Databasemodellering og DBMS

Oppsummering

Alexander Nossum 2006
alexander@nossum.net

Innholdsfortegnelse

1. Modellering

1.1 ER vs. Relasjonsmodellen

ER-modellen er et diagram som abstraherer modellen fra den faktiske databasen.
Relasjonsmodellen er mer konkret og viser i detalj hvilke tabeller/relasjoner som skal lages. Her tas også beslutningen om hvordan relasjoner mellom to entiteter skal realiseres.

1.2 ER-modellen

1.2.1 Egenskaper

Entiteter

- Tabeller hvor data lagres i rader som inneholder felter/attributter

Attributter

- Felter hvor data blir lagret.

Primærnøkkel (PK)

- Unik identifikator for en rad i tabellen.

Fremmednøkkel (FK)

- Peker til en annen tabell som (ofte) inneholder relevant informasjon.

Kardinaliteter

- Begrensning på relasjonen mellom to (eller flere) tabeller.

Svake entiteter

- Entiter hvor radene ikke kan bli identifisert unikt av entitetens egne attributter. Primærnøkkelen dannes da av en fremmednøkkel fra en relatert entitet og en attributt fra entiteten selv. Primærnøkkelen kan også dannes ut i fra en garantert unikt løpenummer.

1.2.2 Relasjonstyper

En-til-mange (A til B)

- En rad i tabell B kan ha flere pekere til seg fra tabell A. “En B har flere A”.

Mange-til-mange (A til B)

- “Mange A har Mange B”
- Ikke realiserbart
- Løses ved å lage relasjonen mellom A og B til en egen “koblingstabell” som inneholder primærnøklene til både A og B og eventuell annen informasjon som kun oppstår idet relasjonen oppstår.

1.2.3 Subklassing

- Subklassing av tabeller er når man vil ha flere tabeller som deler mye lik informasjon. For eksempel entitetene Lærer og Elev kan subklasse fra Person, siden både en lærer og elev deler egenskapene som en person har.

1.2.3.1 Disjunkt vs. overlappende subklassing

- Disjunkt
  - Markeres med d mellom "super" og subklassene
  - Når en entitet kun kan tilhøre en og bare en subklasse.
  - For eksempel en person kan ikke være både lærer og elev.
- Overlappende
  - Markeres med o mellom "super" og subklassene
  - Når en entitet kan tilhøre flere av subklassene
  - En film kan være både være tegnefilm og barnefilm.

1.2.3.2 Delvis vs. fullstendig subklassing

- *Fullstendig subklassing*
  - Markeres med dobbel strek
  - Når en entitet må være en av subklassene
- *Delvis subklassing*
  - Markeres med enkel strek
  - Når en entitet ikke nødvendigvis trenger å være en av subklassene

1.2.4 3- og 4-veis relasjoner

- 3 eller 4 entiteter kobles sammen.
- Typisk en entitet som kobles til 2/3 andre med en-til-mange-relasjon
- For eksempel en person jobber på *en* avdeling og *en* lokasjon

1.3 Relasjonsmodellen

- Realiserer ER-diagrammet
- Relasjoner mellom entiteter blir realisert gjennom fremmednøkler og eventuelt egne tabeller.
- Notasjon: Person(persId, navn) Merk: fremmednøkler har ikke nødvendigvis noen egen notasjon.

1.3.1 Metoder for realisering av subklasser

1.3.1.1 ER-stil

- En tabell for hver subklasse
- Medfører at alle subtabellene får fremmednøkkel fra "super"-tabellen
- Kan føre til at en overlappende rad får tuppel i alle tabellene.

1.3.1.2 Objekt-orientert-stil

- Hver tabell "arver" alle attributtene til "super"-tabellen

1.3.1.3 Null-stil

- En tabell, med alle attributtene til alle subtabellene, hvor NULL er gyldig verdi på feltene.

2 Relasjonsalgebra

- Relasjonsalgebra er et matematisk språk for å kommunisere med et DBMS. SQL blir ofte oversatt til ekvivalent relasjonsalgebra når den kjøres i DBMS'et.
- Består av operander som gjør de mest vanlige operasjonene og relasjoner som operandene brukes på. Variabler kan også forekomme, disse representerer isåfall relasjoner.

2.1 Bag vs. sett

- Bag forhindrer duplikater i resultatet
- Sett har duplikater, men disse kan fjernes ved eksplisitt uttrykke det.
- SQL-resultater kommer ut i sett.

2.2 Kjernealgebra

Union, snitt, differans (U, ∩, -)

- Utfører vanlig sett-operasjoner på relasjoner med likt sett attributter og samme domene (likt relasjonsskjema)
- Baserer seg på matematisk logikk

Seleksjon ($\sigma_{vilkår}$)

- Velger ut bestemte rad(er) i relasjon(er) basert på vilkår

Projeksjon ($\pi_{vilkår}$)

- Velger ut bestemte kolonner ut i fra seleksjonen
- "Projiserer kolonner"

(Kartesisk) Produkt og join ($\sigma_{vilkår}(R \times S)$ eller $R|X|_{(vilkår)}S$)

- Kombinerer tupler fra flere relasjoner
- I praksis brukes til å få ut tilhørende informasjon fra tilhørende relasjoner
- *Natural-join*
  - Vilkåret settes automatisk (er logisk) basert på like attributtnavn i relasjonene.
- *Theta-join*
  - Er vanlig join med boolsk uttrykk.
  - Henter som regel ut flere rader (tupler)

2.3 Utvidet relasjonsalgebra

Gruppering og aggregering (γ)

- Gjør operasjoner på hele kolonner i en seleksjon.
- SUM(), COUNT() osv..

Ytre (outer) join

- Tar også med rader som ikke "joiner" med andre rader.
- Attributter som ikke joiner blir fylt med NULL-verdier.
- Vil typisk ha med relevant informasjon, men vil også ha med alt av informasjon, selv om de ikke er koblet sammen.
- Kan også være right/left join. Mer eksentrisk, left er default. For å styre hvilke kolonner som er "matchende".

3. Normalisering

3.1 Hvorfor

- Normalisering er en designteknikk for å planlegge gode relasjonsskjemaer
- Redundans
  - Ved redundans blir oppdateringer vanskelige - må oppdatere flere steder.
  - Sletting av rad betyr ikke nødvendigvis at informasjonen blir borte - kan være flere steder
  - Innsetting blir ofte problematisk. Må lagres flere ganger → redundans.
- Normalisering forhindrer redundans
  - Dekomponerer tabeller
  - Vil oppnå tapsløs join
    - Ved join av tabeller forsvinner ikke informasjonen
  - Vil bevare funksjonelle avhengigheter

3.2 Funksjonelle avhengigheter (FA/FD)

3.2.1 Nøkler

$\{A_1 , .. , A_n\}$ er nøkkel til R hvis:

1. Attributt $A_1, .. ,A_n$ bestemmer alle andre attributtene i R
2. Ingen delmengder $\{A_1 , .. , A_n\}$ bestemmer attributter i R → minimal nøkkel

3.2.2 Nøkkeltyper

- Kandidatnøkkel
  - Alle nøklene i en tabell
- Primærnøkkel
  - Brukerbestemt nøkkel
- Supernøkkel
  - Nøkkel som tilfredstiller betingelse 1. ovenfor. (Merk: Ikke nødvendigvis betingelse 2.)
  - Kan være en supernøkkel uten å være en minimal nøkkel!

3.2.3 Funksjonell avhengighet

- FA = en restriksjon
- En FA setter en restriksjon på hvilke rader (tupler) som kan finnes samtidig i en tabell.

Eks:

A	***B***	***C***
A1	B1	C1
A1	B1	C2
A1	B2	C3

FA'er
a → b gjelder ikke
b → a gjelder
a → c gjelder ikke
b → c gjelder ikke
c → a gjelder
ab → c gjelder ikke
bc → a gjelder
ac → b gjelder

3.2.4 Full funksjonell avhengighet

- X → Y er en full funksjonell avhengighet hvis vi ***ikke*** kan fjerne noe i X og ha:

$$(X - \{A\}) \rightarrow Y$$

Eks:

$$snr, pnr \rightarrow navn \text{ (delvis FA)}$$
$$snr \rightarrow navn \text{ (full FA)}$$

3.2.5 Utledning av FA'er

- For en mengde FA kan det utledes andre FA'er som også *må* gjelde.

Eks:

$$studnr \rightarrow pnr$$
$$pnr \rightarrow navn$$
$$\Rightarrow studnr \rightarrow navn$$

3.2.6 Tillukning (closure)

F^+ = $\{X \rightarrow Y \mid F \models X \rightarrow Y\}$
= Alle FA som kan utledes fra F
= FA som er samme når F er sant.

3.2.7 Armstrongs aksiomer — Utledningssystem

Reflexive:	Hvis X delmengde av Y så $X \rightarrow Y$
Augmentation:	Hvis $X \rightarrow Y$ så $XZ \rightarrow YZ$
Transitive:	Hvis $X \rightarrow Y$ og $Y \rightarrow Z$ så $X \rightarrow Z$

3.2.8 Triviell FA

$X \rightarrow Y$, der Y delmengde av X

3.2.9 Supernøkler og tillukning

- Tillukning av supernøkkelen er alle attributtene i tabellen
- Medfører vi kan teste om $\{A_1 , .. , A_n\}$ er en supernøkkel ved å beregne tillukningen.

3.3 Normalformer

3.3.1 1. NF

1. Attributtens domener inneholder atomiske (enkle) verdier, ikke mengder (arrays)
2. Verdien til attributtet er atomiske (enkle)

Eks:

avdeling(anr, navn, lokasjoner) → Ikke 1.NF

avdeling(anr, navn, lokasjon) → 1.NF

3.3.2 2. NF

1. Ingen ikke-nøkkelattributter er delvis funksjonelt avhengig av nøkkel

Eks:

Ikke 2NF

Utlån(bokId, boktittel, dato, låneId) , der bokID→boktittel, bokID,dato→låneId

2.NF (egentlig BCNF)

Bok(bokId, boktittel)

Utlån(bokId,dato, låneId)

3.3.3 3. NF

En tabell er på 3. NF hvis det for alle ikke trivielle avhengigheter er slik at:

$A_1...A_n \rightarrow B_1$

Der

1. $A_1...A_n$ er en supernøkkel

eller

2. B er et nøkkelattributt, altså en del av en nøkkel

Eks:

Ikke 3.NF

person(pnr, navn, adr, postnr, poststed) der pnr→Alle attrib og postnr→poststed

3.NF (alt.1)

person(pnr, navn, adr, postnr)

poststed(pnr, poststed)

Avhengighet mellom postnr og poststed.

Dårlig skikk! MEN: lovlig 3. NF

3.NF (alt.1)

person(pnr, navn, adr, postnr)

poststed(postnr, poststed)

3.3.4 Boyce-codd normalform (BCNF)

- En tabell R er på BCNF hvis det for alle ikke-trivielle FA'er:

  $A_1...A_n \rightarrow B_1$

  Så må

  1. $\{A_1, ... ,A_n\}$ være er en supernøkkel til R

- Forskjell mellom BCNF og 3. NF oppstår kun når det eksisterer overlappende kandidatnøkler.

Eks:

3.NF – ikke BCNF

eksamen(pnr, snr, fag, karakter) , der pnr↔snr og pnr,fag→karakter

BCNF

Student(pnr, snr)

eksamen(pnr, fag, karakter)

- Av og til vil det ikke finnes noen dekomponering til BCNF som er FA-bevarende (se lenger ned). Det vil alltid finnes en dekomponering til 3NF som er *FA-bevarende*, men hvor det kan oppstå noe *redundans*.

3.3.5 Tapsløs join

En dekomponering av:

$$R = \{R_1, ... , Rn\}$$

er tapsløs hvis det for alle instanser av R som tilfredstiller F (alle funksjonelle avhengigheter) er slik at:

$$\pi_{R1}(R) \,|X|\, ... \,|X|\, \pi_{Rn}(R) = R$$

dvs

$$\text{projiser}(R_1) \text{ JOIN } ... \text{ JOIN projiser}(R_n) = \text{Hele R}$$

"Hvert eneste attributt i R taes ut av R og slås sammen med JOIN for å bli hele R igjen"

- En dekomponering gir tapsløs join hvis de felles attributtene er en supernøkkel for en eller flere av prosjeksjonene (relasjonene?).

eks:

$R(\underline{a}, b, \underline{c})$, $a \rightarrow b$

$a_1\ b_1\ c_1$

$a_2\ b_1\ c_2$

dekomp1 (feil):

$R_1(\underline{a}, b)$, $a \rightarrow b$		$R_2(\underline{b, c})$	
a_1	b_1	b_1	c_1
a_2	b_1	b_1	c_2

R_1 |X| R_2 :

a_1	b_1	c_1	
a_1	b_1	c_2	← Adderende rad
a_2	b_1	c_1	← Adderende rad
a_2	b_1	c_2	

dekomp2 (riktig):

$R_1(\underline{a}, b)$, $a \rightarrow b$		$R_2(\underline{a, c})$	
a_1	b_1	a_1	c_1
a_2	b_1	a_2	c_2

R_1 |X| R_2 :

a_1	b_1	c_1	Ingen tap (addering) → tapsløs
a_2	b_1	c_2	

Dekomp1	***Dekomp2***
BCNF	BCNF
FA-bevarende	FA-bevarende
Tapsfull (adderende)	Tapsløs (ikke-adderende

3.3.6 Bevaring av FA

- Dekomponering av R = {R_1, ... , R_n}
- Mål:
  - Alle FA'er i F finnes i en eller flere av R_n-ene

Eks:

Person(<u>pnr,</u> postnr, poststed) , pnr→postnr, pnr→poststed, postnr→poststed

P_1(<u>pnr</u>, postnr) , pnr→postnr

P_2(<u>pnr</u>, poststed) , pnr→poststed

→ IKKE FA bevarende, mangler postnr→poststed

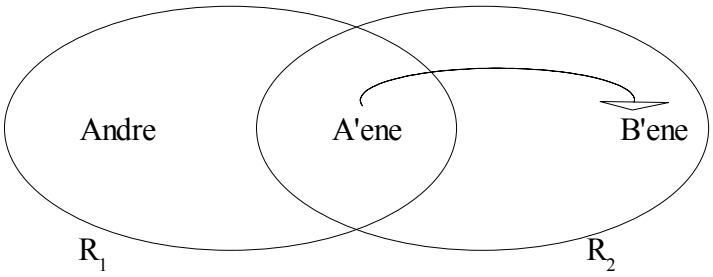
Dekomponering til BCNF

Finne en ikke triviell FA

$$A_1, \ldots, A_n \rightarrow B_1, \ldots, B_n$$

som bryter med BCNF, dvs:

{A_1, ... , A_n} er ikke en supernøkkel



Gir **<u>ALLTID</u>** tapsløs join

Eks:

Eksamen(<u>pnr,</u> snr, <u>fag</u>, karakterer) , pnr↔snr, snr_fag→karakter, pnr_fag→karakter

Supernøkler:

pnr, fag

snr, fag

Dekomponering

Student(<u>pnr</u>, snr) , pnr↔snr

Eksamen(<u>snr, fag</u>, karakter) , snr_fag→karakter

3.3.7 Flerverdiavhengigheter (FVA)

X—>>Y (X bestemmer en mengde Y-verdier)

- Mengden Y-verdier som er assosiert til en X-verdi i en tabell er bare avhengig av andre attributter i tabellen.

Eks:

R(X, Y, Z) , X—>>Y

Dersom vi har tuplene: $t_1(1,2,3)$

t2(1,3,5)

Må vi også ha tuplene: t3(1,2,5)

for at X—>>Y skal være oppfylt t4(1,3,3)

Eks2:

Relasjonen

Fag	***Lærer***	***Bok***
TDT4145	Kjetil	Rama
TDT4145	Kjetil	UML
TDT4145	Svein E	Rama
TDT4145	Svein E	UML

Fører til

Fag —>> Lærer

Fag —>> Bok

Triviell FVA

X —>> Y er triviell i en tabell dersom:

1. Y⊆X (Y delmengde av X) eller
2. X U Y (X union Y) utgjør alle attributtene i tabellen.

En triviell FVA innebærer ingen restriksjon.

3.3.8 4. NF (noe eksentrisk)

- En tabell R er på 4.NF med hensyn på F (funksjonelle avhengigheter) hvis, for alle ikke-trivielle FVA og F,

$$X \twoheadrightarrow Y \text{ er slik at X er en supernøkkel for R}$$

Eks:

Fag(<u>fag</u>, lærer, bok) , fag→lærer, fag→bok

lærer(<u>fag</u>, lærer) , fag→lærer

bok(<u>fag</u>, bok) , fag→bok

Dekomponering til 4.NF

- Finn en

$$A_1 , \ldots , A_n \twoheadrightarrow B_1 , \ldots , B_n$$

hvor

$\{A_1 , \ldots , A_n\}$ <u>ikke</u> er en supernøkkel

Dekomponering

R_1(A-ene, B-ene)

R_2(A-ene, ikke B-ene)

Eks (?):

Studveil(fag, student, studass) , fag→student, fag→studass

Studfag(<u>fag, student</u>) , fag→student

Studass(<u>fag, studass</u>) , fag→studass

4. SQL

- Svært høynivå kommunikasjonsspråk mot en database.

4.1 Mest brukte kommandoer

4.1.1 CREATE

- Lager en tabell i db'en
- CREATE TABLE [tabellnavn] (
  [feltnavn] [felttype] [parametere]
  );
- CREATE TABLE fag (
  fagkode VARCHAR(50),
  fag VARCHAR(100)
  PRIMARY KEY (fagkode)
  );

4.1.2 INSERT

- Legger til en rad i tabellen
- INSERT INTO [tabellnavn]([felter]) VALUES('[verdier]');
- INSERT INTO fag(fagkode, fag) VALUES('tdt4145', 'datamod');

4.1.3 SELECT

- Henter ut informasjon fra databasen.
- SELECT [attributter] FROM [tabell(er)] WHERE [boolsk uttrykk];
- SELECT fagkode, fag FROM fag WHERE fagkode = 'tdt4145';

4.1.4 UPDATE

- Oppdaterer en eller flere rader i tabellen
- UPDATE [tabellnavn] SET [feltnavn] = '[verdi]' WHERE [boolsk uttrykk];
- UPDATE fag SET fag = 'Databamodellering og databasesystemer' WHERE fagkode = 'tdt4145';

4.1.5 DELETE

- Sletter en eller flere rader i tabellen.
- DELETE FROM [tabellnavn] WHERE [boolsk uttrykk];
- DELETE FROM fag WHERE fagkode = 'tdt4145';

4.1.6 Andre operatorer

- IN, EXISTS, ALL, COUNT, JOIN .. ON, NATURAL JOIN ++++

4.2 Sub-spørringer/Nøstede spørringen

- En spørring kan inneholde sub-spørringer for å gjøre spørringen mer dynamisk i forhold til informasjonen allerede lagret i databasen.
- Resultatet fra sub-spørringen blir brukt i den delen av spørringen hvor sub-spørringen opptrer.
- Sub-spørringer kan opptre som felt i FROM-delen eller som verdi(er) i både "verdi"-delen og WHERE-delen
- SELECT fag FROM (SELECT * FROM ... WHERE ...) kunstigtabellnavn WHERE ... ;
- SELECT fagkode FROM fag WHERE fagkode = (SELECT fagkode FROM student WHERE snr = '2');
- INSERT INTO studentTarFag(fag, fagkode) VALUES (SELECT fag, fagkode FROM fag WHERE fagkode = 'tdt4145');

4.3 Views

- Views er kunstige tabeller som ofte kun kan leses fra (SELECT).
- Innholde i et view baserer seg på en SELECT-spørring mot databasen.
- Nyttig hvis man hyppig har bruk for en bestemt spørring, fks. SELECT * FROM fag NATURAL JOIN studentTarFag; og for å kontrollere rettigheter.
- INSERT og UPDATE kan veldig for bli veldig intrikat mot views, i praksis bør det aldri gjøres. Støttes dårlig i SQL92, noe bedre i SQL99

4.4 Restriksjoner

- Restriksjoner er regler som databasen forholder seg til ved bruk av databasen
- Setter regler på en tabell.
- CREATE TABLE fagkoder (

  fagkode VARCHAR(50) REFERENCES fag(fagkode)

  { ON DELETE CASCADE

  ON UPDATE NO ACTION }
  - ON DELETE CASCADE – sletter automatisk raden hvis tilhørende rad i fag.fagkode blir slettet
  - ON UPDATE NO ACTION – gjør ingenting hvis tilhørende rad i fag.fagkode blir oppdatert
- CREATE TABLE pris (

  pris int CHECK(pris > 1000)

  );
  - Sjekker om verdien stemmer overens med det boolske uttrykket
- Finnes mange forskjellige constraints, mange DBMS-spesifikke.

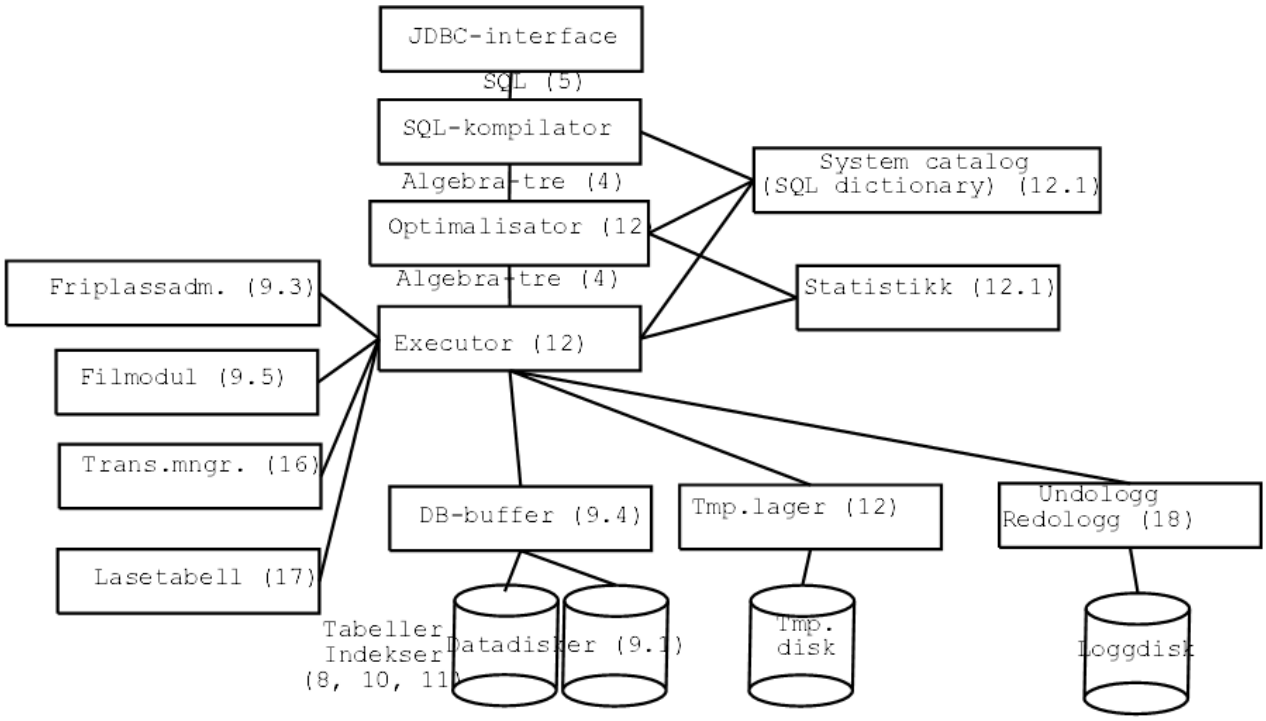
4.5 Triggere

- Mulighet for å spesifisere når noe skal utføres
- Tror dette er SQL/PL som er Oracle-spesifikt
- Baserer seg på EVENT-CONDITION-ACTION-rules
- CREATE TRIGGER fagTrig

```
AFTER INSERT ON fagkode
REFERENCING NEW as  Ny
FOR EACH ROW
WHEN (Ny.fagkode = 'tdt4145')
BEGIN
INSERT INTO idiFag
VALUES (:Ny.fagkode)
END;
```

5. Fysisk design

5.1 DBMS-arkitektur



5.2 Indekser og lagring

5.2.1 Clustered vs. Unclustered

- Clustered indekser baserer seg på den samme nøkkelen og lagrer typisk de faktiske postene ordnet på disk men i datastrukturen. Som for eksempel løvnoder i et tre.
  - Kalles også ordnet indeks siden postene ligger i "nesten" samme rekkefølge som de originalt ville ligget.
- Unclustered indekser kan basere seg på hvilken som helst nøkkel og lagrer typisk de faktiske postene uordnet utenfor datastrukturen.
- Dette kan føre til at clustered indekser er raskere enn unclustered.

5.2.2 B-trær

5.2.2.1 Typer B-trær

1. B-tre lagrer postene i datastrukturen dvs at vanlige poster ligger i løvnivået av treet mens resten er indeksposter (indeksnivåer). Løvnodene er en lenket liste, "first + next". Clustered B-tree kalt B+ tree.
2. B-tre lagrer postene i en heapfile (vanlig fil/haug). Unclustered B-tree.

- B+ tree er standard i DBMS. Kan gå fortere enn vanlig B-tree

5.2.2.2 Oppbygging/struktur

- Begynner med løvnodene
- Når en løvnode er full splittes den i to og en peker fra den "største" av de to nye løvnodene lager en "rotnode" hvor pekeren har en "mindre-enn" og en "større-enn" peker til de respektive nodene.
- Løvnodene må bli en lenket liste!
- Data i nodene er sortert! (B+)
- Antall poster i hver node bestemmes selv = "rangen"
- Antall nøkler = n^h der h er høyden på treet og n "rangen" til treet
- Veldig bra på range og sekvensiell aksess.
- God men ikke like god som hash på direkteaksess.

5.2.3 Hash

- Består av "bøtter" som inneholder blokker med pekere. Blokkene kan inneholde pekere til videre blokker.
- En nøkkelfunksjon bestemmer hvilken "bøtte" verdien skal inn i. Typisk [nøkkel] mod 5.
- Særdeles rask direkteaksess
- Oppdatering/vedlikehold er dyrt.

- Dårlig på rangesøk
- Lagrer ikke poster i datastrukturen.

5.2.4 Lagringsstrukturer

5.2.4.1. Heapfile

- Et "rålager" dvs. en vanlig fil uten noe mer om og men.
- RecordId = (BlokkNo, 1) , 1 = indeks i blokken
- God til tabell scan (full scan)
- Dårlig til direkteaksess, tilsvarer full scan.
- Dårlig på rangesøk.

5.2.4.2. B-trær

- Kan brukes til å lagre data strukturert (clustered) istedefor fks. en heapfile
- Kan også brukes som indeks!
- God på rangesøk (clustered).
- God på direkteaksess (struktur).
- Oppdatering/vedlikehold av struktur er dyrt.

5.2.4.3. Hashfil

- Brukes som oftest ikke som lagringsstruktur men som ren indeks.
- Særdeles god på direkteaksess!

5.2.4.4 Ulemper med RecordId

(BlokkId, Idx, blokken)

- BlokkId tar 6 byte, lagres i både deviceId og IndexDevice
- Dyrt å flytte poster → Må oppdatere alle indeksene!

Alternativt

- Clustered index
  - RecordId byttes ut med primærnøkkelen.

5.3 Kostnadsanalyse

5.3.1 Kostnader

- B = Antall blokker
- R = Antall poster pr. B (blokk)
- D = tiden for les eller skriv B på disk.

	Scan (hele tabellen)	***Likhetssøk (direkteaksess)***	***Range***	***Insert***	***Delete***
Heap-file	BD	0.5BD (gjsn. halve filen)	BD (søke igjennom hele filen	2D (lese + skrive)	Søk + D
Sortert fil	BD	$D\log_2(B)$ (binærsøk)	$D\log_2(B)$ + D*#treff	2*0.5BD (?)	2*0.5BD
Clustered files (b tre)	1.5BD (ca 50% ekstra blokker, 67% fyllgrad)	$\log_F(B)*D$ (høyden på treet)	$D\log_F(B)$ + D*#treff	$D\log_F(B) + D$	$D\log_F(B) + D$
Unclustered tre-index **	0.15*BD+BRD (scan index) eller BD (scan heap-file)	$D\log_F(0.15B) + D$ (søk i index)	$D\log_F(0.15B)$ + D*#treff eller BD (scan heap-file)	$D\log_F(0.15B) + D + D$ (scan idx + skriv idx + skriv blokk)	$D\log_F(0.15B) + D + D$ (scan idx + skriv idx + skriv blokk)
Unclustered hash-index ***	BD(R+0.125)	2D (les nøkkel + les post?)	BD	4D	Søk + 2D

** Antar indeksposten er 10% av datapostene. → størrelsen på indeksen er:

$$0.1*B*1.5B = 0.15B$$

*** Antar søkenøkkel + postId 10% = 0.125B ??

5.4 Valg av indekser

- Indekser er generelt lurt hvis det er mye mer lesing enn skriving til db'en.
- B-trær er best hvis det er en betydelig andel av rangesøk og/eller sekvensielle søk
- Hashindeks er best hvis det er en betydelig andel likhetssøk (direkteaksess) på attributten

6 Queryevaluering

6.1 Metoder for join

6.1.1 Simple nested loops

- Looper enkelt gjennom alle verdier til feltene i kravet for å se om det blir match.
- Krever ingen indeksering
- Fyfy! Gigantisk kostnad, men brukes antakligvis mye.

6.1.2 Index nested loops

- Looper igjennom tuplene på samme hvis som *simple nested loops* men oppslaget er fra en datastruktur så kostnaden på I/O blir mindre.

foreach tuple r in R do

foreach tuple s in S where $r_i = s_i$ *do*

add <r,s> to result

Eks:

R = 100.000 blokker, S = 40.000 blokker

$R|X|_{rid=sid}S$

Antar Hashindeks på sid i S

Scan R + (for hver post i R) * (oppslag i indeks + les heap)

1000 blokker + 100.000*(1.2+1) = 221000

Antar Hashindeks på rid i R

Scan S + (for hver post i S) * (oppslag i indeks + les heap)

500 blokker + 40.000*(1.2+(¤)) = 88500 (clustered)

der ¤ er enten 1 for clustered eller 2.5 for unclustered

2.5 kommer fra:

blokker i R / blokker i S = 100.000 / 40.000 = 2.5

6.1.3 Sort-merge-join

- Baserer seg på merge-sort, krever **ikke indeksering**
- Sorterer en stor fil på typisk 2-4 gjennomganger/pass (antar videre 2)
- Antall I/O = antall pass * 2 (les+skriv)

Eks:

Sorter R	2*2*1000	= 4000 I/O
Sorter S	2*2*500	= 2000 I/O

Flettesortere R og S	1000 + 500	= 1500 I/O (lese R og S)
Totalt	4000 + 2000 + 1500	= **7500** I/O UTEN IDX!!!

6.2 Planer

- Optimalisatoren lager endel planer og velger den planen som den tror er billigst

Eks:

Relasjoner

Sailors(sid, sname, rating, age)

Reserves(sid, bid, day, rname)

Spørring

```
SELECT S.sname
FROM Reserves R, Sailor S
WHERE
    R.sid = S.sid
    AND R.bid = 100
    AND S.rating > 5;
```

Relasjonsalgebra (relasjonstre)

π_{sname}

↓

$\sigma_{bid = 100 \text{ AND } rating > 5}$

↓

$|X|_{sid = sid}$ (nested loop)

↓ ↓

Reserves Sailors (scan)

Første plan

Simple nested loop:

Scan R 1000 + 1000*500 = 501000 I/O

eller

Scan S 500 + 500 *1000 = 500500 I/O

Andre plan

Push down selection:

π_{sname}

↓

$|X|_{sid = sid}$ (sort-merge-join)

↓ ↓

$\sigma_{bid = 100}$ $\sigma_{reting > 5}$

(Tmpfil $F_{1)}$ (Tmpfil $F_{2)}$

↓ ↓

Reserves Sailors (scan)

Scan R + Scan S = 500 + 1000

Skriv F_1 + Skriv F_2 = 10 blokker + 250 blokker (antar halvparten kvalifiserer)

Sorter F_1 = 2*2*10

Sorter F_2 = 2*4*250

Join-merge = 10+250

Totalt: 4060 I/O

7. Transaksjoner og concurrency-controll (cc)

- En DBMS støtter deling og samtidig aksess av data
  - Medfører transaksjoner og låsing.

Eks, samtidighetsproblem:

Relasjoner:

Selger(bar, øl, pris, volum)

Merk, ikke god modell, kun for eksempelets skyld!

Handling

Baren selger:

"øl1" til 38

"øl2" til 53

Anne finner den billigste og den dyreste ølen på Baren

1. SELECT MAX(pris) FROM selger WHERE bar='Baren';
2. SELECT MIN(pris) FROM selger WHERE bar='Baren';

Frank fjerner "øl1" og "øl2" og setter inn "øl3" til 63

3. DELETE FROM selger WHERE bar='Baren' AND (øl='øl1' OR øl='øl2');
4. INSERT INTO selger VALUES ('Baren', 'øl3', 63, '0.33');

Problem

Hvis sekvensen ble:

1,3,4,2

Ville Anne fått at den minste prisen er større enn den største.

1 = 53

3 = true

4 = true

2 = 63

Løsning:

Bruk av transaksjoner, i praksis, bruk av COMMIT

7.1 ACID – Egenskaper ved transaksjoner

- Transaksjon: Gruppering av operasjoner (SQL)

7.1.1 ACID

A – Atomiske

- Enten kjøres operasjonene fullstendig eller overhodet ikke

C – Consistency

- Overholder konsistenskrav
  - Primærnøkkel, References, Check, osv..

I – Isolation

- Transaksjonene er isolerte fra hverandre
  - Merker ikke at noen kjører i parallell

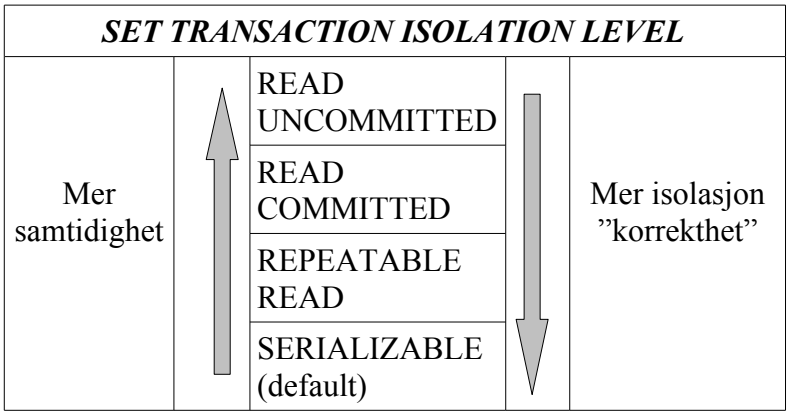
D – Durability

- Transaksjonene er permanente etter COMMIT

7.2 COMMIT/ABORT

- En transaksjon slutter med COMMIT
  - Alt gikk bra, endringer er garantert lagret
- Slutter med ROLLBACK (ABORT)
  - endringer rulles tilbake (undo)
- Autocommit
  - Hver SQL-setning er en egen transaksjon
  - Som oftest er dette default i DBMS

7.3 SQLs isolasjonsnivåer



7.4 Transaksjoner og samtidighet

- Vi ønsker samtidighet
  1. Mye samtidighet
     - Kan bruke CPU til en annen transaksjon så lenge
  2. Muliggjør for parallelle maskiner

7.5 Historie (Schedule), H/S

7.5.1 Historie

- Liste (sekvens av aksjoner fra en mengde transaksjoner
  - READ(A)
  - WRITE(A)
  - ABORT(A)
  - COMMIT(A)
- Fks: $R_1(A)$, $W_2(B)$, C_1 , C_2

7.5.2 Seriell historie

- En historie som ikke fletter aksjoner fra forskjellige historier

7.5.3 Serialiserbare historier

- En historie som har nøyaktig samme effekt på databasen som en seriell historie, uten å egentlig være det.

7.5.4 Problemer med flettet utføring

Dirty read

- Reading uncommitted data
- W-R-konflikt (WRITE-READ-conflict)

Eks

T₁	R(A), W(A)	R(B), W(B), ABORT
T₂	R(A), W(A), C	

T_2 leser og skriver før T_1 kjører abort (angrer)

T_2 får altså informasjon som ikke eksisterer

Unrepeatable reads

- Overwriting uncommitted data
- R-W-konflikt (READ-WRITE-conflict)

Eks

T1	R(A)	R(A), W(A), ABORT
T2	R(A), W(A), C	

T_2 skriver til A og Committer før T_1 er ferdig
T_1 leser først A og så noe annet fra A

7.6 Isolasjonsnivåer og problemer med flettet transaksjoner

- Problemer som kan oppstå ved de forskjellige isolasjonsnivåene

	Dirty read	***Unrepeatable read***	***Fantom***
Read uncommitted	JA	JA	JA
Read committed	NEI	JA	JA
Repeatable reads	NEI	NEI	JA
Serializable	NEI	NEI	NEI

- Metode for å unngå fantomer
  - Hvis T leser en mengde verdier basert på en søkebetingelse, så vil ikke denne mengden verdier endres før T er ferdig.

7.6 Låsing

7.6.1 Strict locking

- Strict 2PL – tofaselåsing (two-phase-locking)
  1. Hvis en transaksjon ønsker å lese (eller skrive) et objekt så må den sette en delt (evt. ekslusiv) lås på objektet før den kan utføre operasjonen.
  2. Alle låser som er holdt av en transaksjon slippes når transaksjonen er ferdig (commit).
- Strict 2PL tillater *kun* serialiserbare historier.

7.6.2 Implementasjon av låser

- Låsetabell i minnet
- Låsetyper
  1. Postlåser
  2. Blokklåser
  3. Tabellåser
  4. Verdiområdelåser
  5. Predikatlåser
- 3, 4 og 5 forhindrer fantomer

7.6.3 Vranglås

- Vranglås oppstår når to (flere) transaksjoner venter gjensidig på hverandres låser.
- T_1 venter på T_2 sin lås samtidig som T_2 venter på T_1 sin lås.
- Løsning
  1. Wait-for-graphs (komplekst)
     - Finn sykler i grafen og aborter noen transaksjoner
     - "bulletproof"
  2. Transaksjonstimeout (intuitivt enkel)
     - Hver transaksjon har en estimert timeout
     - Aborter hvis timeouten løper ut
     - Kan slå feil, lite sannsynlig ved god timeout-estimering.

7.6.4 Konfliktserialiserbarhet

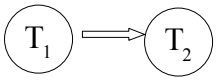
- Konflikt mellom to operasjoner hvis

Konflikt 1	Konflikt 2	Konflikt 3
$R_1(A)$	$W_1(A)$	$W_1(A)$
$W_2(A)$	$R_2(A)$	$W_2(A)$

- To historier er konfliktekvivalente hvis historiene kan bli like ved å bytte plass for aksjoner uten konflikter (swappe)
- En konfliktserialiserbar historie er konfliktekvivalent med en seriell historie.

7.6.5 Presedensgraf

- *Noder:*
  - Transaksjoner i en historie
- *Kanter:*
  - Oppstår mellom T_1 og T_2 når det finnes en aksjon i T_1 som er i konflikt med en aksjon i T_2 og T_1's aksjon skjer før T_2's



Eks

Transaksjoner

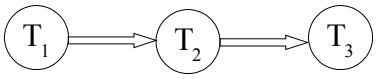
T_1 , T_2 , T_3

Historie:

$R_2(A), R_1(A), W_2(A), R_3(A), W_1(B), W_3(A), R_2(B), W_2(A)$

Konflikter:

$W_2(A) \rightarrow R_3(A)$
$W_1(B) \rightarrow R_2(B)$



- Hvis presedensgrafen har *sykler* er historiene *ikke* konfliktserialiserbare
- Bruker låser på dataelement for å garantere konfliktserialiserbarhet

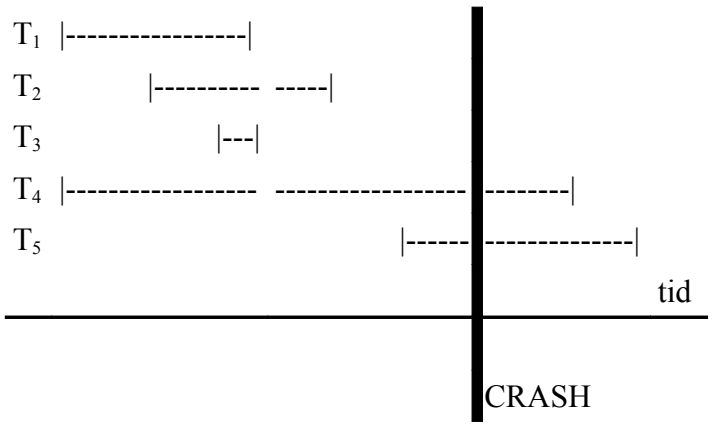
7.6.6 Mer om historier og samtidighetskontroll

- Les selv emner, wikipedia for de interesserte..
- Recoverable history (gjenopprettbar historie)
- Historier som unngår gallopperende abort (cascading abort), (ACA, ACR)

8. Logging & Recovery

- En DBMS støtter sikker atomisk aksess til store mengder data.
- Transaksjoner er
  - A – Atomiske: Etter commit har transaksjonene enten kjør fullstendig eller overhodet ikke.
  - C
  - I
  - D – Durable: Etter commit mistes ikke endringene

8.1 Transaksjoner etter recovery



T_1, T_2 og T_3 overlever - "vinner"
T_5 og T_5 aborteres - "taper"

8.2 Force/Steal – klassifisering av L&R algoritmer

- Hvor fleksibel (uavhengig) er buffermanageren din?
- Når kan sider skrives?
- Når må sider skrives?

8.2.1 Force

- Må skrive en skitten dataside til disk ved commit
- Tregt

8.2.2 Steal

- Kan en transaksjon stjele fra en annen transaksjon?
- ???

8.2.3 Egenskaper ved Force og Steal

	No Steal	***Steal***
Force	(shadowing) uten logg	undo (logging)
No Force	(redo logging)	ARIES (ønskelig)

8.2.4 Write-Ahead-Logging (WAL)

1. Skriv loggpost som endret en side til disk *før* du skriver dataside. (Atomicity, STEAL, tillater UNDO)
2. Før du committer en transaksjon. Skriv loggen til disk. (NO FORCE, Durability)

8.2.5 Logging & recovery

8.2.5.1 Loggbuffer

- Struktur til loggbufferen.

<table>
<tr><td></td><td>LSN, loggpost med nøkkel LoggSekvensNummer (LSN)</td><td></td><td></td><td></td></tr>
<tr><td colspan="2">Skrevet til disk</td><td colspan="3">FlushedLSN</td></tr>
</table>

8.2.5.2 DB-buffer

- PageLSN: LSN til loggposten som sist endret datasiden.
  - Lagres i datasiden
- Ved skriving av dataside til disk sjekkes PageLSN <= FlushedLSN
- Hvis ikke PageLSN <= FlushedLSN skrives loggen først.

8.2.5.3 Loggpost

Innhold

LSN	TransId	PrevLSN	Type	PageId	Offset/index	Before image	After image

- PrevLSN: LSN til forrige loggpost i samme transaksjon
- Type: INSERT, UPDATE, DELETE
- PageId: Hvilken dataside er berørt (ble oppdatert)
- Offset/index: Hvor på datasiden ble det skrevet.
- Before image: Verdi *før* oppdatering
- After image: Verdi *etter* oppdatering

8.2.5.4 Andre datastrukturer

- Transaksjonstabell
  - Registrerer aktive transaksjoner
  - For hver transaksjon
    - TransId
    - Tilstand (aktiv, aborting, committed ...)
    - LastLSN: Peker til nyeste loggpost

- DirtyPageTable
  - For hver skitten side i DB-buffer registreres:
    - RecLSN: Peker til første loggpost som gjorde siden skitten

8.2.5.5 Sjekkpunkting

- Periodisk lager DBMS'et et sjekkpunkt i loggen for å minimalisere recoverytiden.
- Sjekkpunktloggpost består av:
  - Transaksjonstabell
  - DirtyPageTable
  - Lagrer LSN til sjekkpunktloggposten på sikkert sted på disk (Master record)
  - I noen systemer er sjekkpunkting knyttet til skriving av skitne sider

8.2.5.6 Abortering av transaksjoner

- Skriv abort-loggpost i loggen
- Finn LastLSN fra transaksjonstabellen
- For hver loggpost i transaksjonen (fra nytt til gammelt)
  - Lag CLR (kompenseringsloggpost)
  - Gjør redo på CLR

8.2.5.7 Recovery etter krasj

a) Start fra siste sjekkpunkt i loggpost

b)

1. analyse – finn vinnere og tapere
2. REDO – redo av alle loggposter
3. UNDO – undo effekten av alle taper transaksjonene

- Redo av loggpost

if(loggpost.LSN > loggpost.PageLSN)
Skriv AFTER IMAGE inn i blokken;
Oppdater PageLSN til loggpost.LSN

9. Informasjonsgjenfinning (IR) vs. DBMS

- For spesielt interesserte.

IR	*DBMS*
Nøkkelordsøk	SQL
Ustrukturert data	Strukturert data
Hovedsaklig lesing	Relativt mye oppdateringer
Top-K-resultat (rangering)	Lager fullt (urangert) resultat

9.1 Tekstsøking

- Eksempeldokument:
  - D_1: "Database: Dette er et dokument om databaser"
- Typisk IR-data-modell:
  - Et dokument er en "bag" med ord
  - Høyfrekvente ord, stopp-ord, er ikke med i bag'en.
    - ("er", "in", "is" osv)
  - Stemming
    - Lagrer kun grunnstamme av bøyde ord
      - Databaser → Database
  - Boolsk tekstsøk:
    - "dokument" AND "database"
  - Resultatset blir rangert

9.2 Vektorrom-modellen (VSM)

- Lager frekvenstabell av ordene i alle dokumentene
- Term-frekvens (TF) av term *k* i et dokument *i* → H_{ik}
- Enkel spørring: Q("Kjetil" OR "morsomt")
- To treff.. Hva er det beste treffet?

9.2.1 TF-IDF

- Ord som er med i mange dokumenter er lite nyttig i søket og rangeringen
- IDF – Invers DokumentFrekvens av term *k* som er i n_k dokumenter av totalt N dokumenter:

$$\log(N/n_k)$$

- Vekt av term *k* i dokumentet

$$W_{ik}' = H_{ik} * \log(N/n_k)$$

- Lengdenormalisering – vekt bør være mindre i store dokument.

9.2.2 Semistrukturert data (XML)

- Data representert som graf
- Spørringer representert som path (XQuery)