

Minikompendium

TDT4145 – databasemod og dbsys

Pages og records

Her er det viktig å holde tunga rett i munnen så man ikke blander begrepene.

Page

Den minste dataenheten databasesystemet leser og skriver til disk. Kan forveksles med blokk, som på norsk ofte brukes som synonym for page. I læreboken brukes derimot page konsekvent.

Block

Den minste enheten selve disken opererer med for lesing og skriving. Det vil si disken adresseres på blokkbasis. Det er ikke uvanlig at størrelsen til en page settes lik størrelsen til en blokk.

Record

Synonymt med poster (eller rader), for eksempel (2964, Gunnar, Nilsen) i en persontabell.

File

For databasesystemet er en fil en samling med pages, eller en samling av records. En fil kan være en heap file, eller en indeks (clustered eller ikke).

Lagringsmetoder

Forskjellige måter å strukturere blokker på disk, som til sammen gir en fil.

Sekvensielle lagringsmetoder (Heap)

Det finnes fire alternativer:

- Fysisk sekvensiell, usortert
Blokkene ligger sekvensielt (etter hverandre) på disk, men postene i blokkene er usortert.
- Fysisk sekvensiell, sortert
Blokkene ligger sekvensielt (etter hverandre) på disk, og postene i blokkene er sortert.
- Logisk sekvensiell, usortert
Blokkene er logisk knyttet til hverandre vha lenkede strukturer, men postene er ikke sortert. Kan ikke finne blokkene uten å gå veien om kjeden.
- Logisk sekvensiell, sortert
Blokkene er logisk knyttet til hverandre vha lenkede strukturer, og postene er sortert. Kan ikke finne blokkene uten å gå veien om kjeden.

Direkte adressering

Som hashing bare at antall poster i hvert bønne uansett er maksimalt en. Alltid best hvis den kan brukes. Unngår bruk av nøkler - må bare markere at posten eksisterer. Må ta høyde for alle mulige forekomster. Må kunne forsvares ut fra plassforbruk. Hver verdi må entydig kunne bestemmes (f.eks regnes ut)

Eksempel: Tabell for hver dag i et år: Lager tabell med 12×31 rader (6 dager for mye). Da vil 1 januar ligge på rad 0 ($1 \times 31 + 1 - 32$). 15 februar vil ligge på rad 45 ($2 \times 31 + 15 - 32$). 17 mars vil ligge på rad 78 ($3 \times 31 + 17 - 32$). Og sånn kan man direkte slå opp alle datoer i året (veldig kostnadseffektivt). Gir alltid bare 1 I/O når den skal hente data, siden den allerede vet hvilken rad den er ute etter, og dermed kan hente den ut uten å søke.

Randomisering (hashing)

Adresse regnes ut av hashformelen (tar inn nøkkel som parameter). Hashformelen må gi adresser mellom 0 og $n-1$. Problem oppstår når flere nøkler gir samme hash verdi. Problemet må håndteres og den enkleste måten er å lage lenkede lister. Kan også bruke lineær probing og andre liknende teknikker, men disse er ikke pensum i dette faget.

Indeks sekvensiell lagring

Filen lagres som et clustered B+-tre. Dataene, som ligger i løvnodene, blir da automatisk sortert i logisk sekvens.

Indekser

Primary index

En indeks på nøkkelattributtet.

Clustered index

En clustered index er unik for hver tabell. Dvs. at man kan bare ha en clustered index per tabell. I en clustered index så vil løvnodene bestå av selve dataene.

Non-clustered index

Ved bruk av en non-clustered index vil ikke løvnodene inneholde faktiske data, men pekere til dataene.

Join-algoritmer

Vi vil joine to tabeller A og B. I hovedsak er det antall diskaksesser som er dyrt (~ 10 ms), så databasesystemet vil gjerne prøve å velge en join-strategi som er best mulig.

Pages Records per page

Tabell A 100 150

Tabell B 1000 50

Nested loops join

Den enkleste måten å joine A og B er å sjekke alle rader i kryssproduktet ($A \times B$) og legge til de som passer i resultatet. En **simple nested loops join** gjør nettopp dette, ved å gå gjennom alle poster i en ytre tabell (A) og for hver post sjekke alle poster i en indre tabell (B).

```
for alle poster i A:
  for alle poster i B:
    legg til raden fra (A x B) i resultatet dersom join-betingelsen
    stemmer;
```

Her må vi lese hele A, med en diskaksess per page blir det 100 aksesser. For hver av de 15000 postene i A må vi lese hele B, som er 1000 aksesser hver gang. Total kostnad blir derfor $100 + (100 * 150) * 1000$.

En **index nested loops join** utnytter indekser på join-attributtet i den indre tabellen (B). I stedet for å måtte scanne hele B for hver post i A, kan vi heller slå kjapt opp i indeksen til B for hver post i A. For hashindeks er kostnaden ~ 1.2 aksesser per oppslag, og for B-trær er det 2-4 aksesser.

En annen forbedring av algoritmen er **block nested loops join**. Her må ikke ordet "block" forveksles med disk-blokker, her betyr det bare en del av en tabell. Algoritmen utnytter ledig plass i minnet til å lagre blokker av tabell A (hvor en blokk er x antall pages), i stedet for å se på en og en post. For hver blokk av A som leses, opprettes typisk en hashtabell i minnet (ikke disk). Vi kan da scanne B én gang per blokk av A, og for hver post slå opp i hashtabellen. Merk at hashtabell kun opprettes for å gjøre oppslag i minne effektivt, det påvirker ikke antall diskaksesser.

Når hashtabell benyttes er block nested loops join er egentlig bare et spesialtilfelle av hash join.

Hash join

Hash join utnytter så mye ledig plass i minnet som mulig. Vi leser A samtidig som vi bygger opp en hashtabell i minnet. Når minnet er fullt leser vi hele B og slår opp i hashtabellen. A må leses én gang totalt sett. Hvor mange ganger vi må lese B avhenger av størrelse på minnet. Dersom vi klarer å få plass til hele A hashet i minne, trenger vi bare scanne B én gang.

Sort merge join

Dersom vi sorterer A og B kan join utføres ved å scanne begge tabellene parallelt. For å sortere tabellene brukes gjerne **external sorting**, som er en teknikk for å sortere større datamengder enn man har plass til i minne. Som eksempel, si at vi skal sortere en fil på 900MB, og minne tilgjengelig er 100MB. Først kopieres 100MB fra filen inn i minnet, og sorteres vha. for eksempel quicksort. Dette gjøres 9 ganger, så vi får $9 * 100$ MB sorterte partisjoner. Disse flettes (merges) sammen til den ferdig sorterte filen. I praksis er external sorting det samme som merge stort.

Dersom man har en B-tre-indeks på join-attributtet kan man scanne indeksen direkte, siden postene allerede er sorterte i løvnodene.

SQL

Et utvalg av de "sære" tingene i SQL. På eksamen gjelder det derimot å ha kontroll på basics, som SELECT, JOIN osv.

Integrity Constraints

Et eksempel på hvordan man kan opprette en tabell for å holde reserverasjoner på en restaurant. Man har i dette tilfellet en tabell for bord og for gjester. Denne constrainten gjør at vi ikke kan reservere utebord. Merk at CHECK er en constraint på lik linje med PRIMARY KEY, FOREIGN KEY, NOT NULL, UNIQUE osv.

```
CREATE TABLE Reservasjoner (  
    gid INTEGER,  
    bid INTEGER,  
    dag DATE,  
    FOREIGN KEY (gid) REFERENCES Gjester(gid),  
    FOREIGN KEY (bid) REFERENCES Bord(bid),  
    CONSTRAINT IngenUtebord CHECK  
    (  
        'Utebord' <> (  
            SELECT B.Type  
            FROM Bord B  
            WHERE B.bid = Reservasjoner.bid  
        )  
    )  
);
```

Oppretting av egne domener

```
CREATE DOMAIN Karakter INTEGER DEFAULT 1 CHECK (VALUE >= 1 AND VALUE <= 6);
```

Denne gjør at en karakter ikke kan være mindre enn 1 eller større enn 6. Når du oppretter en tabell f.eks vitnemål kan man skrive MatteKarakter Karakter der man ellers ville skrevet MatteKarakter INTEGER. Da vil det ikke være mulig å sette inn karakteren 7.

Assertion

Fungerer som en IC over flere tabeller

```
CREATE ASSERTION assertion_name CHECK (<expression>);
```

View

Et view er en tabell som ikke ligger eksplisitt lagret i databasen, men en tabell som beregnes fra andre tabeller. Tabellen bergenes ved å legge inn en view-definisjon. Dette viewet oppfører seg som en vanlig tabell, som man kan spørre på. Problemer oppstår når man vil oppdatere data (via viewet) . Views hvor poster kan spores entydig tilbake til en tabell, kan oppdateres.

Eksempel: Anta at man har en tabell som heter Student (studnr, fornavn, etternavn) og en tabell som heter AvlagtEksamen(fagkode, studnr, dato, karakter). Vi ville sikkert hatt en tabell Fag og Eksamen også, men de dropper vi for unngå unødvendig kompleksitet.

Vi vil ha en egen tabell med alle de som har fått karakteren F i eksamener etter januar 2010. Kanskje praktisk for å se hvem som skal ta konten og i hva. I stedet for å oprette en ny tabell, og lagre informasjon flere ganger så oppretter vi et view.











```
CREATE VIEW FailedStudents(sid, fornavn, etternavn, fagkode) AS
(
  SELECT studnr, fornavn, etternavn, fagkode
  FROM Student
  NATURAL JOIN AvlagtEksamen
  WHERE dato > '2010-01-01' AND karakter = 'F'
);
```

Når man har gjort dette kan man bruke viewet som det var en vanlig tabell (fysisk lagret på disk).

```
SELECT * FROM FailedStudents WHERE sid = '1337'
```

Relasjonsalgebra

Her er det mye forskjellig notasjon, men vi holder oss til forelesers versjon.

-  seleksjon
-  projeksjon
-  union
-  snitt
-  differanse
-  naturlig forening
-  forening
-  kartesisk produkt
-  divisjon
-  grupperingsoperasjon

Det er to måter å uttrykke relasjonsalgebra på; som en graf, eller som tekst. På eksamen bør begge deler være godkjent, så lenge det ikke eksplisitt står hva som foretrekkes.

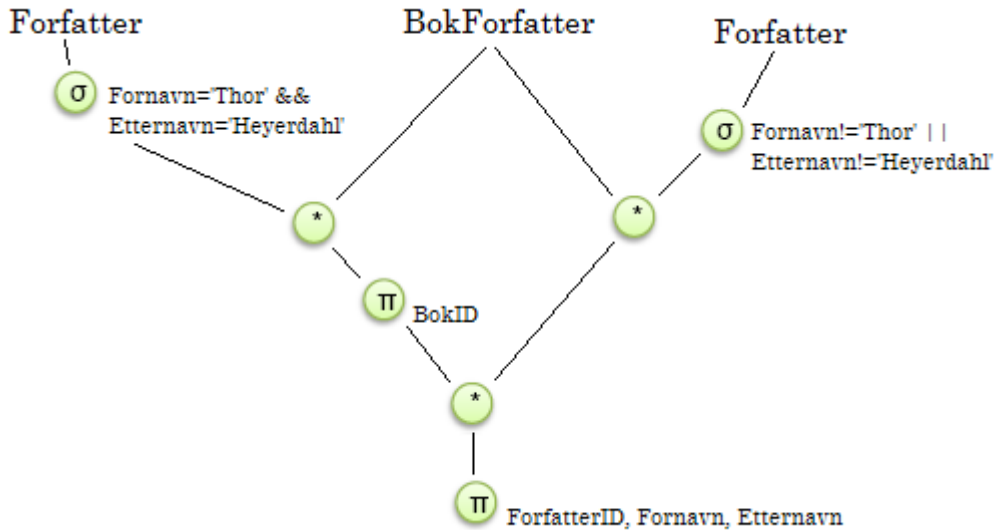
Eksempel fra eksamen 2002:

Bok (BokID, Tittel, ForlagID)

Forfatter (ForfatterID, Fornavn, Etternavn, Nasjonalitet, Url)

BokForfatter (BokID, ForfatterID)

En spørring uttrykt i relasjonsalgebra som finner alle forfattere som har skrevet en eller flere bøker sammen med Thor Heyerdahl. Thor Heyerdahl skal ikke bli med i resultatet, og det er bare en forfatter som heter Thor Heyerdahl:



$$R = \pi_{\text{ForfatterID, Fornavn, Etternavn}}(\pi_{\text{BokID}}(\sigma_{\text{Fornavn}='Thor' \ \&\& \ \text{Etternavn}='Heyerdahl'}(\text{Forfatter}) * \text{BokForfatter}) * (\text{Bokforfatter} * \sigma_{\text{Fornavn}!='Thor' \ || \ \text{Etternavn}!='Heyerdahl'}(\text{Forfatter})))$$

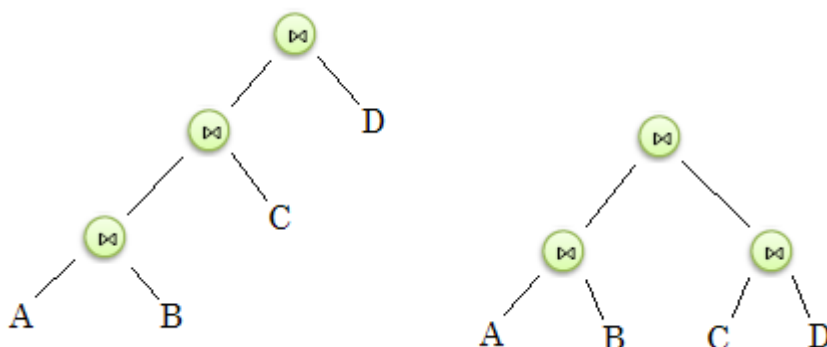
Det er garantert lettest å konstruere dette som graf først, for så å skrive ned på "vanlig" form.

Optimalisering

SQL-spøringer blir behandlet som relasjonsalgebra av query-evaluatoren i DBMS-et. En viktig observasjon her er at flere forskjellige løsninger (trær) er mulige for hver spørring, og man vil gjerne velge den planen som er billigst. Dette kan løses ved dynamisk programmering.

Left deep plans

Det lønner seg som regel ikke å bruke tid på å beregne alle løsninger, så optimalisatoren fokuserer kun på planer som er "left deep". Tenk deg at du skal joine tre eller flere tabeller: $A * B * C * D$. Det er $5!$ måter å gjøre dette på. Nedenfor er det vist to av alternativene, merk at algebratreet er tegnet "opp ned" i forhold til hvordan vi gjorde det i eksempelet over.



Optimalisatoren vil her velge alternativet til venstre, hvorfor? Grunnen er pipelining, som betyr at operasjoner gjøres "on the fly", altså mens tabellen leses fra disk. I det andre alternativet må vi faktisk lage tabellene $A \text{ JOIN } B$ og $C \text{ JOIN } D$ i egne filer, for så å joine

disse igjen. I det første alternativet trenger vi ikke det; vi joiner A og B, men i stedet for å lagre, filtreres resultatet direkte på C, deretter D. Vi unngår derfor masse overhead.

Transaksjoner

En transaksjon er et sett med operasjoner på databaseobjekter, og den minste enheten for arbeid i et databasesystem. En transaksjon skal enten fullføre alle sine endringer på databasesystemet (commit) eller avbryte (abort, rulle tilbake) om transaksjonen opplevde problemer. Vi sier en transaksjon er atomisk, enten fullføres den i sin helhet - eller ikke i det hele tatt.

Et enkelt praktisk eksempel er en banktransaksjon. Om du overfører 1000 kr fra din lønnskonto til din sparekonto vil banken først trekke 1000 kr fra lønnskontoen, deretter legge inn 1000 kr på sparekontoen. Det er ikke akseptabelt at denne transaksjonen bare gjennomføres halvveis om det oppstår problemer underveis. Da kunne penger forsvinne i luften eller dukke opp ut av ingenting.

Transaksjoner flytter databasen fra en konsistent tilstand til en annen. Med konsistent tilstand mener vi at gitte integritetsregler for dataene oppfylles. I eksempelet over skal bankens pengebeholdning og det totale omløpet av penger inn og ut være konstant.

Serialiserbarhet

For å øke ytelsen på transaksjonsutførelsen, vil et databasesystem utføre operasjonene til transaksjoner parallellt ved å flette de i hverandre. Så lenge resultatet av denne flettingen vil være ekvivalent (oppfattes likt som) den serielle utførelsen T1, T2, T3, T4, så vil jo ikke brukeren se noen forskjell og det vil heller ikke være noen forskjell i dataene. Vi sier at serialiserbarhet garanterer korrekt utførelse av transaksjonene.

Vi kan legge til at serialiserbarhet egentlig ikke sier noe om rekkefølgen på transaksjonene. T2, T1, T4, T3 vil fortsatt være en seriell utførelse av transaksjonene og "korrekt" utførelse i denne sammenheng, selv om denne rekkefølgen ikke vil gi samme resultat.

Derimot kan en fletting av operasjoner veldig lett føre til at resultatet ikke blir ekvivalent med en seriell utførelse av transaksjonene. Dette kan gi uventede resultater.

Historier

I et databasesystem blir alle operasjoner som blir gjort logget som en historie. Dette er spesielt nyttig dersom database vil "prøve seg frem" med en sekvens operasjoner for etterpå å sjekke om denne sekvensen av operasjoner var utført sekvensielt og dermed korrekt. Vi kan sjekke om en historie er utført sekvensielt ved bruk av det vi kaller en presedensgraf.

Vi har at en operasjon - read (r), write (w) eller commit (c) - blir gjort av en transaksjon x på et objekt Y, for eksempel: $w_x(Y)$. Eksempel på en historie kan være:

$w_2(X); w_1(X); r_3(X); r_1(X); w_2(Y); c_1; r_3(Y); r_3(Z), c_3; r_2(X); c_2;$

Vi kan avgjøre om en historie er serialiserbar ved å se om presedensgrafene til historien inneholder sykler. Har den en sykel er historien ikke serialiserbar.

- RAW - read after write (WR - dirty read)
- WAR - write after read (RW - unrepeatable read)
- WAW - write after write (WW - blind write)

Dersom to transaksjonsoperasjoner følger et av disse mønstrene, er de to operasjonene i konflikt med hverandre. Vi ser at hvis en av operasjonene er en skriveoperasjon (W-write), så oppstår det en konflikt. Alternativt: det oppstår ikke en konflikt dersom vi har RAR, og konflikt ellers. Disse konfliktene kan tegnes opp i en presedensgraf.

I en presedensgraf er hver transaksjon representert med en node. Pilene mellom nodene skal vise en konflikt mellom disse to transaksjonene. Vi betegner transaksjon nr. x til Tx (transaksjon nr. 1 blir T1). Det skal gå en pil fra Ti til Tj hvis noe av det Tj gjør skjer etter og er i konflikt med noe av det Ti gjør. Det holder at vi har én konflikt, av totalt mange operasjoner, for å lage en pil i grafen.

Recovery-egenskaper for historier

Her gjelder det å holde tunga rett i munnen, læreboka gjør dette vanskelig å få tak på. For en grei innføring anbefales Wikipedia: [Schedule](#).

Recoverable

$$G = \begin{bmatrix} T1 & T2 \\ R(A) & \\ W(A) & \\ & R(A) \\ & W(A) \\ & Com. \\ Abort & \end{bmatrix}$$

I en recoverable historie blir commit gjort i riktig rekkefølge. Det vil si at en transaksjon ikke commiter før alle dirty objekter som den har lest blir committed av sine transaksjoner. Først da kan vi garantere at dataene som ble lest inn er riktige (fordi objektet ble committed), og så kan vi commite selv.

I eksemplet til høyre commiter T2 før T1, selv om T2 har lest data som har blitt forandret av T1. T1 aborter like etterpå, som betyr at dataene som T2 har brukt ikke er gyldige. Vi har en unrecoverable historie.

Avoids cascading aborts / cascadeless

$$F = \begin{bmatrix} T1 & T2 \\ R(A) & \\ W(A) & \\ & R(A) \\ & W(A) \\ Com. & \\ & Com. \end{bmatrix} F2 = \begin{bmatrix} T1 & T2 \\ R(A) & \\ W(A) & \\ & R(A) \\ & W(A) \\ Abort & \\ & Abort \end{bmatrix}$$

Avoids cascading aborts (eller "unngår forplantende aborts", cascadeless) vil si at vi vil unngå at en transaksjon T som aborter kan føre til at andre transaksjoner også må aborte (fordi de har lest data fra T). Transaksjonen F til høyre er ikke cascadeless. Vi ser at T2 leser inn A, som er endret av T1, som enda ikke har committed. I F2 har vi samme historie men T1 velger å aborte - det fører til at også T2 må aborte, fordi dataene T2 har lest inn ikke er gyldige.

For å ha en historie som er cascadeless må dirty reads unngås, det vil si lesing av objekter som har blitt endret av en pågående transaksjon som ikke er avsluttet.

Strict

Forskjellen på Strict og Cascadeless er liten, men Strict forhindrer også at vi kan skrive over objekter som er dirty, det vil si allerede skrevet av en annen transaksjon som ikke har committed.

Samtidighetskontroll

Isolasjon er en av ACID-egenskapene ved et databasesystem. Vi ønsker å øke effektiviteten til systemet ved å kjøre transaksjoner i parallell, men samtidig må vi sikre at transaksjonene ikke ødelegger for hverandre.

Om vi har flere aktive transaksjoner, vil en gyldig utførelse av transaksjonene være enhver rekkefølge av operasjoner som tilsvarende en seriell utførelse.

Fenomener relatert til isolasjon

Read Uncommitted (dirty reads)

Dersom vi tillater å lese data som ikke er committet, risikerer vi å lese fra en tilstand i databasen som ikke er konsistent.

```
T1: SELECT * FROM Studenter WHERE studnr='1000';
T2: UPDATE Studenter SET antall_eksamener='0' WHERE studnr='1000' // IKKE COMMIT
T1: SELECT * FROM Studenter WHERE studnr='1000';
```

T2 har ikke committed sine endringer. For alt vi vet er dette en midlertidig endring som ikke skal være offisiell. Transaksjonen kan oppdatere raden med riktige verdier senere. Det kan også hende at transaksjonen ruller tilbake (gjenoppretter gammel tilstand) i stedet for å commite. Om T1 får tilbake den midlertidige (ikke-committede) raden fra databasesystemet vil vi få det vi kaller en dirty read (lesing av un-committede data).

Read committed (Non-repeatable reads)

Problemet oppstår når andre transaksjoner forandrer rader som har blitt hentet inn. Om vi prøver å gjøre en identisk spørring vil vi ikke få samme resultatet neste gang - selv om transaksjonen selv ikke har gjort noen forandringer på settet.

```
T1: SELECT * FROM Studenter WHERE studnr='1000';
T2: UPDATE Studenter SET dob='13-12-1989' WHERE studnr='1000'; COMMIT
T1: SELECT * FROM Studenter WHERE studnr='1000';
```

Phantom reads

Hvis vi i løpet av en transaksjon kjører samme spørring to ganger forventer man å få tilbake samme sett av rader tilbake. En annen transaksjon kan i mellomtiden forandret settet av rader ved å legge inn en ny rad som passer inn i intervallet vi søkte på.

```
T1: SELECT * FROM Studenter WHERE studnr BETWEEN 1000 AND 2000
T2: INSERT INTO Studenter (studnr, navn) VALUES (1500, "Bob"); COMMIT
T1: SELECT * FROM Studenter WHERE studnr BETWEEN 1000 AND 2000
```

Isolasjonsnivåer i SQL

SQL tilbyr fire forskjellige isolasjonsnivåer, fire forskjellige kompromiss mellom ytelse og isolasjon av transaksjoner. Under nevnt i rekkefølge fra færrest låser til flest låser.

```
SET TRANSACTION ISOLATION LEVEL { ... }
```

READ UNCOMMITTED

Dirty reads er tillatt. Med andre ord kan transaksjoner få tilbake data som en annen transaksjon har endret, men enda ikke committed. Ingen låser benyttes.

READ COMMITTED

Det benyttes ikke leselåser. Det betyr at andre kan forandre innholdet i rad(er) en transaksjon har lest inn og medføre en non-repeatable read. Transaksjoner vil imidlertid få skrivelås på rader og beholde disse til slutten av transaksjonen. Andre vil bli hindret å lese rader som er skrevet av en ikke avsluttet transaksjon.

REPEATABLE READ

Transaksjoner låser alle radene som hentes ut fra databasesystemet. Dette hindrer at andre kan forandre disse radene i mellomtiden. Transaksjonen vil derfor alltid få forventet resultat tilbake om samme spørring utføres flere ganger (repeatable read). I tillegg vil transaksjoner låse rader de har oppdatert helt til slutten av sin egen levetid. Om SELECT-spørringen inneholder søk på et intervall kan vi få phantom reads, fordi andre transaksjoner kan legge inn nye rader som tilfeldigvis passer inn i det aktuelle intervallet.

SERIALIZABLE

Det strengeste isolasjonsnivået. Benytter i tillegg til skrivelås og leselås intervall-lås, som hindrer at rader kan bli lagt inn som passer inn i gitte intervaller. I eksempelet ovenfor vil

databasesystemet stoppe alle transaksjoner som prøver å legge inn nye rader med studentnr mellom 1000 og 2000.

	Skrivelås Leselås Intervall-lås			Problemer
READ UNCOMMITTED	Nei	Nei	Nei	Dirty read, non-repeatable read, phantom
READ COMMITTED	Ja	Nei	Nei	Non-repeatable read, phantom
REPEATABLE READ	Ja	Ja	Nei	Phantom
SERIALIZABLE	Ja	Ja	Ja	

Logging

Det er to forskjellige logger: UNDO-logg og REDO-logg. Et DBMS kan implementere en kombinasjon av disse, som gir fire muligheter:

- NO UNDO/NO REDO
- NO UNDO/REDO
- UNDO/NO REDO
- UNDO/REDO

Hvilken loggstrategi som benyttes henger sammen med hvordan DBMS-et behandler pages i buffer.

Steal

Dersom det tillates at en page, som har blitt endret av en transaksjon som enda ikke har commitet, skrives til disk (kanskje fordi en annen transaksjon ønsker å bruke denne plassen i minnet), sier vi at plassen stjeles (steal). En steal-egenskap tilsier at vi må ha en UNDO-logg, fordi vi må kunne angre skrivingen til disk dersom transaksjonen aborter, eller systemet krasjer. Motsatt vil no-steal bety at ingen pages en transaksjon endrer kan skrives til disk før transaksjonen committer. No-steal er det enkleste for databasesystemet, men det antar at alle pages en pågående transaksjon bruker, får plass i minnet.

Force

Om vi tvinger at alle objekter endret av en transaksjon skrives til disk i det øyeblikket transaksjonen commiter, bruker vi en force-tilnærming. Tilsvarende vil no-force bety at objektene ikke nødvendigvis skrives til disk når transaksjonen commiter. En no-force-egenskap betyr at vi slipper å skrive til disk 20 ganger om 20 transaksjoner endrer samme objektet i rask rekkefølge, noe som sparer mange forespørsler mot disken.

Om vi bruker en no-force-tilnærming, trenger vi en REDO-logg. Vi må skrive i REDO-loggen før hver commit, slik at transaksjonen kan gjentas selv om systemet krasjer.

Typer logg

Hvilke logger som benyttes avhenger av hvilken policy man har til steal og force.

Ingen logg

Dersom man har no-steal/force er det ganske greit; transaksjoner får ha pages i fred helt til de committer. Da skrives alt umiddelbart til disk.

Kun REDO

Når man bare har REDO må alle endringer transaksjon T gjør skrives til loggen før T committer. Det er no-force-policyen som gjør at vi trenger redo, og når vi kun har redo betyr det no-steal.

Et eksempel, anta at en transaksjon T vil forandre objekt A, som har initialverdi 8.

```
Log <START T>
READ (A, s)
s = s + s
Log <T, A, 16>
WRITE (A, s)
Log <COMMIT, T>
FLUSH Log
OUTPUT (A)
```

Et problem her er at vi ved systemkrasj ikke vet hvilke transaksjoner i loggen som faktisk ble skrevet til disk og hvilke som ikke ble det. Husk at vi ikke tvinger pages til disk ved commit (no-force), så hvis vi forestiller oss vi har et stort buffer vil DBMS-et sin buffer manager la være å skrive til disk veldig lenge. Dette betyr at vi faktisk må gjøre alle transaksjonene, som er committet i loggen, på nytt ved systemkrasj.

For å delvis unngå dette problemet kan vi bruke **dynamisk checkpoint**:

1. Skriv loggpost <START CHECKPOINT T1, T2, ... Tn>, med liste over aktive transaksjoner.
2. Skriv (force) alle dirty pages som er endret av transaksjoner som allerede har committet.
3. Skriv en <END CHECKPOINT>.

Ideen her er at hvis du støter på en END CHECKPOINT i loggen vet du at alle committede transaksjoner som startet før START CHECKPOINT er skrevet til disk. Det betyr at vi ikke trenger å analysere hele loggen ved recovery lenger, det holder å finne det siste checkpointet og ta alle transaksjoner derfra. En variant av dette brukes også i ARIES.

Eksempel fra eksamen august 2005:

I et system som bruker redologging finner man følgende logg ved recovery:

```
<START T1>
<T1, X, 42> (*)
<START T2>
<T2, Y, 24>
<T1, Z, 21> (**)
<COMMIT T1>
<START T3>
<T3, Z, 50> (***)
<START T4>
<COMMIT T3>
<T4, X, 30>
```

Hvilke verdier kan X, Y og Z ha før recovery starter, og hvilke verdier kan de ha etter recovery? Angi svaret for verdien på disk og i minnet (RAM), dvs. til sammen 12 svar.

Det benyttes redo-logging. Det vil si at hvert logginlegg gir oss informasjon om hvordan vi skal "gjøre om" (redo-e) operasjoner. Det betyr at det er no-force, vi kan ikke garantere at transaksjoner som har committet har fått sine endringer på disk. Samtidig kan vi anta at det brukes no-steal, som betyr at for eksempel siste linje ikke plutselig skrives til disk før T4 commiter.

Vi finner transaksjoner som er committede, i dette tilfellet T1 og T3. Vi ser hvilke endringer de har gjort på objekter, finner her $X = 42$ (*), $Z = 50$ (***)). Observer at endring (**), som T1 gjør på Z, blir overskrevet av T3 lenger ned.

Fordi det brukes no-force må vi gjøre disse operasjonene på nytt. Det vi altså vet er at $X = 42$ og $Z = 50$. Vi vet ingenting om hva objektene kan ha av verdier før recovery startet. Tabellen vil se slik ut:

Minne før Disk før Minne etter Disk etter

X	Vet ikke	Vet ikke 42	Vet ikke
Y	Vet ikke	Vet ikke	Vet ikke
Z	Vet ikke	Vet ikke 50	Vet ikke

Kun UNDO

Her har vi kun mulighet til å gå bakover i historien. Loggen skrives før hver operasjon (WAL), og inneholder kun verdien før endring. Vi er også avhengig av at dirty pages skrives til disk rett etter at loggen er skrevet (gjelder alle endringer, ikke bare etter commit!), dette fører til mye skrivearbeid og gjør dette til en slitsom strategi for systemet.

Man kommer faktisk ikke unna REDO-logg likevel, for hva skjer hvis disken(e) som selve databasen ligger på går fyken? Da hjelper det lite med historie om hva som må reverseres.

Både UNDO og REDO

De fleste (alle?) databasesystemer har både UNDO- og REDO-logg, typisk samlet i én stor loggfil. Det er flere måter å håndtere dette på i praksis, i pensum er det fokusert på ARIES.

ARIES

Aries er en algoritme for gjenoppretting av et databasesystem ved systemkrasj, strømbrudd eller lignende. Siden databasesystemet flyttes mellom ulike konsistente tilstander av kjørende transaksjoner, kunne et uplanlagt avbrudd føre til at databasen sannsynligvis ikke lenger er konsistent.

Aries-algoritmen har et ganske enkelt mål. Den skal bringe databasesystemet tilbake til den siste konsistente tilstanden før krasjet.

Algoritmen er laget for et steal/no-force-databasesystem. De fleste databasesystemer bruker i praksis steal/no-force av optimaliseringsgrunner.

Algoritmen går i tre faser:

- Analyse: analyserer loggen, lager en oversikt over aktive transaksjoner og data som enda ikke er skrevet til disken (dirty pages).
- Redo: Skriver "dirty pages" til databasen. Når denne fasen er ferdig representerer disken et øyeblikksbilde av systemet i krasjøeblikket.
- Undo: Alle transaksjoner som er aktive (altså enda ikke fullført) angres/rulles tilbake. Dermed er systemet tilbake i konsistent tilstand.

Et annet prinsipp er viktig for at denne algoritmen kan fungere, **Write Ahead Logging**. WAL innebærer at alle forandringene på et objekt først lagres i en logg og skrives til disk (stable storage), før selve objektet kan lagres permanent. Loggen må skrives på et stabilt lagringsmedium, et medium som vil overleve alle tenkelige feil som kan skje på systemet.

Databaseloggen består blant annet av disse feltene:

- Log Sequence Number (LSN): Et unikt nummer gitt dette logg-innlegget, gis i inkrementerende rekkefølge.
- Transaction ID: En referanse til transaksjonen som er ansvarlig for logginnelegget.
- Prev_LSN: En referanse til forrige logginnelegg for aktiv transaksjon. Gir oss mulighet til å traverse logginnelegg for en transaksjon som en lenket liste.
- Type: Informasjon om det er snakk om en update/commit/end osv. Avhengig av verdien her får loggposten flere felter. En update-loggpost må for eksempel ha before_image og after_image for å dokumentere endringen.

Når en transaksjon starter blir en "Begin Transaction"-logg skrevet, tilsvarende en "End" når den avslutter.

Analysefasen

Databasesystemet holder hele tiden oversikt over (i minnet) tabeller med oversikt over data som enda ikke er skrevet til disk samt alle aktive transaksjoner, henholdsvis Dirty Page Table (DPT) og Transaction Table (TT). DPT inneholder, for hver dirty page, referanse til det første logginnelegget som gjorde den dirty (rec_LSN). TT inneholder siste logginnelegg generert per transaksjon.

Aries vil under analysefasen benytte loggen til å gjenoppbygge disse tabellene. Algoritmen begynner på starten av loggen (eventuelt et dynamisk sjekkpunkt) og bygger opp tabellene. Hver gang den treffer en "Begin Transaction" legges transaksjonen til TT, og ved en "End" blir transaksjonen fjernet.

DPT blir en oversikt over alle pages som har blitt forandret av systemet. Det er verdt å merke seg at vi enda ikke vet om disse pageene er skrevet til databasen eller ikke.

Redo-fasen

DPT gir oss referanse til det første logginnelegget per dirty page - rec_LSN. Vi benytter denne til å gjøre operasjonene på nytt og skrive endringene til databasen. Når denne fasen er ferdig gir disken et riktig øyeblikksbilde av databasen rett før krasjøeblikket.

Mer detaljer: Vi har en liste over rec_LSN-nummer i DPT som peker til rader i loggen med beskrivelse av endringer som gjorde pagen dirty første gang. Vi må begynne å gjøre om igjen endringer som beskrevet i loggen (husk WAL) fra og med den tidligste rec_LSN. For alle loggposter fra det punktet, gjøres operasjonen på nytt dersom ingen av følgende stemmer:

- Den aktuelle pagen finnes ikke DPT, den er altså allerede skrevet til disk.
- Den aktuelle pagen finnes i DPT men dens rec_LSN er større enn LSN til det loggelementet vi sjekker mot. Det betyr at pagen ble dirty etter denne loggposten, og vi slipper å gjøre det om igjen.
- page_LSN er større eller lik til LSN som blir sjekket. Enten var denne oppdateringen eller en senere oppdatering av pagen skrevet til disk. (Hver page inneholder LSN til loggposten som beskriver endringen som førte pagen til sin nåværende tilstand)

Undo-fasen

Etter redo må vi rulle tilbake alle transaksjoner som var aktive under systemkrasjet (og dermed ikke ble fullførte) for å få systemet tilbake i konsistent tilstand.

Algoritmen går ganske enkelt gjennom loggen i bakvendt rekkefølge og benytter informasjonen i Undo-feltet til å angre operasjonen. For hver endring som blir gjort i denne fasen skrives en **Compensating Log Record**, som dokumenterer at en operasjon ble rullet tilbake. CLR-loggen sikrer at vi også kan gjenopprette databasen om det skjer en ny krasj i løpet av gjenopprettingen.

Normalisering

Med normalisering menes å få en relasjon R over på en eller annen normalform, for eksempel 3NF eller BCNF. Typiske eksamensoppgaver er å avgjøre hvilken normalform en gitt relasjon er på, hvilke funksjonelle avhengigheter som eksisterer, finne et sett med kandidatnøkler/supernøkler, og gjøre om en relasjon til en høyere normalform.

Vi ser på relasjonen R(Studnr, Navn, Fagkode, Fagnavn, Karakter), som skal fungere som gjennomgående eksempel.

Studnr	Navn	Fagkode	Fagnavn	Karakter
90678	Per	TDT4100	Java	E
84700	Lise	TDT4100	Java	A
90678	Per	TFY4125	Fysikk	E
75866	Nils	TDT4100	Java	C

Anomalier i forbindelse med unormaliserte relasjoner

Målet med normalisering er å unngå en rekke anomalier som kan oppstå når informasjon lagres på flere steder (redundans):

- **UPDATE:** Ingenting hindrer oss i å endre navnet fra "Per" til "Pål" på linje 3, som vil føre til inkonsistens.
- **INSERT:** Tilsvarende kan vi sette inn en ny rad hvor fagkoden er TDT4100 men navnet er "C++".
- **DELETE:** Sletter vi en rad i relasjonen kan vi miste informasjon om forholdet mellom fagkode og fagnavn, eller studentnr og studentnavn.

Funksjonelle avhengigheter

En funksjonell avhengighet (forkortet FD av *functional dependency*) skrives som $X \rightarrow Y$ hvor X og Y er sett av attributter i en relasjon R. Det kan leses som "X bestemmer Y", med andre ord - dersom man vet X vet man også Y.

Vi kan analysere en oppgitt relasjon for å finne funksjonelle avhengigheter. Det er to muligheter:

1. Dersom kun attributtene er kjent, vi får for eksempel kun oppgitt R(Studnr, Navn, Fagkode, Fagnavn, Karakter), må vi vurdere ut ifra hva som logisk virker å være avhengig av hverandre. Vi vet for eksempel at hver Fagkode har ett Fagnavn. Med andre ord, hvis vi vet Fagkode vet vi også Fagnavn, eller som en FD: {Fagkode} \rightarrow {Fagnavn}.
2. Dersom man får oppgitt data (som i tabellen over) kan man bli spurt om å finne *alle funksjonelle avhengigheter som gjelder for de oppgitte dataene*. Da er det ikke sikkert attributtene har noe fornuftig navn, så punkt 1 kan ikke brukes. Da må man heller prøve seg fram, vi ser for eksempel at hver gang Studnr = 90678 og Navn = Per er også Karakter = E. Altså gjelder {Studnr, Navn} \rightarrow {Karakter} *for de oppgitte*

dataene. Merk at dette ikke impliserer at denne FD-en gjelder for R (litt drøyt å påstå at Per uansett alltid får E).

I relasjonen vår over kan vi identifisere følgende funksjonelle avhengigheter:

$\{\text{Studnr}\} \rightarrow \{\text{Navn}\}$

$\{\text{Fagkode}\} \rightarrow \{\text{Fagnavn}\}$

$\{\text{Studnr}, \text{Fagkode}\} \rightarrow \{\text{Karakter}\}$

Hvis man antar unike fagnavn får man også $\{\text{Fagnavn}\} \rightarrow \{\text{Fagkode}\}$. I resten av kapitlet antar vi at dette ikke er tilfelle.

Tillukning av F

I forrige eksempel fant vi 3 funksjonelle avhengigheter. Ved hjelp av Armstrongs aksiomer kan vi finne tillukningen av F, betegnet som F^+ , som vil si alle mulige FD-er som gjelder for R. Noen av reglene er:

- $A \rightarrow B$ gir $XA \rightarrow XB$ for vilkårlig X element i R.
- $A \rightarrow B$ og $B \rightarrow C$ gir $A \rightarrow C$.
- $A \rightarrow B$ og $A \rightarrow C$ gir $A \rightarrow BC$.

Denne framgangsmåten gir et sett som gjerne blir ganske stort, mer interessant er det å finne et *minimalt* sett FD-er.

Tillukning av attributter

Må ikke forveksles med tillukning av F. Vi definerer tillukningen for et sett av attributter X som settet av alle attributter i R som er funksjonelt avhengig av X, betegnet med X^+ . Her er altså X også et sett attributter i R.

Basert på FD-ene vi fant tidligere, kan vi sette $X = \{\text{Studnr}\}$ og få $\{\text{Studnr}\}^+ = \{\text{Studnr}, \text{Navn}\}$.

Dette blir relevant når man snakker om nøkler, som vi snart skal se på.

Minimalt sett av FD

Vi kan finne et minimalt sett, eller *minimal cover*, G for settet med funksjonelle avhengigheter F. Definisjonen for G er som følger:

1. Alle FD-er i G er på formen $X \rightarrow A$ hvor $|A| = 1$.
2. Tillukningen $F^+ = G^+$.
3. $H^+ \neq F^+$ dersom vi sletter en FD eller fjerner en attributt fra en FD i G slik at vi får settet H.

Vi kan få oppgitt et sett FD-er og bli bedt om å finne G. Algoritmen er som følger:

1. Få hver FD på standardform, $X \rightarrow A$ hvor $|A| = 1$.
2. Minimer X.
3. Fjern redundante FD-er.

Med andre ord: her må man bruke hodet. Vi ser på et nytt eksempel:

$F = \{ A \rightarrow B, ABCD \rightarrow E, EF \rightarrow G, EF \rightarrow H, ACDF \rightarrow EG \}$.

Første steg gjør om $ACDF \rightarrow EG$ til $ACDF \rightarrow E$ og $ACDF \rightarrow G$.

Så kan vi bruke transitivitet til å se at $ABCD \rightarrow E$ er ekvivalent med $ACD \rightarrow E$ på grunn av $A \rightarrow B$.

Vi kan bruke transitivitet igjen og se at $ACDF \rightarrow G$ er ekvivalent med $EF \rightarrow G$ på grunn av $ACD \rightarrow E$.

Nå har vi både $ACD \rightarrow E$ og $ACDF \rightarrow E$. Vi kan fjerne den siste (beholder den enkleste, ellers taper vi informasjon).

Vi ender opp med det minimale settet $G = \{ A \rightarrow B, EF \rightarrow G, EF \rightarrow H \}$. Ved hjelp av Armstrongs regler kan vi komme tilbake til utgangspunktet, G inneholder nøyaktig den samme informasjonen som F .

Nøkkel og supernøkkel

Vi skiller mellom hva som er nøkkel (key) og hva som er supernøkkel (superkey) for en relasjon R .

Superkey: Et sett attributter som bestemmer alle attributter i R .

Key: Et *minimalt* sett attributter som bestemmer alle attributter i R .

En **kandidatnøkkel** er et sett attributter som kvalifiserer som *key*. Det kan altså være flere mulige minimale sett attributter som bestemmer alle andre attributter i R .

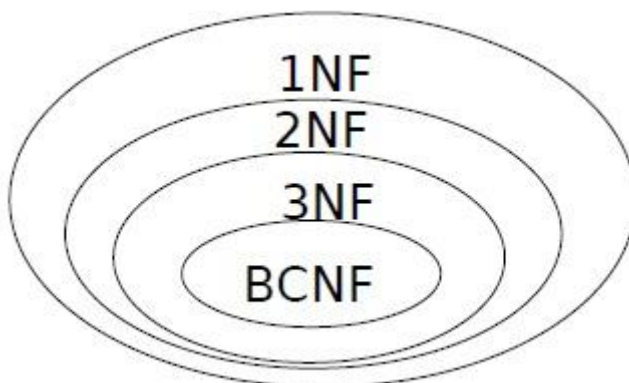
En **nøkkelattributt** er en attributt som er en del av en eller flere kandidatnøkler.

I eksempelet vårt er det ganske enkelt å se at en kandidatnøkkel (faktisk den eneste) er $\{ \text{Studnr, Fagkode} \}$. Vi har at tillukningen $\{ \text{Studnr, Fagkode} \}^+ = \{ \text{Studnr, Navn, Fagkode, Fagnavn, Karakter} \}$, av det kan vi slutte at det er en superkey. Det er også en key fordi ved å fjerne enten Studnr eller Fagkode så mister vi tillukningen.

I mer kompliserte tilfeller kan det lønne seg å tegne FD-ene som en graf for å finne kandidatnøkler.

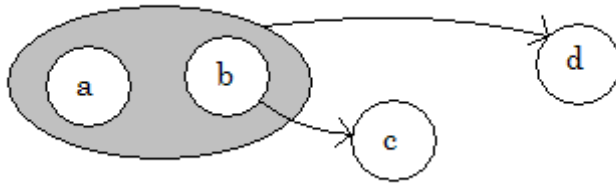
Normalformer

Normalformene bygger oppå hverandre. Det vil si at om vi vet at en relasjon er på 3NF, kan vi automatisk konkludere med at lavere normalformer er oppfylt. Historisk sett ble de ulike normalformene formulert etter at det ble oppdaget hvilke feil den forrige normalformen kunne føre til. Vi nevner at det finnes flere normalformer utover BCNF, men det er ikke pensumrelevant.



Mengder av relasjoner etter normalform.

Første normalform (1NF)

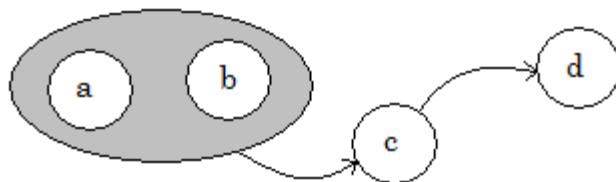


Første normalform

Alle attributter er atomiske, altså lagres ikke arrays eller lignende i noen felter. Relasjonen antas alltid å være i 1NF i dette faget, det er mer ett formelt krav til databasesystemer. Lurt å nevne i besvarelsen hva som er den formelle definisjonen av denne normalformen.

$R(a, b, c, d)$ er på 1NF, men ikke 2NF.

Andre normalform (2NF)

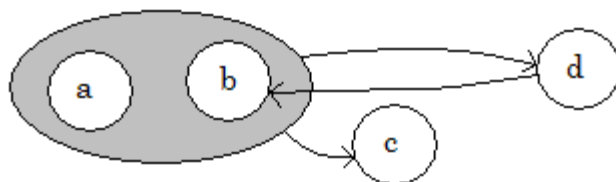


Andre normalform

Alle attributter skal være funksjonelt avhengige av **hele** nøkkelen. Transitive avhengigheter er tillatt.

$R(a, b, c, d)$ er på 2NF, men ikke 3NF. Vi ser at d er avhengig av nøkkelen (a, b) , men gjennom c .

Tredje normalform (3NF)



Tredje normalform

Alle attributter som *ikke* er del av en nøkkel, er *direkte* avhengig av hele nøkkelen. Transitive avhengigheter godtas altså ikke.

For hver avhengighet $X \rightarrow A$ må X minst være en del av en nøkkelkandidat. $d \rightarrow b$ går bra i dette tilfellet, siden nøkkelkandidater er (a, b) og (a, d) .

$R(a, b, c, d)$ er på 3NF, men ikke BCNF.

Dekomponering til 3NF

En relativt enkel metode for å gjøre om en relasjon R til 3NF er som følger:

1. Finn minimal cover for F
2. For hver FD $X \rightarrow A$, lag en relasjon XA med X som nøkkel.
3. Gjenstående attributter + nøkkel samles i en siste relasjon.

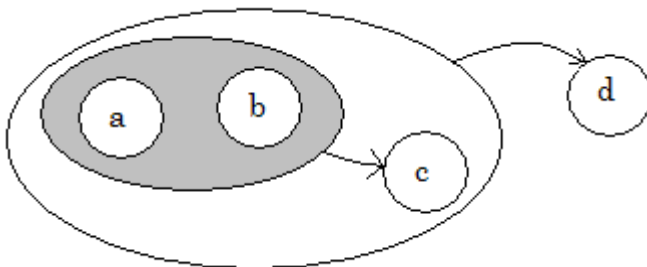
Ved å gjøre det på denne måten har vi en **dependency preserving decomposition**, altså en dekomponering som ivaretar alle FD-er. Merk at dette ikke er noe krav, fremgangsmåten vi viser for BCNF vil også lage gyldige 3NF-oppdelinger, men garanterer ikke denne egenskapen.

En annen viktig egenskap ved oppdelinger er **lossless join decomposition**, som vil si at oppdelingen i R_1, R_2, \dots, R_n kan joines og gi tilbake den opprinnelige relasjonen R. Man kan enkelt sjekke om en oppdeling er lossless:

Oppdelingen i R_1 og R_2 er lossless hvis og bare hvis $(R_1 \cap R_2) \rightarrow R_1$ eller $(R_1 \cap R_2) \rightarrow R_2$.

Hvis vi for eksempel deler $R(\text{Studnr, Navn, Fagkode, Fagnavn, Karakter})$ i $R_1(\text{Studnr, Navn})$ og $R_2(\text{Studnr, Fagkode, Fagnavn, Fagnavn, Karakter})$ får vi at $R_1 \cap R_2 = \{ \text{Studnr} \}$, og $\{ \text{Studnr} \} \rightarrow \{ \text{Studnr, Navn} \} = R_1$. Oppdelingen er altså lossless, og vi kan joine R_1 og R_2 uten å tape informasjon vi hadde i R.

Boyce-Codd Normal Form (BCNF)



Boyce-Codd-normalform

3NF + Ingen avhengigheter peker tilbake på en nøkkelkandidat.

Det betyr at for alle FD-er $X \rightarrow A$ må X være en *superkey*. Her har vi $\{a, b\} \rightarrow \{c\}$ og $\{a, b, c\} \rightarrow \{d\}$. $\{a, b\}$ er en key (en key er også per def. en superkey), og $\{a, b, c\}$ er åpenbart en superkey siden den har en nøkkel som subset.

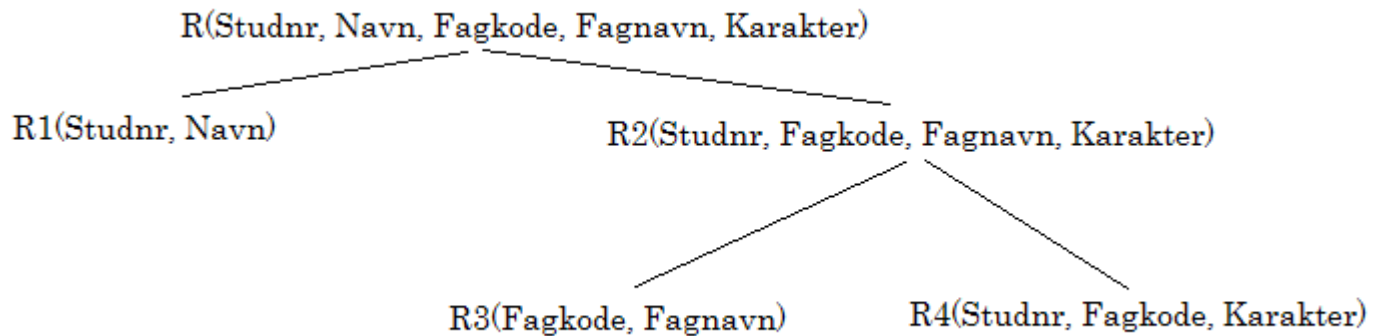
I praksis vil de aller fleste relasjoner på 3NF også være på BCNF. Det er også tilfeller hvor en relasjon ikke lar seg omskrive til BCNF.

Dekomponering til BCNF

For å gjøre om en vilkårlig relasjon R til en samling med relasjoner R1, R2, ..., Rn hvor alle er på BCNF kan vi bruke følgende framgangsmåte:

Så lenge det finnes en relasjon R som ikke er på BCNF:

1. Finn en mengde attributter X i R, slik at FD-en $X \rightarrow A$ gjør at kravet for BCNF ikke oppfylles for R.
2. Dekomponer R i R - A og XA.



Dekomponeringsprosessen illustrert som trestruktur

Oppdelingen vi får som tilfredsstillende BCNF (løvnode) kan vi nå skrive som:

Student (*Studnr*, Navn)

Fag (*Fagkode*, Fagnavn)

Eksamen (*Studnr*, *Fagkode*, Karakter)