

ER til tabell

- Dersom vi har en **(0,n)-(0,n)**-relasjon, så lager vi tabell for relasjonen med fremmednøkkel mot de to tabellene
- Dersom vi har **(0,n)-(1,1)** så kan vi
 - enten legge til fremmednøkkel i (1,1)-klassen. Her kan vi også legge til attributt for egenskap til relasjonen. Legger det på (1,1)-siden
 - eller lage en koblingstabell
- Dersom vi har **(1,1)-(1,1)**
 - legger vi til en fremmednøkkel i en av tabellene
 - eller lager egen koblingstabell
- Dersom vi har **flerverdiattributter** må vi lage egen tabell for denne (for eksempel tlf.nummer) og legge til en fremmednøkkel til entitetstabellen i denne.
- Dersom vi har **mer enn 2 entitetsklasser til en relasjon**, så lager vi tabell til hver av de tre pluss en tabell for relasjonen med ID-ene til de andre tabellene pluss ev. egenskap til relasjonen
- Dersom vi har **spesialisering** (super-/subklasse) så kan vi enten:
 - Lage tabell for superklasse og en for hver av subclassene. Hver tabell inneholder superklassen sin ID
 - Lage tabell for hver av subclassene med ID og attributter til superklassen. Krever disjunkt og total
 - Lage én tabell med attributter til alle pluss et typefelt.
 - Gir mange NULL-verdier
 - Vanskelig å holde oversikt over hvilke attributter som hører til hvilken underklasse
- Dersom vi har **kategorier** så har vi egen tabell for superklassen og hver av subclassene med surrogatnøkkel til superklassen og med egen ID.
 - Legger til surrogatnøkkel til kategorien, og legger denne til som fremmednøkkel hos relasjoner som korresponderer med superklassen til kategorien. NULL hvis den ikke er med.
 - En kategori er en subklasse av unionen av to eller flere superklasser. Den kan ha ulike nøkler fordi superklassene kan være ulike entitetstyper. Subklassen har et typefelt som forteller hvilken entitetstype den er.

Datamodellering

Entitetsintegritet: betyr at alle rader en tabell er unike og realiseres ved at tabellen har en primærnøkkel som ikke kan ha NULL-verdier

Referanseintegritet: Betyr at alle fremmednøkler enten refererer til en tuppel som finnes eller har en NULL-verdi.

Restriksjoner med fremmednøkler: Ut over å spesifisere hvilken tabell fremmednøkkelen refererer til og om den kan ta nullverdi eller ikke, kan man spesifisere hva som skal skje dersom det skjer endring i eller sletting av det tuppelet fremmednøkkelen refererer til. Aktuelle handlinger kan være å blokkere endring eller sletting i den refererte tabellen, å videreføre endringen ved enten å endre fremmednøkkelen verdi eller å slette tuppelet med fremmednøkkelen, eller å sette fremmednøkkelen til NULL.

Svak entitetsklasse:

- Det finnes ikke noe attributt eller mengde av attributter som kan tjene som entydig identifikator for entitetsklassen.
- Entitetsklassen A har en relasjonsklasse (R) til en annen entitetsklasse (B). Klassen A er eksistensavhengig av relasjonsklassen R og kan ha denne typen relasjon til maksimalt en entitet i entitetsklassen B. A har altså (1,1)-kardinalitet i forhold til relasjonsklassen R.

Delvis nøkkel: Dersom det finnes et attributt eller en mengde attributter i A som alltid er entydig blant de A-entitetene som har en R-relasjon til samme B-entitet, kaller vi dette en **delvis nøkkel**. I slike tilfeller kan vi modellere A som en svak entitetsklasse.

Relasjonsklassen R fungerer da som en **identifiserende relasjonsklasse** for entitetsklassen A.

En globalt entydig identifikator for A vil bestå av attributtene i den delvise nøkkelen pluss attributtene i identifikatoren til B. Alternativet til å modellere A som en svak entitetsklasse, vil være å legge til et ekstra attributt i A, ofte kalt en surrogatnøkkel, som vil fungere som identifikator entitetsklassen.

Join

Join:

- Kritisk sammenstilling
- Kombinerer relaterte data og vi får flere kolonner
- Kan ha duplikate kolonner
- $\bowtie =$

Equijoin (en type join)

- Bruker kun = når de sammenlignes.
- Vil ha flere attributtpar som har identiske verdier i hver tuppel i resultatet
- Dette er den vanlig joinen.

Naturlig join:

- Implisitt join-betingelse – likhet i all par av kolonner med like navn

- Fjerner alle duplikatkolonner
-
- Risiko: kan bli større join-betingelse enn man ønsker
- Spesifiserer ikke hva man joiner på
- De må ha samme navn i begge tabellene for å fjerne den ene kolonnen
 - Hvis ikke må vi først rename. Bruker da ρ

Outer join:

- Får med alle rader i en eller begge operand-tabellene – left, right eller full
- Filtrerer ikke bort rader slik som inner-join. Beholder de man spesifiserer
 - Attributter som mangler data får NULL-verider

Funksjonell avhengighet: En mengde attributter (Y) er funksjonelt avhengig av en annen mengde attributter (X), $X \rightarrow Y$, når alle tupler med samme verdier i X alltid har samme verdier i Y.

Mengdeoperasjoner

UNION: Bruker $S \cup R$. en relasjon som inkluderer alle tupler som enten er i S eller R, eller i begge. Duplikater blir eliminert.

INTERSECTION(snitt): $S \cap R$. Resultatet er en relasjon som inkluderer de tupler som er i både S og R.

SET DIFFERENCE(minus): Resultatet $(R-S)$ er en relasjon som inkluderer alle tupler som er i R, og ikke i S.

DIVISION: $R \div S$. Resultatet er alle restriksjoner av tupler i R til attributtnavnene som er unike i R. Resultatet er der alle kombinasjoner med tupler i S er i R.

Nøkler

Alle tabeller må ha minst én nøkkel

Supernøkkel: Unik identifikator. Supernøkkel for en tabell R er en mengde attributter S slik at

- det ikke i noen forekomst av tabellen kan finnes to tupler, t_i og t_j , med like verdier for S ($t[i]=t[j]$)

- supernøkkel vil være en unik identifikator

- $S^+ = R$

- Minimal supernøkkel: hvis vi ikke kan fjerne noe attributt fra nøkkelen og fortsatt ha en supernøkkel. Alle nøkler er supernøkler, men alle supernøkler er ikke nøkler.

(Kandidat)nøkkel: En tabells nøkler utgjør kandidatnøkklene. Dette er en unik, minimal identifikator. Samme som minimal supernøkkel

Primærnøkkel: Den viktigste blant kandidatnøkklene

Alternativ nøkkel: Sekundærnøkkel. Kandidatnøkkel som ikke er primærnøkkel.

Nøkkelattributt: Et attributt som inngår i en eller flere kandidatnøkler.

Ikke-nøkkelattributt: Et attributt som ikke inngår i kandidatnøkkel

Normalformene

Bakgrunn:

- minimere redundans
- minimere innsetting-, sletting- og oppdatering-anomalier (forventet, avvik fra det normale)
- filtrering, rensing for bedre kvalitet på design
- dersom vi har en tabell som ikke er på NF så må vi dekomponere i mindre relasjoner

Normalform: en relasjon refererer til høyeste normalform-betingelse som den møter, og svarer til hvilken graf den er normalisert

Når vi normaliserer må vi passe på

- at vi har tapsløs og ikke-additiv join (VIKTIGST)
- bevaring av FA
- attributtbevaring

Første normalform (1NF)

- tillater ikke relasjoner i relasjoner eller relasjoner som attributtverdier med tupler
- attributtdomener inneholder atomiske (udelelige) verdier
- verdien til attributtet er en enkelt verdi fra domenet
- sikrer flate 2D-tabeller
- unngår sammensatte attributter, flere verdier og nøstede tabeller

Oppnå 1NF

- Del opp
- ++

Andre normalform (2NF)

- Basert på fulle funksjonelle avhengigheter (FFA)
 - Dersom man kan fjerne et attributt A fra X gjør at avhengigheten ikke lengre holder (ingen overflødige attributter)
 - Delvis avhengighet dersom det finnes et attributt A i X som kan fjernes og avhengigheten fortsatt holder
- Et relasjonsskjema er på 2NF hvis alle ikke-nøkkel-attributter i R er FFA av primærnøkkelen til R. (det finnes ingen ikke-nøkkel-attributter som er delvis avhengig av en kandidatnøkkel). Altså hvis kandidatnøkkelen er bokID, dato og man det finnes en FA bokIDàboktittel, så er dette en delvis FA fordi man kan fjerne et attributt (dato) fra kandidatnøkkelen og likevel ha avhengigheten
- Teste om det finnes FA der venstre-side-attributtet er en del av primærnøkkelen. Dersom primærnøkkelen inneholder et enkel-attributt trenger man ikke å gjøre testen en gang.

Oppnå 2NF

- Splitte opp i flere tabeller

Tredje normalform (3NF)

- Basert på transitiv avhengighet: om det finnes et sett med attributter Z i R som verken er kandidatnøkkel eller et subset av en nøkkel i R, og at både $X \rightarrow Z$ og $Z \rightarrow Y$ holder.
- Det må være 2NF og ingen ikke-nøkkel-attributter i R er transitivt avhengig av primærnøkkelen
- Forelesning: Hvis og bare hvis det for alle FA på formen $X \rightarrow A$ som gjelder for tabellen er slik at
 - X er en supernøkkel i tabellen, eller
 - A er et nøkkelattributt i tabellen
- Ved splitting: unngår redundans og innsettings-, oppdaterings- og slettingsanomalier (avvik).

Boyce-Codd Normalform (BCNF)

- Hvis det for alle FA på formen $X \rightarrow Y$, som gjelder for tabellen er slik at
 - X er en supernøkkel i tabellen
 - Dvs at all venstresider i FA må være supernøkler

Splitte opp tabeller for å oppnå høyre normalform:

- Fordeler: Unngår redundans og problemer med innsetting, oppdatering og sletting.
- Ulemper: Flere tabeller, må joine mer, mer kompliserte spørringer

4 forhold som må tas i betraktning når vi splitter:

1. Normalform: ser på hver tabell hver for seg. Bør ha høyere normalform
2. Attributtbevaring: alle attributtene må fremdeles finnes
3. Bevaring av funksjonelle avhengigheter
4. Tapsløs sammensetning til utgangspunktet: ikke skape falske data. En joining må gi den samme tabellen som vi startet med før vi splittet.

Transaksjoner

A – Atomic: Enten så kjører transaksjonen fullstendig eller ikke i det hele tatt

C – Consistency: Overholder konsistenskrav (primary key, references, check osv.). Dersom en transaksjon er helt utført fra start til slutt uten forstyrrelser fra andre transaksjoner så skal den ta databasen fra en konsistent tilstand til en annen.

I – Isolation: Transaksjoner er isolert fra hverandre, og merker ikke at noen andre kjører samtidig. Transaksjoner skal ikke bli forstyrret av andre transaksjoner som kjører samtidig

D – Durability: Transaksjonene er permanente. Dvs. at de ikke mistes etter commit.

Disse er alle egenskaper ved transaksjoner.

Samtidighetsproblemer:

- **Dirty read**
 - Når en transaksjon oppdaterer et dataelement og så feiler transaksjonen. I mellomtiden blir dette dataelementet lest fra en annen transaksjon før verdien blir abortert eller rullet tilbake. Hvis man leser fra en verdi som er skrevet til og ikke committed (i en annen transaksjon)
- **Lost update**

- Når to transaksjoner som kan aksessere de samme dataelementene, fletter operasjonene sammen. En verdi blir da feil. Hvis to transaksjoner skal skrive til samme verdi, men ingen har committed i mellom
- **Unrepeatable read**
 - Transaksjonen T leser det samme dataelementet to ganger og dataelementet er endret av en annen transaksjon T' mellom lesingene.
- **Incorrect summary**

Commit: alt gikk bra, endringen ligger i databasen

Rollback (abort): transaksjonen rulles tilbake og ingen endring i databasen

Autocommit: avslutt spørring med ;

Konflikt:

- Samme dataelement
- To forskjellige transaksjoner
- Minst én write

Gjenopprettbarhet

- **Strict:**
 1. Tj reads a data item X after Ti has written to X and Ti is terminated (aborted or committed)
 2. Tj writes a data item X after Ti has written to X and Ti is terminated (aborted or committed)
- **ACA:**
 1. Tj reads X only after Ti has written to X and terminated (aborted or committed).
- **Recoverable**
 1. Tj commits after Ti if Tj has read any data item written by Ti.

Serialiserbarhet

- Seriell historie: historie hvor du ikke fletter operasjoner fra ulike transaksjoner. Kjører etter hverandre
- Serialiserbar historie: historie som har samme effekt på databasen som en seriell historie
- To historier er konfliktekvivalente hvis de har samme rekkefølge for operasjoner med konflikt
- En historie er konfliktserialiserbar hvis den er konfliktekvivalent med en seriell historie
- Konfliktserialiserbar à serialiserbarhet. Men ikke nødvendigvis omvendt
- Kan bruke en presedensgraf. Dersom presedensgrafen ikke har noen sykler, så er historien konfliktserialiserbar.

Låsing

- Brukes for å oppnå serialiserbarhet
- Låser poster eller blokker
- Låse bare poster à mer samtidighet
 - Read_lock(x), kan deles

- Write_lock(x), eksklusiv lås
-
- **2PL (tofaselåsning)** – impliserer serialiserbarhet
 - Problemer som kan oppstå er vranglås og cascading rollback
 - Cascading rollback er at en transaksjon T1 gjør en endring på en verdi, unlocker låsen (men ikke commiter) og så bruker en annen transaksjon T2 verdien videre. Hvis T1 failer uten å ha committed etter dette, betyr det at T2 bruker en ugyldig verdi. Da må både T1 og T2 rulles tilbake.
 - Har en growing fase og en shrinking fase
 - I growing settes shared(lese) -og exclusive(skrive)låsene, i shrinking unlockes disse igjen
 - **Basic:** Låser settes i growing fase, unlocks i shrinking. Når man har begynt å unlocke, er det ikke mulig å sette nye. Derfor settes låsene gjerne tidlig. Kan få vranglås og cascading rollback.
 - **Konservativ:** Setter alle låser med en gang, derfor ingen growing phase. Tillater lite samtidighet, men unngår vranglåser. Kan ha cascading rollback. Praktisk implementasjon av dette er vanskelig.
 - **Strict:** Garanterer en Strict historie. Releaser bare skrivelåser etter commit, leselåser unlockes i shrinking phase. Unngår cascading rollback, men kan ha vranglås
 - **Rigorous:** Som strict, men slipper alle låser til slutt, også leselås. Enkel å implementere, men tillater mindre samtidighet enn strict. Unngår cascading rollback, kan ha vranglås. Mer populær enn strict, og litt bedre.
 - *“Rigorous two-phase locking has the advantages of strict 2PL. In addition it has the property that for two conflicting transactions, their commit order is their serializability order. In some systems users might expect this behavior.”*
 - *“Consider two transactions conducted at the same site in which a long running transaction T1 which reads x is ordered before a short transaction T2 that writes x. T2 returns first, showing an update version of x long before T1 completes based on the old version.”*
- **Vranglås**
 - To eller flere transaksjoner venter gjensidig på hverandres låser. Kan løses ved:
 - § Unngåelse – med konservativ 2PL settes alle låser på forhånd, og vi unngår vranglåser
 - § Oppdagelse – vanskeligst, tvinger abort. Konstruer wait-for-grafer og søk etter sykler. Aborter en transaksjon og se om sykkelen blir borte
 - § Timeout – veldig enkel. Hver transaksjon har en timer. Start tilbakerulling når timeouten går. Men vanskelig å sette timeouten riktig.
- **Multiversjons-currency control**

- Veldig vanlig
- La en leseoperasjon som er i konflikt lese en gammel versjon
- Basert på tidsstempelrordning
- Ulempe: administrasjon av mange versjoner, mer plass, men ikke update-in places
- 2 måter i praksis:
 - § Lagrer flere versjoner
 - § Lagrer kun den siste versjonen

Recovery

Betyr: en database blir restaurert/reparert til den siste konsistente tilstanden før feilingen. Da må systemet ha informasjon om endringene som ble gjort på dataenheter av transaksjonene. Denne informasjonen holdes i **systemloggen**.

2 main policies for recovery:

- **Deferred update:** oppdaterer ikke databasen fysisk på disk før etter en transaksjon commiter, oppdaterer kun loggen i hurtigbufferet. Det er da oppdateringene blir med i databasen. Før commit lagres alle oppdateringer i loggfilen på disk i main memory bufferet. Etter commit så blir oppdateringene skrevet til databasen. Hvis det skjer krasj før commit er ingen endringer gjort på databasen, og UNDO er ikke nødvendig. Kan være nødvendig med REDO. Samme som **NO-UNDO/REDO**.
- **Immediate update:** databasen kan bli oppdatert av noen operasjoner i transaksjoner *før* transaksjonen har commitet. Disse operasjonene må også bli registrert i loggen på disk ved force-writing før de er lagt til databasen på disk. Dersom feiling; må UNDOe operasjonene på databasen (roll back). UNDOer ved rolling back transaksjonen og gjøre om transaksjonens operasjoner som har gjort en effekt. Bruker STEAL når den UNDOer for å bestemme når oppdaterte hovedminnebufre kan bli skrevet tilbake til disk.
UNDO/REDO
 - trenger ikke REDO dersom alle oppdateringer er registrert av databasen på disk før commit. **UNDO/NO-REDO. STEAL/FORCE** for å bestemme når oppdatert hovedminnebuffere skal bli skrevet tilbake på disk.
 - Hvis ikke alle endringer må være skrevet til databasen før commit har vi **UNDO/REDO** som bruker **STEAL/NO-FORCE**

UNDO og REDO for at operasjoner som kjører mange ganger skal virke på samme måte som om den kjører kun en gang.

Flushe et endret buffer tilbake på disk:

- **In-place updating:** skriver bufferet (med after image) til den samme diskplasseringen, og overskriver den gamle (before image) verdien til alle dataelementer på disken. En enkel kopi av hver database diskblokk blir opprettholdt. Trenger en log for gjenoppretting. Da må vi

passe på at loggplassen er flushet til disk før vi skriver over before image med after image: **WAL** (helt nødvendig for UNDO).

- **Shadowing**: skriver en oppdatert buffer til en annen disklokasjon, så flere versjoner av data blir opprettholdt. Blir ikke ofte brukt.

Force/steal

- Utgangspunkt: hvor fleksibel/uavhengig er buffer manager (den som håndterer pages i minnet og hvem som skal skrive) til logging/recovery
 - Når skal skitne blokker skrives
 - Når må skitne blokker skrives
- **Force**: må en skitten blokk tvinges til disk ved commit. Det er tregt fordi datablokkene kan være spredd over hele disken. Hvis alle pages som oppdateres av en transaksjon blir skrevet til disk med den gang før transaksjonen commiter. Trenger aldri REDO under recovery, fordi alle committede transaksjoner har sine endringer på disk før committed.

NO-REDO

- **No-force**: trenger ikke bli skrevet til disk med en gang.
- **Steal**: kan en transaksjon stjele plass i buffer til en skitten blokk. Hvis ikke må en aktiv transaksjon ha alle skitne blokker i bufferet inntil commit. Hvis man kan skrive et endret buffer før transaksjonen commiter. Når man trenger å erstatte plass.
- **No-steal**: hvis en hurtigbufferside som oppdateres av en transaksjon ikke kan bli skrevet til disk før transaksjonen commiter. Trenger ingen UNDO her, siden committede transaksjoner ikke vil ha noen oppdateringer på disk før commit. **NO-UNDO**
- **FORDEL MED STEAL**: unngår å trenge veldig store bufferplass for å lagre endrede sider i minnet.
- **FORDEL MED NO-FORCE**: en oppdatert page til en ucommitet transaksjon kan fortsatt være i buffer når en annen transaksjon trenger å oppdatere den, så trenger ikke skrive med page flere ganger til disk og muligens måtte lese den igjen fra disk. **NO-FORCE** må skrive loggen, men ikke de skitne blokkene. Loggen er sekvensiell og kan skrive raskt ut til disk. **FORCE** krever utskrift av datablokker, og disse kan være mange, og de kan være svært spredd utover disken. Kan også slå sammen skriving av datablokker, for eksempel oppdatering av samme blokk.

- **No-force/steal: UNDO/REDO**

FRA EKSAMENSOPPGAVE:

Spørsmål: En transaksjon har oppdatert en tabell og gjort ei blokk i bufferet "skitten" (dirty). Hvilke metoder finnes for å la transaksjonen oppfylle D-en (durability) i ACID-egenskapene?

Svar: Her er det **FORCE** eller **NO-FORCE** som er alternativene:


- 1) **NO-FORCE**: (Redo-)loggen må skrives til disk ved commit.
- 2) **FORCE**: De skitne datablokkene må skrives til disk ved commit.



Buffer Mgmt Plays a Key Role

- **Force policy** – make sure that every update is on disk before commit.
 - Provides durability without REDO logging.
 - But, can cause poor performance.
- **No Steal policy** – don't allow buffer-pool frames with uncommitted updates to overwrite committed data on disk.
 - Useful for ensuring atomicity without UNDO logging.
 - But can cause poor performance.

Of course, there are some nasty details for getting Force/NoSteal to work...



Preferred Policy: Steal/No-Force

- This combination is most complicated but allows for highest performance.
- **NO FORCE (complicates enforcing Durability)**
 - What if system crashes before a modified page written by a committed transaction makes it to disk?
 - Write as little as possible, in a convenient place, at commit time, to support REDOing modifications.
- **STEAL (complicates enforcing Atomicity)**
 - What if the Xact that performed updates aborts?
 - What if system crashes before Xact is finished?
 - Must remember the old value of P (to support UNDOing the write to page P).

Algoritme for UNDO/REDO:

- Bruk to lister med transaksjoner; en for de committede siden siste sjekkpunkt, og en for de aktive
- UNDO alle write_item-operasjonene til aktive transaksjoner. Operasjonene skal UNDOes i reversert rekkefølge som de ble skrevet til loggen.
- REDO alle write_item til committede transaksjoner fra loggen, i rekkefølgen de ble skrevet til loggen.

Shadow paging

- Betrakter databasen som den er laget av et fastsatt antall disksider (-blokker), n.
- Et **directory** (katalog) med n oppføringer blir laget, der det i-te oppføringen peker på den i-te databasesiden på disk. I hovedminnet om den ikke er for stor.
- Bruker ikke logging, men lager kopier av data ved oppdatering. Commiter transaksjonen ved å kopiere inn pekere til nye data. Må ha katalog med pekere til data.

Write-ahead logging (WAL)

- Basis for undo/redo-logging
- Hver endring har en loggpost i loggen
- Regler:
 - Skriv en loggpost som endret en datablokk til disk før du skriver datablokken
 - Skriv loggen til disk før en transaksjon committer. «force log at commit»
- WAL-konsepter i ARIES:
 - LSN - loggsekvensnummer. ID for loggpost. Stigende nummer
 - PageLSN – LSN til loggpost som sist endret en blokk. Det er et felt i datablokker som forteller om hvilken loggpost so sist endret datablokken. Brukes for å vurdere om en loggpost trenger å gjøre REDO av blokken, og sjekker om loggen er skrevet før blokka. Det sjekker den når den skal skrive data, om loggen er skrevet helt fr etl og med pageLSN. Hvis ikke, skriv loggen først. Dette er konseptet med WAL.
 - FlushedLSN – LSN til nyeste skrevne loggpost til disk
 - Ved skriving av datablokk til disk. Sjekk pageLSN < flushedLSN

§ Hvis ikke: skriv (flush) logg først

ARIES

- En recovery-algoritme
- Bruker Steal/no-force-tilmærming for skrijving og er basert på 3 konsepter
 - Write-ahead-logging
 - Repeating history during redo
 - § ARIES vil gjenopprette alle handlinger i databasesystemet før krasjet skjedde.
 - § Alle transaksjoner som var ucommitted da det krasjet (aktive transaksjoner) blir undone.
 - Logging changes during undo
 - § Hindrer ARIES fra å gjenta de ferdige undo-operasjonene dersom svikt skjer under gjenoppretting, som forårsaker restart av gjenopprettingen.
- 3 hovedsteg i ARIES:
 1. analyse
 - Identifiserer dirty (uoppdaterte) pages i bufferet og de transaksjonene som var aktive ved krasjet. Fastslår også steder der REDO-operasjonen bør starte. REDO-fasen gjentar endringer fra loggen til databasen (som regel kun på committede transaksjoner, men ikke på ARIES)
 2. REDO
 - ARIES finner ut av hvilke som må REDOes og ikke, slik at man ikke trenger å REDOe alt.
 - REDOer kun det som er nødvendig
 - Loggposten trenger ikke REDO hvis:
 - i. Den tilhørende blokka ikke er i DPT
 - ii. Blokken er i DPT, men blokkens recLSN > LSN til loggposten
 - iii. Blokkens pageLSN >= loggpostens LSN. Her må blokken leses inn.
 - Hvis REDO:
 - i. Sett inn / skriv after image inn i blokka
 - ii. Oppdater blokkens pageLSN til loggpostens LSN.
 3. UNDO
 - Loggen blir skannet baklengs og operasjonene til transaksjonene som var aktive ved krasjet blir UNDONE i motsatt rekkefølge
- ARIES trenger informasjon om logg, Transaction Table og DPT. Bruker også sjekkpunkt
- Hver logrecord har et **log sequence number (LSN)** – Denne øker og indikerer adressen til denne loggposten på disken
 - Hver LSN korresponderer til en spesiell endring / handling av en transaksjon

- Hver data page vil lagre LSN til den siste loggposten som tilhører en endring for den siden
- Loggpost for write, commit, abort, undo og end.
- Når en oppdatering er undone, en kompensasjonsloggpost blir skrevet i loggen slik at undo ikke må bli gjentatt
- Når en transaksjon er ferdig (abort eller commit), så lages en sluttloggpost (end log record)
- Har også **transaksjonstabell** og **dirty page table** i tillegg til loggen
- **Transaksjonstabell**
 - Har en linje for hver aktive transaksjon
 - transaksjonsID, transaksjonsstatus og LSN til den siste loggposten for denne transaksjonen.
- **Dirty Page Table**
 - En linje for hver skitne side (page) i DBMS-bufferet
 - Har PageID og LSN til den tidligste oppdateringen av siden
- **Checkpointing**
 - Skriver begin_checkpoint-post til loggen, skriver end_checkpoint og skriver LSN til begin_checkpoint-posten til en spesiell fil.
 - Denne filen blir aksessert under recovery for å allokere den siste sjekkpunkt-informasjonen.
 - Med e_c-posten, så blir innholdet av TT og DPT lagt til i enden av loggen.
 - DBMS lager et sjekkpunkt i loggen periodisk. Dette skal minimere tiden det tar for å gjøre recovery
 - § Slipper å scanne hele loggen ved recovery (b_c, e_c, lagre LSN til sjekkpunktloggpost på et sikkert/fast sted)
- Abortering av transaksjon
 - Finn LastLSN fra transaksjonstabellen
 - For hver loggpost i transaksjonen (bakover): lag CLR (kompenserende loggpost) som gjør det motsatte, og gjør REDO av CLR-en
 - Fjern transaksjonen fra transaksjonstabellen
 - CLR-en er grunnlag for låser på radnivå (mer presise)
- Etter et krasj tar ARIES recovery manager over.
 - Aksesserer først last checkpoint fra den spesielle fila
 - Analysefasen starter med b_c-posten, og går til slutten av loggen'
 - E_c-posten er funnet, så aksesserer TT og DPT
 - **REDO-fasen**
 - § Finner den minste LSN, M , fr DPT. Det er denne loggposten der ARIES må starte REDO-fasen. Der $LSN < M$, så må endringer ha blitt skrevet til disk.
 - § Hvis en endring i loggen (med $LSN=N$) som gjelder page P og DPT inneholder en linje for P med $LSN > N$, så er endringene allerede present.

- § Hvis ingen av disse to forholdene stemmer, page P blir skrevet fra disk og LSN som er lagret på denne siden (LSN(P)) blir sammenlignet med N.
 - Hvis $N < \text{LSN}(P)$, så har endringene blitt gjort og siden trenger ikke å bli skrevet på nytt til disk.
- **UNDO.fasen**
 - § Skanner seg bakover fra slutten og angret alle passende handlinger
 - § Så skriver man en kompenserende loggpost for hver handling som blir angret
 - § UNDO leser baklengs i loggen inntil hver handling i i undo-settet har blitt angret.
- SÅ FERDIG MED RECOVERY-PROSESSEN

UNDO/REDO

- Alle oppdateringer av en transaksjon må bli registrert på disk før en transaksjon commiter, slik at REDO ikke er nødvendig. Må bruke **steal/force-strategy** for å bestemme når oppdaterte hovedminne-buffer blir skrevet tilbake på disk.
- Dersom transaksjonen kan commite før alle endringer er skrevet til database, så har vi **UNDO/REDO recovery algoritmen** Da blir **steal/no-force strategy** lagt til. Mest vanlig og mest kompleks.
- UNDO/REDO med sjekkpunkter
 1. bruke t lister med transaksjoner opprettholdt av systemet: de committede transaksjonene fra siste sjekkpunkt og de aktive transaksjonene
 2. Undo alle write_item-operasjonene til de aktive (ikke committede) transaksjonene, med UNDO-prosedyren. Operasjonene bør gjøres i motsatt rekkefølge av hvordan de ble skrevet til loggen
 3. Redo alle write_item-operasjonene fra de committede transaksjonene fra loggen, i rekkefølgen de ble skrevet til loggen, med REDO-prosedyren.
- UNDO: ser på old_value og setter verdien til denne – dette er before image.
 - Må gjøres i reversert rekkefølge

PageLSN: Et felt i datablokkene som forteller hvilken loggpost som sist endret blokken. Brukes for å

- Avgjøre om en loggpost på gjøre redo mot denne blokken
- Sjekke at loggen er skrevet før datablokken, dvs. når datablokken skal skrives, sjekkes det om loggen er skrevet frem til og med pageLSN. Hvis nei: skriv loggen først (WAL).

Lagring

- SQL-dictionary (katalog) beskriver hvordan en tabell/post er lagret: lengde, datatyper++

Heapfiler

- Rått og usortert lager av poster

- Poster settes inn på slutten av filen
- Aksesseres med en RecordIDBlockId, indeks annen blokk)
- Vanligvis har man indekser i tillegg til heapfiler
- Bra: lett å sette inn en post, god til tabellscan, bra skriveytelse
- Dårlig: dårlig til søk på attributter og rangesøk

Hashbaserte indekser

- Bra for direkte aksess på søkenøkkel
- Mod-funksjonen gir god spredning
- Håndtere overflyt
 - o Åpen adressering: lagre posten i første ledige etterfølgende blokk i fila
 - o Separat overløp: lenk sammen overløpsblokker
 - o Multippel hashing: bruk en ny hashfunksjon når det blir kollisjoner

Hashing

- Rask aksess under enkelte omstendigheter
 - o Hash key
- Ha en hash-funksjon eller random-funksjon
- Interne filer: hash-tabell (array of records)
- Garanterer ikke distinkte verdier hashes til distinkte adresser
 - o Hash field space (så mange mulige verdier som hashfeltet tar) er ofte mye større enn adresse-space
 - o Kollisjoner: må finne en løsning, annet sted: collision resolution
 - § Open addressing
 - § Chaining
 - § Multiple hashing
- Burde være 70-90% full for å unngå kollisjoner og ikke sløser bort mye plass
- Hashing for diskfiler: ekstern hashing
 - o The target address space is made of buckets – hver holder mange records/poster
 - § Mindre kollisjoner siden det kan være flere i en
 - § Kan lage kjeder – må da ha en record pointer

Statisk hashing

- Det over er statisk hashing
- Statisk fordi vi har et fast antall blokker M. Key-to-address-mapping
- Ulempe: the hash address space is fixed – kan ikke øke eller minske filen dynamisk
 - o Mange overløp kan ødelegge ytelsen
- Derfor innført extendible hashing

Extendible hashing

- En array med 2^d blokkadresser er laget, der d er den globale dybden
- Problem: Utvider filen, altså doubler antall blokker. Les alle blokker og skriv alle blokker på nytt. Dette er stress
- Bruker derfor extendible hashing i stedet
- Bruker katalog med pekere til blokker og doubler katalogen ved behov

- Splitt (les og skriv) kun den blokken som ble full
- Bra hvis antall poster er usikre og kan bli stor, og aksess skjer via primærnøkler
- Lokal og global dybde
- Hvis en blokk er full og lokal dybde==global dybde

Indekser

- Indeksfelt – felt/attributt av posten som indeksen bruker
- Primærindeks: indeks på primærnøkkel
- Clustered indeks: indeks på tabell hvor postene er fysisk lagret sammen med indeksen.
- Sekundærindeks: ekstra indeks på et annet felt
 - o Kan være brukt for å tvinge igjennom UNIQUE, dvs. unik verdi for hver post i tabellen

Lagrings-og indekseringsmuligheter

- Clustered B+-tre / clustered index
 - o På primærnøkkel
 - o Løvnivå er selve posten
 - o SQL server: clustered index når primærnøkkel er definert
- Heapfil og B+-tre
 - o Tabell lagret i heapfil
 - o B+-tre på primærnøkkel
 - o SQL server: Heap + unclustered index
- Heapfil (uten indeks)
 - o Postene lagres fortløpende uten organisering
 - o SQL server: hvis primærnøkkel ikke er definert
- Clustered hash index
 - o Hashindex på primærnøkkel
 - o Nyttig når du ikke trenger sortering, kun lett oppslag
 - o Posten lagret i indeksen
- LSM-trees
 - o Moderne
 - o «cacher de nyeste innsatte/oppdaterte postene
 - o Viktig i Big Data
 - o Høy skriveytelse, bedre komprimering
 - o Eldre poster flyttes til langtidslager
- Column stores
 - o Lagres i kolonner i stedet for rader
 - o Leser mindre data ved queries
 - o Slipper å gå igjennom alt

B+-trær

- Den mest brukte indeksen
- Høyde: 2-4
- Postene sortert på nøkkelen og treet støtter
 - o Likhetsøk

- Verdiområdesøk
- Sekvensielle, sorterte scan
- Gode også på dynamiske datamengder
- Ikke så bra for Big data

Teknikker for å utføre relasjonsalgebraoperasjoner

- Indeksering: bruke WHERE-uttrykk til å trekke ut små mengder poster (seleksjon, join)
- Iterasjon: ofte er det raskest å scanne hele tabeller
- Partisjonering: sortering og hashing av input gir operasjoner på mindre datamengder

Aksessvei

- Access path / search method
 - Optimalisatoren velger den billigste aksessveien
 - Måles i antall blokker som aksesseres
 - 1. filscan
2. indeks (indeksscan, rangescan, index lookup)

Flettesortering

1. init sortering
 - b: antall blokker
 - n_r: antall sorterte delfiler
 - n_b: plass i buffer
 - n_r er takfunksjonen av b/n_b
2. fletting
 - flettegrad, d_m: $\min(n_r, n_b - 1)$
 - flettepass: takfunksjonen av $\log_{d_m}(n_r)$
 - total I/O:
 - 1) $- 2 * b$
 - 2) $- 2 * b * \text{flettepass}$

Seleksjon

- CASE: vi skal finne alle med etternavn på «L», 10% tilfredsstillende, 1024 blokker i heapfil, 200 poster i hver blokk = 204800 poster
- 1. Heapfil: scanne gjennom alle = 1024
- 2. clustered B+-tre: antar høyde 3. 1536 ($1,5 * 1024$) blokker på løvnivå. 10% av 1536 = 156 blokker
- 3. unclustered og heapfil: antar B+-tre er 20% av 1536 (=308). 10% av 308 + 2 = 33. Så følger vi RecordId til alle poster som kvalifiserer. $204800 * 10\% = 20480$. Til sammen 20513 blokker.

JOIN

- Nested loop join – for hver blokk i den ene tabellen, scan hele den andre og se etter match
- Single loop join – loop gjennom den ene og bruk indeks for å slå opp i den andre
- Sort merge join – hvis begge tabellene er sorterte, kan de flettes,

- Partition hash join – partisjoner tabeller ved hashing på joinattributtene. Får mange partisjoner som kan joines parvis i RAM
- OPPGAVE: vi har 2 tabeller. Employee med 10 diskblokker, department med 2000. RAM har 7 blokker. Da får vi ha alltid den minste tabellen ytterst

Diverse teori fra eksamensoppgaver

Redundans:

- Samme info lagret flere steder
- Fører til inkonsistens
- Kan føre til anomalier ved innsetting, oppdatering og sletting av data i databasen
- Gjennom normaliseringsprosessen vil vi fjerne inkonsistens og blir kvitt anomaliene
- Ulempen er at vi kan ende opp med mange tabeller som vi må joine, og vi kan misse en av kravene når vi splitter
- Ved å tillate noe redundans kan vi noen ganger få enklere relasjonsskjema som gir enklere og mer effektive spørringer, men vi må håndtere redundansbivirkninger
- .
- Kan være nyttig i lagring for å oppnå høyere tilgjengelighet, ytelse og sikkerhet.

NULL-verdi: en spesiell verdi som kommer i tillegg til en datatype sine ordinære verdier. Denne brukes når et attributt ikke har noen verdi eller at den er ukjent. Attributter er av ulike typer, så uten NULL ville en av verdiene i verdimengden tatt seg av denne oppgaven, og da hadde det vært forskjellig for ulike datatyper som hadde krevd dokumentasjon og forutsatt kjennskap til denne tilordningen blant alle som bruker databaren. Den er standardisert. I tillegg ville da denne verdien (for eksempel 0) ikke kunne brukes til det den egentlig skal.