SAMENVATTING LES 10

donderdag 14 mei 2020 13:56

Indexstructuren 1

Wat en waarvoor een index?

Wat is database-index?

Een database-index is een structuur die tot doel heeft selecties en selectieve bewerkingen op een database te versnellen
Een index reduceert het aantal vergelijkingen dat nodig is om een of meerdere database-records te vinden

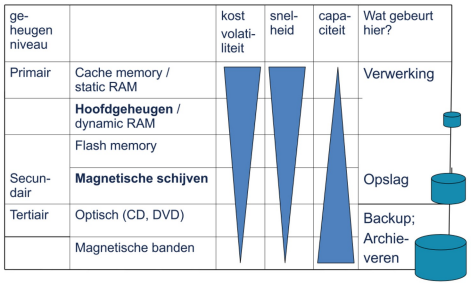
=> Zo wordt voorkomen dat een zogeheten **full table scan** moet worden gedaan, waarbij alle records in de tabel sequentieel moeten worden doorlopen
=> wijzen

Waarvoor dient een index?

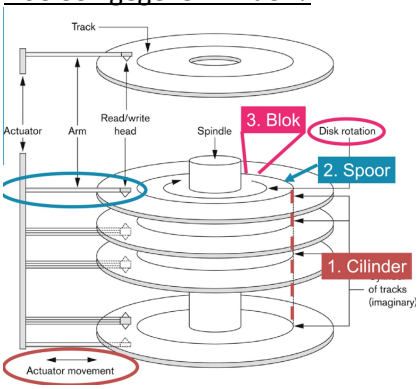
- Niet alles moeten lezen
- Zoeken naar verschillende criteria
- Vinden
- Verstoppen (censureren op internet)

Media, bestandsorganisaties en indexen

Geheugenhiërarchieën



Hoe een gegeven vinden?



Fundamenten van geheugen voor een gegevensbank

- Voor een gegevensbank willen we snelheid + ruimtegebruik optimaliseren
- Gegevens zijn fysisch gestructureerd in bestanden
- Deze bestanden zijn fysisch opgeslagen in een medium
    - Harde schijf (nog steeds heel populair voor GBs), SSD, USB, flash stick, hoofdgeheugen
    - Ook: banden (archief)
- Medium en gekozen structuur hebben gevolgen voor de snelheid van verschillende toegangsoperaties, en voor ruimtegebruik

=> we moeten structuren zo kiezen dat ze de meest belangrijke operaties van onze GB-toepassing goed ondersteunen

- We gaan uit van een "row-oriented database"

- Lezen/schrijven: transport schijf ⇔ hoofdgeheugen
- **Je leest/schrijft nooit een record, maar een hele blok!**
    - Traag: lezen/schrijven van/naar schijf
    - Snel: zoeken, sorteren, … binnen een blok in hoofdgeheugen
- Blok(NL) = block of page (EN)
- Bucket(EN, hier ook gebruikt) = 1 blok of cluster van opeenvolgende blokken
- Snelheid wordt geoptimaliseerd als
    - Zo **weinig records mogelijk** opgezocht worden
    - De opgezochte records **zo klein mogelijk** zijn

- Records **naast elkaar in** een/meerdere blok/**blokken** zitten

Zoeken in bestanden

Zoeken in lijsten : basisstrategieën

Datastructuur/bestand	Zoekstrategie	Complexiteit (in-memory, n: aantal elementen)
Ongeordend	Lineair	O(n)
Geordend	Lineair	O(n)
	Binair	$O(\log_2 n)$
Hashed	Lineair	O(n)
	Direct	O(1) (in een "goede" hash)

Indexeren: definitie en soorten

Indexstructuren indexeren velden

- Het is een **index op een bestand**
  = een gegevensstructuur die de toegang op dat bestand via een bepaald veld (of een groep velden) efficiënter maakt
  - D.w.z.: laat efficiënt zoeken naar een bepaalde waarde van dat veld toe
- Index kan opgeslagen zijn:
  - In centraal geheugen (enkel redelijk kleine indexen)
  - In een bestand in het externe geheugen

Soorten indexen

- **Primaire index**: op veld dat
  - De ordening van het bestand bepaald
  - Records zijn uniek geïdentificeerd (d.w.z. elke waarde voor het veld is uniek)
- **Secundaire index**: index op een ander veld dan dat wat de ordening bepaalt

Primaire indexen (als geordende bestanden)

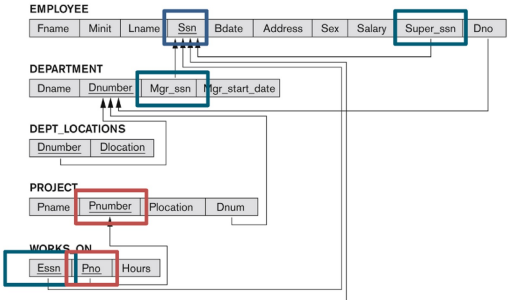
Primaire indexen

- Primaire index:
  - Bestand
    - Met vaste lengte records
    - **Fysisch geordend volgens de sleutelwaarden**
  - Index: bevat 1 record per blok in het gegevensbestand:
    - Sleutel van "ankerrecord" van het blok (= eerste of laatste record in het blok)
    - Adres van het blok
  - Gegeven een sleutelwaarde, kan adres van blok waar overeenkomstig record zit, gevonden worden door zoeken in index i.p.v. gegevensbestand
    - D.i. dankzij de ordening in het bestand

- Eigenschappen en voordelen
  - Index bevat kleinere records dan gegevensbestand
    - Enkel sleutel + adres, geen andere info
  - Index bevat meestal minder records dan gegevensbestand
    - Is een niet-dichte of ijle (nondense, sparse index)

  - Index is kleiner dan gegevensbestand
  - Doorlopen van index gaat sneller dan doorlopen van gegevensbestand
  - Veel minder toegang tot schijf nodig

Op welke velden kan je een primaire index definiëren?



Secundaire indexen (als geordende bestanden)

Secundaire index

- Index op een ander veld dan het veld dat de ordening bepaalt
  - Index zelf is wel geordend volgens dat veld
  - Veld kan al dan niet een sleutel zijn
- Indien dit een secundair-sleutel-veld is:
  - 1 record in index per waarde van het geïndexeerd veld in gegevensbestand
    - Geen ordening → enkel ankerrecords is onvoldoende

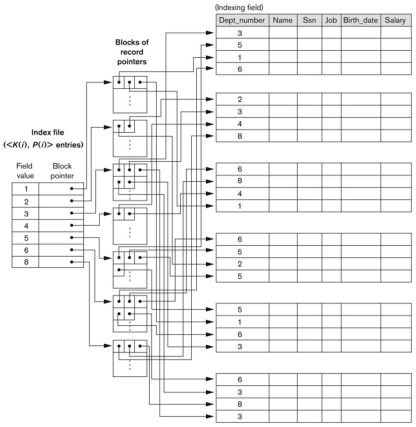
=> dichte index

- Nog steeds kleiner dan gegevensbestand omdat records zelf kleiner zijn (maar minder spectaculair)
- Hoewel index zeer groot kan zijn: toch grote tijdswinst
  - Index is geordend → binair zoeken mogelijk
  - Vs. gegevensbestand: lineair zoeken nodig!
- Eens blok gevonden: enkel nog lineair zoeken binnen blok
  - In intern geheugen → gaat snel
  - Bfr meestal relatief klein

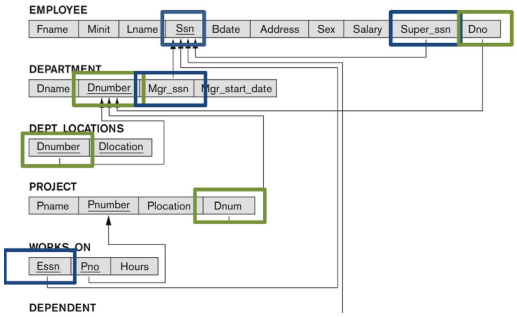
=> verwaarloosbaar vs. inlezen van blokken

Secundaire index op niet-sleutel veld

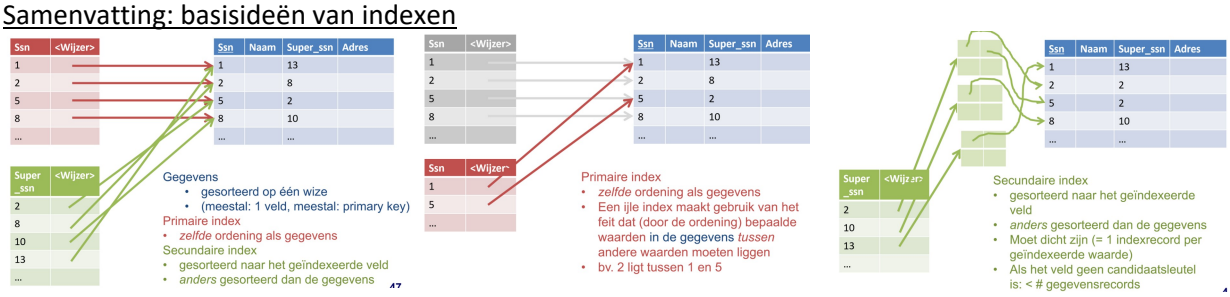
- Dichte index
  - Elke waarde komt even vaak in index voor als in gegevensbestand
- Index met variabele lengte records
  - Per waarde een lijst wijzers naar blokken
- Index met verwijzingen naar blok recordswijzers
  - M.a.w. 1 adres per waarde
  - Adres wijst naar blok (evt. lijst van blokken) met wijzers naar blikken in gegevensbestand
    → 1 indirectie meer



Op welke velden kan je een secundaire index defiiniëren



Samenvatting: basisideën van indexen



Hash-indexen

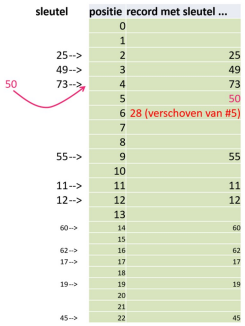
Hashing-bestandsorganisatie

- Methode om uit recordsleutel meteen recordadres in bestand te vinden: hashing
- Principe van hashing:
  - Hash-functie beeldt waarde k (getal, string, …) af op adres f(k)
  - Goede hashfunctie "spreidt" waarden zoveel mogelijk over verschillende adressen
  - Botsing ("collision") : f(k1) = f(k2) met k1 ≠ k2
- Interne hashing
  - Gegevens en hashfunctie zijn in centraal geheugen voorgesteld
- Externe hashing
  - Gegevens (evt. ook hashfunctie zelf) opgeslagen in een bestand

Botsingsafhandeling

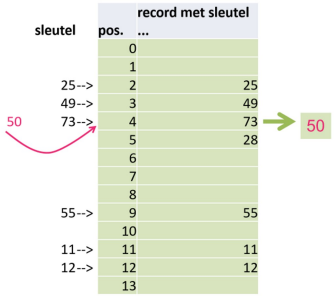
- Botsing:
  - Meerdere records met verschillende sleutel komen terecht in dezelfde cel
- Meestal: #sleutelruimten >> #adresruimten
  - Botsingen zijn dan onvermijdeling
  - Moeten op een andere manier opgevangen worden ("collision resolution")
- Eenvoudige manier van botsingsafhandeling:
  - Tabel van cellen, meerdere records in 1 cel toelaten
  - Celdiepte = max # records in 1 cel (te groot: kan niet!)
- Indien teveel records aan 1 cel toegekend worden: overloop
  - Meerdere technieken van overloopafhandeling

Open adressering



Hashfunctie
h(k) = k mod 23

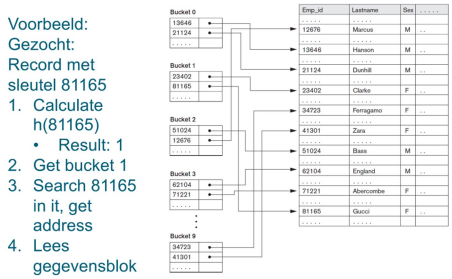
Ketening / gesloten adressering



Hash-indexen

- Secundaire structuur
  - (er bestaat ook hashing als bestansorganisatie, waarbij de hashfunctie de ordening van de gegevens bepaalt)
- Structuur: <sleutel, blok/recordwijzer>

Zoeken in Hash-indexen



Indexen in MySQL

More common usage of "clustered index"

- A **clustered index** is a special type of index that reorders the way records in the table are physically stored. Therefore tables can have only one clustered index. The leaf nodes of a clustered index contain the data pages.

- A **nonclustered** index is a special type of index in which the logical order of the index does not match the physical stored order of the rows on disk. The leaf node of a nonclustered index does not consist of the data pages. Instead, the leaf nodes contain index rows.

Clustered index

The InnoDB term for a **primary key** index. InnoDB table storage is organized based on the values of the primary key columns, to speed up queries and sorts involving the primary key columns. For best performance, choose the primary key columns carefully based on the most performance-critical queries. Because modifying the columns of the clustered index is an expensive operation, choose primary columns that are rarely or never updated.

In the Oracle Database product, this type of table is known as an index-organized table.

Clustered and primary indexes

Every InnoDB table has a special index called the **clustered index** where the data for the rows is stored. Typically, the clustered index is synonymous with the primary key. To get the best performance from queries, inserts, and other database operations, you must understand how InnoDB uses the clustered index to optimize the most common lookup and DML operations for each table.

1. When you define a PRIMARY KEY on your table, InnoDB uses it as the clustered index. Define a primary key for each table that you create. If there is no logical unique and non-null column or set of columns, add a new **auto-increment** column, whose values are filled in automatically.

2. If you do not define a PRIMARY KEY for your table, MySQL locates the first UNIQUE index where all the key columns are NOT NULL and InnoDB uses it as the clustered index.

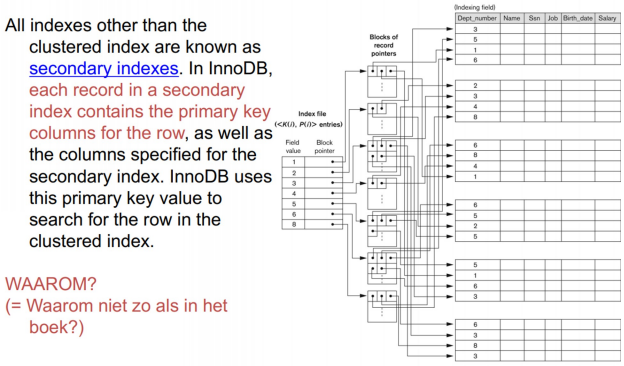
3. If the table has no PRIMARY KEY or suitable UNIQUE index, InnoDB internally generates a hidden clustered index on a synthetic column containing row ID values. The rows are ordered by the ID that InnoDB assigns to the rows in such a table. The row ID is a 6-byte field that increases monotonically as new rows are inserted. Thus, the rows ordered by the row ID are physically in insertion order.

How the clustered index speeds up queries

Accessing a row through the clustered index is fast because the index search leads directly to the page with all the row data. If a table is large, the clustered index architecture often saves a disk I/O operation when compared to storage organizations that store row data using a different page from the index record. (For example, MyISAM uses one file for data rows and another for index records.)

How secondary indexes relate to the clustered index

All indexes other than the clustered index are known as **secondary indexes**. In InnoDB, each record in a secondary index contains the primary key columns for the row, as well as the columns specified for the secondary index. InnoDB uses this primary key value to search for the row in the clustered index. If the primary key is long, the secondary indexes use more space, so it is advantageous to have a short primary key.



Worden NULL waarden geïndexeerd?

- Ja, maar door de meeste DBMS alleen voor UNIQUE kolommen (zie volgende pagina).
- “A search using col_name IS NULL employs indexes if col_name is indexed.”
- “MySQL can perform the same optimization on col_name IS NULL that it can use for col_name = constant_value. For example, MySQL can use indexes and ranges to search for NULL with IS NULL. “
- Met een voorbeeld dat queryverwerking toont (EXPLAIN, zie later in de lessen over queryverwerking):

SAMENVATTING LES 11

donderdag 14 mei 2020 16:46

Indexstructuren 2

Motivatie en samenvatting

Overzicht zoektijd- kan dat beter?

Structurele types indexen (bestandsorganisatie van de index)	Functionele types indexen		
	Primaire index – zoeken naar uniek veld dat de ordening van gegevens bepaald	Secundaire index – zoeken naar andere velden	
Geen index	O(log 2 n)	O(n)	n = # gegevensrecords
lineair	O(log 2 b) (ijle index) O(log 2 n) (dichte index)	O(log 2 n) (moet dicht zijn)	b = # gegevensblokken
hash	O(1)	O(1)	Maar: dit is "best case", verschillende problemen (zie later)

Multiniveau-indexen

Terminologie: "Sleutel"

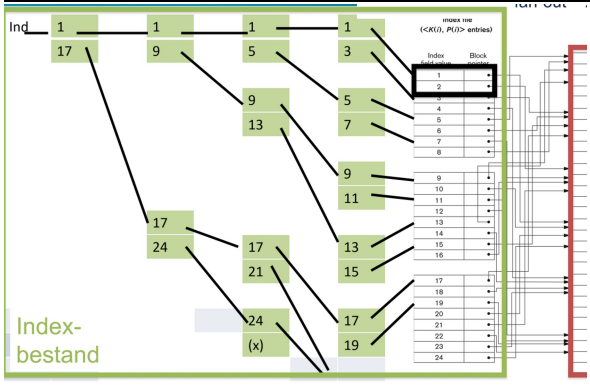
- Relationeel model, SQL:
  - Primaire sleutel
  - Secundaire sleutel
  - Foreign key

- Indexen
  - Sleutel = geïndexeerde waarde
  - Kan uniek zijn in de geïndexeerde tabel (bv. Primaire index)
  - Maar hoeft niet uniek te zijn (bv. Meeste secundaire indexen)

Vereenvoudiging: aanname "sleutel is uniek"

Structurele types indexen (bestandsorganisatie van de index)			
	zoeken naar uniek veld (dat niet noodzakelijk de orde van de geg. bepaald()		
Geen index	O(n)		n = # gegevensrecords
lineair	O(log 2 n) (dichte index)		
hash	O(1)		(best case)

Binair zoeken qua datastructuur: multi-niveau indexen



- Padlengte = x
- Op het diepste niveau zijn alle n waardes → $2^{x+1} \geq n$

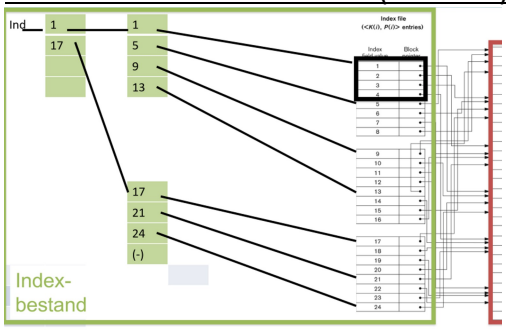
$$x = \lceil \log_2 n \rceil - 1$$

Niveau	0	1	2	3	4
# blokken / knopen	2^0	2^1	2^2	2^3	$2^4 * 2$ = ruimte voor 24 waardes

Winst?

Structurele types indexen (bestandsorganisatie van de index)	zoeken naar uniek veld (dat niet noodzakelijk de orde van de geg. bepaald()		
Geen index	O(n)		n = # gegevensrecords
lineair	O(log 2 n)		
Multi-niveau, fo=2	O(log 2 n)		
hash	O(1)		(best case)

Multi-niveau indexen met fo (=fan-out) > 2



Algemeen: indexen met meerdere niveaus

- Principe:
  - Gewone index op gegevensbestand kan nog steeds groot zijn
    -> opnieuw index bouwen bovenop deze index
    - ◇ Laat toe waarden sneller terug te vinden in deze index
    - ◇ = niveau 2 index
  - Eventueel hierbovenop nog een index, enz.
  - Tot top-index maar 1 blok groot is

- Fan-out (fo) even groot voor alle indexen
  = hoeveel records passen in één blok?
- Winst?

Structurele types indexen (bestandsorganisatie van de index)	zoeken naar uniek veld (dat niet noodzakelijk de orde van de geg. bepaald()		
Geen index	O(n)		n = # gegevensrecords
lineair	$O(\log_2 n)$		
Multi-niveau, fo=2	$O(\log_2 n)$		
Multi-niveau, fo>2	$O(\log_{fo} n)$		
hash	O(1)		(best case)

Problemen van statische indexstructuren

Multi-niveau indexen

- Wat als de gegevenstabel krimpt of groeit?

Hash-indexen

- Weinig geïndexeerde waarden → ruimteverkwisting
- Veel geïndexeerde waarden → zoeken vertraagd
- Herhaling: botsingsafhandeling: ketening, openadtressering

Problemen van hash-indexen met ketening

- Veel geïndexeerde waardes, te veel botsing → zoeken vertraagd
- Andere operaties ook vertraagd

Problemen van hash-indexen met openadressering

- Veel geïndexeerde waardes, te veel botsing → zoeken vertraagd

- Andere opties ook vertraagd

Statische en dynamische structuren

- Problemen met toevoegen / weglaten van records
  - Doordat elk niveau van de indexboom fysisch geordend is
  - Hele boom van indexen moet aangepast worden
  - Oplossing: van tijd tot tijd reorganiseren (bv. Re-hashing)
  - Dit is duur
- Dynamisch indexstructuren groeien + krimpen met hoeveelheid geïndexeerde data
  - Hier: bomen (B-bomen, B+-bomen)
  - Bestaat ook voor hashing

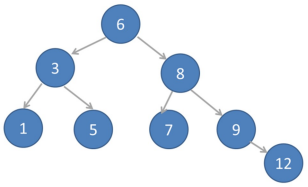
Bomen: introductie

(Zoek)bomen

- We bespreken deze eerst als datastructuur / bestandsstructuur
  = de gegevens zelf zitten in de knopen
- Daarna bespreken we ze als indexstructuren
  = de knopen verwijzen naar de gegevens

Bomen

- Kan een gegevens- of bestandsstructuur zijn
  - Vb. Binaire boom: knoop =
    - Waarde / gegevensitem
    - Wijzer naar linker kinderknoop
    - Wijzer naar rechter kinderknoop
  - Speciale knopen: wortel (hier: 6), bladeren (hier: 1, 5, 7, 12)



- Kan ook een indexstructuur zijn
  - Zoekattribuutwaarde ("sleutelwaare") + wijzer naar gegevens ipv. waarde

Boomstructuren als indexen

- Binaire zoekboom is geordend:
  - 1 waarde in knoop
  - In linkerdeelboom enkel kleinere waarder
  - In rechterdeelboom enkel grotere waarden
- Opzoeken van waarde vraagt tijd evenredig met hoogte h van boom
  - 'gewoonlijk"
    - **$H \approx \log_2 n$**
    - Met n = #waarden in de boom
  - Dus: zoeken is efficiënt
  - **Boom groeit en krimpt naarmate van de gegevens → dynamisch**
- Maar: als je waardes toevoegt die al geordens zijn is de zoektijd O(n)

Evenwichtigheid

- Aanpassen vna boom(toevoegen, weglaten): ook tijdscomplexiteit evenredig met h
  - Gemiddeld dus ook efficiënt
- MAAR: aanname van "evenwichtigheid" van bomen wordt gemaakt!
  - Niet onmogelijk dat h ≈ n i.p.v. $\log_2 n$
  - Vb. bij eenvoudig toevoeg-algoritme dat waarden reeds in volgorde krijgt
- → concept van evenwichitge zoekbomen
  - Toevoegen, weglaten worden zo geïmplementeerd dat evenwicht steeds bewaard blijft

Zoektijden

Structurele types indexen (bestandsorganisatie van de index)	zoeken naar uniek veld (dat niet noodzakelijk de orde van de geg. bepaald()		
Geen index	O(n)		n = # gegevensrecords
lineair	$O(\log_2 n)$		In de praktijk niet relevant
Multi-niveau, fo≥2	$O(\log_{fo} n)$		
(B bomen,) B+ bomen	$O(\log_p n)$		P = orde van de boom, ~ fo
hash	O(1)		(best case)

B-bomen

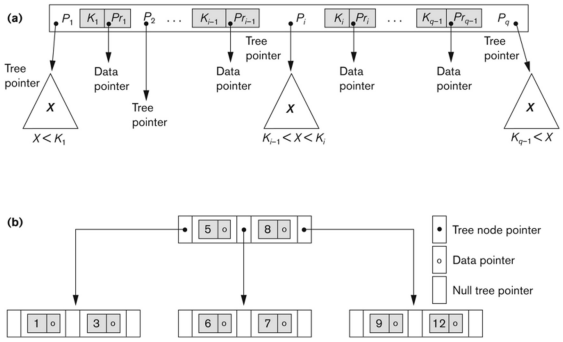
- B-boom van orde p (p>2) is zoekboom waarvoor:
  - Elke inwendige knoop heeft hoogstens p kinderen
  - De wortel heeft minstens 2 kinderen, elke andere knoop minstens ⌈ p/2 ⌉
  - Alle bladeren zitten even diep
  - "waarde" in B-boom = sleutel+ adres
  - Speciale gevallen: 2-3 bomen, 3-5 bomen,…

  - Beperkingen i.v.m. min en max aantal kinderen garanderen
    - Redelijk gebalanceerdheid
    - Beperkte verspilling van geheugen

Maximale hoogte van B-bomen

- Orde p → minstens d = ⌈ p/2 ⌉ deelbomen per knoop
- Op niveau 1 (onder wortel)
  - Minstens 2 knopen
- Op niveau i
  - Minstens 2 d^{i-1} → 2 d^{i-1} (d-1) waarden

⇒ $h \leq \log_d ((n+1)/2)$



Voorbeeld: berekening orde B-boom

- Stel:
  - Grootte van veld waarop gezocht wordt V = 9 bytes
  - B = 512 bytes
  - Recordadres P_r = 7 bytes, blokadras P = 6 bytes
- 1 knoop van B-boom moet in 1 blok passe
  → max aantal deelbomen p van een knoop:
  - $p*P + (p-1)*(P_r + V) \leq B$
  - $6p + 16(p-1) \leq 512$
  - $p \leq 24$
  - Meestal nog wat extra (administratieve info in blok) → kies p = 23

Operaties en hun kost/efficiëntie

- Opzoeken: $O(\log_d n)$
- Toevoegen, weglaten:
  - Eerst positie opzoeken
  - Wijziging aanbrengen en doorvoeren
  - Alles in $O(\log_d n)$ tijd
- Sequentiele verwerking
  - Boom doorlopen in in-orde (links, knoop, rechts)
  - Interne knopen vaak opnieuw gelezen, tenzij ze centraal in geheugen onthouden worden
  - Kan beter: met B+-bomen

Ook: B-boom alleen zinvol als records klein (orde groot)

B+-bomen

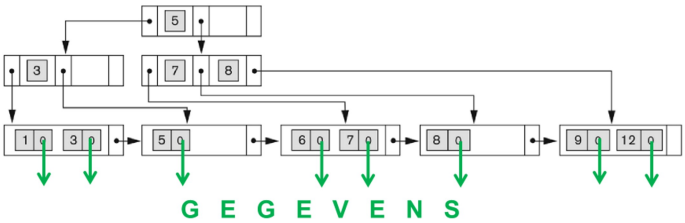
Van B-bomen naar B+ bomen: idee

1) Alle datapointers verwijderen uit interne knopen
   → veel meer ruimte in een interne knoop!
   → Orde kan veel groter zijn
2) Alle datapointers naar een aparte structuur

B+-bomen

- Bij B-bomen
  - Sommige record-wijzers in interne knopen, andere in bladeren
- Bij B+-bomen
  - Interne knopen bevatten enkel sleutels, geen adressen van records
    - Recordadressen enkel in de bladeren
    - Interne knopen bevaten enkel "wegwijzers"

    - Orde pi van interne knopen is nu groter → betere prestaties; orde pb van bladeren ongeveer even groot
- Extra:
  - Aan het eind van een blad wijzer naar volgend blad
    - Maakt sequentieel doorlopen eenvoudiger



Voorbeeld: berekening orde B+-boom

- Gegeven:
  - V = 9 bytes, B=512 bytes, Pr = 7 bytes, P = 6 bytes
- Orde van interne knopen:
  - $p_i * P + (p_i-1) * V \leq B$
  - $6p_i + 9(p_i-1) \leq 512$
  - → $p_i = 34$
- Orde van bladeren
  - $p_b * (Pr + V) + P \leq B$
  - $p_b * (7 + 9) + P \leq B$
  - $16\ p_b + 6 \leq 512$
  - → $p_b = 31$

Hoogte B+-boom

- Maximale hoogte h nog steeds bepaald door logd(n) met d = ⌈ p/2 ⌉
- (formule: huiswerk!)
- Maar:

  d is veel groter dan in een B-boom!

  → h veel kleiner!

  → veel minder blokken lezen

Aantal sleutels en diepte

- Stel 69% vol
- dan:
  - 0.69 * 34 = 23 wijzers per knoop (22 waarden)
  - in blad: 0.69 * p_b = 0.69 * 31 = 21 recordwijzers
- gemiddeld aantal sleutels op elk niveau:
  - wortel: 1 knoop, 22 sleutels
  - niveau 1: 23 knopen, 506 sleutels
  - niveau 2: 529 knopen, 11 638 sleutels
  - bladeren: 12 167 knopen, 255 507 recordwijzers
- Vgl. met 65 536 recordwijzers voor B-boom

Algoritmes

B+-boom: opzoeken van een sleutelwaarde

```
{ K = gezochte sleutel }
n := blok dat wortel van B+-boom bevat;
lees blok n;
zolang n geen blad is:
q := #deelbomen van n;
v0=- ∞, v1..vq-1 waarden in knoop, vq=+ ∞
kies i zo dat vi < K <= vi+1;
n := bi;
lees blok n;
zoek in n een koppel (vi, Pri) met vi=K;
indien gevonden: lees record met adres Pri
anders: meld 'niet gevonden'
```

B+-boom: toevoegen van een record met sleutel K

- zoek blad waar sleutel hoort
- indien niet vol: voeg sleutel gewoon toe
- indien blad al vol: splits blad
  - 1e helft blijft, 2e helft naar nieuw blad
  - voeg sleutel toe aan juiste blad
  - pas ook bladwijzers aan
  - voeg laatste waarde van blad 1 in ouderknoop toe
- herhaal zolang ouderknoop overvol is:
  - splits knoop : helft van waarden naar nieuwe knoop; verhuis laatste waarde van 1e knoop naar ouder

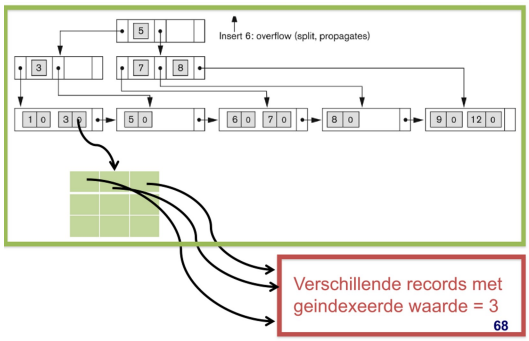
Nieuw

~ B-boom

B+-boom: verwijderen van een sleutel K uit gegevens

- zoek blad met sleutel, verwijder sleutel daaruit
- indien sleutel ergens in interne knopen voorkomt:
  - vervang door waarde net links ervan
- indien onderloop (te weinig waarden in blad):
  - steel enkele waarden van naburig blad (en pas bovenliggende knoop aan)
  - indien nog niet voldoende: voeg 2 bladeren samen
  - verwijder 1 wegwijzer uit bovenliggende knoop
- indien onderloop in interne knoop:
  - herverdeel of voeg samen (met evt. verwijdering van 1 waarde uit bovenliggende knoop...)

Sleutels zijn niet noodzakelijk uniek in de data -> oplossing



Point en Range queries

Verschillende indexstructuren

- Zijn min of meer geschikt voor het zoeken naar 1 waarde of een waardeinterval
- Point queries vs. Range queries

Vooruitblik NoSQL: Indexstructuren in MongoDB

NoSQL

- 'Not only SQL'
- Geen tabellen, maar opslag van bv.
  - Collecties van documenten
  - Sleutel-waarde paren
  - Kolommen van waarden
  - Grafen
  - Objecten
  - ...
- **Voordelen**
  - Horizontaal schaalbaar
  - Geen statisch schema of datamodel
  - Goedkoper in onderhoud
- **Nadelen**
  - Mogelijkheden zeer systeemspecifiek
  - Gevolg: geen universele querytaal
  - Minder (sterke) theoretische garanties

RDBMS		MongoDB
Database		Database
Table, View		Collection
Row		Document (JSON, BSON)
Column		Field
Index		Index
Join		Embedded Document
Foreign Key		Reference
Partition		Shard

MongoDB is "schema-free"!

Samenvatting les 12

donderdag 14 mei 2020 11:47

00

Queryverwerking en -optimalisatie 1

Motivatie & Samenvatting

Relationele vraagtalen

Calculustalen:
- Beschrijving gewenste info (adhv relationele calculus)
- Declaratief: **WAT**

Alebraïsche talen:
- Operatoren & relationele algebra
- Proceduraal: **HOE**

SQL:
- Vooral declaraties (WAT)

declarativiteit

Voor verwerking en optimalisatie

Overzicht les: welke implementatie van relationele algerbra is het best/meest efficiënt voor een bepaalde SQL query?

Query-bomen:
- □ Wat is slimmer? (Heuristische optimalisatie van querybomen)
- □ Wat helpt daarbij? (B+-bomen, indexen)
- □ Waar moeten we op letten? (niet te veel blokkentransport)

Inleiding

Overzicht queryverwerking

1. Lezen en ontleden van query
   a. Scanne: zet string om in tokens
   b. Parser: controleert syntactische correctheid en bouwt interne structuur op die query voorstelt (= meest intuïtieve structuur: canonieke vorm (zie later))
2. Query-optimalisatie

   Algemeen: zoek strategie met minimale kost (kost ≈ (schatting van) uitvoeringstijd)

   Doelen, benaderingen en concrete methodes:

   Doel 1: zo "klein" mogelijke tussenrelaties
   1) Benadering 'op schema-niveau': gebruik transformatieregels voor relationele algebra
      - Concreet: operaties in query-boom van boven naar beneden verschuiven
   2) Benadering 'op gegevens-niveau': gebruik info over aantal records, selectiviteit,…
      - Concreet: operaties in query-boom van links naar rechts verschuiven

   Opmerking: gegevens als # tupels in een relatie, # waardes voor bepaald attribuut,… worden bijgehouden om te bepalen welke tussenrelaties klein zijn.

   Doel 2: weinig lees-, schrijf- en vergelijkingsoperaties
   3) Benadering 'op algoritmes-niveau': gebruik efficiënte implementaties
      - Concreet: gebruik indexen (indien zinvol, zie volgend voorbeeld), gebruik selectie-, join-, enz. implementatie

   Opmerking: Doel 1 bereiken helpt ook bij doel 2
3. Codegeneratie

   Resultaat van 2. is uitvoerbare code (vaak niet de allerefficiëntste, wel een zeer efficiënte)
4. Uitvoering van query

Omzetting SQL ⟶ relationele algebra

Nodig aangezien relationele algebra meer geschikt is om uitvoeringsstrategie te bepalen

```
SELECT    att-list
FROM      R1,...,Rn
WHERE     condities;
```

$\pi_{\text{att-list}}(\sigma_{\text{condities}}(R_1 \times \ldots \times R_n))$

geneste queries: omzetting per "query block"

```
SELECT Lname, Fname
FROM   EMPLOYEE
WHERE  Salary > (SELECT   MAX(Salary)
                 FROM     EMPLOYEE
                 WHERE    Dno = 5 );
```

$c := \Im_{\text{MAX Salary}}(\sigma_{\text{Dno = 5}}(\text{EMPLOYEE}))$

$\pi_{\text{Lname, Fname}}(\sigma_{\text{Salary > c}}(\text{EMPLOYEE}))$

Aanpakken voor query-optimalisatie

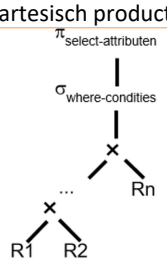
- Heuristische (=systematisch tot een oplossing komen) regels voor ordenen van operaties binnen een query-boom
- Systematische schatting van kosten van verschillende uitvoeringsstrategieën
  - # tuples in tussenresultaten, hoeveel blokkentransport, …
  - Vaak alleen voor beste strategieën volgens heuristische regels
- Semantische query-optimalisatie: gebruik extra kennis (in vorm van restricties) om query te transformeren

Heuristische optimalisatie van query-bomen

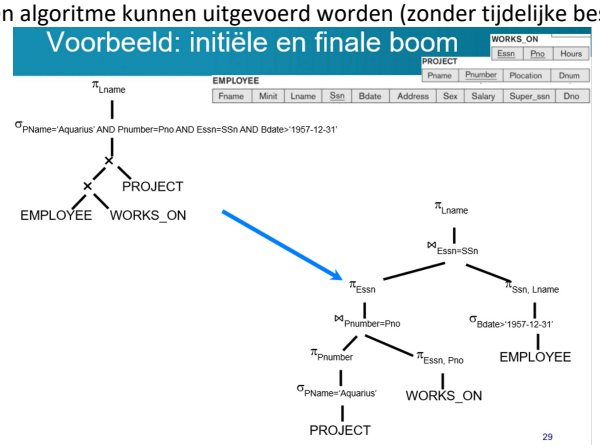
Heuristische optimalisatie (~ doel 1 (zie hoger) van query-optimalisatie)

Stappenplan

1. Bouw queryboom in canonieke vorm: eerst cartesisch product (FROM), dan selectie (WHERE), dan projectie (SELECT)



   Opmerking: hier is nog sprake van conjunctie van selecties (geschreven als ... AND …)
2. Splits conjunctie van selecties en schuif elke selectie op zich zo ver mogelijk naar beneden
3. Plaats meest restrictieve selectie 'eerst' (= naar links) **(Waarom? Zie later)**
   Opmerking: houd join condities wel bij de cartesische producten
4. Vervang cartesisch product + selectie door join
5. Projecteer in begin al wat enkel gebruikt zal worden
   Opmerking:
   Is niet enkel wat in SELECT staat; ook wat in join/selecties gebruikt wordt
6. Identificeer deelbomen die door één algoritme kunnen uitgevoerd worden (zonder tijdelijke bestanden) **(? Geen voorbeeld van ?)**



Algemene transformatieregels

- σ-cascade
  Selectie op conjunctie van condities opsplitsen
  $\sigma_{c1 \text{ AND } c2 \text{ AND } \ldots \text{ AND } cn}(R) \equiv \sigma_{c1}(\sigma_{c2}(\ldots(\sigma_{cn}(R))\ldots))$
- Commutativiteit van σ
  Binnenste selectie restrictiever dan buitenste is beter
  $\sigma_{c1}(\sigma_{c2}(R)) \equiv \sigma_{c2}(\sigma_{c1}(R))$
- π-cascade
  Aanname: list(i) $\subseteq$ list(i+1)
  $\pi_{list1}(\pi_{list2}(\ldots(\pi_{listn}(R))\ldots)) \equiv \pi_{list1}(R)$
- Omwisselen van σ en π (onder voorwaarden!)
  Voorwaarde: c slaat enkel op de attributen A1,…,An uit projectielijst
  $\pi_{A1, A2, \ldots, An}(\sigma_c(R)) \equiv \sigma_c(\pi_{A1, A2, \ldots, An}(R))$
- Commutativiteit van $\bowtie$ (of $\times$)

- $R \bowtie S \equiv S \bowtie R$
- $R \times S \equiv S \times R$

- Omwisselen van σ en $\bowtie$ (of $\times$)

Indien c enkel slaat op attributen van R:	Indien c= c1 AND c2 en c1 slaat enkel op R, c2 enkel op S:
$\sigma_c(R \bowtie S) \equiv \sigma_c(R) \bowtie S$ $\sigma_c(R \times S) \equiv \sigma_c(R) \times S$	$\sigma_c(R \bowtie S) \equiv \sigma_{c1}(R) \bowtie \sigma_{c2}(S)$ $\sigma_c(R \times S) \equiv \sigma_{c1}(R) \times \sigma_{c2}(S)$

- Omwisselen van π en $\times$
  - $\pi_L(R \times S) \equiv \pi_{L(R)}(R) \times \pi_{L(S)}(S)$
  - L(R) zijn de attributen van R in L, L(S) die van S
- Omwisselen van π en $\bowtie$

Als join-conditie c alleen attributen in L gebruikt:	Anders:
$\pi_L(R \bowtie_c S) \equiv \pi_{L(R)}(R) \bowtie_c \pi_{L(S)}(S)$	• R en S projecteren op join-attributen + attributen in projectie-lijst • join • op het einde nogmaals projecteren op L $\pi_L(R \bowtie_c S) \equiv \pi_L(\pi_{L(R),c(R)}(R) \bowtie_c \pi_{L(S),c(S)}(S))$ bv. $\pi_{R.A,S.D}(R \bowtie_{B=C} S) \equiv \pi_{R.A,S.D}(\pi_{A,B}(R) \bowtie_{B=C} \pi_{D,C}(S))$ B en C meenemen in projectie om te kunnen joinen

- Commutativiteit van verzameling-operaties
  - $\cap$ en $\cup$ commuteren, maar $\setminus$ niet
  - $R \cap S \equiv S \cap R$
  - $R \cup S \equiv S \cup R$
- Associativiteit van $\cap$, $\cup$, $\bowtie$ en $\times$
  - $(R \cap S) \cap T \equiv R \cap (S \cap T)$
  - $(R \cup S) \cup T \equiv R \cup (S \cup T)$
  - $(R \bowtie S) \bowtie T \equiv R \bowtie (S \bowtie T)$
  - $(R \times S) \times T \equiv R \times (S \times T)$
- Commutativiteit van σ met verzameling-operaties
  - $\sigma_c(R \cap S) \equiv \sigma_c(R) \cap \sigma_c(S)$
  - $\sigma_c(R \cup S) \equiv \sigma_c(R) \cup \sigma_c(S)$
  - $\sigma_c(R \setminus S) \equiv \sigma_c(R) \setminus \sigma_c(S)$
- Commutativiteit van π met verzameling-operaties
  - $\pi_L(R \cup S) \equiv \pi_L(R) \cup \pi_L(S)$
- Samenvatten van (σ,$\times$) in $\bowtie$
  - $\sigma_c(R \times S) \equiv R \bowtie_c S$

  met c een join-conditie
- Herschrijven van join-/selectiecondities
  - NOT(c1 AND c2) ≡ NOT(c1) OR NOT(c2)
  - NOT(c1 OR c2) ≡ NOT(c1) AND NOT(c2)
- … (lijst is niet volledig (ook in slides niet))

Kleine relaties – histogrammen

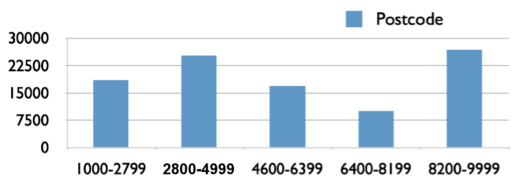
Histogrammen:

- Geven meer info over verdeling van waardes

  Vorm groepen van waardes ⟶ houd # tupels per groep bij ⟶ veronderstel uniforme verdeling binnen groep
- In praktijk vaak gebruikt voor 'belangrijke' attributen (attributen die in veel queries voorkomen)
- 2 soorten:
  - Equi-depth: groepen met ongeveer evenveel tupels
  - Equi-width: groepen met ongeveer evenveel waardes

vb:

- … WHERE postcode=3000 …
  - in groep 2800-4999 met 2200 mogelijke waardes
  - 25200 tupels in die groep
  - schatting: $\lceil 25200/2200 \rceil$=12 tupels voldoen aan conditie



Informatie in catalogus: schatting grootte van operaties

Te gebruiken waardes om grootte van resultaten te schatten:

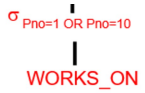
- n_R: aantal tupels in R
- histogrammen voor attributen van R
- V(A,R): aantal verschillende waardes van attribuut A in R
- max(A,R): maximale waarde van A in R
- min(A,R): minimale waarde van A in R

Schatting grootte: enkelvoudige selectie

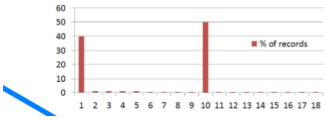
Met histogram betere schattingen mogelijk, zonder histogram:

Point selectie	Range selectie
$\sigma_{A=v}(R)$ – $n_R/V(A,R)$ tupels in resultaat – 1 tupel indien A sleutel is	$\sigma_{A\leq v}(R)$ – als min(A,R) en max(A,R) in catalogus: aantal tupels is • 0 als v<min(A,R) • $n_R \times (v-min(A,R)) / (max(A,R)-min(A,R))$ anders – geen informatie gegeven: gebruik $n_R/2$

Vb.



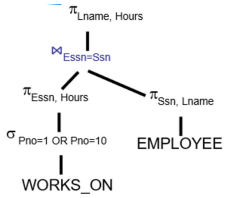
- 1000 records in EMPLOYEE, 5000 in WORKS_ON
- 18 unieke waarden van Pno in WORKS_ON
- Zonder extrainfo: (5000 / 18) * 2 = 556
- Met histogram: 0.4 * 5000 + 0.5 * 5000 = 4500



Schatting grootte: join $R\bowtie_{R.A=S.B}S$

Indien R.A sleutel: n_S tupels in resultaat

Indien S.B sleutel: n_R tupels in resultaat



- R (gebaseerd op WORKS_ON) heeft 5000 records, zie 2 slides eerder
- S (gebaseerd op EMPLOYEE) heeft 1000 records
- Ssn is een sleutel in S

➔ Schatting aantal records in R⋈S = 5000

Indien R.A en S.B geen sleutels:

- V(A,R) verschillende waardes voor R.A
- veronderstel uniforme verdeling:
- voor elke waarde a
  - $n_R/V(A,R)$ tupels in R met A=a
  - $n_S/V(A,R)$ tupels in S met B=a
  - dus $n_R/V(A,R) * n_S/V(A,R)$ tupels in join met waarde a
- voor alle waardes samen: $V(A,R) * n_R/V(A,R) * n_S/V(A,R)$ tupels
- opnieuw: betere schatting met histogrammen

- bv. 10000 tupels in R, 5000 tupels in S, 100 waardes voor R.A:
  - per waarde (10000/100)*(5000/100)=100*50=5000 join tupels
  - in totaal: 100*5000=500000 join tupels

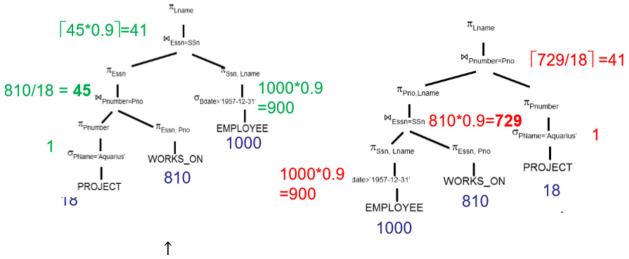
Terzijde: waarom selectievere relaties naar links?

Selectiviteit = % records die aan constraint voldoen

Selectiecardinaliteit = Gemiddeld # records die aan constraint voldoen

Aannamen:

- 1000 tupels in E, 810 in WO, 18 in P
- *Pname* uniek, 90% van de werknemers niet ouder dan 60
- Uniforme verdelingen



↑ Betere query-boom

Schatting grootte: andere operaties:

Projectie $\pi_A(R)$

n_R met duplicaten ⟶ V(A,R) na verwijderen van duplicaten

Verzamelingen (met duplicaten)

R ∪ S: $n_R + n_S$

R ∩ S: $\min(n_R, n_S)$

R \ S: n_R

Van query-boom naar uitvoeringsplan

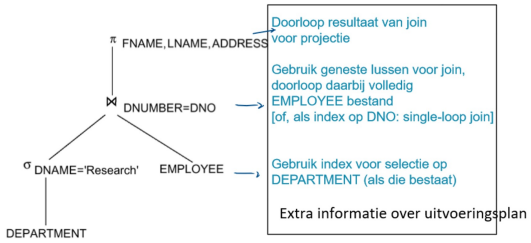
Na heuristische optimalisatie geeft queryboom de volgorde van operaties en een indicatie van deelbomen waarvoor één algoritme bestaat, maar we missen nog:

Welke implementaties van operaties gebruikt worden

Welke indexen gebruikt worden

Of we met materialised of pipelined evaluatie werken (met/zonder tijdelijke bestanden, zie later)

```
SELECT  Fname, Lname, Address
FROM    EMPLOYEE, DEPARTMENT
WHERE   Dname = 'Research' AND Dnumber = DNO ;
```



Extern sorteren

Sorteren is vaak nodig (ORDER BY, duplicate weghalen, implementatie JOIN,…), maar vaak niet voldoende geheugen voor algoritmes zoals quicksort (intern sorteren) ⇒ **extern sorteren met merge-sort** (in kort: splits te sorteren gegevens in stukken, sorteer (intern) elk stuk en voeg samen)

Merge-sort

Stappenplan

FASE 1:

Lees een deel van het bestand in buffer (= beschikbaar werkgeheugen) met k blokken

Sorteer records in buffer (bv. Intern sorteren met quicksort)

Schrijf resultaat naar schijf als tijdelijk bestand

Herhaal tot hele bestand overlopen is

FASE 2:

Meng (maximaal) k-1 bestanden tot een gesorteerd tijdelijk bestand:

Lees 1 (gesorteerd) blok per bestand in buffer, gebruik laatste buffer voor resultaat

Vergelijk volgend record van alle buffers, schrijf kleinste naar resultaat-buffer

Alle records van blok gezien? Lees volgende blok van dat bestand

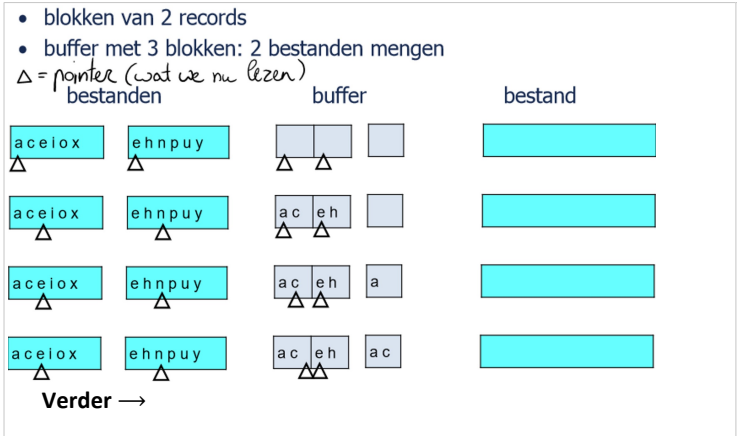
Resultaat-buffer vol? Voeg toe aan bestand op schijf

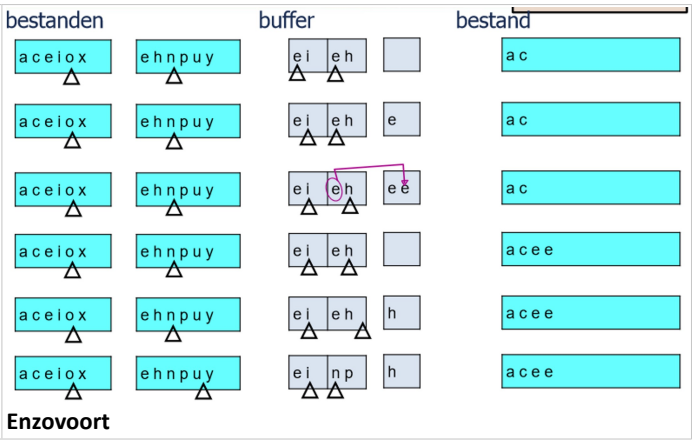
Herhaal tot er maar één gesorteerd bestand meer is

herhalen tot alle deelbestanden volledig geschreven zijn

geheel ook blijven herhalen

Voorbeeld





Complexiteit

Complexiteit totaal = complexiteit sorteerfase + complexiteit mengfase

Sorteerfase	Mengfase
Zoveel mogelijk blokken samen in het geheugen gelezen en gesorteerd (afh van de beschikbare bufferruimte dus) ⟶ Complexiteit: $n_R = \lceil b/n_B \rceil$ "runs" nodig Met: b = # blokken in het te sorteren bestand n_B = beschikbare blokken in buffer elk blok wordt een keer ingelezen en een keer geschreven: voor *b* blokken: 2 * *b* voorbeeld: • $n_B = 5$ $b = 1024$ • dan: $n_R = \lceil 1024 / 5 \rceil = 205$ ⇒ 205 gesorteerde deelbestanden	Mengingsgraad d_M: -# deelbestanden die in één keer gemengd kunnen worden = # beschikbare buffers − 1 = $n_B - 1$ -# doorgangen ("passes"): $\lceil \log_{dM} n_R \rceil$ -Elke doorgang leest en schrijft b blokken •voorbeeld: •$d_M = 4$ •aantal gesorteerde deelbestanden: 205 → 52 → 13 → 4 → 1 •complexiteit mengfase: 2 * *b* * ($\log_{dM} n_R$)

$$Totale\ complexiteit = 2 \cdot b + 2 \cdot b \cdot (\log_{dM} n_R) = 2 \cdot b + 2 \cdot b \cdot (\log_{dM} \lceil b/n_B \rceil)$$

Implementaties van selectie (~ doel 2 (zie hoger) van query-optimalisatie)

Voorbeelden

Verschillende strategieën mogelijk:
- Soort selectie criterium
- Bestaan van indexen
- b_R = # blokken in relatie R

Voorbeelden criteria:
- OP1: $\sigma_{Ssn = '123456789'}$ (EMPLOYEE) … query primary key
- OP2: $\sigma_{Dnumber > 5}$ (DEPARTMENT) range query primary key
- OP3: $\sigma_{Dno = 5}$ (EMPLOYEE)
- OP4: $\sigma_{Dno = 5 \text{ AND } Salary > 3000 \text{ AND } Sex = 'F'}$ (EMPLOYEE) conjunctieve selectie
- OP5: $\sigma_{Essn = '123456789' \text{ AND } Pno = 10}$ (WORKS_ON)

Implementatiemethodes

S1: lineair zoeken
- Kan altijd
- Kost? **b_R** blokken transporteren
  Indien selectieconditie primaire sleutel gebruikt: stop indien gevonden (**gemiddeld $b_R/2$** blokken transporteren)

S2: binair zoeken
- Kan als: relatie gesorteerd op selectie-attribuut (primary key bv.) & selectie met '='
  Vb. $\sigma_{Ssn='123456789'}(EMPLOYEE)$ als er geordend is op Ssn
- Kost? = rekenkost van binary search = $\lceil \log_2 b_R \rceil$

S3: primaire index of hash-functie (sleutel ⟶ blok/recordwijzer) om één record op te halen (point query)
- Kan als: index of hash op selectie-attribuut & selectie met '='
  Vb. $\sigma_{Ssn='123456789'}(EMPLOYEE)$ als er index is op Ssn
- Kost? Indien primaire index: # blokken nodig voor vinden van waarde in index
  Indien hashing: meestal 1 of 2 blokken

S4: gebruik primaire (range query)
- Kan als:
  index op selectie-attribuut
  bv. OP2: $\sigma_{Dnumber > 5}$ (DEPARTMENT) met index op Dnumber
- Kost?
  gemiddeld: $b_R/2$ + aantal blokken nodig voor vinden waarde in index

S5: gebruik cluster-index om meerdere records op te halen
- Kan als:
  = op attribuut dat geen sleutel is
  cluster-index voor dat attribuut
  bv. OP3: $\sigma_{Dno = 5}$ (EMPLOYEE) met index op Dno
- Kost?

⌈(s/bfr)⌉+ aantal blokken nodig voor vinden waarde in index
s = schatting van aantal tupels dat aan conditie voldoet
bfr = aantal records per block van relatie

S6: gebruik secundaire index (B⁺-boom) (uitbreiding van S5 -- ook range query makkelijk in B⁺-boom)
- Kan als:
  voor = en ongelijkheden
  als index voor dat attribuut bestaat
  bv. OP3: $\sigma_{Dno = 5}$ (EMPLOYEE) met index op Dno
- Kost?
  - afhankelijk van
    - vergelijkingsoperatie
    - hoeveel tupels voldoen aan conditie
    - waardes attribuut uniek of niet
    - kost index-gebruik

S7: conjunctieve selectie c_1 AND … AND c_N
- Kan als:
  als voor een ci een van de methodes S2-S6 bruikbaar is
  bv. OP4: $\sigma_{Dno = 5 \text{ AND } Salary > 3000 \text{ AND } Sex = 'F'}$ (EMPLOYEE) met index op Dno
- Kost? Afhankelijk van gekozen methode + van de grootte van de bekomen tussenrelatie

S8: conjunctieve selectie met samengestelde index
- Kan als:
  alle subcondities met =
  er bestaat een samengesteld index voor de selectie-attributen
  bv. OP5: $\sigma_{Essn = '123456789' \text{ AND } Pno = 10}$ (WORKS_ON) met index op (Essn,Pno)
- Kost? Afhankelijk van type index

S9: conjunctieve selectie door intersectie van recordpointers
- Kan als: secundaire indexen met recordpointers bestaan voor aantal subcondities met =
  - principe:
    - voor elke =-conditie met secondaire index:
      haal verzameling pointers uit index
    - bereken doorsnede van die verzamelingen
    - haal records op en filter volgens overblijvende condities
  - voordeel: groot deel van selectie-werk alleen op indexen
- Kost? Afhankelijk van type index

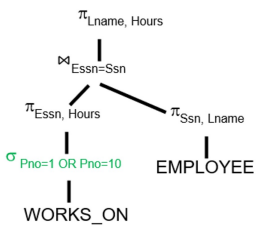
Welke kiezen?

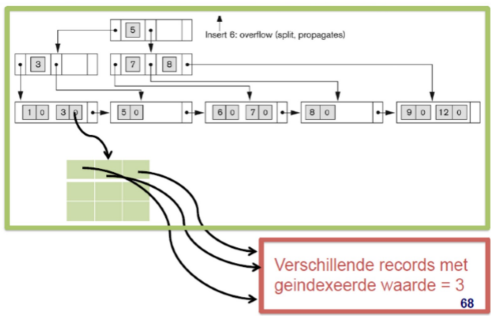
- Enkelvoudige voorwaarde
  - Keuze uit toepasbare methodes S2-S6 volgens schatting kost
  - Indien geen ervan mogelijk: S1 (lineair zoeken is worst case)
- Conjunctieve voorwaarde
  - Vergelijk kosten S7-S9
  - Selectiviteit van selectie-condities mee in rekening brengen (hoe minder tupels in resultaat verwacht, hoe beter)
- Disjunctieve voorwaarde
  - Indien voor elke deelvoorwaarde efficiënte methode bestaat ⟶ gebruik die & neem dan unie ∪ om alle records te vinden
  - Zodra er voor één deelvoorwaarde lineair zoeken nodig is, is lineair zoeken de beste oplossing

Voorbeelden

1	**2**

Stel dat er een index is op Pno, geïmplementeerd zoals in les 11 (zie afbeelding).
Hoeveel blokken moet je lezen als je de query zo afwerkt en de index gebruikt?
Is dat efficiënt?



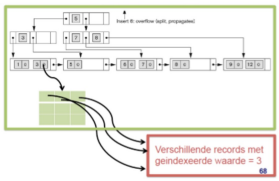


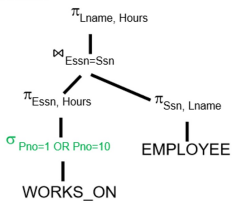
1. Stel dat er een index (B+ boom) index is op Pno. Bij gebruik van de index:

→ voor elke waarde van de selectie (1, 10):

– Opzoeken waarde in index
  ➔ boomdiepte blokken lezen (2 keer)
– Alle records opzoeken
  ➔ 2000 blokken lezen (Pno=1) en 2500 blokken lezen (Pno=10)

2. Alternatief: index niet gebruiken, full-table scan van WORKS_ON
– Stel dat 10 records van WORKS_ON in 1 blok passen
  ➔ 5000/10 = 500 blokken lezen





Besluit: 3 opmerkingen over indexen

Niet noodzakelijk voor elke query beter een index te gebruiken
Daarnaast: een index op een veld waar je nooit naar zoekt, kost tijd en ruimte en levert geen voordelen op
(herhaling uit les 11:) het is niet per se sneller voor elke query een hashindex te gebruiken
(point query ⟶ hashindex, range queries ⟶ beter B^+-boom)

Samenvatting les 13

donderdag 14 mei 2020 11:47

Queryverwerking en -optimalisatie 2

Implementaties van andere operaties (join,…)

Join

Hier enkel 2-way (uit 2 bestanden) equijoin / 2-way natuurlijke join

Algemeen bekijken we dus:

$$R \bowtie_{R.A=S.A} S$$

Notatie en naamgeving:

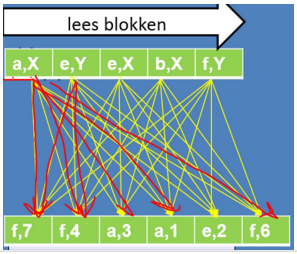
R = buitenste lus: staat links in joinnotatie

S = binnenste lus: staat rechts in joinnotatie

J1: Geneste lussen

meest naïeve manier; werkt ook meest algemeen (meer dan enkel een equijoin)

Bovenste rij ⟶ R, onderste rij ⟶ S



Gewone nested loops	**Beter =** Blocked nested loops

For (tuple)r in R:
For (tuple)s in S:
Check if r[A] = s[A]
 	
Geheel telt mee in blokkentransport
	For (block) Br in R:
	For (block) Bs in S:
	For (tupel)r in Br:
	For (tupel)s in Bs:
	Check if r[A]=s[A]
	
	Enkel eerste twee forloops in blokkentransport, rest gebeurt in werkgeheugen

Kost (van block nested loops) algemeen:

- Kost in werkgeheugen om de join uit te voeren ⟶ minimum 3 blokken nodig: 1 voor R, 1 voor S, 1 voor resultaat
- Kost voor blokkentransport (geen gebruik maken van werkgeheugen
  ⟶ indien R buitenste lus alle blokken van R overlopen + voor elke blok alle blokken van S overlopen
  ⟶ $blokkentransport = b_R + b_R * b_S$

Opmerking:

Kleinste tabel aan buitenkant levert minder blokkentransport op (kleinere som)

Link met belangrijk overzicht van query-optimalisatie (zie les 12): de volgorde aanpassen heeft invloed op **doel 1** 2) en **doel 2** 3)

Kost als S (binnenste lus) volledig in werkgeheugen past:

- Hiervoor nodige geheugen ⟶ minstens b_S + 2 blokken geheugen nodig
- Blokkentransport vermindert dan tot:
  ⟶ $blokkentransport = b_R + b_S$

Kost als er voor de buitenste tabel n blokken als buffer in het werkgeheugen gebruikt kunnen worden:

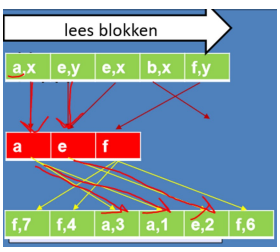
- Hiervoor nodige geheugen ⟶ minstens n + 2 blokken geheugen nodig
- Blokkentransport vermindert dan tot:
  ⟶ $blokkentransport = b_R + b_R/n * b_S$

Zelfde **opmerking** als bij kost algemeen

J2: Single-loop join (= geneste lussen met index)

Analoog aan gewone nested loops

Verschil hier = tussenliggende laag van indexen zodat je niet per blok van R (bovenste rij) alle blokken van S (onderste rij) moet controleren, maar enkel de overeenkomstige)



Belangrijk hier: kies relatie met hoogste Join Selection Factor (JSF) voor buitenlus

Single-loop join
For (tupel) in R:

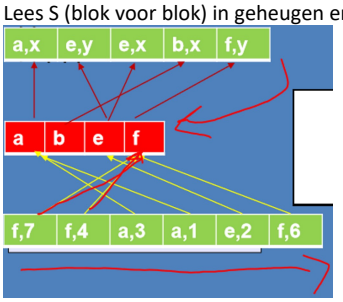
Gebruik index om tupels in s te vinden waarvoor s[A]=r[A]

Kost:

b_R = aantal blokken R ; n_R aantal records in R ; c = kost voor het vinden van s in index (hangt af van type index, zie lessen over indexing)

$\longrightarrow blokkentransport = b_R + b_R * c$

J4.1: Hash join

Hash join: Alles in hoofgeheugen	Block nest join: Een groep (niet alle!) records in hoofdgeheugen
1. Lees R (in zijn geheel) in geheugen en bouw hash index 2. Lees S (blok voor blok) in geheugen en *probe*(=onderzoek) hash 	1. Lees R (1 of zoveel mogelijk blokken) in geheugen en bouw hash index 2. Lees S (blok voor blok) in geheugen en *probe* hash 3. Herhaal voor volgende blokken / groepen van blokken van R totdat R verwerkt is. 

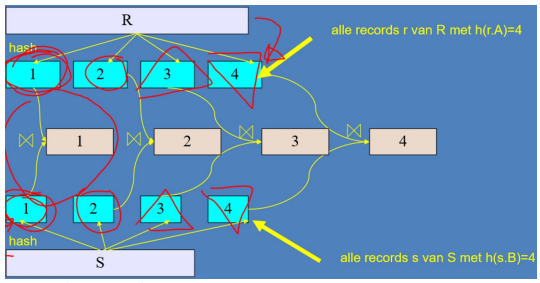
J4.2: Partition hash-join

Fase 1:

splits beide relaties met (beide dezelfde!) hash-functie h1
schrijf delen naar tijdelijke bestanden

Fase 2: (⟶ per partitie ~ for-loop)

Lees R_i en bouw hash-index (een andere hash-functie h2)
Lees S_i blok per blok en gebruik hash-index



Kost:

- b_R = aantal blokken R ; b_R aantal blokken in S ; b_{RES} aantal blokken van het resultaat

  $\longrightarrow blokkentransport = 3 * (b_R + b_S) + b_{RES}$

  Opmerking:
  - veel beter dan J1 geneste lussen onder zelfde condities
  - Hoe kunnen we zeker zijn dat R_i in geheugen past? Hash-functie met voldoende verschillende waardes voorzien (vb. modulo een groot genoeg getal); maar werkt alleen als waardes ongeveer even vaak voorkomen

J3: Sort-merge join

Voorwaarde: relaties moeten gesorteerd op hun join-attribuut, anders eerst sorteren
Zelfde principe als bij extern sorteren (?): bekijk tupels in volgorde

Kost indien al gesorteerd:

$\longrightarrow blokkentransport = b_R + b_S$

Zelfs met sorteren is dit vaak nog een goede optie omdat het resultaat ook gesorteerd is (groot voordeel!), dus volgende operaties gaan mogelijk veel makkelijker

R

A	B
a	1
a	3
e	2
f	4
f	6
f	7

S

A	C
a	x
b	x
e	x
e	y
f	y

$R \bowtie_{R.A=S.A} S$

A	B	C
a	1	x
a	3	x
e	2	x
e	2	y
f	4	y
f	6	y
f	7	y

Stappenplan (zie volledig voorbeeld in slides - hier in woorden; notatie *tabel.kolom.rij*):

$R.A.1 = a = S.A.1 \longrightarrow$ rechts wegschrijven & volgende rij in tabel 2 checken: $R.A.1 \neq S.A.2$ dus 'cursor' in tabel 1 opschuiven
$R.A.2 = S.A.1 \longrightarrow$ rechts wegschrijven & volgende rij in tabel 2 checken: $R.A.2 \neq S.A.2$ dus 'cursor' in tabel 1 opschuiven
$R.A.3 \neq S.A.1$ dus 'cursor' in tabel 2 opschuiven $\longrightarrow$ $R.A.3 \neq S.A.2$ opnieuw cursor in tabel opschuiven
$R.A.3 = S.A.3 \longrightarrow$ rechts wegschrijven & volgende rij in tabel 2 checken: $R.A.3 = S.A.4 \longrightarrow$ wegschrijven enz.

Overzicht

- Geneste lussen:
  - Zonder iets: vrij nutteloos
  - Met blokken (BNL): zeer algemeen & kan voor alle joincondities (t.o.v. 'zonder iets' toch beetje geoptimaliseerd
  - Met index (Single-loop join): kan alleen als nodige index bestaat & heel nuttig als er zeer selectieve selecties zijn (wanneer je weinig tupels terugkrijgt)

- (hybrid) hash join
  - Goede keuze voor grote relaties
- Sort-merge join
  - Veel gebruikt (aangezien relaties vaak al gesorteerd zijn)
  - Gesorteerd resultaat is ook handig om mee verder te werken

Projectie $\pi_{attribuutlijst}(R)$

Als attribuutlijst een sleutel van R bevat:	Als attribuutlijst geen sleutel van R bevat:
Opmerking: resultaat heeft sowieso evenveel tupels als R • Overloop R en schrijf voor elke tupel r r[attribuutlijst] naar resultaat	Opmerking: mogelijk dubbels in resultaat • Doe zelfde als wanneer er wel sleutel zou zijn en verwijder daarna duplicaten

Duplicaten verwijderen

Best door sorteren (hashing kan evt ook)

Sorteren	Hashing

Sorteer R
Lees tupels in die volgorde (laatste = null)
For (tupel) r in R:
  If r != laatste:
  Schrijf r na resultaat, laatste = r (?)
  Else:
  Volgend tupel
	For (tupel) r in R:
	  If r niet in hashtabel:
	  Wegschrijven in hashtabel & resultaat
	  Else:
	  Volgend tupel

Verzameling-operaties

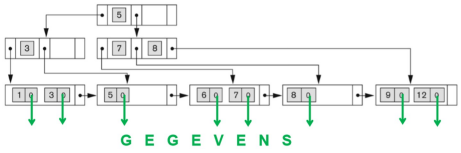
- Cartesisch product $R \times S$
  - Duur + groot resultaat = vermijd zoveel mogelijk
- Unie, doorsnede, verschil $\cup$, $\cap$, $-$
  - Enkel voor relaties met zelfde attributen
  - Implementatie: sorteer beide relaties volgens zelfde attributen en doorloop in parallel en voeg samen
  - Hashing gebruiken kan ook

	Sorteren	Hashing
Unie	Sorteren niet nodig; enkel duplicaten vinden	R sowieso al wegschrijven & dan S overlopen: Indien nog niet weggeschreven, dan ook wegschrijven
Doorsnede	Enkel duplicaten toevoegen: Beide relaties doorlopen: als beide pointers op zelfde tupel, dan toevoegen in resultaat	R niet wegschrijven en S overlopen: Indien tupel van S ook in R, dan wegschrijven
Verschil	Per tupel in R checken of die in S zit: Indien die niet in S zit, dan wegschrijven	OVT xd

Aggregatie-operaties

Zonder GROUP BY

Indien geen index op attribuut: doorlopen
Indien dichte index op attribuut bestaat: resultaat kan zuiver uit index berekend worden
bv. B⁺-boom als index ⟶ min/max = meest linkse/rechtse wijzer volgen
⟶ avg/count/sum = bladeren van B⁺-boom doorlopen en '..._so_far' bijhouden



Met GROUP BY

```
SELECT A, count(B)
FROM R
GROUP BY A;
```

met hashing:
- maak een hash tabel op A, houd count(B) bij voor elke waarde
- voor elk tupel r in R
  - kijk of r[A] in hash tabel
  - ja: verhoog count met 1
  - nee: voeg toe aan hash tabel en initiële count op 1

- met sorteren:
  - sorteer R op A
  - de tupels binnen een groep staan naast elkaar
  - lees R in volgorde en tel (of bereken andere aggregatie-operatie)

Opmerkingen:
Sum count min en max houden 1 waarde per groep bij, avg 2 waardes (sum & count)
Count(distinct R): eerst duplicaten weghalen en daarna tellen

Queryuitvoering

Situering:
Queryboom en implementatiemethode voor elke knoop gekozen
Laatste vraag: wat met tussenresultaten?

Materialization

- Operaties één per één evalueren

- Schrijf tijdelijke relatie naar harde schijf
- Lees voor volgende operatie

Opmerkingen:
- Tijdelijke relaties naar harde schijf schrijven kan altijd
- Maar: mogelijk veel extra tijd omwille van lezen en schrijven van tijdelijke relaties
- Daarom: "double buffering"
  - Gebruik 2 buffers voor elke operatie: wanneer eerste vol is, schrijf deze naar harde schijf en gebruik tegelijk de tweede voor verdere resultaten ⇒ sneller door overlappend werk qua datatransport

Pipelining

- Evalueer meerdere operaties tegelijk
- Schrijf niet naar harde schijf
- Meestal sneller, maar meer geheugen nodig
- Niet altijd mogelijk (bv. Sort-merge join **(J3)**)

Opmerkingen:
- Resultaten direct doorgeven aan volgende operatie vermijdt wegschrijven / terug lezen ⟶ (veel) sneller
- Maar: niet altijd mogelijk & meer geheugen nodig
- Sommige operaties blokkeren: volledige invoer nodig om berekening te starten
- Operaties moeten tupels in resultaat genereren terwijl invoer tupels binnen komen:
  - selectie: pipelining meestal mogelijk
  - sorteren: blokkeert
  - sort-merge join: de laatste (merge) fase kan via pipelining, sorteerga deelte niet
  - hash-join: partitioning blokkeert, tweede fase kan via pipelining
  - aggregaten: hebben meestal volledige invoer nodig, wel stap per stap pipelinen is mogelijk
  - duplicaten weghalen: sorteren blokkeert
  - operaties op verzamelingen: zie duplicaten weghalen } hashing wel mogelijk

Queryoptimalisatie in MySQL

In XAMPP gebruik je MariaDB, een fork van MySQL: bijna identiek aan MySQL

Algemeen

MySQL heeft basic queryoptimalisatie (= goed genoeg)
± heuristieken die gezien hebben: voorkeur voor kleine relaties en gebruik van indexen

Je kan voor elke query MySQL …

… naar queryboom en uitvoeringsplan vragen	… helpen een betere boom/plan te bouwen	… dwingen je query anders uitvoeren
EXPLAIN	ADD INDEX, …	STRAIGHT JOIN, FORCE INDEX, …

Belang van EXPLAIN

MySQL heeft zoveel heuristieken om queries te optimaliseren dat je niet alles kan kennen (en er zelfs geen volledig overzicht over hebt)
Zelfde in wilde weg optimaliseren is wss niet beter ⟶ gebruik EXPLAIN als startpunt om evt. verbeteringen te zoeken

EXPLAIN

Resultaat = tabel met info over uitvoering van statement (een rij voor elke relatie, waarin ze gelezen worden)
MySQL gebruikt altijd geneste lussen voor join, **MAAR** is niet de 'J1 geneste lussen' van hoger (MySQL werkt met buffers, block nested loops, …)
Tabel bestaat uit:
- **Id**: hoeveelste SELECT binnen query (buitenste heeft id 1)
- **Select_type**: soort SELECT (indien genest ⟶ PRIMARY, SUBQUERY,… // indien niet genest ⟶ SIMPLE)
- **Table**: over welke relatie gaat deze rij?
- **Type**: soort join **(⟶ zie vooral oefenzitting)**
- **Possible_key**: welke indexen zijn er?
- **Key**: welke index is er gebruikt?
- **Key_len**: lengte van gekozen sleutel (in aantal bytes)
- **Ref**: welke kolomo of constante vergelijken met index?
- **Rows**: schatting van aantal te lezen tupels
- **Extra**: meer informatie over uitvoering

Op welke info baseren queryplannen zich?

- Informatie uit de catalogus:
  - **Voor ons:** Wat de *data dictionary* (zie pagina phpMyAdmin) bevat: schema (= exact) + cardinaliteiten van tabellen en indexen (= schatting)
- Sampling ("random dives") van de data voor verdere cardinaliteiten ⟶ vaak inschattingen van aantal records (heuristieken!)
  - **Voor ons:** gebruik common sense om de schattingen van de EXPLAIN te beoordelen
- Kostenschatting , gebaseerd op blokkentoegang

Voorbeelden (oefenzitting 9 bespreekt ook EXPLAINs van queries)

- De gegevensbank

Table	Action	Rows
department	Browse Structure Search Insert Empty Drop	3
dependent	Browse Structure Search Insert Empty Drop	7
dept_locations	Browse Structure Search Insert Empty Drop	5
employee	Browse Structure Search Insert Empty Drop	8
employee2	Browse Structure Search Insert Empty Drop	8
project	Browse Structure Search Insert Empty Drop	7
works_on	Browse Structure Search Insert Empty Drop	16

Constraints in het schema zijn onvolledig:
- department, employee, project: hebben hun primaire sleutels
- de andere constraints ontbreken
- nieuwe tabel: employee2 ~ employee, maar *zonder* primaire sleutel

Stuk uit de data dictionary

department

Column	Type	Null	Default	Links to
dname	char(15)	Yes	*NULL*	
dnumber *(Primary)*	char(1)	No		
mgrssn	char(9)	Yes	*NULL*	
mgrstartdate	char(9)	Yes	*NULL*	

Indexes

Keyname	Type	Unique	Packed	Column	Cardinality	Collation	Null	Comment
PRIMARY	BTREE	Yes	No	dnumber	3	A	No	

dependent

Column	Type	Null	Default	Links to
essn	char(9)	No		
dependent_name	char(12)	No		
sex	char(1)	Yes	*NULL*	
bdate	char(9)	Yes	*NULL*	
relationship	char(10)	Yes	*NULL*	

- Query1 van les 12

EXPLAIN SELECT E.Fname, E.Lname, E.Address **FROM** employee2 E, department D **WHERE** D.Dname = 'Research' **AND** D.Dnumber = E.Dno

Zonder index

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	D	ALL	PRIMARY	*NULL*	*NULL*	*NULL*	3	Using where
1	SIMPLE	E	ALL	*NULL*	*NULL*	*NULL*	*NULL*	8	Using where; Using join buffer (flat, BNL join)

Id: 1e SELECT statement (er is maar 1)
SIMPLE: simple SELECT (geen UNION of subqueries)
Table: eerst wordt D (= department) gelezen, dan E (= employee)
Type = ALL: full table scan, for each combination of rows from previous table
Possible keys; key (rij 1): primary key bestaat op department en is een kandidaat aangezien dat er een constraint op bestaat in de query, maar wordt niet gebruikt
Ref = NULL: kan met niets vergeleken worden
Rows: MySQL verwacht alle tupels van de tabellen (D: 3, E: 8) te moeten lezen
8 * 3 vergelijkingen verwacht
Using where: zoeken in tupels ("condition not checked at storage engine level, but all records returned and filtered by mysql")
Using join buffer (*block nested loop*): tupels van D moeten niet noodzakelijk opnieuw gelezen worden; een E tupel haalt te matchen D tupel(s) uit buffer indien mogelijk

Met index: (enkel wat veranderd is t.o.v. zonder index)
create index dname_i on department (Dname)

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	D	ref	PRIMARY,dname_i	dname_i	16	const	1	Using where; Using index
1	SIMPLE	E	ALL	*NULL*	*NULL*	*NULL*	*NULL*	8	Using where; Using join buffer (flat, BNL join)

Type = ref: gebruik van de index (key dname_i) met referentie naar een constante waarde ('Research')
Possible keys; key (rij 1): kandidaten, alleen dname_i is zinvol en wordt ook gebruikt
Rows: DBMS verwacht maar 1 D tupel en alle E tupels te moeten lezen
1 * 8 vergelijkingen worden verwacht
Using index:
* "the columns selected are part of an index that is used to return the results,
* there is no need to read the full table record
* and the columns will be returned using the already read index record"

- Query2 van les 12

EXPLAIN SELECT Lname, Fname **FROM** employee **WHERE** Salary > (**SELECT MAX**(Salary) **FROM** employee **WHERE** Dno = 5)

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	PRIMARY	employee	ALL	*NULL*	*NULL*	*NULL*	*NULL*	8	Using where
2	SUBQUERY	employee	ALL	*NULL*	*NULL*	*NULL*	*NULL*	8	Using where

Select_type = PRIMARY: outermost SELECT, buitenlus
Select_type = SUBQUERY: 1st SELECT in subquery

- Query4 van les 12 zonder primaire sleutel (en employee 2)

EXPLAIN SELECT E.Lname, Hours **FROM** employee2 E, works_on WO **WHERE** E.Ssn = WO.Essn **AND** (WO.Pno = 1 **OR** WO.Pno = 10)

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	e	ALL	*NULL*	*NULL*	*NULL*	*NULL*	8	
1	SIMPLE	wo	ALL	*NULL*	*NULL*	*NULL*	*NULL*	15	Using where; Using join buffer (flat, BNL join)

Id: 1e SELECT statement (er is maar 1)
SIMPLE: simple SELECT (geen UNION of subqueries)
Table: eerst wordt E (= employee) gelezen, dan WO (= works_on)
Type = ALL: full table scan, for each combination of rows from previous table
Possible keys; key : geen (en daarom ook geen key_length)
Ref = NULL: kan met niets vergeleken worden
Rows: MySQL verwacht alle tupels van de tabel E (8) te moeten lezen, en alle 16 van WO voor elk van hen, 8 * 16 vergelijkingen verwacht
Using where: zoeken in tupels ("condition not checked at storage engine level, but all records returned and filtered by mysql")
Using join buffer (*block nested loop*): tupels van E moeten niet noodzakelijk opnieuw gelezen worden; een WO tupel haalt te matchen E tupel(s) uit buffer indien mogelijk

Totaal # gelezen rijen = 8 + 8*15 = schatting door MySQL (eigenlijk 8 + 8*16)

- Query4 van les 12 (met employee)

EXPLAIN SELECT E.Lname, Hours **FROM** employee E, works_on WO **WHERE** E.Ssn = WO.Essn **AND** (WO.Pno = 1 **OR** WO.Pno = 10)

Standaard (zonder indexen/foreign key)

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	WO	ALL	*NULL*	*NULL*	*NULL*	*NULL*	16	Using where
1	SIMPLE	E	eq_ref	PRIMARY	PRIMARY	9	Company.WO.essn	1	

Type = eq_ref: gebruik van de PRIMARY key van E (Ssn),
waarbij WO.Essn gelijk moet zijn aan deze waarde, zie join conditie;
➔ alleen 1 row van E verwacht te lezen per (in stap 1) gelezen tupel van WO

Met secundaire index -- zonder FK:

create index pno_i on works_on (Pno)

```
explain SELECT E.Lname, Hours FROM employee E, works_on  WO WHERE E.Ssn = WO.Essn AND ( WO.Pno = 1 OR WO.Pno = 10 )
```

[Edit inline] [Edit] [Skip Explain SQL] [Analyze Explain at maria

➔ Index niet gebruikt,
zoals in onze eerdere theoretische analyse

+ Options

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	WO	ALL	pno_i	*NULL*	*NULL*	*NULL*	15	Using where
1	SIMPLE	E	eq_ref	PRIMARY	PRIMARY	9	company_small.WO.essn	1	

Opm: index niet gebruikt, kan wel, maar als we bij de disjuncte OR veel waarden/resultaten moeten wegschrijven kan het evengoed zonder index

Met secundaire index -- met FK:

```
alter table works_on add CONSTRAINT `fk_1`
FOREIGN KEY (`Essn`) REFERENCES `employee` (`Ssn`) ON DELETE CASCADE
```

```
explain SELECT E.Lname, Hours FROM employee E, works_on WO WHERE E.Ssn = WO.Essn AND ( WO.Pno = 1 OR WO.Pno = 10 )
```

+ Options

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	E	ALL	PRIMARY	*NULL*	*NULL*	*NULL*	8	
1	SIMPLE	WO	ref	fk_1,pno_i	fk_1	9	Company.E.ssn	1	Using where

Foreign key constraints en indexen in mySQL

MySQL vereist indexen op foreign keys en referenced keys zodat key checks geen hele table scan vereisen. Zo'n index wordt automatisch gecreëerd in de 'referencing table' als die nog niet bestaat.

⇒ Foreign Key ⟶ index

Handmatige queryoptimalisatie

Twee kleine voorbeelden (voor de rest enkel referenties naar extra links/documentatie:

… **STRAIGHT JOIN** … **ON** …: eisen dat linkse relatie eerst wordt gelezen (kleinste tabel links dus)

… **FORCE INDEX** (indexnaam) **JOIN** … **ON** …: eisen dat MySQL deze index gebruikt

Te onthouden 3

woensdag 17 juni 2020 16:30

Transacties

Inleiding tot concurrentiecontrole en herstel

- Transactie = uitvoering van een programma dat de gegevensbank raadpleegt of wijzigt
- Interleaved uitvoering = 1 processor behandelt afwisselend verschillende transacties
- Simultane uitvoering = meerdere processoren werken in parallel

Concurrentie

- = gelijktijdige verwerking van transacties kan problemen veroorzaken
  - Verloren aanpassingen (T2 doet T1 teniet)
  - Tijdelijke aanpassing (dirty reads)
  - Foutiieve sommering (gebruik van inconsistente waarden door aggregaatfunctie)
  - Niet herhaalbare lezing (waarde verandert tussen 2 reads kort na elkaar)

Herstel

- Een transactie moet ofwel volledig uitgevoer worden ofwel helemaal niet
- Bij een fout: oorspronkelijke toestand moet hersteld worden
- Mogelijke falingen
  - Computer-crash (inhoud van geheugen verloren)
  - Transactie- of systeemfout (verkeerde parameter, overflow, deling door 0…)
  - Uitzonderingscondities (bestand kan niet gelezen worden)
  - Opgelegd door concurrentiecontrole (T afgebroken wegens deadlock)
  - Schrijf-fout (beschadigd spoor)
  - Fysieke problemen, catastrofes (brand, stroomonderbreking)

Transacties: begrippen

- Transacites
  - Read-only transactie => ophalen van gegevens
  - Update of read-write transactie => aanpassing van gegevens
- Granulariteit van lees- en schrijfbewerkingen
  - Lezen en schrijven van/naar geheugen altijd per blok
  - Gegevenselement X in transactie: blok of attribuut
  - DMBS buffer: blok
- Status van de transactie (bepaald door operaties)
  - BEGIN_TRANSACTION
  - READ/WRITE
  - END_TRANSACTION (controle=> transactie doorvoeren of ongedaan maken)
  - COMMIT_TRANSACTION
  - ROLLBACK (of ABORT)
- Systeemlog
  - = alle transactie operaties worden bijgehouden in log om zo nodig te kunnen herstellen
  - Twee mogelijkheden voor herstel
    - Transactie volledig ongedaan maken -> log achterwaarts doorlopen, UNDO alle writes
    - Transactie goed afwerken -> log voorwaarts doorlopen, REDO alle writes
    - Kiezen? -> commit point
- ACID properties = gewenste eigenschappen transacties
  - Atomicity (transactie volledig uitgevoerd, of helemaal niet) => herstel
  - Consistensy preservation (behoud van een consistente gegevensbank) => programmeur, DBMS
  - Isolation (effect van transactie moet zijn alsof het de enige transactie is) => concurrentiecontrole
  - Durability (effect van transactie moet persistent zijn) => herstel

Transactieroosters

- = operaties van meerdere transacties in chronologische volgorde opgeschreven
- Conflicterende operaties:
  - Horen bij verschillende transacties
  - Gebruiken hetzelfde gegevenselement
  - Minstens 1 ervan is een write_item
- Een rooster S voor n transacties T_i is volledig:
  - S bevat alle operaties van alle transacties met inbegrip van een commit of abort als laatste transactie
  - Elk paar operaties van één transactie T_i in dezelfde volgorde voorkomt in S als in T_i
  - Voor elk paar conflicterende operaties ligt de volgorde éénduidig vast

Herstelbaarheid van rooster

- **Herstelbaar** = een transactie die gecommit is moet nooit meer ongedaan gemaakt worden
  - Voldoende voorwaarde: T commit enkel na commit van elke transactie die een waarde schrijft die T leest
  - Mogelijk "cascading rollback" = tijdsrovend

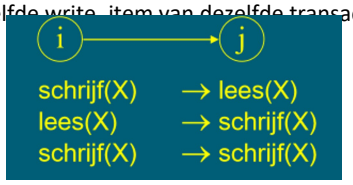
    - Minst restrictief
- **Cascadeloos** = garanderen dat er geen cascading rollbacks nodig zijn
    - Voldoende voorwaarde: elke transactie T leest enkel waarden geschreven door transacties die al gecommit hebben
    - Meer restricief
- **Strikt** = elke transactie T leest en schrijft enkel items na commit of abort van de laatste transactie die dat item geschreven heeft
    - UNDO write_item = gewoon oorspronkelijke waarde terugzetten
    - Meest restrictief, maar eenvoudigst herstelbaar

Serialiseren van roosters

- Serieel rooster
    - Tuseen eerste en laatste opdracht van een transactie T worden geen opdrachten van eender welke andere transactie uitgevoerd (geen interleaving)
    - Indien de transacties onafhankelijk zijn, is elk serieel rooster correct
    - Nadeel = beperking op concurrentie

Serialiseerbaarheid van rooster

- **Serialiseerbaar** = het rooster is equivalent met een serieel rooster met dezelfde n transacites
    - Resultaat-equivalentie = gegeven beginvoorwaarde, zelfde resultaat (te zwak!)
    - Conflict-equivalentie= volgorde van 2 conflicterende operaties is steeds dezelfde in beide roosters
    - View-equivalentie = roosters zijn view equivalent als:
        - Voor elke read_item(X) geldt: de laatste write_item(X) voor die read_item(X) moet in beide roosters dezelfde write_item van dezelfde transactie zijn
          => elke leesopdract in S1 leest dezelfde waarde als overeenkomstige leesopdracht in S2
        - Voor elke X waarvoor een write_item(X) voorkomt: de laatste write_item(X) moet dezelfde write_item van dezelfde transactie zijn in beide roosters
          => laatst geschreven waarde voor een item is dezelfde in beide roosters



- **Conflict serialiseerbaar** = conflict-equivalent met een serieel rooster
    - Testen d.m.v. precedence graph = graaf die volgorde van transacties aanduidt
      => rooster is serialiseerbaar a.s.a. de graaf geen cycli bevat
    - Equivalent rooster S' bekomen door topologisch te sorteren
      => als er een boog (T_i, T_j) bestaat, moet T_i voor T_j komen
- **View serialiseerbaar** = view equivalent met een serieel rooster
- Verschil view-equivalentie en conflict-equivalentie
    - Zijn allebei hetzelfde indien "constrained write" aanname geldt
        - CWA: aan elke write_item(X) gaat een read_item(X) vooraf, en de geschreven waarde hangt enkel af van de gelezen waarde
        - Bij ''unconstrained write" is view-equivalentie minder restrictief dan conflixt-equivalentie

Testen of verzekeren van serialiseerbaarheid

- Problemen met testen
    - Interleaving van operaties niet vooraf te voorspellen
    - Transacties continu aangeboden (begin/einde moeilijk te voorspellen)
    - Indien niet serialiseerbaar: herstel (duur)
- Oplossingen
    - Gebruik bij opstellen van transacties regels om seriealiseerbaarheid te vermijden

Transacties in SQL

- Dirty read
- Nonrepeatable read
- Phantom = record wordt zichtbaar bij een 2de keer lezen van een tabel

Concurrentiecontrole

Vergrendeling (locking)

- Grendel = variabele horend bij een gegevenselement in de gegevensbank
    - Beschrijft status van dat element t.o.v. mogelijke bewerkingen die erop kunnen worden uitgevoerd
- Soorten grendels
    - Binaire grendels = 2 mogelijke toestanden
        - 1: X is niet toegankelijk, 0: X is toegankelijk
        - Lock_item(X), Unlock_item(X) => bewerkingen zijn steeds atomair
    - Gedeelde/exclusieve grendels (read/write locks) = 3 mogelike toestanden
        - Niet vergrendeld, lees-grendel, schrijf-grendel
        - Read_lock(X), Write_lock(X), Unlock(X)
- Regels voor binaire grendels
    1) T moet lock_item(X) uitvoeren voor read_item(X) of write_item(X)
    2) T moet unlock_item(X) uitvoeren nadat alle read_item(X) en write_item(X) van T zijn uitgevoerd
    3) T mag geen lock_item(X) uitvoeren als het al een grendel op X heeft
    4) T mag geen unlock_item(X) uitvoeren als het geen grendel op X heeft
- Regels voor lees/schrijf-vergrendeling

1) T moet read_lock(X) of write_lock(X) uitvoeren voor eender welke read_item(X)
2) T moet write_lock(X) uitvoeren voor write_item(X)
3) T moet unlock(X) uitvoeren nadat alle read_item(X) en write_item(X) uitgevoerd zijn
4) T mag geen read_lock(X) uitvoeren als het al een grendel heeft op X
5) T mag geen write_lock(X) uitvoeren als het al een grendel heeft op X
6) T mag geen unlock(X) uitvoeren als het geen grendel heeft op X

=> afzwakken 4 en 5: conversie van grendels
- ◇ Als T al een leesgrendel op X heeft, en het is de enige transactie met een leesgrendel op X, dan kan dit met een write_lock(X) een schrijfgrendel worden
- ◇ Als T een schrijfgrendel op X heeft, kan dat met read_lock(X) verlaagd worden tot een leesgrendel

- Gebruik van de regels vermijdt bepaalde problemen maar garandeerd geen serialiseerbaarheid

Twee-fasen-vergrendeling
- = alle vergrendelingen van een transactie gebeuren voor de eerste ontgrendelbewerking
- 2 fasen
  - Expanding phase: plaatsen grendels
  - Shrinking phase: vrijgeven grendels
  - Fasen zijn strikt geschijden!
- Garandeert serialiseerbaarheid
- **Deadlock** = 2 processen wachten op elkaar
  - Hebben elk grendel die de ander wil
  - Geven die grendel niet vrij voor ze de andere grendel krijgen
  - Benaderingen van deadlock: deadlock prevention en deadlock detection
- Varianten van 2FV
  - Basisversie: zoals gezien
  - Conservatieve 2FV = alle grendels plaatsen voor transactie begint
    - Vermijdt deadlocks
    - Probleem: niet altijd bekend welke grendels nodig zullen zijn
  - Strikte, resp. rigoureuze 2FV = geen enkel schrijfgrendel, resp. grendel vrijgeven voor commit of abort
    - Garandeert een strikt rooster
    - Niet deadlock-vrij

Tijdstempels en multiversie-technieken

Deadlock-preventie met grendels en tijdstempels
- Lussen in wachtpatroon vermijden door wachten maar in 1 richting toe te laten
- Jongere transacties moeten worden afgebroken als er in de andere richting gewacht zou worden
  - Minder locks
  - Minder lees/schrijf operaties gedaan
  - Bij herstart behoudt de transacties zijn originele time stamp
- **Wait-die schema** = wanneer de jongere een lock wilt nemen op X waar een oudere een lock op heeft, sterft de jongere
  - Meer aborts, maar vaak nog (bijna) niks gedaan
  - Deadlock vrij
  - Sommige transacties afgebroken zelfs al zouden ze geen deadlock veroorzaken
- **Wound-wait schema** = wanneer de oudere een lock wilt nemen op X waar een jongere een lock op heeft, wordt de jongere vermoord door de oudere
  - Minder aborts, maar meer verloren werk
  - Deadlock vrij
  - Sommige transacties afgebroken zelfs al zouden ze geen deadlock veroorzaken

Deadlock-preventie zonder tijdstempels
- Niet wachten (no waiting)
  - Als een transactie een grendel niet kan krijgen, wordt ze direct afgebroken en na een zekere tijd herstart
- Voorzichtig wachten (cautious waiting)
  - Transactie mag alleen wachten op een grendel als de transactie die die grendel heeft niet zelf aan het wachten is
  - Deadlock vrij!
- Time-outs
  - Als een transactie langer wacht dan een welbepaalde tijd, wordt ze automatisch afgebroken en herstart
  - Detecteert niet echt deadlocks

Deadlock-detectie
- Periodieke controle of systeem in deadlock is interssant wanneer er weinig interferentie tussen transacties is
- Op basis van 'wacht op"-graaf
  - Lus in graaf = deadlock
- Indien deadlock
  - Kies slachtoffer om af te breken (victim selection)
    - Bij voorkeur jongere transacteis of transacties die in meerdere cycli in de graaf betrokken zijn
    - Oppassen voor starvation!
- **Starvation** = een transactie moet steeds maar blijven wachten, terwijl de andere transacties vooruitgaan

- Reden: andere transacties krijgen steeds vrijkomende grendel of altijd slachtoffer van victim selection
- Oplossingen:
  - Eerlijke toekenning van grendels (first come, first serve)
  - Eerlijke slachtofferselectie (selecteren transactie met laagste prioriteit)

Concurrentie d.m.v. tijdstempels (zonder grendels)

- = serialiseerbaarheid garanderen door tijdstempels (geen grendels nodig => geen deadlock)
- Er is 1 tijdstempel per transactie (uniek)
- Transactie rooster met tijdstempelordening is equivalent met serieel rooster met precies die volgorde van transacties
- Elk item X heeft:
  - Read_TS(X) = grootste tijdstempel van alle transacties die met succes X gelezen hebben
  - Write_TS(X) = grootste tijdstempel van alle transacties die met succes X geschreven hebben
- Bij afbraak transactie
  - Ongedaan maken
    - Cascading rollback
  - Opnieuw aanbieden met latere TS
- Vermijdt deadlock, maar geen starvation
- Transactie T voert **write_item(X)** uit:
  als read_TS(X) > TS(T) of write_TS(X) > TS(T):
  (write_item komt te laat, jongere transacties hebben intussen al een oudere waarde van X gelezen of een jongere geschreven)
  breek T af en maak T ongedaan
  anders
  write_item(X);
  write_TS(X) := TS(T)

  Transactie T voert **read_item(X)** uit:
  als write_TS(X) > TS(T):
  (te lezen waarde is intussen al overschreven door jongere transactie)
  breek T af en maak T ongedaan
  anders
  read_item(X);
  read_TS(X) := max(TS(T), read_TS(X))
- Problemen: cascading rollbacks en niet herstelbaar
- **Strikte tijstempelordening**
  - Een transactie T die een read_item(X) of write_item(X) doet met TS(T) > write_TS(X) = TS(T') van een andere transactie wacht met deze operatie tot de transactie T' met timestamp write_item(X) gecommit of afgebroken is
  - Garandeert strikt rooster

Multiversietechnieken

- Meerdere versies van een item X door een het systeem bewaart
  - Elke Xi heeft
    - Read_TS(Xi)
    - Write_TS(Xi)
- Voordeel: laat view-serialiseerbare roosters toe => meer concurrentie
- Nadeel: meer gegevens bijhouden + timestamps
- Transactie T wil een **write_item(X)** uitvoeren:
  zij X_i de versie van X met de grootste write_TS(X_i) <= TS(T)
  als TS(T) < read_TS(X_i) *(d.w.z. versie X_i zou dan moeten gelezen zijn nadat T geschreven heeft)*
  dan breek T af en maak T ongedaan
  anders
  creëer nieuwe versie X_j van X
  read_TS(X_j) := TS(T)
  write_TS(X_j) := TS(T)

  Transactie T wil een **read_item(X)** uitvoeren:
  zoek de versie i van X met hoogste write_TS(X_i) <= TS(T)
  geef de waarde van X_i terug aan T
  read_TS(X_i) := max(TS(T), read_TS(X_i))

Optimistische concurrentiecontrole

- Er gebeurt geen controle terwijl transactie wordt uitgevoerd
- Alle aanpassingen gebeuren in lokale kopies van gegevens
- Einde transacite: valideringsfase
  - Serialiseerbaarheid voldaan => commit transactie
  - Niet voldaan => herstart transactie later
- 3 fasen
  - Leesfase = T kan gegevensbank lezen en aanpassingen maken in lokale kopies
  - Valideringsfase = controle op serialiseerbaarheid
  - Schrijffase = als controle positief blijk, worden aanpassingen in GB geschreven
- Voordeel: alle controles voor transacties in 1 keer (handig bij kleine interferentie tussen transacties)
- Nadeel : bij grote interferentie moeten veel transacties herstart worden

Granulariteit van items

- Keuze van item
  - Gegevensbank, bestand, blok op schijf, record, veld in record
- Hoe groter item, hoe minder concurrentie mogelijk
- Hoe kleiner item, meer items
  - Meer grendels, lock/unlock bewerkingen, tijdstempels
  - Extra ruimte en verwerkingstijd
- Ideale granulariteit is afhankelijk van soort transactie
- Eventueel mogelijk om vergrendeling op verschillende niveaus van granulariteit uit te voeren
- Intention locks
  - **IS: intention shared locks**
    - Gedeelde grendels zullen gevraagd worden op lagere niveaus
  - **IX: intention exclusive locks:**
    - Een exclusieve grendel zal gevraagd worden op een lager niveau
  - **SIX: shared-intention-exclusive locks:**
    - Actuele knooppunt is in gedeelde mode vergrendeld, maar een exclusieve grendel zal op lager niveau gevraagd worden
- Compatibiliteitsmatrix

Geeft aan of een transactie T een knooppunt kan locken indien daar al een lock op staat

S shared lock
X exclusive lock

	IS	IX	S	SIX	X
IS	yes	yes	yes	yes	no
IX	yes	yes	no	no	no
S	yes	no	yes	no	no
SIX	yes	no	no	no	no
X	no	no	no	no	no

- Meervoudige granulariteitsvergrendelingsprotocol
  1. de vergrendeling moet rekening houden met de compatibiliteit
  2. de wortel van de boom moet eerst vergrendeld worden
  3. een knooppunt N kan slechts vergrendeld worden door een transactie T in S of IS modus indien het ouder knooppunt van N reeds vergrendeld is in IS of IX modus
  4. een knooppunt N kan slechts vergrendeld worden door een transactie T in X, IX of SIX modus indien het ouder knooppunt van N reeds vergrendeld is in IX of SIX modus
  5. een transactie T kan een knooppunt slechts vergrendelen indien het geen knooppunt ontgrendeld heeft (om aan 2-fase protocol te voldoen)
  6. een transactie T kan een knooppunt N enkel ontgrendelen indien geen van de kinderen van N op dat ogenblik vergrendeld zijn door T

Herstel

Herstel concepten

- Doel van herstel = gegevensbank terugzetten naar laatste consistente status voor het valen
- Informatie hiervoor is opgeslagen in system log
- 2 types:
  - Herstel van back-up kopie (bij grote schade)
  - Identificeer veranderingen die inconsistentie kunnen veroorzaken (kleine problemen)
    - REDO/UNDO

Herstelstrategieën

- **Uitgestelde updates**
  - Update database op schijf na commit point
  - NO-UNDO/REDO algoritme
    - Falen voor commit point: geen herstel nodig
    - Falen na commit point: REDO
- **Directe updates**
  - Updates database zijn mogelijk voor commit point
  - UNDO/REDO of UNDO/NO-REDO
    - Falen voor commit point: UNDO
    - Falen na commit point: REDO of geen herstel nodig

Crash tijdens herstel

- Nieuw herstel
- UNDO/REDO operaties moeten idempotent zijn
  - Herstel kan dus meermaals uitgevoerd worden

Caching (buffering)

- Caching: buffer per blok
- Cache directory: overzicht van buffers in de cache
- Gegevens element uit blok nodig => eerst cache checken

    - Niet in cache: laad blok als buffer in cache
    - Cache vol: flush andere buffer
- '**Dirty bit**' per buffer
    - 1: buffer is gewijzigd, 0: buffer is ongewijzigd
    - Bij flushing: enkel dirty buffers wegschrijven
- '**Pin-unpin bit**' per buffer
    - Gepinde buffer mag niet weggeschreven worden
    - Gebruikt door sommige herstelprotocol
- Twee caching strategieën
    1) **In-place updating**
        - Schrijft buffer weg naar origineel schijfadres
        - Overschrijft origineel blok
        - 1 versie van elk gegevenselement
    2) **Shadowing**
        - Schrijft buffer weg naar nieuw schijfadres
        - Meerdere versies gegevenselement
        - Weinig gebruikt in praktijk
- BFIM (before image) = oudere waarde gegevenselement
- AFIM (after image) = nieuwe waarde gegevenselement

Write-ahead logging
- System log is nodig voor in-place-updating
- Log moet op schijf staan voor BFIM overschreven wordt met AFIM
- Wat wordt gelogd?
    - REDO-type log entry: AFIM
    - UDNO-type log entry: BFIM
    - UNDO/REDO algortime: beide waarden
    - Cascading rollbacks mogelijk => lees_items zijn UNDO-type entries

Steal/No-Steal en Force/No-Force
- **Steal**
    - Buffer kan weggeschreven voor commit
    - Cache ruimte nodig voor andere transactie: buffer kan vervangen
    - No-steal
        - Gebruikt pin-unpin bit
        - UNDO niet nodig
- **Force**
    - Alle buffers moeten worden weggescreven voor commit
    - REDO niet nodig
- **Steal/no-force**
    - Steal => kleinere buffer ruimte nodig
    - No force => minder blokkentransport

Checkpoints
- Pauzeer actieve transacties
- Schrijf alle aangepaste buffers naar schijf
- Schrijf checkpoint record (=lijst van actieve transacties)
- Zet actieve transacties voort
- Bij herstel zijn REDOS van transacties die gecommit zijn voor laatste checkpoint niet nodig
- **Fuzzy checkpoints**
    - Checkpoinst kunnen lang duren wegens blokkentransport
    - Fuzzy checkpoints laat transacties terug sneller opstarten
    - Checkpoint wordt gemaakt in parallel met transacties
    - Zolang niet voltooid, wordt het vorige checkpoint gebruikt

Rollbacks
- **Rollback**
    - Transectie vaalt na update maar voor commit
    - Herstel oude waarden met UNDO
- **Cascading rollback**
    - Rollback transactie T, S heeft waarde gelezen die geschreven was door T => rollback S
    - Erg kostelijk
- Meeste herstelmechanismen vermijden cascading rollbacks
    - Strikte/cascadeloze roosters
    - Lees_item niet nodig in log

NO-UNDO/REDO herstel met uitgestelde update
- Uitgestelde updates

    - Nodige log entries: REDO met nieuwe waarde (voor het geval dat systeem faalt na commit point, maar voordat alle updates zijn weggeschreven)
- Protocol:
    1) Transactie voert geen updates uit op schijf, totdat commit point bereikt is
        - Alle gewijzigde buffers worden 'gepind' tot commit (no-steal)
    2) Transactie commit pas nadat alle REDO-type entries in log staan en de log weggeschreven is
        - Write-ahead logging

- Voordelen: simpele herstelprocedure: nooit UNDO
    - Niet voor herstel: uitgestelde update schrijft niets weg voor commit
    - Niet voor concurrentie: strikte 2FV geeft geen grendels vrij voor commit
- Nadelen:
    - Enkel bruikbaar bij korte transacties met weinig updates (bufferruimte)
    - Gelimiteerde concurrectie door 2FV

Hersteltechnieken voor directe update

Shadowing

- Schrijft buffer naar nieuw schijfadres
- Meerdere versies van gegevenselementen
- Zonder concurrentie: geen log nodig
- Zonder concurrentie
    - Directory: pointers naar blokken met gegevenselementen op schijf
    - Hou 2 directories bij
        - Huidige directory: recentste versies gegevenselementen
        - Shadow directory: originele versies gegevenselementen
    - Herstelprocedure NO-UNDO/NO-REDO
        - Geef nieuwe blokken terug vrij
        - Verwijder huidige directory
- Moeilijkheden
    - Na update: blokken veranderen va locatie op schijf (geralateerde blokken samen houden)
    - Mogelijks grote overhead bij commit
    - Garbage collection
    - Migratie oude naar nieuwe versie moet atomair gebeuren

ARIES herstelalgortime

- Herstelalgoritme van IBM
- Steal/no-force (UNDO-REDO)
- 3 concepten
    - Write-ahead logging
    - Geschiedenis herhalen bij REDO
    - Verandering loggen tijdens UNDO = vermijden om succesvolle UNDOs te herhalen bij crash tijdens herstel
- Systeem log
    - Records: update, commit, abort, undo en and
    - Elk record: LSN (log sequence number)
        - Elk blok in de database houdt LSN bij van laatste weggeschreven update

- Elk record bevat vorige LSN voor zijn transactie
  - Dit linkt de log records voor elke transactie in omgekeerde volgorde
- 2 tabellen worden bijgehouden naast de log
  - Transactie tabel: actieve transacties [ID, status, most recent LSN]
  - Gewijzigde blokken tabel= gewijzigde blokken in de buffer die nog niet zijn weggeschreven: [blok ID, LSN van recentste update]
  - Tabellen gemaakt tijdens fuzzy checkpoints

Algoritme

Analyse fase

Doel: Correcte tabellen verkrijgen tot einde log, zodat die de correcte en volledige informatie bevatten voor de REDO en UNDO stappen

Werkwijze:

- Vindt recentste checkpoint en verzamel zijn tabellen (bij end_checkpoint)
- Doorloop log van begin_checkpoint tot einde log
  - Bij 'end' record: verwijder deze transactie uit transactie tabel
  - Ander log record voor transactie:
    - Indien niet aanwezig, voeg deze transactie toe aan transactietabel
    - Update last LSN in transactietabel
    - Indien 'update' record: update gewijzigde blokken tabel met LSN

REDO fase

Doel: gegevensbank status hetzelfde maken als op het moment van de crash, door zo efficient mogelijk er voor zorgen dat alle updates werkelijk zijn uitgevoerd.

- Ook van niet gecommitte transacties
- Vermijden updates opnieuw uit te voeren

Werkwijze:

- Vindt de LSN M waarbij alle vorige updates zeker al zijn weggeschreven, of overschreven in buffer
  - Laagste LSN in gewijzigde blokken tabel
- Overloop log van M tot einde en herhaal update records indien nodig:
  - Op basis van LSN van record, LSN van blok in gewijzigde blokken tabel en LSN van blok op schijf wordt er bepaald of de update moet herhaalt worden

UNDO fase

Doel: Transacties die actief waren op het moment van de crash ongedaan maken

Werkwijze:

- Achteruit door de log gaan
- Voor elk 'update' record van een niet-gecommitte transactie in de transactietabel
  - UNDO update
  - Schrijf UNDO record in log