chlívečku → buď špatná funkce, a nebo jen pitomý data (moc hodnot stejnýho atributu na kterým dělá join),

Overflow resolution (řešení přetoků) - použiji dvě různé hashovací funkce na obě relace (snížím kolizní doménu, pořád problém stejných atributů- vždy lze použít block nested loop join)

Overflow avoidance (vyhýbání se overflow) - udělej mnoho chlívečků a pak spojuj ty které jdou, zase může failnout na stejných atributech - vždy lze použít block nested loop join

Hybridní hash join - vhodné pokud paměť M >> sqrt(počet bloků)

- **drž první chlíveček jedné relace (build) stále v paměti**, další bloky použij na bufferování zbylých chlívečků této build relace a jeden blok na bufferování celé probe relace

chlívečky R1 R2 R3 R4 R5 S1 S2 S3 S4 S5

v paměti R1, buffry na R2 R3 R4 R5 a jeden pro S

Joiny s komplexní podmínkou - konjunkce

block nested loop nebo vypočítej jednodušší konjunkci pouze jednoho, a nad výsledkem počítej konjunkci zbylých podmínek

- **disjunkce**

vypočítej pro každou podmínku join zvlášť a výsledky spoj

nebo se na to vykašli a použij block nested loop join na všechno zaráz

Odstranění duplicit

setřídit a smazat duplicity (to lze již dělat při samotném třízení) / hashoováním smazat stejné věci v chlívečku

Agregace - třízením nebo hashováním spojit související prvky tříd, aplikuj funkce nad těmito skupinami

- lze si držet opět částečné výsledky s prvky které jsme již našli, - např merge, mám výsledek pro každý chlíveček a při spojení spojím výsledky (max - max(ch1 ch2), min, sum⋯)

Průnik/sjednocení/rozdíl

rozděl na chlívečku hashovací funkcí

jinou hashovací funkcí vytvoř index nad každým chlívečkem relace 1 a každý chlíveček i relace 2 vyhledej v i chlívečku relace 1 → proveď operaci (průnik = přidej pokud našel pro relaci 2 odpovídající hodnotu podle indexu relace 1, sjednocení = přidej do hash indexu relace1 prvky relace2 pokud tam nejsou, pak dej do výsledky prvky celého indexu, rozdíl 2 - 1 = pokud najdeš pro prvek v chlívečku relace 2 odpovídající prvek v indexu chlívečku relace 1, smaž z 2)

Outer join

buď inner join kdy doplníme nakonec ty prvky co chybí

opravením join algoritmů:

merge A LEFT OUTER JOIN B - při lineárním procházení pokud nenajde pro řádek z A žádný match B, pak jej přidej tento řádek doplněný hodnotami NULL do výsledku

hash A LEFT OUTER JOIN B - pokud je A sonda (probe), pak pokud nenajdeš přidej do resultu

pokud je A build input, poznamenávej si které řádky A nebyly přidány do výsledku při porovnávání chlívečků, a nakonec je tam přidej

Vyhodnocování výsledků

Materializace - spočítej mezivýsledek, ulož a pokračuj - spočítá join, uloží, provede WHERE..

- vždy použitelná, ale drahý zápis mezivýsledků: cena je cenou všecj operací a cenou zápisu všech mezivýsledků
- doublebuffering - ukládej mezivýsledek do bufferu a když skončíš, pracuj dál nad druhým bufferem zatímco ten první se ukládá na disk

Pipelines - předávej částečné výsledky hned dál na další zpracování - počítá join a hned to sype podmínce WHERE zatímco počítá další JOIN

- levnější než materializace, ale nemusí být nutně použitelné (hash join, nebo když dělám join konjunkcí více atributů a ještě bude potřeba vyhazovat další prvky)

demand driven / pull model (řízené na žádost, také jako líné vyhodnocování)

- dostane další mezivýsledek když potřebuje, když si o něj řekne (musí si pamatovat co zatím bylo posláno)
- implementace jako iterátory - open() - inicializuje si pointery, next() - pracuj dokud nenajdeš výsledek pak ho vraťa a ulož pointery jako svůj stav

producer driven / push model (řízené na dostupnost, hladové vyhodnocování)

- vždy předej výsledek pokud máš, vyšší operace si musí buffrovat vstup aby stíhaly zpracovat, pokud se zaplní tak předchozí operace musí počkat

Optimalizace Dotazů

Evaluační plán definuje jaký algoritmus je použit pro kterou operaci, a jak jsou operace řízeny (tedy jejich pořadí).

Optimalizace vzhledem k ceně dotazu - Cost-based query optimization

- vygeneruj ekvivalentní dotazy s použitím **pravidel ekvivalence** a získej ekvivalentní plány
- vyber nejlevnější plán podle **předpokládané ceny**

Předpoklad ceny je počítán na základě

- statistických informací o relacích - počet řádků, velikost domén atributů
- cenu algoritmů (matematicky vyjádřená cena / statisticky ověřená cena)
- statistických informací pro mezivýsledky - mám již uloženy ceny jednoduchých operací, dražší vypočítám pomocí nich

Dva výrazy relační algebry jsou **ekvivalentní**, jestliže pro všechna správná naplnění (nezájem pokud jsou špatné) relací oba výsledky vrací stejnou množinu(nehledě na pořadí) ntic.

Dva výrazy SQL jsou ekvivalentní, ⋯ stejnout multimnožinu(tedy i stejný počet duplikátů) ntic.

Pravidlo ekvivalence říká, že lze nahradit jeden výraz jiným tak, že je zachována ekvivalence výrazů.

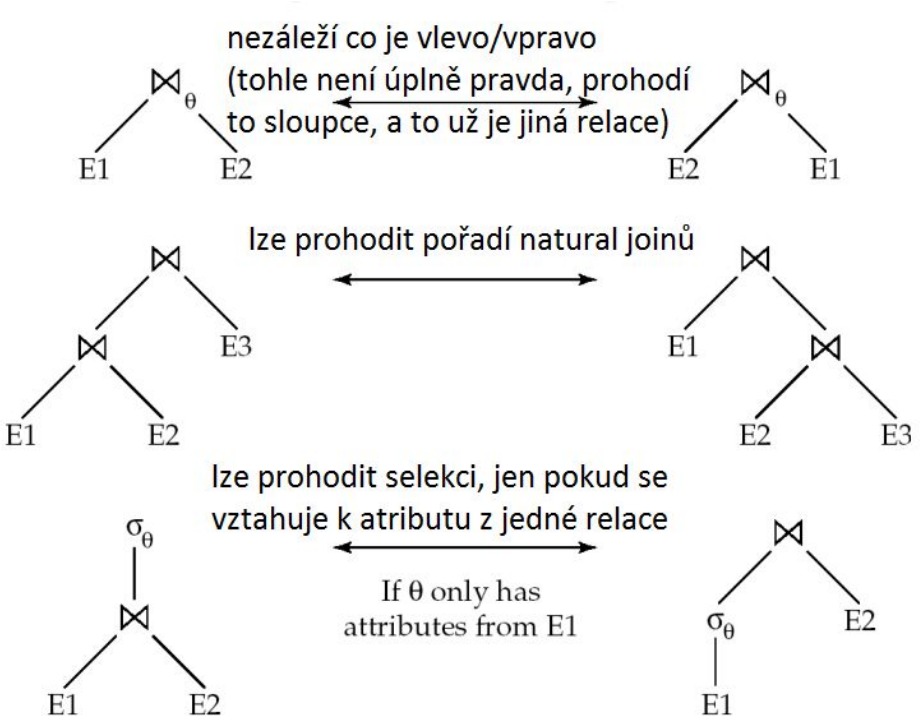
- $\sigma_{P \wedge Q}(E)$ === $\sigma_{P}(\sigma_{Q}(E))$ jak sériové tak paralelní zpracování je stejné

- $\sigma_P(\sigma_Q(E)) \equiv \sigma_Q(\sigma_P(E))$ nezáleží na pořadí podmínky
- $\pi_{L1}(\pi_{L2}(\pi_{L3}(...(E)))) \equiv \pi_{L1}(E)$ pouze poslední projekce určuje které sloupce

  !! neplatí pokud je mezi nimi jiná operace
- $\sigma_P(E \times F) \equiv E \bowtie_P F$ to jsme dělali, když jsme neměli joiny
- $\sigma_P(E \bowtie_Q F) \equiv E \bowtie_{P \wedge Q} F$ pokud má platit i P na obou E,F, pak to dát rovnou do joinu
- $E \bowtie_P F \equiv F \bowtie_P E$ až na pořadí sloupců, informačně ekvivalentní
- $(E \bowtie F) \bowtie G \equiv E \bowtie (F \bowtie G)$ NATURAL: pokud všechny tři mají mít to samé, tak jedno v jakém pořadí
- $(E \bowtie_P F) \bowtie_{Q \wedge R} G \equiv E \bowtie_{P \wedge R} (F \bowtie_Q G)$

asi nejsložitější,tohle lze udělat jen pokud Q má atributy pouze z F a G (např E student | F zápis| G předmět → nezáleží na podmínce P protože když ji hodím ven, pořád tam bude mít všechny atributy co potřebuje - provede se až poslední, ALE pokud chci Q/R provést dřív - tedy studenta vyhodnocovat až posledního, nesmí v nich být podmínka např na učo (protože předmět v sobě nemá učo)

- $\sigma_P(E \bowtie_Q F) \equiv \sigma_P(E) \bowtie_Q F$ lze nejdřív vytřídit, ale jen pokud P obsahuje jen atributy z E (jinak přidat do joinu - o pár pravidel výš)
- $\sigma_{P \wedge Q}(E \bowtie_R F) \equiv \sigma_P(E) \bowtie_R \sigma_Q(F)$ naprosto samé, jen obecnější



1. Vždy je nejlepší provést podmíněný výběr prvků před joinem (pokud to lze)

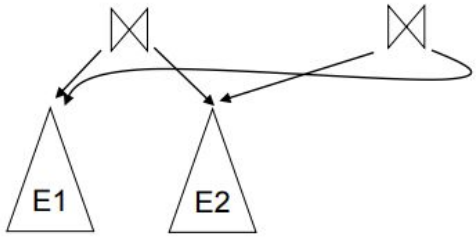
   a. raději vyber auta značky MBW než udělat velký join a pak z toho vybírat BMW
2. Lze použít asociativity joinu k tomu aby šel provést podmíněný výběr prvků co nejdříve (pokud mám (A nat join B) nat join C a pak vybírám podle atributů C, je lepší prvně dělat join na protřízeném C a ne na A,B → A nat join (B nat join C)
3. Dělat projekci co nejdřív pokud to lze - v joinu bude méně sloupců, menší paměťová náročnost
4. (E ⋈ F) ⋈ G vs E ⋈ (F ⋈ G) vždy prvně dělat ten který je menší (první pokud E,F udělají menší množinu) - už třeba jen proto, že při MATERIALIZED se mezivýsledky ukládají

Není vhodné počítat všechny možné execution plány (a strávit "ušetřený čas" výběrem "nejlepšího řešení")

- Optimalizované generování plánů pomocí pravidel transformací (1.-4.)
- Speciální řešení pro speciální (a časté) typy úloh obsahující jen selekci, projekci a joiny

Šetření časem: Počítáme pouze ty plány, které mají naději být "jedny z nejlepších" (není zaručeno že najdu ten nejlepší, ale nebude to ten nejhorší)

Šetření pamětí: Sdílení podvýrazů: např. aplikováním komutativity JOINU se nemění výsledek podvýrazů E1, E2 - detekuj a sdílej, nepočítej 2x



Výběr nejlepšího plánu - nezáleží pouze na součtu jednotlivých cen operací, volba *pomalejšího* merge joinu může být *lepší* než hashjoinu, pokud dál dělám operace které budou *rychlejší nad seřazenými daty* jako důsledku marge joinu

Prakticky

- buď vyber plá na základě cost-based přístupu (tedy nějak získej cenu při znalosti statických dat - počet řádků, velikost domén atributů⋯ + ceny algoritmů)
- nebo používej heuristiky (metody jak vybrat tak že často vyjdou dobře, semtam se netrefím ale nemusím draze počítat nejlepší řešení a pálit čas na něčem co mě nezajímá)

Dynamické programování - optimalizace cast-based (jen pro joiny)

- vypočítej cenu S1 join (S2, S3⋯ Sn) kde S1 je neprázdná množina relací, které joinujeme, rekurzivně řeš problém pro (S2, S3⋯ Sn) → takto vypočítej 2^n - 2 alternativ, (báze rekurze je pokud mám pouze jednu relaci)
- uložíme si nejlepší volbu pořadí pro relace 1,2...n a tu poté budeme používat, ne počítat znovu a znovu

Algoritmus: bestPlan(S): pokud je znám nejlepší plán pro S: return plán, **if** S je pouze jedna relace nastav best plan jako nejlepší způsob přístupu k této relaci, **else** pro každou podmnožinu T z S --> left = bestPlan(S), right = bestplan(S - T), A = algoritmus pro nejlepší spojení (S - T) a T, spočítej cenu jako left.cost + right.cost + cost(A), pokud je výsledek lepší než dosavadní nejlepší, aktualizuj a **end if**, vrať zatím nejlepší výsledek

Ledft-deep join - vyhodnocuji r1 a r2, pak výsledek spojím s r3 a tak až do konce - tedy vždy přidávám do výsledku následující relaci, pokud si to naktreslím do stromu dostanu "hrábě" - cestu doleva dolů

Algoritmus pro left-deep-join: použij předchozí akorát nedělej pro každou podmnožinu z S, ale pro každou relaci z S - tedy generuj vyhodnocení pouze pro různá pořadí jednotlivých relací v dané posloupnosti, varianty:

((r1 join r2) join r3) join r4 NEBO ((r1 join r3) join r4) join r2 ALE NE (r1 join r2) join (r3 join r4)

Cost-based optimalizace pomocí pravidel ekvivalence

fyzická pravidla ekvivalence - taková která převedou logický plán (operace relační algebry a jejich pořadí) na fyzický plán (konkrétní algoritmy a jejich pořadí) výběrem algoritmů pro provedení operací

Obsahuje (už bylo výše)

- optimální reprezentaci která neplýtvá místem - vyhýbá se dělání kopií sub-výrazů které jsou stejné
- efektivní detekce duplicitních výrazů
- ukládání nalezených optimálních řešení pro použití (memorization)
- chytrý výběr který se vyhýbá generování všech plánů

Heuristická optimalizace

cost-based je výpočetně náročný, použij heuristiku k výběru těch plánů, které jsou potenciálně efektivní, typicky je lepší

dělat selekci co nejdřív (snižuje počet řádků relace)

dělat projekci co nejdřív (snižuje počet sloupců relace)

dělat nejdříve selekce a joiny s nejmenším výsledkem

často se používá pouze left-deep join s heuristikou pro zmenšení relací

nebo se používá heuristika pro vnitřní bloky (agregace, další výrazy) a každý blok se pak optimalizuje joiny pomocí cost-based přístupu

"**Optimalizační cenový rozpočet**" (optimization cost budget) - hranice po kterou se počítá optimalizace, je přerušeno pokud překročí - abychom někdy začali počítat i ten sql dotaz

Plan caching - podobné dotazy mohou znovu využít plán, i když jsou mírně odlišné (konstantami)

Statistické informace pro cost-based optimalizaci (kolik hodnot)

nr - počet řádků v tabulce / ntic v relaci r

br - počet bloků, které obsahují řádky relace - počet bloků nutných k uložení relace r

lr - velikost relace r

fr - faktor bloku - kolik ntic se vleze do jednoho bloku (nr / fr = br) v relaci r

V(A, r) - počet různých hodnot v relaci r pro atribut A == $|\Pi_A(r)|$ (== nr pokud A je superklíč)

min(A,r) / max (A,r) - nejmenší/největší hodnota atributu A v relaci r

Running example:

student ⋈ *takes*

Catalog information for join examples:

- $n_{student}$ = 5,000.
- $f_{student}$ = 50, which implies that $b_{student}$ =5000/50 = 100.
- n_{takes} = 10000.
- f_{takes} = 25, which implies that b_{takes} = 10000/25 = 400.
- *V(ID, takes)* = 2500, which implies that on average, each student who has taken a course has taken 4 courses.
  - Attribute *ID* in *takes* is a foreign key referencing *student*.
  - *V*(*ID, student*) = 5000 (*primary key!*)

$\sigma_{A=value}(r)$ **nr / V(A, r)** je počet vyhovujících hodnot selekce, 1 pokud A je klíčem (nr / nr)

$\sigma_{A<=value}(r)$ **0** pokud value < min(A,r)

nr * (value - min(A,r)) / (max(A,r) - min(A,r))

nr / 2 (průměrně polovina) pokud neznáme hodnoty k výpočtu předchozího

$\sigma_{P1 \wedge P2 \wedge \ldots \wedge Pm}(r)$ $n_r * \frac{s_1 * s_2 * \ldots * s_m}{n_r^m}$

s = selektivita - pravděpodobnost, že řádek splňuje predikát P (pokud P splňuje 5 řádků z 20 = 5/20 → tedy počet řádků / nr) pokud např P1: A=value pak S1 = cost(P1)/nr = (nr/V(A,r))/nr = V(A,r)

$\sigma_{P1 \vee P2 \vee \cdots \vee Pm}(r)$ $n_r * \left(1-(1-\frac{s_1}{n_r})*(1-\frac{s_2}{n_r})*\ldots*(1-\frac{s_m}{n_r})\right)$

$\sigma_{\neg P}(r)$ **nr - size(**$\sigma_P(r)$**)**

r x s **nr * ns**, každý řádek má sr + ss velikost

r ⋈ s pokud nemají společné atributy tak **nr * ns** (jako r x s)

pokud společné atributy tvoří superklíč pak **max(nr, ns)**, každý řádek má velikost sr+ss

pokud r obsahuje cizí klíč pro s, pak **nr** - tedy počet řádků r

jinak cca **nr * ns / V(A,s)** nebo **nr * ns / V(A,r)** ,menší hodnota je přesnější

A jsou společné atributy r,s

$\Pi_A(r)$ **V(A, r)**

$A\,\gamma_F(r)$ **V(A, r)**

Množinové operace (nepřesné, horní odhad, stačí)

$\sigma_P(r) \cup \sigma_Q(r) == \sigma_{P \vee Q}(r)$ selekce na stejné relaci je jako disjunkce

r ∪ s **nr + ns**

r ∩ s **min(nr, ns)**

r - s **nr**

Outer join

r LEFT OUTER JOIN s **sizeof(r ⋈ S) + nr**

r FULL OUTER JOIN s **sizeof(r ⋈ S) + nr + ns**

Selekce **různých** hodnot A

$\sigma_{A=value}(r)$ V(A, $\sigma_{A=value}(r)$) **= 1**

$\sigma_{A=v1 \vee A=v2 \vee \cdots \vee A=vn}(r)$ V(A, $\sigma_{A=value}(r)$) = **n** (počet podmínek na A)

$\sigma_{A\ <op>\ value}(r)$ **V(A, r) *s** s = selektivita - pravděpodobnost, že řádek vyhovuje predikátu A <op> value

ostatní případy $\min(V(A,r), n_{\sigma\theta(r)})$ n je asi počet řádků dotazu select (r)

Vnořené příkazy

SQL se chová k vnořeným dotazům jako k funkcím, které vrací množinu výsledků či jeden výsledek a berou jako argumenty proměnné vnější relace - **korelační proměnné**

SELECT * FROM auto a WHERE znacka IN
(SELECT znacka FROM auto b WHERE b.znacka = **a.znacka** AND b.rok > **a.rok**)

SELECT * FROM auto a WHERE znacka IN *function(a.znacka, a.rok)*

Korelační evaluace - pro každý řádek relace je vnořený dotaz vyhodnocen znovu (logicky).

- nemusí být úplně optimální, volat pro každý řádek znovu, zbytečně moc IO, snaha transformovat na join, což dovolí použít techniky optimalizace joinu

Transformace vnořených dotazů na join

*Obecně je problematické zachovat přesně stejný počet duplikací, řešení je **dekorelace**:*

SELECT ⋯ FROM t1 WHERE p1 AND EXISTS(SELECT... FROM t2 WHERE p2)

-> CREATE TABLE nested AS SELECT **DISTINCT**... FROM t2 WHERE (**p2 bez korelačních proměnných**)
-> SELECT ⋯ FROM t1, nested WHERE p1 AND (**korelační proměnné z p2**)

- ale nemusí být vždy takto jednoduché, zvláště pokud je použita agregace, NOT EXSISTS⋯

Zhmotněné VIEWs (materialized VIEWs)

- uložení VIEW jako tabulky pro znovupoužití, výhodou pokud často VIEW chceme, ale potřeba aktualizace jak původních tabulek tak i VIEW současně-dražší modifikace

Top-N dotazy (jak zpracovat ORDER BY attr LIMIT N?)

chceme prvních N hodnot SELECT * FROM **a,b** WHERE **a.id = b.id** ORDER BY **a.id** DESC LIMIT 5

- buď indexed nested loop join s relací **a** jako vnější - tedy vezmu jen vyhovujících 5 řádků A, pokud bych začal s **b,** tak bych musel projít všechno, abych zjistil které odpovídající A je největší
- předpokládej maximální hodnotu **a.id** a přidej podmínku **AND a.id <= maxValue**, pokud dostaneš méně výsledků než uvedeno v LIMIT, zvyš maxValue

Aktualizace hodnot na kterých provádím výběr

vybírám prvky tabulky, které pak měním - pod rukama si ničím výběr

UPDATE produkt SET price=price*1.2 WHERE price < 5000

Přístupy

- Vždy odkládej modifikace: získej množinu dat a pak ji celou modifikuj, není moc úsporné

- Odkládej pokud nutné: aktualizuj data jen pokud nejsou ve WHERE, odkládej pokud jsou

Join minimalizace

nemusí být vždy nutné počítat JOIN, lze vypustit například b2, lze smazat:

SELECT...FROM a,b as b1,b as b2 WHERE ⋯ AND **b1.attr < 5** AND **b2.attr <= 8**

Transakce

je posloupnost instrukcí, které pracují (čtení/zápis) s daty - požadavek na to, aby proběhly ATOMICKY:

1) buď všechny a nebo vůbec žádná
2) nebylo možné mezitím co běží modifikovat používaná data (např. změnit hodnotu prvku poté co jsme ho četli)

snaha chránit se před chybami (poruchy disků / pády systému) a paralelním přístupem (vlákna)

není přeloženo "commit" ale ani to moc nemá smysl, je to jako překládat v pseudokódu "return"..

Nutné požadavky na transakce - ACID

ATOMICITY: Nutnost atomičnosti transakcí

Transakce se chová jako jedna operace, nedělitelná - částečné výpočty se "neprojevují v databázi".

CONSITENCY: Konzistence dat před i po transakci

Data musí být v konzistentním stavu, transakce *v izolaci* tuto konzistenci může porušit během svého běhu, ale na konci musí být data opět konzistentní.

1) **Explicitní omezení** na integritu dat (cizí a primární klíče databáze, specifikováno v návrhu)
2) **Implicitní omezení** na integritu dat (z logiky systému, není nikde "specifikováno" např. při převodu peněz z jednoho účtu do druhého musí platit že součet zůstatku obou účtů je před i po stále stejný = tedy peníze se nikam neztratily)

ISOLATION Požadavek izolace transakcí

Pokud je databáze v nekonzistentním stavu kvůli běhu transakce A a začne transakce B, pak B neumí zaručit navrácení do konzistence, protože začala v nekonzistentním stavu. (A čte, B čte, A uloží, B uloží → A jakoby neproběhla i když systém řekne že ano)

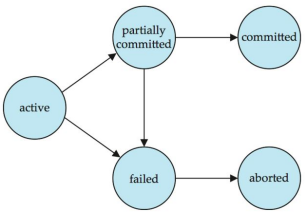
Lze zaručit sériovým během transakcí (ale chceme umět paralelně → vyšší využití procesoru, lepší odezva: kratší operace nemusí čekat na dokončení dlouhých) a tedy:

Transakce mohou probíhat paralelně, ale každá transakce neví o ostatních, její stav/mezivýsledky jsou skryté před ostatními (lepší by bylo říci: nerozlišitelné) → transakce A se jeví při vykonávání B tak, že buď ještě nezačala, a nebo již skončila.

DURABILITY: Trvanlivosti transakcí

Pokud je operace dokončena, musí být uložena i přes selhání disku/software.

Stavy transakcí



Active – transakce je vykonávána
Partially commited – poslední instrukce je vykonána, výsledek zatím neuložen
Failed – transakce z nějakého důvodu selhala (chyby / podmíny)
Aborted – změny doteď provedeny byly vráceny do stavu před začátkem transakce, lze 1) znovu transakci spustit / 2) ukončit
Commited – transakce je vykonána a uložena do databáze

Schéma kontroly souběžnosti/paralelismu (Concurrency control scheme)
kontrola interakce transakcí běžících souběžně tak, že je zachována izolace

Rozvrh (schedule) - posloupnost instrukcí které určují pořadí ve kterém jsou instrukce transakcí vykonány (tedy transakce se mohou navzájem prolínat)

rozvrh množiny transakcí musí obsahovat všechny instrukce těchto transakcí (nic nepřeskočí)

musí dodržet pořadí instrukcí jedné transakce (každá je provedena ve správném pořadí)

na konci každé úspěšné/neúspěšné transakce musí být commit/abort instrukce (předpoklad, že to má v sobě už ta sama instrukce a nemusí to řešit rozvrh)

Serializovatelný rozvrh (serializable chedule) je posloupnost instrukcí několika transakcí, jejíž výsledek je ekvivalentní tomu, kdyby všechny transakce běžely sériově.

Konfliktnost instrukcí (uvažujeme jen read() a write())

2 instrukce jsou **konfliktní**, pokud existuje jeden objekt, nad kterým alespoň jedna instrukce zavolala write() (read, read není konfliktní, čtou to samé ALE read write už bez daného pořadí vedou k chybě)

Konfliktní serializovatelnost

Rozvrhy R a S jsou **konfliktně ekvivalentní**, pokud R umíme převést na S nekonfliktními instrukcemi. (tedy R a S mají stejnou "konfliktnost")
A tedy rozvrh S je **konfliktně serializovatelný**, pokud je konfliktně ekvivalentní se sériovým rozvrhem.

Rozvrh S: je konfliktně serializovatelný se Sériový rozvrh:

T_1	T_2
read (A)	
write (A)	
	read (A)
	write (A)
read (B)	
write (B)	
	read (B)
	write (B)

T_1	T_2
read (A)	
write (A)	
read (B)	
write (B)	
	read (A)
	write (A)
	read (B)
	write (B)

Pohledová (view) serializovatelnost

Rozvrhy R a S jsou **pohledově (view) ekvivalentní**, pokud pro všechny datové bloky Q

1) Pokud transakce T_i čte počáteční hodnotu Q v R, pak ji musí číst jako počáteční také v S
   - tedy, obě musí číst počáteční hodnotu Q před tím, než byla změněna
2) Pokud transakce T_i čte hodnotu Q modifikovanou transakcí T_k v rozvrhu R, pak v S musí T_i číst hodnotu Q jako výsledek té samé modifikace způsobené transakcí T_k
3) Pokud jakákoliv transakce provede finální zápis do objektu Q, musí být jako poslední v R i S.

A tedy rozvrh S je **pohledově serializovatelný**, pokud je pohledově ekvivalentní se sériovým rozvrhem.

Každá conflict-serializable je také view serializable (ale ne naopak). View-serializable která není conflict-serializable má tzv. **blind writes** (zápis naslepo).

Rozvrh S: je pohledově serializovatelný se Sériový rozvrh:

T_{27}	T_{28}	T_{29}
read (Q)		
	write (Q)	
write (Q)		
		write (Q)

T_{27}	T_{28}	T_{29}
read (Q)		
write (Q)		
	write (Q)	
		write (Q)

S není konfliktně serializovatelný, neboť pro serializaci musíme vzájemně prohodit konfliktní instrukce (write a write), na čemž ale "nezáleží", protože hodnota uložená v Q je vždy závislá na T29 - které tam poslepu zapíše, aniž čte co tam bylo

Konfliktnost s libovolnými instrukcemi

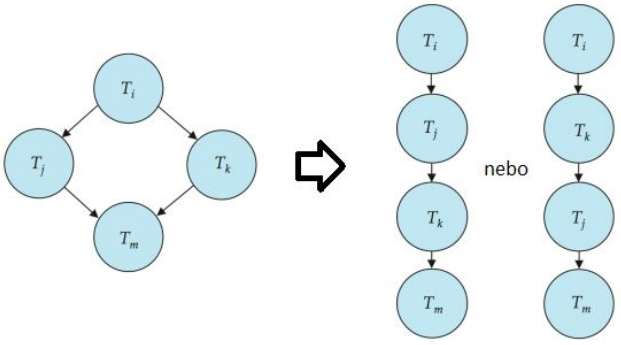
T_1	T_5
read (A)	
$A := A - 50$	
write (A)	
	read (B)
	$B := B - 10$
	write (B)
read (B)	
$B := B + 50$	
write (B)	
	read (A)
	$A := A + 10$
	write (A)

T_1	T_5
read (A)	
$A := A - 50$	
write (A)	
read (B)	
$B := B + 50$	
write (B)	
	read (B)
	$B := B - 10$
	write (B)
	read (A)
	$A := A + 10$
	write (A)

Tyto nejsou ani pohledově ani konfliktně ekvivalentní, i když mají stejný výsledek a nejsou zde konflikty read/write (proč? zkus si místo sčítání dát násobení a odčítání dělení)

Graf předností/priorit (precedence graph)
je orientovaný (direct) graf kde
vrcholy jsou transakce
hrana z T1 do T2 vede, pokud dvě transakce jsou kolizní a T1 má přístup ke koliznímu objektu jako první
popis hrany je objekt, na kterém dochází ke kolizi

Rozvrh je **konfliktně serializovatelný** pouze pokud je hraf předností acyklický.
Získat konfliktně ekvivalentní sériový rozvrh lze pomocí **topologického uspořádání** acyklického grafu předností.



Pohledová serializovatelnost rozvrhu nemůže být jednoduše řešena pomocí grafu předností (NP-úplný problém), existují ovšem algoritmy které umí zkontrolovat několik **dostačujících podmínek**.

Vrácení transakcí

Obnovitelný (recoverable) rozvrh

Pokud transakce T2 čte objekt Q, který byl předtím modifikován transakcí T1, pak commit transakce T1 musí být před commit-em transakce T2 (pokud T2 skončí dříve, a T1 provede abort, pak se musí vrátit změna zápisu na Q, ale vzhledem ke změněné hodnotě již proběhl commit T2, ten vrátit nelze).

Cascading rollback (kaskádový návrat)
Chyba jedné transakce T způsobí abort na dalších transakcích (které mohou způsobit abort na dalších⋯) jen proto, že četly hodnotu prvku Q modifikovanou transakcí T.

Cascadeless schedules (bezkaskádové rozvrhy) **← a to chceme, i když to nemusí vždy jít**
Pokud transakce T2 čte objekt Q, který byl předtím modifikován transakcí T1, pak commit transakce T1 musí být před čtením Q transakcí T2.

Řízení paralelního přístupu

Databáze musí zajistit že vykonávané rozvrhy jsou
→ konfliktně nebo pohledově serializovatelné
→ obnovitelné a pokud možno bezkaskádové
tradeoff mezi náročností výpočtu a počtem transakcí, kterým umožní běžet paralelně
Někdy se povolují pouze konfliktně serializovatelné rozvrhy, jindy i pohledové, ale vždy musí být obnovitelné. Protokoly obvykle nekontrolují přednostní graf, pouze se vyhýbají neserializovatelným rozvrhům (heuristiky).

Někdy povolujeme nepatrné chyby v konzistenci (statistická data databáze pro optimalizaci - stačí přibližně) - takové transakce nemusí být serializovatelné s ostatními (pokud nemodifikují data).

Úrovně konzistence
1) Serializovatelné - defaultní
2) Opakované čtení (repeatable read) - čte pouze commited záznamy - více čtení nevadí, výsledek je stejný ALE nemusí být serializovatelné (nemusí najít některé údaje)
3) Commited čtení - pouze commited záznamy jsou čtené, ale nemusí vracet stejné výsledky (někdo mezitím udělal další commit).
4) Uncommited čtení - číst lze cokoliv.

SQL

COMMIT WORK - uloží výsledek provedené transakce a začne novou
ROLLBACK WORK - zahodí transakci
většinou je implicitní commit na konci každé správně provedené transakce (lze vypnout).