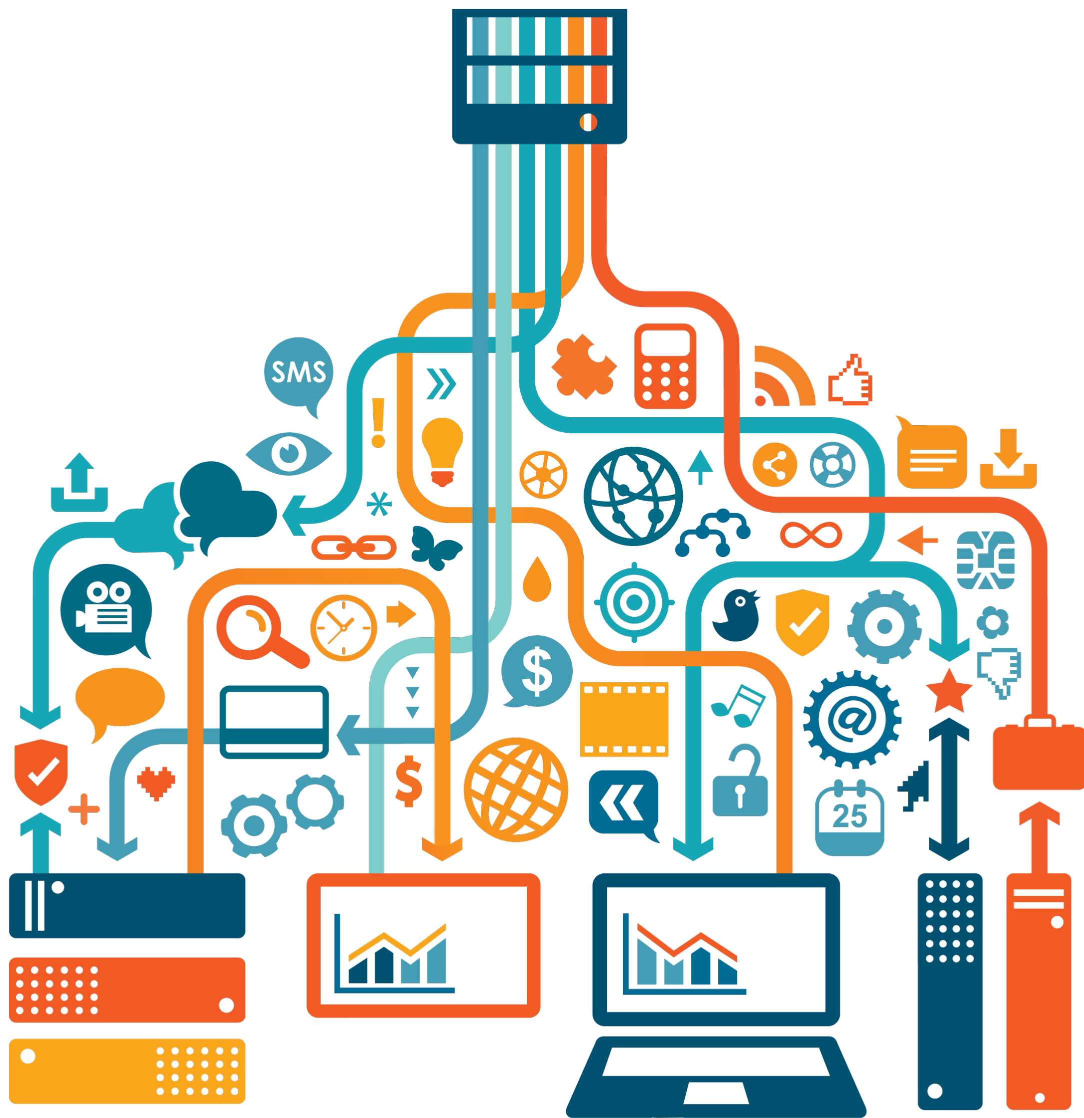


Kompendium – Datamodellering og Database

Mathilde Haukø Haugum

Dette er et kompendium for emnet TDT4145 Datamodellering og Database som ble undervist våren 2019 ved NTNU. Kompendiet er basert på boka Fundamentals of Database Systems av Elmasri og Navathe. Hver del av kompendiet består av notater fra boka etterfulgt av en oppsummering av det viktigste basert på videoforelesningene. Disse oppsummeringene vil også være mer rettet mot praksis arbeid med databaser

Del 1 – Introduksjon til databaser

Kapittel 1 – Databaser og databasebrukere

Databaser og databasesystemer er en essensiell komponent i det moderne samfunnet. Når vi henter penger fra en bankkonto eller kjøper noe på nettet vil en person eller en datamaskin aksessere en database. Disse interaksjonene er eksempler på **tradisjonelle database applikasjoner**, der meste av informasjon er enten tekst eller nummer. Sosial media applikasjoner, slik som Facebook, krever enorme databaser som lagrer ikke-tradisjonell data, slik som poster, bilder og videoklipp. Derfor har det blitt utviklet nye typer databasesystemer som ofte kalles **big data** lagringssystem eller **NOSQL system**.

1.1 Introduksjon

En database er en samling av relaterte data, der data er kjent fakta som kan tas opp og har en mening. For eksempel vil mobiler ha en database programvare som lagrer navn og telefonnummer til personer du kjenner. En database har følgende egenskaper:

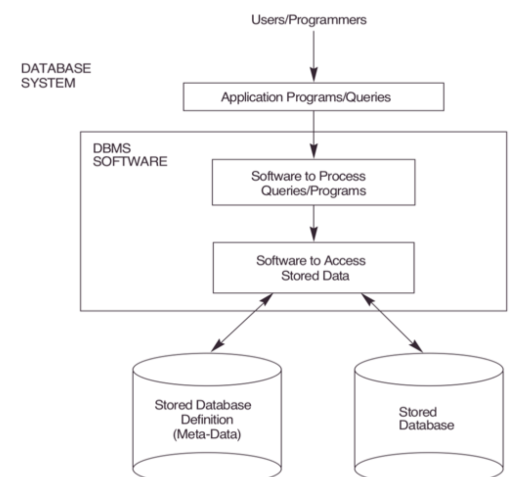
- Representerer en del av den virkelige verden, kalt en **miniverden** eller **universet av diskurs (UoD)**. Endringer i miniverden reflekteres i databasen
- Er en sammenhengende samling av data med en inneboende mening. Et tilfeldig utvalg av data kan ikke kalles en database
- Blir designet, bygget og fylt med data for et spesielt formål. Databasen har en tiltenkt gruppe brukere.

En database har en kilde der dataen kommer fra, påvirkes av hendelser i den virkelige verden og har et publikum som er aktivt interessert i innholdet. **Databasen må være en sann refleksjon av miniverden den representerer og må derfor oppdateres så fort som mulig ved endringer**. Databasen kan genereres og opprettholdes manuelt (eks: bibliotekkort katalog) eller den kan være datastyrt.

Et database håndteringssystem (DBMS) er et datastyrt system som lar brukere lage og opprettholde en database. DBMS er et programvaresystem (*software*) som legger til rette for:

- **Definering** = spesifisere datatyper, strukturer og begrensninger til dataen
- **Konstruksjon** = lagre dataen på et medium som kontrolleres av DBMS
- **Manipulasjon** = henting av spesifikk data, oppdatering pga. endring i miniverden, osv.
- **Deling** = lar flere brukere og programmer få tilgang til databasen samtidig.

Programmer får tilgang til databasen ved å sende forespørsler om data til DBMS. Ved en transaksjon kan noe data bli lest eller skrevet inn i databasen. Andre viktige funksjoner ved DBMS inkluderer beskyttelse av database og vedlikehold/bruk over lengre tid (oppnås vha tuning). Beskyttelse innebærer systembeskyttelse fra funksjonsfeil og sikkerhet mot uautorisert eller skadelig tilgang. **DBMS og databasen kombinert kalles databasesystemet.**



STUDENT	Name	StudentNumber	Class	Major
	Smith	17	1	CS
	Brown	8	2	CS

COURSE	CourseName	CourseNumber	CreditHours	Department
	Intro to Computer Science	CS1310	4	CS
	Data Structures	CS3320	4	CS
	Discrete Mathematics	MATH2410	3	MATH
	Database	CS3380	3	CS

SECTION	SectionIdentifier	CourseNumber	Semester	Year	Instructor
	85	MATH2410	Fall	98	King
	92	CS1310	Fall	98	Anderson
	102	CS3320	Spring	99	Knuth
	112	MATH2410	Fall	99	Chang
	119	CS1310	Fall	99	Anderson
	135	CS3380	Fall	99	Stone

GRADE_REPORT	StudentNumber	SectionIdentifier	Grade
	17	112	B
	17	119	C
	8	85	A
	8	92	A
	8	102	B
	8	135	A

PREREQUISITE	CourseNumber	PrerequisiteNumber
	CS3380	CS3320
	CS3380	MATH2410
	CS3320	CS1310

1.2 Et eksempel

Figuren viser UNIVERSITET database som består av fem filer, der **hver fil lagrer datapost av samme type** (eks: STUDENT lagrer data om studenter). For å definere databasen må vi spesifisere datatypene som skal lagres i hver post, f.eks. skal hver post i STUDENT inkludere dataelementene Name, StudentNumber, Class og Major. Videre må vi oppgi datatypen til hvert dataelement (eks: Name = streng). For å konstruere databasen vil vi lagre dataen som poster i filene. **Det vil være relasjoner mellom poster i de ulike filene**. For eksempel kan registret for Ola i STUDENT filen være relatert til to registre i GRADE_REPORT og COURSE filene som gir hhv. Karakter og emne. Manipulasjon involverer forespørsler og oppdateringer, for eksempel liste navnet til studentene som tok emnet 'Database' eller legge til karakter 'A' for 'Ola' i 'Database'.

1.3 Karakterisering av database tilnærming

Ved tradisjonell **filbehandling** vil hver bruker definere og implementere filene den trenger som en del av programmeringen av applikasjonen. For eksempel kan karakterkontoret ha filer for studenter og karakterer og emnekontoret har filer for studenter og emner. Begge er interessert i data om studenter og har separate filer, noe som fører til overflødig definering og lagring av data. **Ved database tilnærmingen vil dataen defineres én gang og kan deles av flere ulike brukere.** Vi skal nå se på egenskaper ved database tilnærmingen.

Selvbeskrivende natur

Databasesystemet inneholder en **DBMS katalog** som lagrer beskrivelsen av databasen (eks: datastruktur, typer og begrensninger). Denne beskrivelsen kalles **metadata**. **Databasesystemer har en selvbeskrivende natur, siden det ikke bare inneholder databasen, men også metadata som definerer og beskriver dataen** (dvs. gir datastruktur, typer og begrensninger). DBMS programvaren blir ikke skrevet for spesifikke databaser, og må derfor referere til katalogen for å få vite blant annet typen til dataen den skal aksessere. **Metadata gjør at DBMS kan arbeide med flere ulike typer databaser og generelle databaseapplikasjon**. Figuren viser noe av DBMS katalogen for eksempelet over. Ved forespørsel om aksess til Name i en STUDENT post, vil DBMS programvaren undersøke katalogen for å bestemme strukturen til STUDENT filen og posisjonen og størrelsen til Name elementet i STUDENT posten.

RELATIONS

Relation_name	No_of_columns
STUDENT	4
COURSE	4
SECTION	5
GRADE_REPORT	3
PREREQUISITE	2

COLUMNS

Column_name	Data_type	Belongs_to_relation
Name	Character (30)	STUDENT
Student_number	Character (4)	STUDENT
Class	Integer (1)	STUDENT
Major	Major_type	STUDENT
Course_name	Character (10)	COURSE
Course_number	XXXXNNNN	COURSE
....	....	.....
....	....	.....
....	....	.....
Prerequisite_number	XXXXNNNN	PREREQUISITE

Dataabstraksjon – skille mellom program og data

Ved tradisjonell filbehandling vil strukturen til filene være en del av applikasjonsprogrammet, så endring i filstrukturen kan kreve endring av alle programmer som har tilgang til filen. Dersom vi ønsker å legge til Birth_data til hver STUDENT post, må vi endre hele programmet. DBMS aksessprogrammer krever ingen slike endringer, fordi **strukturen til datafilene er lagret i**

DBMS katalogen separat fra aksessprogrammene. Dette kalles **program-data uavhengighet**. For å legge til Birth_data trenger vi kun å endre beskrivelsen av STUDENT registrene i katalogen og ingen program blir endret. Neste gang DBMS programmet refererer til katalogen vil den nye strukturen til STUDENT posten brukes. **Denne uavhengigheten avhenger av dataabstraksjon, og det går ut på at DBMS bruker datamodeller for å gi lite informasjon om lagring og implementasjon av dataen til brukeren**. Brukerne får heller informasjon om objekter og deres egenskaper og relasjoner (= konseptuell representasjon av dataen). Ved referanse til Name i en STUDENT post, vil brukeren som regel være mer interessert i at riktig verdi blir returnert enn at lokasjonen til dataelementet blir gitt. Det finnes flere typer datamodeller som brukes for å gi en slik dataabstraksjon til brukerne av databasen.

Støtter flere visninger av data

En database vil ofte ha flere brukere som har ulikt perspektiv eller syn på databasen, altså kan de være interessert i ulike deler av databasen. DBMS med flere brukere må derfor ha tjenester som definerer flere typer visninger. Figuren viser to syn som er hentet fra UNIVERSITET databasen. Bruker A er interessert i å hente karakterene til studentene, mens bruker B er interessert i å sjekke forutsetningene for emnene.

(a)

TRANSCRIPT	StudentName	Student Transcript				
		CourseNumber	Grade	Semester	Year	SectionId
	Smith	CS1310	C	Fall	99	119
		MATH2410	B	Fall	99	112
	Brown	MATH2410	A	Fall	98	85
		CS1310	A	Fall	98	92
		CS3320	B	Spring	99	102
		CS3380	A	Fall	99	135

(b)

PREREQUISITES	CourseName	CourseNumber	Prerequisites
	Database	CS3380	CS3320
			MATH2410
	Data Structures	CS3320	CS1310

Deling av data og flerbruker transaksjonsbehandling

En flerbruker DBMS må la flere brukere få tilgang til databasen samtidig. DBMS må ha programvare for **samtidighetskontroll**, slik at når flere brukere prøver å oppdatere den samme dataen, så blir dette gjort på en kontrollert og effektiv måte. For eksempel ved salg av flyseter må DBMS sikre at kun én bruker får reservere ett bestemt sete. Dette kalles **OLTP** (Online Transaksjon Prosessering) og lar flere hundre samtidige transaksjoner utføres per sekund.

Transaksjoner er utførende programmer som inkluderer aksess til en eller flere databaser, for eksempel lesing eller oppdatering av database post. **Samtidighetskontroll garanterer at hver transaksjon blir riktig utført eller avbrutt.** DBMS må sikre flere egenskaper ved transaksjoner, for eksempel **isolasjon** (hver transaksjon ser ut til å utføres isolert fra andre som utføres samtidig) og **atomisitet** (alle operasjonene i en transaksjon blir utført eller ingen).

1.4 Brukertyper

I store organisasjoner vil mange personer være involvert i **designet og opprettholdelsen av en større database**:

- **Database administratorer (DBA)** – styrer databasen, DBMS og relatert programvare som blir brukt av mange personer. DBA autoriserer aksess til databasen, koordinerer bruken, henter program- og maskinvare som trengs og sørger for sikkerhet og god responstid.
- **Database designere** – identifiserer dataen som skal lagres i databasen og velger passende strukturer for å representere og lagre denne dataen.
- **Endebrukere** – brukerne som er grunnen til at databasen eksisterer.
- **System analytikere og applikasjonsprogrammerere** – utvikler programvaren og bestemmer kravene til brukerne, tester, debugger, osv.

1.5 Arbeidertyper

Andre personer er ansvarlig for **design, utvikling og operasjon av DBMS programvaren og systemmiljøet**. Disse er som regel ikke interessert i selve databaseinnholdet. Arbeidstypene er DBMS systemdesignere og implementere, verktøysutviklere (designer og utvikler programvare pakker som legger til rette for design av databaser) og personell for operasjon og vedlikehold

1.6 Fordeler med DBMS tilnærming

Noen flere fordeler ved å bruke en DBMS:

- **Kontroll av redundans** – ved tradisjonell filbehandling må hver bruker opprettholde sine egne filer, noe som fører til duplikater av samme fil. Dette gir bortkastet lagringsplass og mer arbeid for å oppdatere dataen. Ved DBMS tilnærming blir dataen ideelt sett lagret ved kun én plass i databasen = data normalisering. I noen tilfeller ønsker brukerne å hente ut samlinger av data samtidig (eks: emne, student og karakter), og derfor kan det hende at duplikater blir lagret i bestemte filer = denormalisering. Dataen blir plassert samlet for at DBMS skal slippe å søke gjennom flere filer. DBMS bruker redundanskontroll for å sikre at disse duplikatene er riktige.
- **Begrenser uautorisert tilgang** – som regel vil databasen inneholde informasjon som kun autoriserte brukere skal ha tilgang til. Det kan også hende at det er begrenset hvilken operasjon som er tillatt, for eksempel kun lesing. En DBMS skal gi et delsystem for sikkerhet og autorisering.
- **Gir vedvarende lagring for programobjekter** – et kompleks objekt skrevet i C++/Java kan lagres permanent i en objektorientert DBMS, slik at den «overlever» terminering av programmet og kan senere direkte hentes av et annet program.
- **Gir lagringsstruktur og søketeknikker for effektiv forespørsel** – databasen blir ofte lagret på disken, så DBMS må gi spesialiserte datastrukturer (eks: buffer) og søketeknikker for å gjøre søk raskere og mer effektivt.
- **Gir backup og gjenoppretting** – DBMS legger til rette for gjenoppretting etter feil i maskinvare eller programvare.
- **Gir grensesnitt for flere brukere** – siden flere brukere med ulik grad av teknisk kunnskap bruker databasen, må DBMS gi ulike brukergrensesnitt (GUIs).

1.8 Tilfeller der DBMS ikke bør brukes

I noen tilfeller vil DBMS føre til unødvendig driftskostnader som ikke er tilfellet ved tradisjonell filbehandling. Disse kostnadene skyldes:

- Høy initial investering i maskinvare, programvare og trening
- Generaliteten DBMS gir for å definere og behandle data
- Drift som gir sikkerhet, samtidighetskontroll, gjenoppretting og integritet

Situasjoner der det er ønsket å utvikle tilpasset database applikasjoner er når applikasjonen er enkel, veldefinert, ikke endres og har spesielle krav for sanntid, når systemet har begrenset lagringskapasitet og når det ikke er flere brukere som vil ha tilgang.

Oppsummering – Kapittel 1 (F2, Øv1)

En **database** er en samling av relaterte data, og data kan vi tenke på som «recorded facts» (dvs. informasjon satt i et system). Et **databasehåndteringssystem** (DBMS) er et kort forklart en software-pakke som støtter det å lage og vedlikeholde databaser mellom brukere og applikasjoner. Noen fordeler ved å bruke et databasesystem i motsetning til tradisjonelle filsystemer er:

1. **Program-data uavhengighet** – databasesystemet separerer databasekoden fra annen programvarekode, altså strukturen til datafilene er lagret i DBMS katalogen isolert fra programmene. Dette gjør at strukturen til filene kan endres uten at det krever noen endring i alle programmene som har tilgang til filene.
2. **Flerbrukerstøtte** – databasesystemet gir mange brukere tilgang til å hente data eller gjøre endringer i databasen, gjerne samtidig.
3. **Selvbeskrivende** – databasesystemet kan i stor grad beskrive sin egen struktur med mye metadata. Dette er nødvendig for at et DBMS skal fungere for en generell databaseapplikasjon, altså at den skal kunne arbeide med flere ulike typer databaser. Dette gjør også at man kan hente ut veldig konkret data fra databasen som matcher gitte krav spesifisert i et spørrespråk og at systemet kan lagre dataen i datastrukturer som er effektive, basert på spørringene som gjøres og hva slags data man har.

Kapittel 2 – Konsepter og arkitektur

Den moderne DBMS bruker en klient/server arkitektur som er distribuert, dvs. består av flere tusen datamaskiner som styrer lagringen av data. Store sentraliserte hovedrammemaskiner blir erstattet av mange distribuerte arbeidsstasjoner som er koblet til servermaskiner over nettverk. Datamiljø over skyen består av tusenvis av store servere som styrer big-data for brukere på nettet. DBMS arkitekturen består av en klient modul som håndterer brukerinteraksjon og server modul som håndterer data lagring, aksess, søk, osv.

2.1 Datamodeller, skjemaer og instanser

Database tilnærmingen gir **dataabstraksjon**, der detaljer om organisering og lagring av data blir «skjult» og de essensielle detaljene blir fremhevet. Det er flere nivåer med dataabstraksjon, slik at ulike brukere kan få ulike mengde detaljer etter behov. **En datamodell beskriver strukturen til en database og brukes for å oppnå abstraksjon**. Strukturen til databasen er datatyper, relasjoner, begrensninger og grunnleggende operasjoner. Datamodellen kan også inneholde konsepter som spesifiserer oppførelsen til en database applikasjon. Database designeren kan dermed spesifisere hvilke brukeroperasjoner som er gyldige på database objektene.

Kategorier av datamodeller

Vi kan kategorisere datamodellene etter hvilke typer konsepter de bruker for å beskrive strukturen til databasen:

- **Høy-nivå/konseptuelle datamodeller** – konseptene ligner måten brukerne oppfatter dataen og er basert på entiteter, attributter og relasjoner. En **entitet** er et objekt fra miniverden som databasen beskriver (eks: ansatt), en **attributt** er en egenskap som beskriver entiteten (eks: navn) og en **relasjon** er en assosiasjon mellom entiteter (eks: kollegaer).
- **Lav-nivå/fysiske datamodeller** – konseptene beskriver detaljer om hvordan data lagres på lagringsmediet (eks: magnetisk disk). De beskriver hvordan data blir lagret som filer i datamaskiner ved å gi postformat, postordning og aksessbaner (gjør søk etter bestemt database post effektivt, eks: indeks og hashing).
- **Representasjon/implementasjon datamodeller** – mellomting, dvs. gir konsepter som endebrukere lett kan forstå, samtidig som det ikke er så ulikt måten dataen er organisert i lagringen. Dataen blir representert vha poststrukturer og derfor blir de ofte kalt post-basert datamodeller. Disse modellene er mye brukt og inkluderer blant annet relasjonsmodellen og nettverk og hierarki modeller.
- **Selv-beskrivende datamodeller** – databeskrivelsene og dataverdiene blir kombinert i lagringen. Inkluderer XML, NOSQL systemer, osv.

Skjemaer, instanser og database tilstander

I en datamodell er det viktig å skille mellom beskrivelsen av databasen og selve databasen. **Database skjemaet er beskrivelsen av databasen** som blir spesifisert ila designet av databasen og blir ikke ofte endret. Et skjemadiagram vil vise strukturen, men ikke instansene (se figur). Skjema diagrammet vil kun vise deler av skjemaet og vil for eksempel ikke inkludere datatyper eller

STUDENT

Name	StudentNumber	Class	Major

COURSE

CourseName	CourseNumber	CreditHours	Department

PREREQUISITE

CourseNumber	PrerequisiteNumber

SECTION

SectionIdentifier	CourseNumber	Semester	Year	Instructor

GRADE_REPORT

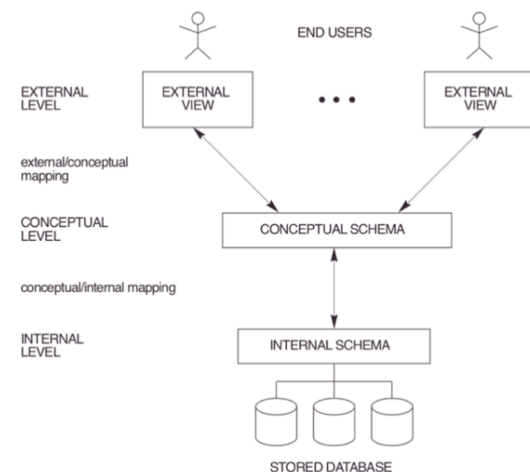
StudentNumber	SectionIdentifier	Grade

kompliserte begrensninger. Dataen i en database kan endres ofte, og **databasetilstand er dataen ved et bestemt øyeblikk**.

Dette skillet er svært viktig! Når vi definerer en ny database vil vi spesifisere databaseskjemaet til DBMS. Da vil vi få en database tilstand som er tom. Den initiale tilstanden blir laget når databasen blir lastet med initial data. Fra da av vil hver oppdatering gi en ny database tilstand. DBMS er delvis ansvarlig for å sikre at disse tilstandene er gyldige, altså tilfredsstiller strukturen og begrensningene gitt i skjemaet. DBMS vil derfor lagre skjemaet (meta-dataen) i DBMS katalogen, slik at det kan hentes frem ved behov. Skjemaevolusjon er endring av skjemaet, for eksempel at vi legger til Date_of_birth til STUDENT skjemaet.

2.2 Tre-skjema arkitektur og datauavhengighet

Tre viktige egenskaper er (1) bruk av katalog for å lagre beskrivelsen av databasen slik at den blir selv-beskrivende, (2) isolering av programmer og data og (3) støtte av flere brukere. For å oppnå disse brukes tre-skjema arkitektur.



Tre-skjema arkitektur

Målet til tre-skjema arkitekturen er å separere bruker applikasjonene fra den fysiske databasen, og den består av tre nivåer:

1. **Internt skjema** – har et internt skjema som bruker en fysisk datamodell for å beskrive lagringsstruktur og aksessbaner.
2. **Konseptuelt skjema** – har et konseptuelt skjema som beskriver strukturen til hele databasen for en samling brukere. Den skjuler detaljer om fysisk lagring og beskriver entiteter, datatyper, relasjoner, brukeroperasjoner og begrensninger. Bruker som regel en representasjon datamodell.
3. **Eksternt skjema** – inkluderer en rekke eksterne skjemaer eller brukervisninger. Hvert eksternt skjema beskriver en del av databasen som en bestemt brukergruppe er interessert i og skjuler resten av databasen fra denne gruppen. Hvert eksternt skjema bruker som regel en representasjon datamodell.
4.

Et database skjema er altså en beskrivelse av databasen, og ved tre-skjema arkitektur blir denne beskrivelsen delt opp i tre nivåer som er separert fra hverandre.

Datauavhengighet

Datauavhengighet er evnen til å endre skjemaet ved ett nivå i databasesystemet uten å må endre skjemaet ved neste høyrere nivå. Det er kun kartleggingen mellom de to nivåene som endres. Det er to typer:

1. **Logisk datauavhengighet** – evnen til å endre det konseptuelle skjemaet uten å må endre eksterne skjema eller applikasjonsprogram. Slike endringer inkluderer å utvide/redusere databasen (legge til/fjerne posttype eller dataelement) eller å endre begrensninger. Ved fjerning vil ikke eksterne skjema som referer til gjenværende data endres.
2. **Fysisk datauavhengighet** – evnen til å endre det interne skjemaet uten å må endre det konseptuelle skjemaet (og dermed også det eksterne skjemaet). Slike endringer inkluderer

reorganisering av fysiske filer for å forbedre aksessbaner. Så lenge den samme dataen er i databasen, trenger vi ikke å endre det konseptuelle skjemaet.

Fysisk datauavhengighet vil eksistere i de fleste databasene der fysiske detaljer (eks: lagringslokasjon) er skjult fra brukerne. Logisk datauavhengighet er vanskeligere å oppnå.

2.3 Database språk og grensesnitt

DBMS må gi passende språk og grensesnitt for de ulike brukertypene.

DBMS språk

Når databasen er designet og en DBMS er valgt til å implementere databasen, vil neste steg være å definere det interne og konseptuelle skjemaene og kartleggingen mellom disse. Hvis det ikke er et skille mellom disse vil **DDL (Data Definition Language)** brukes for å spesifisere begge skjemaene. Hvis det er et skille mellom disse vil DDL brukes for å spesifisere det konseptuelle skjemaet, mens **SDL (Storage Definition Language)** brukes for å spesifisere det interne skjemaet. For å oppnå tre-skjema arkitektur vil et tredje språk kalt **VDL (View Definition Language**) brukes for å spesifisere det eksterne skjemaet, men i de fleste DBMS blir DDL brukt. Når skjemaene er laget og databasen er fylt med data, må brukerne få mulighet til å manipulere databasen, dvs. hente, sette inn, slette og endre dataen. DBMS gir disse operasjonene vha språket **DML (Data Manipulation Language)**.

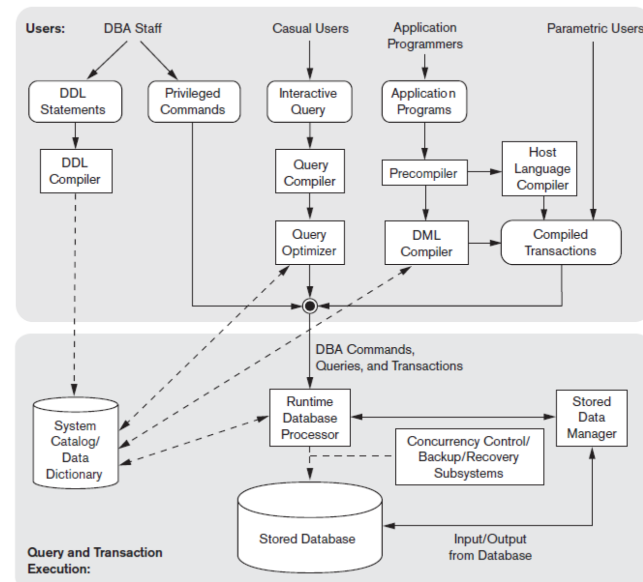
I nåværende DBMS blir språkene heller sett på som ett sammenhengende, integrert språk, for eksempel SQL som er en kombinasjon av DDL, VDL og DML. Definisjoner av lagringen (SDL) blir ofte holdt separat, siden det brukes for å finjustere ytelsen til databasesystemet.

2.4 Databasesystem miljø

En DBMS er et kompleks programvaresystem og består av flere programvare komponenter.

DBMS komponent moduler

Figuren viser de vanlige komponent modulene i en DBMS. Øvre del representerer de ulike brukerne av databasen og deres grensesnitt, mens nedre del representerer de interne modulene ansvarlig for lagring av data og behandling av transaksjoner. Databasen og DBMS blir som regel lagret på disk og aksess kontrolleres av **opererende system (OS)**. Mange DBMS har egne buffer management moduler som styrer buffer lagringen, siden dette har stor effekt på ytelsen.



Øvre del (brukere og grensesnitt):

- **DBA ansatte** bruker DDL påstander og kompilator for å spesifisere skjemaer og lagre beskrivelser av skjemaene (meta-data: lagringsdetaljer, størrelse, ...) i DBMS katalogen.
- **Uformelle brukere** og personer som iblant ønsker informasjon fra databasen interagerer via grensesnittet for interaktiv forespørsel.

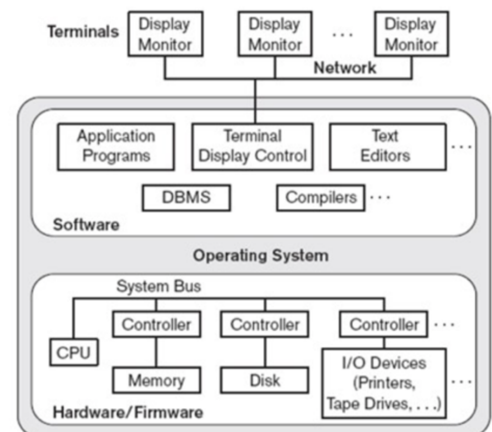
Forespørselen blir omdannet til intern form av kompilatoren og deretter vil optimalisatoren generere utførbar og effektiv kode.
- **Applikasjonsprogrammerere** skriver programmer på et vertsspråk (eks: Java) i et applikasjonsprogram. Prekompilatoren vil omdanne noe av språket til DML kommandoer som sendes til DML kompilatoren for å omdannes til objektkode som gir database aksess. Resten av programmet sendes til vertsspråk kompilatoren og blir deretter koblet til objektkoden for å lage transaksjoner.

Nedre del (interne moduler)
- **Runtime database Processor** vil utføre DBA kommandoene, forespørsler og transaksjoner. Den kan oppdatere system katalogen. Kontrollerer deler av dataoverføring, for eksempel buffere i hovedminnet.
- **Stored data manager** bruker OS tjenester for å utføre lese/skrive operasjoner mellom disk og hovedminne. Den kontrollerer tilgang til DBMS informasjon (database eller katalog) som er lagret på disken.
- **Samtidighetskontroll** og **gjenopprettingssystem** er egentlig integrert som en del av Runtime database prosessoren.

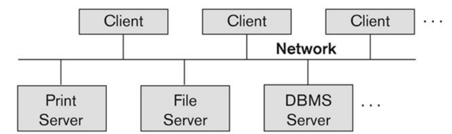
Databasesystem hjelpemidler
I tillegg til programvaremodulene over vil mange DBMS ha database hjelpemidler som DBA bruker for å styre databasesystemet. Disse hjelpemidlene har funksjoner innen lasting av datafiler inn i databasen, backup for gjenoppretting, reorganisering av lagringen, overvåkning av ytelse, sortering av filer, osv.

2.5 Sentralisert og klient/server arkitekturer for DBMS

Sentralisert DBMS arkitektur
Eldre arkitekturer brukte hovedramme datamaskiner, kalt sentralmaskiner, som stod for behandling av alle system funksjoner, inkludert DBMS funksjonalitet, brukergrensesnitt og utføring av applikasjonsprogram. Brukerne hadde tilgang til DBMS via display terminaler som var koblet til sentralmaskinen via kommuniserende nettverk. All behandling ble gjort på hovedramme datamaskinen og kun display informasjon ble sent til terminalene. Terminalene ble etterhvert erstattet med PC og mobile enheter. I begynnelsen brukte disse **sentralisert DBMS der all funksjonalitet ble utført på én maskin** (se figur), men etterhvert begynte DBMS systemet å utnytte behandlingskraften på brukersiden (= klient/server).



Klient/server arkitektur
Klient/server arkitekturen brukes i tilfeller der et stort antall PCer, arbeidsstasjoner, filservere, printere, osv. er koblet via et nettverk. **Klientmaskiner er koblet til spesialiserte servere som har spesifikk funksjonalitet,** for eksempel printserver som lar brukerne skrive ut filer. Brukerne får mulighet til å bruke serverne via passende grensesnitt gitt av klientmaskinene. Lokale

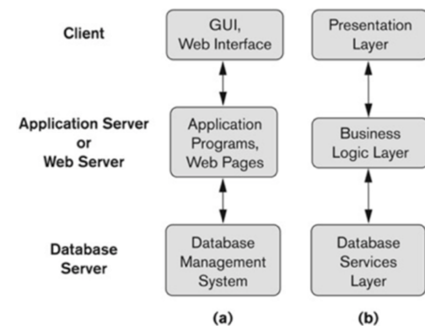


applikasjoner kan også kjøre på klientmaskinen. Figuren viser klient/server arkitektur ved logisk nivå. Maskiner kan være klienter, servere eller begge deler.

Klient/server arkitektur krever et underliggende rammeverk som består av mange PCer, arbeidsstasjoner, mobile enheter og server maskiner som er koblet sammen over et datanettverk. En **klient** er en brukermaskin som gir grensesnitt og lokal behandling, og ved behov for ekstra funksjonalitet vil den koble seg til en bestemt server vha grensesnitt og et kommunikasjonsnettverk (eks: LAN). En **server** er et system av maskin- og programvare som kan gi tjenester til klienter innen forespørsel og transaksjon.

Tre typer klient/server arkitektur for DBMS:

1. **To-tier – komponentene i programvaren er fordelt over to systemer: klienten og serveren**. Klientsiden kan kjøre brukergrensesnitt og applikasjonsprogram, og den må etablere kobling til DBMS som er på serversiden for å få tilgang til DBMS (= datakilde. Klient kan kobles til flere). Funksjonaliteten for forespørsel og transaksjon er fortsatt på serversiden. To-tier arkitektur er enkelt og kompatibelt med eksisterende system.
2. **Tre-tier** (se figur)– legger til et mellomliggende lag, kalt **applikasjonsserveren** eller **web server** (avhengig av applikasjonen), mellom klienten og database serveren. Klientsiden inneholder brukergrensesnitt og nettlesere. Den mellomliggende serveren mottar og behandler forespørsler fra klienten og sender spørsmål til databaseserveren. Serveren vil også videresende behandlet data fra databaseserveren til klienten. Mellomlaget kalles også *business logic layer* siden det håndterer forretningsregler og begrensninger før data sendes opp til brukeren (presentasjonslag) eller ned til DBMS (database tjenestelag). Denne arkitekturen vil øke sikkerheten, siden klientene ikke har direkte tilgang til databaseserveren. Mellomlaget kan også fungere som en Web server.



3. **n-tier** – lagene mellom brukeren og den lagrede dataen deles opp i mindre deler. Hver *tier* kan kjøre på passende system plattform og håndteres uavhengig av andre *tiers*.

2.6 Klassifikasjon av database management system

DBMS kan klassifiseres basert på:

- **Datamodell som brukes (hovedkriteriet)** – ulike system er basert på bestemte datamodeller slik som relasjonelle, nettverk, hierarkiske, objektorienterte, osv. For eksempel er SQL system er basert på relasjonell datamodell. Noen kjente datamodeller:
    - **Relasjonell datamodell**: databasen blir representert som en samling av tabeller, der hver tabell kan lagres som en separat fil.
    - **Objekt datamodell**: databasen defineres i form av objekter, deres egenskaper og deres operasjoner. Objekter med samme struktur og oppførsel hører til en klasse og klassene er ordnet i hierarkier.
    - **Nøkkelverdi datamodell**: en unik nøkkel blir assosiert med hver verdi
    - **Dokument datamodell** – dataen lagres som dokumenter
    - **Graf datamodell** – objekter lagres som grafnoder
    - **XML modellen**: brukes for å utveksle data over nettet. Data representeres som elementer og blir nøstet for å lage komplekse hierarkiske trestrukturer.

- **Antall brukere** – enkeltbruker og flerbruker
- **Distribusjon** – antall steder databasen er distribuert over. Ved sentralisert DBMS vil dataen være lagret i én enkelt maskin, mens ved distribuert DBMS er dataen distribuert over flere maskiner koblet sammen av et datanettverk.
- **Homogenitet** – homogen DBMS bruker samme DBMS programvare ved alle steder, mens heterogen DBMS bruker ulike DBMS programvare ved hvert sted.
- **Kostnad** – jo større database og flere tjenester, desto dyrere er DBMS
- **Aksessbaner** – hvilke aksessbaner som er tilgjengelig for å lagre filer
- **Formål** – generell eller spesifikk

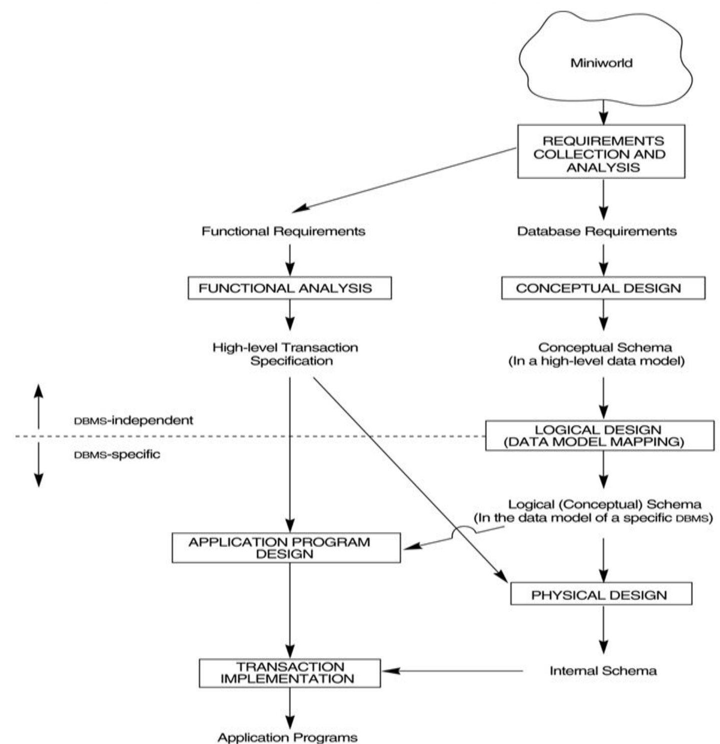
Del 2 – Konseptuell datamodellering

Kapittel 3 – Datamodellering med ER modell

En **database applikasjon** er en spesifikk database og de assosierte programmene som implementerer forespørsler og oppdateringer av databasen. Den tradisjonelle tilnærmingen for design av database er å fokusere på strukturen og begrensningene til databasen, mens designet av applikasjonsprogrammene blir dekket av programvareutvikling. **For å designe en database applikasjon er det svært viktig med konseptuell modellering**. En populær høy-nivå konseptuell datamodell er **entitet-relasjon (ER) modellen**. UML (Unified Modeling Language) er objekt modellering og blir stadig mer populær for design av database og programvare (mer senere). UML gir detaljert design av programvare moduler og deres interaksjoner vha ulike diagram (eks: klassediagram).

3.1 Steg i databasedesign

Figuren viser en forenklet versjon av prosessen bak designet av en database:

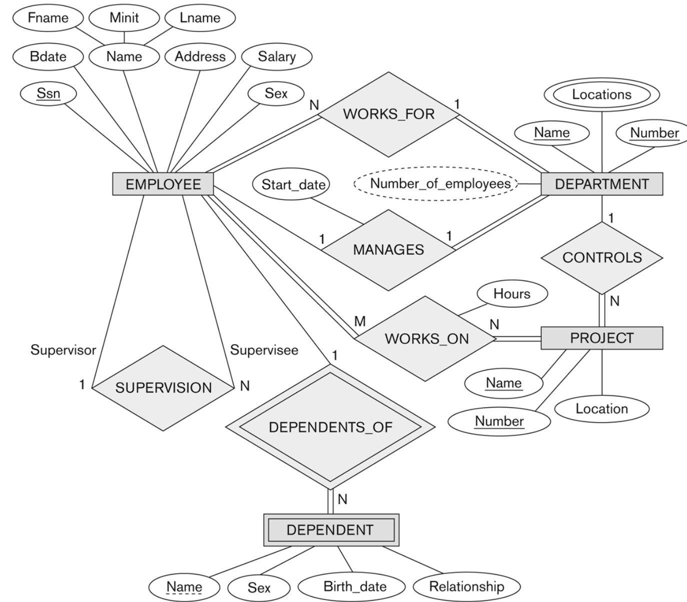


1. **Samling og analyse av krav** – designerne vil intervjue database brukerne for å forså deres datakrav. Designerne bør samtidig definere de funksjonelle kravene som består av operasjonene (transaksjonene) brukerne kan bruke på databasen.
2. **Konseptuell design** – en høy-nivå konseptuell datamodell brukes for å lage et konseptuelt skjema for databasen. Dette skjemaet gir en beskrivelse av datakravene og detaljerte beskrivelser av entitet typer, relasjoner og begrensninger. Designerne kan definere egenskapene til dataen uten å må ta hensyn til detaljer ved lagring og implementasjon. Parallelt med dette vil funksjonskravene brukes for å spesifisere høy-nivå transaksjoner i den funksjonelle analysen.
3. **Logisk design (datamodell kartlegging)** – det konseptuelle skjemaet i en høy-nivå datamodell blir omformet til et dataskjema i en implementasjon datamodell som tilhører en bestemt DBMS.
4. **Fysisk design** – spesifiserer intern lagringsstruktur, filorganisering, indekser, aksessbaner og fysisk designparametere for database filene. Dette kalles en fysisk datamodell. Parallelt med dette vil applikasjonsprogram designes og implementeres som database transaksjoner.

3.2 Eksempel på databaseapplikasjon

COMPANY er et eksempel på en databaseapplikasjon som vi skal bruke for å illustrere de grunnleggende konseptene i ER modellen. Denne databasen holdes styr over selskapets ansatte, avdelinger og prosjekter. Vi skal lage et konseptuelt skjema steg for steg, når følgene er beskrivelsen av miniverden etter samling og analyse av datakravene:

- Selskapet er ordnet i avdelinger som har ulike navn, nummer og avdelingssjef. Vi har datoen når avdelingssjef ble ansatt. Avdelingen kan ha flere lokasjoner
- Avdelingen kontrollerer en rekke prosjekter som har ulike navn, nummer og lokasjoner
- Databasen lagrer navn, personalnummer, adresse, lønn, kjønn og bursdag til hver ansatt. En ansatt er tildelt en avdeling, men kan jobbe med flere prosjekt kontrollert av ulike avdelinger. Det er nødvendig å registrere antall timer i uken en ansatt arbeider med hvert prosjekt og direkte veileder til hver ansatt.
- Databasen vil registrere forsørgeren til hver ansatt pga forsikring. Dette inkluderer fornavn, kjønn, bursdag og relasjon til den ansatte.



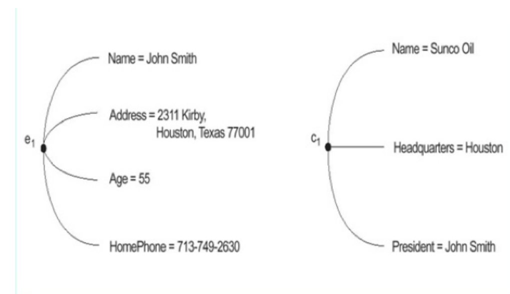
Figuren viser hvordan skjemaet for denne database applikasjonen kan settes opp som et ER diagram. Vi skal nå forklare hvordan denne blir bygd opp.

3.3 Entitetsklasser, entitetssett, attributter og nøkler

ER modellen beskriver data som entiteter, relasjoner og attributter.

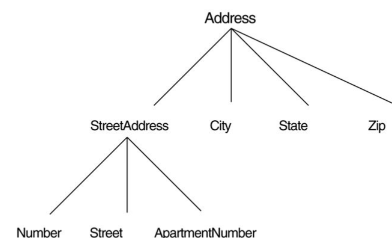
Entiteter og attributter

ER modellen representerer en **entitet, som er en ting eller objekt i den virkelige verden med en uavhengig eksistens.** Dette objektet kan fysisk eksistere (eks: person, bil, hus, ...) eller konseptuelt eksistere (eks: jobb, universitetsemne, ...). Hver entitet vil ha **attributter, som er bestemte egenskaper som beskriver entiteten**. For eksempel kan en EMPLOYEE entitet ha attributter for navn, alder, adresse, lønn og jobb. **En entitet vil ha en verdi for alle attributter og verdiene hentes fra et domene** (datatype). Figuren viser to entiteter og verdiene til deres attributter.



En ER modell kan ha flere typer attributter:

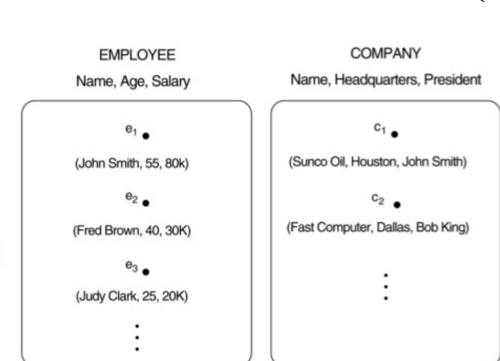
- **Sammensatte/simple attributter** – sammensatte attributter kan deles opp i mer grunnleggende og uavhengige attributter. Eks: adresse attributtet kan deles inn i gateadresse, by, stat, og postkode. Attributter som ikke kan deles opp kalles simple eller atomiske attributter. Sammensatte attributter kan danne et hierarki slik som på figuren.



- **Enkelverdi/flerverdi attributter** – enkelverdi attributter er egenskaper som har en enkel verdi for en bestemt entitet, for eksempel alder hos en person. Flerverdi attributter er egenskaper som kan ha flere verdier for en bestemt entitet, for eksempel farge hos en bil. En flerverdi attributt kan ha nedre og øvre grense på antall verdier som er tillatt.
- **Lagret/utledet attributter** – i noen tilfeller kan attributtverdier være relaterte, for eksempel alder og bursdag hos en person. For en person entitet kan verdien til alderen utledes fra nåværende dato og personens bursdag. Derfor vil alder attributtet være en utledet attributt, mens bursdagsdato attributtet er en lagret attributt.
- **NULL verdier** – det kan hende at en entitet ikke har noen passende verdi for en attributt, og i slike tilfeller brukes en spesiell verdi kalt NULL. Det kan også brukes dersom verdien er ukjent, for eksempel vi ikke kjenner til mobilnummeret til en ansatt.
- **Komplekse attributter** – sammensatte og flerverdi attributter kan nøstes vilkårlig. Sammensatte attributter blir plassert i (...) etter deres felles verdi, mens flerverdi attributter blir plassert i {...}. For eksempel kan en CAR entitet ha en sammensatt attributt Registration (State, Number) og en flerverdi attributt {Color} som kan være {red, black}. Disse attributtene kalles komplekse attributter.

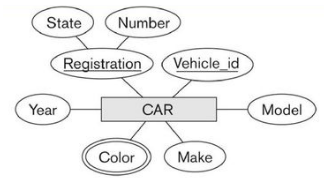
Entitetsklasser, entitetssett, og verdisett

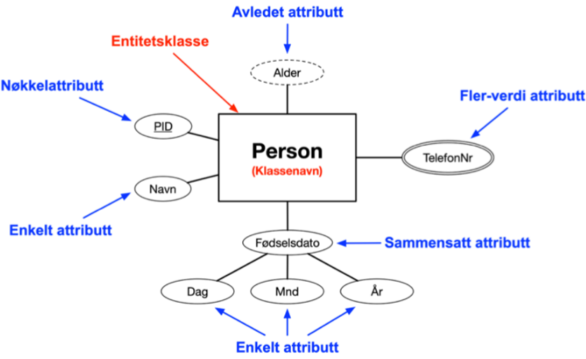
En database vil som regel inneholde grupper av entiteter som er like, for eksempel flere ansatte i et selskap. Slike entiteter vil dele de samme attributtene, men hver entitet har sine egne verdier for hver attributt. **En entitetsklasse (-type) er en samling av entiteter som har samme attributter**. Hver entitetsklasse i databasen blir beskrevet med navn og attributter. Figuren viser to entitetsklasser: EMPLOYEE og COMPANY, og noen entiteter av hver klasse og deres verdier. **Et entitetssett er en samling av alle entitetene til en bestemt entitetsklasse ved et spesifikt tidspunkt**. Navnet på entitetsklassen blir ofte brukt for å referere til entitetssettet, for eksempel vil EMPLOYEE referere til både typen entitet og nåværende samling av alle disse entitetene i databasen.



Nøkkelattributt

En nøkkelattributt er en attributt som har unik verdi for hver entitet i entitetssettet. For eksempel vil personnummer være en nøkkelattributt for Person entiteten, siden ingen personer har samme personnummer. **Verdien til nøkkelattributtet brukes for å identifisere hver entitet og derfor må alle entiteter ha ett eller flere nøkkelattributt.** I noen tilfeller vil flere attributter kombinert danne en nøkkelattributt, altså må kombinasjonen av alle attributtverdiene være unik for hver entitet. Da må attributtene skrives som en sammensatt attributt der felles attributtet er nøkkelen. Når man bestemmer at en attributt er en nøkkelattributt, betyr det at alle entiteter må ha en unik verdi for denne egenskapen. Nøkkelen er derfor en begrensning som hindrer at to entiteter har samme verdi for dette attributtet samtidig. **En entitetsklasse uten nøkkel kalles en svak entitetsklasse og må identifiseres ved å være relatert til en annen entitetsklasse (mer senere).**





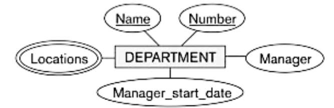
Notasjon i ER diagram

I ER diagrammet vil entitetsklassen representeres av en boks som inneholder navnet til entitetsklassen. Attributt navn er plassert i ovaler og koblet til deres entitetsklasse av en rett linje. Sammensatte attributter er koblet til deres felles attributt av rette linjer og flerverdi attributter har doble ovaler. Nøkkelattributt blir representert med understrek i ovalen.

Verdisett (domene) til attributter

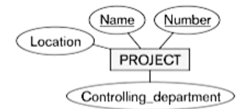
Hver simpel attributt hos en entitetsklasse har et verdisett (domene) som spesifiserer verdiene som kan tildeles dette attributtet. For eksempel for Age attributtet kan domenet være heltall fra 17 til 70. Verdisett blir som regel ikke vist i ER diagrammet og ligner på datatyper som String, osv.

Initialt konseptuelt design av COMPANY databasen

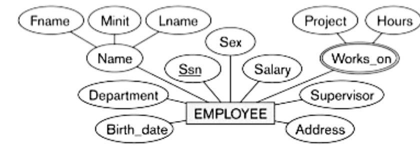


Vi bruker kravene fra seksjon 3.2 for å definere entitetsklassene i COMPANY databasen. Vi kan identifisere fire entitetsklasser:

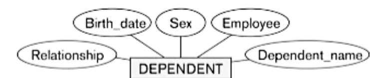
1. DEPARTMENT: har attributtene Name, Number, Locations, Manager og Manager_start_date. Her vil Locations være en flerverdi attributt, mens Name og Number er separate nøkkelattributt.



2. PROJECT: har attributtene Name, Number, Location og Controlling_department. Her vil Name og Number være separate nøkkelattributt.



3. EMPLOYEE: har attributtene Name, Ssn (personnummer), Sex, Address, Salary, Birth_date, Department, Supervisor og Works_on. Her vil Name og Works_on være sammensatte attributter, Ssn og Works_on er nøkkelattributt og Works_on er flerverdi attributt.



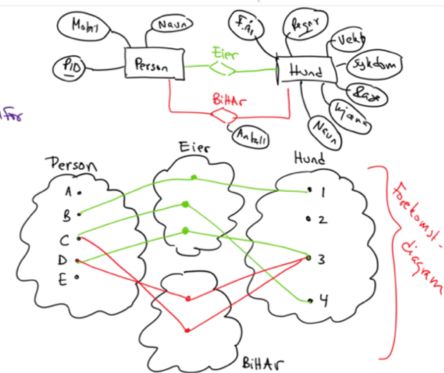
4. DEPENDENT: har attributtene Employee, Dependent_name, Sex, Birth_date og Relationship.

3.4 Relasjonsklasser, relasjonssett, roller og strukturelle begrensninger´

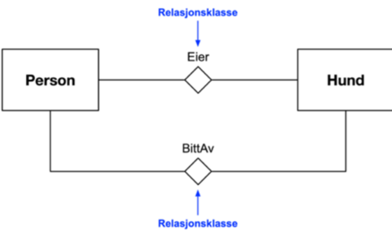
Når en attributt hos en entitetsklasse refererer til en annen entitetsklasse, vil det eksistere en relasjon mellom disse entitetene. For eksempel vil attributtet Manager i DEPARTMENT referere til en Employee som styrer avdelingen, og attributtet Controlling_department i PROJECT refererer til en Department som har ansvar for prosjektet. Slike referanser vil ofte representeres som attributter i det initiale designet, men vil etterhvert erstattes med **relasjoner** mellom entitetsklasser i ER modellen.

Relasjonsklasser, sett og instanser

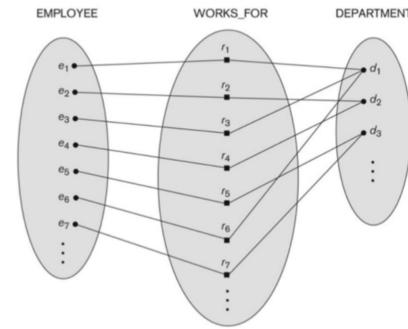
En relasjon er en sammenheng (assosiasjon) mellom to eller flere entiteter, og den vil kun eksistere dersom entitetene som inngår eksisterer. For eksempel kan vi ha to entitetsklasser: Person og

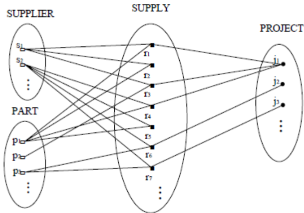


Hund, og to relasjoner: Eier og Bittav. Figuren viser ER modell for entitetsklassene og relasjonen mellom disse og et forekomstdiagram.



En **relasjonsklasse (-type)** er en mengde av likartede relasjoner mellom samme entitetsklasser. I eksempelet over vil Eier være en relasjon fordi det er en lik relasjon mellom Person og Hund. Hvis det er n entitetsklasser $E_1, E_2, \dots, E_n$ vil en relasjonsklasse R definere et sett av assosiasjoner mellom disse, og dette settet kalles **relasjonssett**. Det samme navnet R brukes ofte for både klassen og settet. Et relasjonssett er et sett av **relasjonsinstanser** r_i, der hver r_i assosierer n individuelle entiteter. Entitetene som er en del av en relasjonsinstans vil være relatert på en eller annen måte i miniverden. Figuren til venstre viser relasjonsklassen WORKS_FOR mellom entitetsklassene EMPLOYEE og DEPARTMENT. Hver relasjonsinstans i relasjonssettet vil assosiere en EMPLOYEE entitet og en DEPARTMENT entitet. **I ER diagram blir relasjonsklasser representert med diamantformet bokser**.

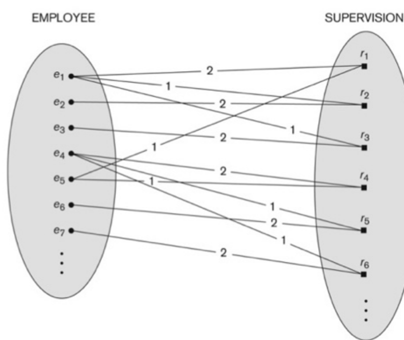




Relasjonsgrad, rollenavn og rekursive relasjoner

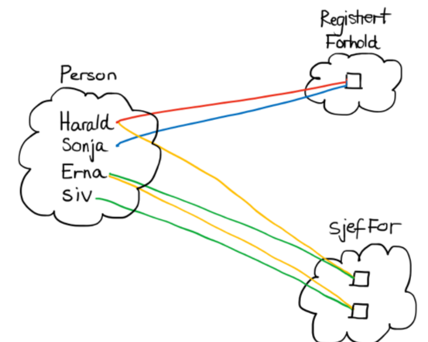
Graden til en relasjonsklasse er antall entitetsklasser som deltar i assosiasjonen. WORKS_FOR er et eksempel på en binær relasjonsklasse (grad to), mens SUPPLY er en ternær relasjonsklasse (grad tre). I SUPPLY vil hver relasjonsinstans assosiere tre entiteter, når leverandør s gir del p til prosjekt j.

Binære relasjoner kan modelleres som attributter. For eksempel for WORKS_FOR kan EMPLOYEE entitetsklassen ha en attributt kalt Department, der verdien til Department for hver EMPLOYEE entitet er DEPARTMENT entiteten som denne ansatte jobber i. Vi kan også ha en Employees attributt i DEPARTMENT entitetsklassen. Dersom begge er brukt må de være inverse. Konseptet av å representere relasjonsklasser som attributter brukes i **forekomstdiagram** (figur ned til høyre).

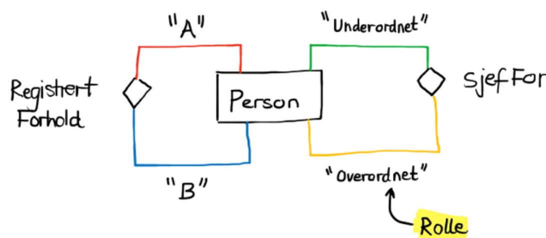


Rekursive relasjoner VIKTIG

Hver entitetsklasse som deltar i en relasjonsklasse har en bestemt rolle i relasjonen. **Rollenavnet beskriver rollen til entiteten i hver relasjonsinstans og forklarer hva relasjonen betyr**. Dette er nyttig i tilfeller der samme entitetsklasse deltar flere ganger i ulike roller i én relasjonsklasse. Slike relasjonsklasser kaller **rekursive relasjoner**. SUPERVISION er en rekursiv relasjonsklasse, siden den relaterer en employee med en supervisor, som begge er medlem av EMPLOYEE entitetsklassen. Hver relasjonsinstans vil assosiere to ulike employee entiteter, der supervisor rollen markeres med '1' og employee rollen markeres med '2'.



Figurene til høyre viser konseptuell modell og forekomstdiagram for to rekursiv relasjonsklasser. Legg merke til fargekoden som skiller rollene fra hverandre. Siden det RegForhold og SjefFor er rekursive relasjoner vil entitetene i relasjonen være fra samme entitetsklasse.



Person = {Harald, Sonja, Erna, Siv}

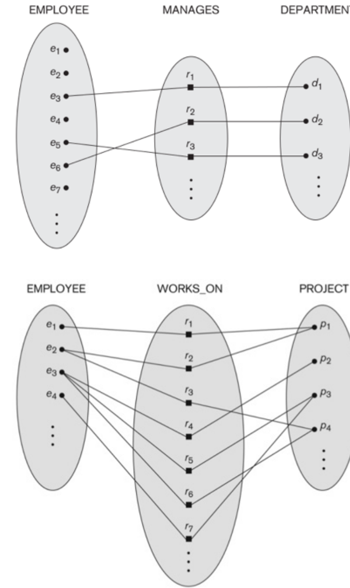
RegForhold = {(Harald, Sonja)}

SjefFor = {(Harald, Erna), (Erna, Siv)}

Restriksjoner på binære relasjonsklasser VIKTIG

Relasjonsklasser vil ofte ha begrensninger på mulige kombinasjoner av entiteter. For eksempel kan det være et krav at hver employee skal jobbe med minst ett prosjekt, og da må dette beskrives i skjemaet. Det er to hovedtyper begrensning for binære relasjoner:

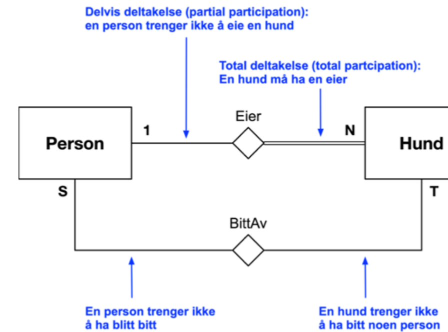
1. Kardinalitetsforhold



Kardinalitetsforhold gir maksimalt antall relasjonsinstanser en entitet kan delta i. For eksempel vil DEPARTMENT:EMPLOYEE kardinaliteten være 1:N, noe som betyr at hver avdeling kan være relatert til et vilkårlig antall ansatte (N), mens en ansatt kan være relatert til maks én avdeling (1). Merk: **N indikerer at det er ingen maksimum**. De mulige kardinalitetene for binære relasjoner er 1:1, 1:N, N:1 og M:N. MANAGES er et eksempel på 1:1 kardinalitet siden én ansatt vil styre en avdeling og en avdeling vil styres av kun én ansatt. WORKS_ON er et eksempel på M:N kardinalitet, siden en ansatt kan ha flere prosjekter og et prosjekt kan ha flere ansatte (se figur). I ER diagram vil kardinaliteten skrives på hver sin side av diamanten. Vi kan også bruke **(min, max) notasjon,** for eksempel vil (0, N) for DEPARTMENT bety at avdelingen kan ha 0 til N antall ansatte.

En hund kan ha Eier-relasjon til maks 1 Person (0-1)
En person kan ha Eier-relasjon til mange hunder (0-N)
Person
1
Eier
N
Hund
S
T
BittAv
En hund kan ha BittAv-relasjon til mange personer (0-S)
En person kan ha BittAv-relasjon til mange hunder (0-T)

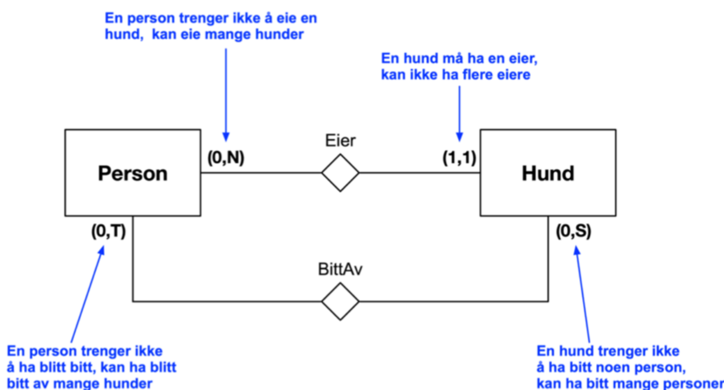
2. Deltakelseskrav

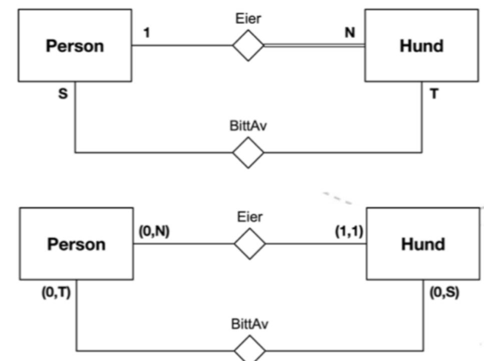


Deltakelseskrav kan spesifisere at en entitet *må* ha en relasjon til minst én annen entitet. Det gir altså minimum antall relasjoner en entitet må delta i for å eksistere og kalles **minimum kardinalitetsforhold**. Det er to typer: **total** og **delvis**. WORKS_FOR er eksempel på total deltakelse, siden en ansatt må jobbe for minst én avdeling for å eksistere (kalles eksistensavhengighet). MANAGER er eksempel på delvis deltakelse, siden det kan eksistere ansatte som ikke styrer en avdeling. I ER diagram vil **total deltakelse vises som doble linjer** (må være minimum en relasjon), mens **delvis deltagelse er enkle linjer.**

Strukturelle restriksjoner

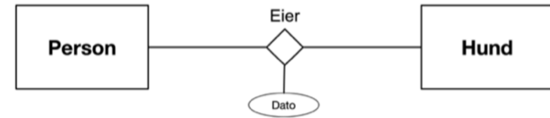
Kardinalitetsforhold og deltakelseskrav utgjør tilsammen strukturelle restriksjoner og skrives i ER diagrammet som min-maks par. På figuren ned til høyre ser vi to alternative måter å spesifisere samme restriksjoner (må forstå begge). Legg merke til at **«maks» bytter side**!





Relasjonsklasser kan ha attributter

Relasjonsklasser kan også ha attributter på samme måte som entitetsklasser. For eksempel kan vi legge til en attributt kalt Hours på WORKS_ON relasjonsklassen for å registrere antall timer en bestemt ansatt jobber på et bestemt prosjekt. Hver relasjon mellom ansatt og prosjekt vil dermed ha en egen timeverdi (merk: kan være NULL). Vi kan også legge til attributtet Dato på Eier relasjonen for å registrere når personen ble eier av hunden (se figur).



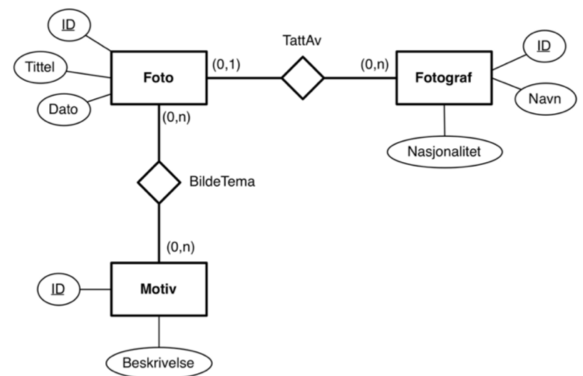
I konseptet vil attributtet høre til relasjonsklassen, men dersom verdien til attributtet skal bestemmes må det bli en del av entitetsklassene. Ved 1:1 kan verdien til attributtet bestemmes ved at det migrerer til en av sidene, mens ved 1:N må attributtet bli en del av entitetstypen på N-siden av relasjonen. Ved M:N relasjonsklasser må attributtet bestemmes av en kombinasjon av entitetsklassene.

Eksempel – Fotodatabase

Følgende er et eksempel på overgang fra beskrivelse av miniverden til en mulig ER-modell:

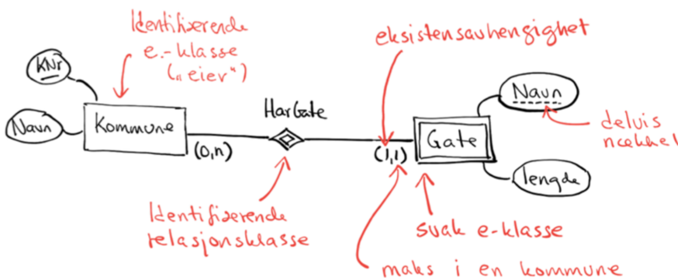
Vi skal lage en ER-modell (ER-diagram) for å holde oversikt over fotografier. Hvert fotografi har en unik identifikator, tittel og en dato da bildet ble tatt. Et fotografi kan ha en fotograf, men ikke flere enn en, og det kan være ukjent hvem som tok bildet. For fotografer skal vi kunne registrere en unik identifikator, navn og nasjonalitet. Vi kan registrere fotografer som (ennå) ikke har tatt noe fotografi som er registrert i databasen. Det er ingen øvre grense på hvor mange bilder en fotograf kan ha tatt. Vi skal kunne registrere et antall bildemotiver som lagres med en unik identifikator og en beskrivelse. Et fotografi kan vise et antall motiver, men det trenger ikke å ha noe motiv. Et motiv kan være knyttet til flere bilder. Det kan finnes motiv som ikke er knyttet til noe bilde.

1) Finn de entitetsklassene som vi trenger
2) Bestem attributter for entitetsklassene
3) Finn de relasjonsklassene som vi trenger
4) Bestem attributter for relasjonsklassene
5) Bestem nøkler for entitetsklassene og restriksjoner for relasjonsklassene
6) Vurder om du har kommet fram til en god modell



3.5 Svake entitetsklasser

Når en entitetsklasse mangler et «naturlig» nøkkelattributt kalles den svak entitetsklasse og kan av og til identifiseres gjennom en relasjon til en annen entitetsklasse. For å identifisere svake entiteter bruker vi en attributtverdi til den svake entitetsklassen og relasjonen til den andre entitetsklassen. **Identifiserende entitetsklasse (eier)** er den andre entitetsklassen, mens **identifiserende relasjonsklassen** er relasjonen mellom disse. For eksempel vil Kommune være eier av Gate entitetsklassen. To ulike gater med samme navn og samme lengde vil identifiseres som ulike etter å ha bestemt hvilken Kommune de er relatert til. En svak entitetsklasse har som regel en **delvis nøkkel som er attributtet som skiller svake entiteter relatert til samme eier.** For eksempel vil gatenavnet være delvis nøkkel som gjør at vi kan skille mellom gater i samme kommune. I verste tilfellet vil en sammensatt attributt av alle attributtene til den svake entiteten være delvis nøkkel. Alternativt kan man legge til en nøkkel og modellere den som en ordinær entitetsklasse, eller representere den svake entitetsklassen som en kompleks attributt.



OBS! En svak entitetsklasse har alltid total deltakelseskrav (dvs. minimum 1 kardinalitet) mht. identifiserende relasjon, fordi den kan ikke identifiseres uten en eier.

I ER-diagram vil svake entitetsklasser og identifiserende relasjonsklasse representeres med doble linjer. Delvis nøkkelattributtet blir understreket med stiplede linjer. Det kan være flere nivåer med svake entitetstyper, altså kan eieren være svak. En svak entitetsklasse kan også ha flere identifiserende entitetsklasser og identifiserende relasjonsklasse med høyere grad enn to.

3.6 Design av COMPANY databasen, fortsettelse

Vi kan nå endre database designet på side 15 ved å endre attributtene som representerer relasjoner med relasjonsklasser. Vi bruker kravene i seksjon 3.2 for å bestemme kardinalitetsforhold og deltakelseskrav. Vi får følgende relasjonsklasser:

- MANAGES – 1:1 relasjonsklasse mellom EMPLOYEE og DEPARTMENT. EMPLOYEE deltakelse er delvis, mens DEPARTMENT er total (avdeling må ha styrer)
- WORKS_FOR – 1:N relasjonsklasse mellom DEPARTMENT og EMPLOYEE, der begge deltakelsene er totale.
- CONTROLS – 1:N relasjonsklasse mellom DEPARTMENT og PROJECT. PROJECT deltakelse er total, mens DEPARTMENT er delvis (noen avdelinger har ikke prosjekt)
- SUPERVISION – 1:N rekursiv relasjonsklasse innen EMPLOYEE, der begge deltakelsene er delvis
- WORKS_ON – M:N relasjonsklasse med attributt Hours. Begge deltakelsene er totale
- DEPENDENTS_OF – 1:N relasjonsklasse mellom EMPLOYEE og DEPENDENT. EMPLOYEE deltakelse er delvis, mens DEPENDENT deltakelse er total (svak entitet).

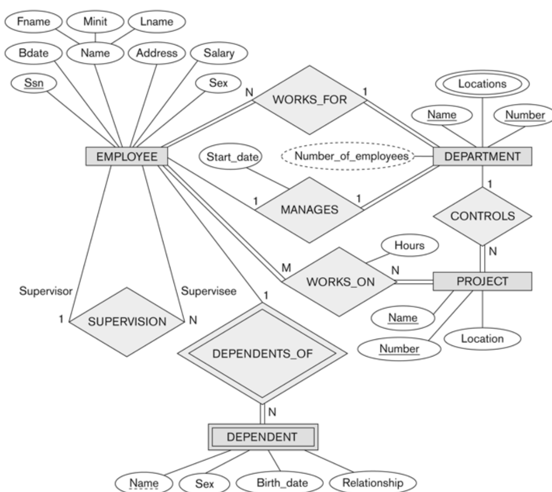
Vi fjerner alle attributtene som har blitt endret til relasjoner (eks: Manager, Manager_start_date, osv.), slik at vi får minst mulig redundans i det endelige konseptuelle skjemaet.

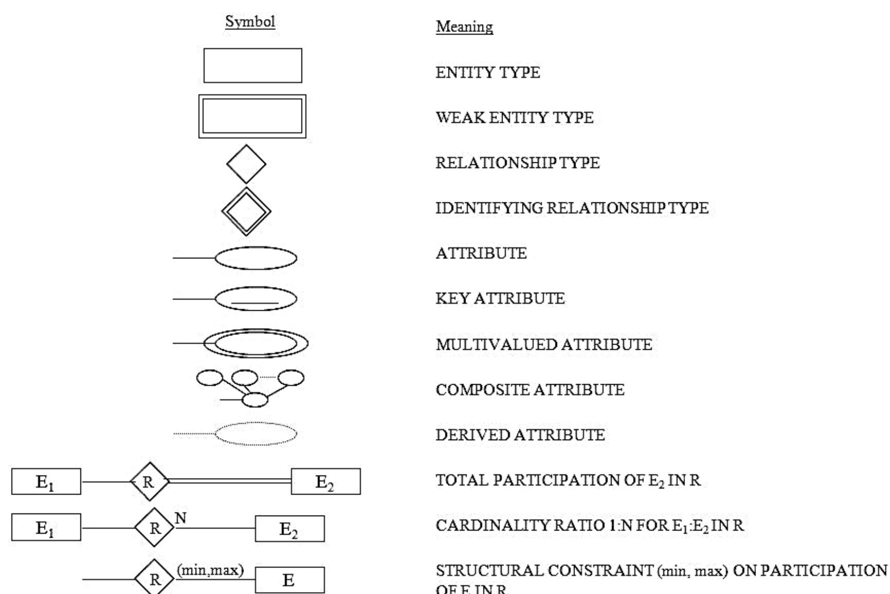
3.7 ER diagram, navngivning og designproblemer

Figuren til venstre viser ER diagrammet til COMPANY databasen. Merk:

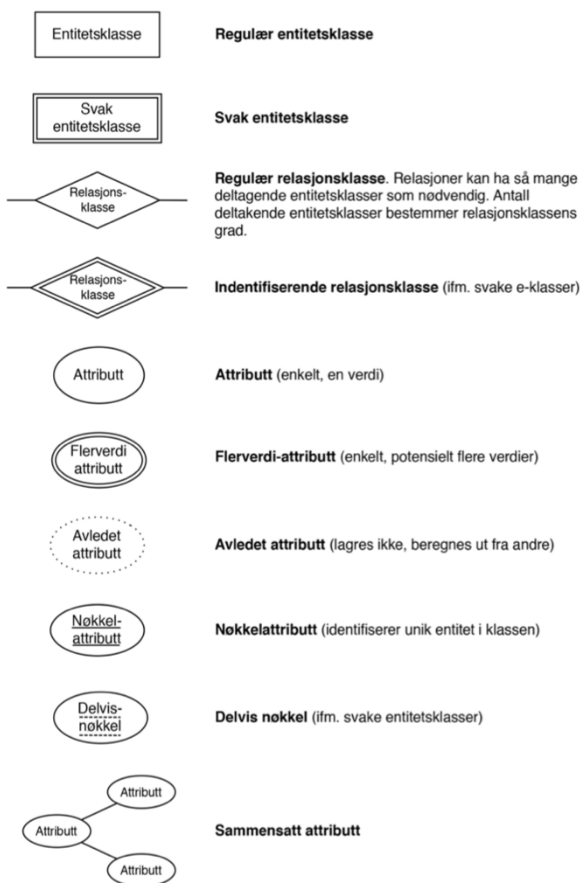
- Sammensatte attributter plasseres i ovaler som er koblet til felles attributt
- Flerverdi attributter har doble ovaler
- Nøkkelattributt er understreket
- Utledet attributter har prikkovaler
- Svake entitetsklasser har doble rektangler og identifiserende relasjonsklasse har doble diamanter.

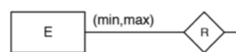
Figuren til høyre er en oppsummering på notasjoner i ER diagram.



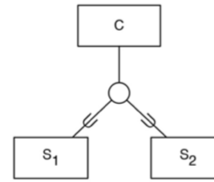


Merk: ved navngivning av relasjonsklasser ser vi ofte på relasjonen fra venstre til høyre og fra toppen til bunnen

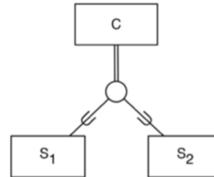




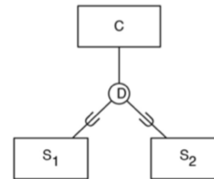
Strukturell restriksjon. En entitet i E er med i minst *min* R-relasjonen og maks *max* R-relasjoner. Vanlige restriksjoner er (0,1), (1,1) og (1,n). Det er også vanlig å bruke (0,n), som strengt tatt ikke definerer noen restriksjon.



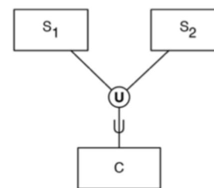
Superklasse-subklasserelasjon. Superklassen (C) spesialiseres i sub-klassene S_1 og S_2.



Total spesialisering. Dobbelt strek spesifiserer total spesialisering -- alle entiteter i superklassen må være med i minst en subklasse. Delvis spesialisering vises med enkelt strek.



Disjunkte subklasser. D markerer disjunkte subklasser -- en entitet i superklassen kan ikke delta i flere enn en subklasse. O eller ingen markering i sirkelen spesifiserer overlappende subklasser.



Kategorier. En kategori (C) representerer en samling av entiteter som er en delmenge av entitetene fra ulike entitetsklasser (kategorienes superklasser). Blir også kalt en union-type.

En kategori kan være delvis (partial) eller total. En total kategori vil inneholde alle entitetene i unionen av superklassene og markeres med dobbelt strek mellom kategorien og sirkelen.

Designprosess

Gitt en beskrivelse av miniverden, kan du lage en ER modell ved å bruke følgende steg:

1. Finn entitetsklasser, og kanskje noen viktige attributter
2. Finn relasjonsklasser (med navn)
3. Lag komplett modell med alle attributter, nøkler og andre restriksjoner
4. Vurder modellen
   a. Oppfylles kravene (den antatt bruken)?
   b. Er det noen relasjonsklasser som burde vært entitetsklasser eller omvendt?
   c. Er det noen relasjonsklasser som bør slås sammen eller deles opp?
   d. Er det noen attributter som burde vært entitetsklasser (eller omvendt)?

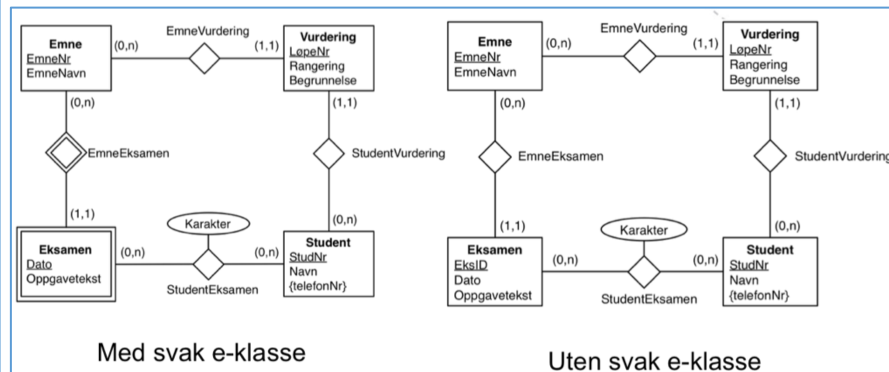
Figuren under viser et eksempel:

a) (10 %) Lag en ER-modell (du kan bruke alle virkemidler som er med i pensum) for følgende situasjon:

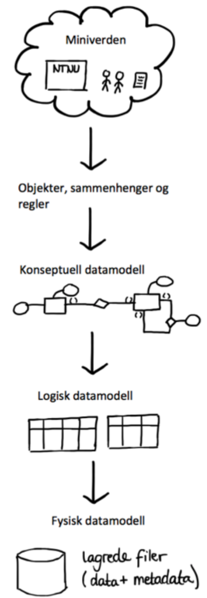
Vi har *emner* som er beskrevet med et unikt emnenummer og et emnenavn. Emner kan ha et antall *eksamener* som er beskrevet med eksamensdato og en oppgavetekst. Et emne har aldri flere eksamener på samme dag. En eksamen er for ett bestemt emne, og den samme oppgaveteksten blir aldri brukt for flere eksamener. *Studenter* er beskrevet med et entydig studentnummer, navn og ett eller flere telefonnummer. Studenter kan ta en eller flere eksamener og oppnår i så fall en karakter for prestasjonen på hver eksamen. Dersom en student tar flere eksamener i samme emne, skal alle resultater lagres. En student kan gi en *vurdering* av et emne. En vurdering består av en rangering fra 1 til 6, der 1 er elendig og 6 er fremragende, og en kort, utfyllende tekst. En og samme student kan gjøre flere vurderinger av et og samme emne.

Gjør kort rede for eventuelle forutsetninger som du finner det nødvendig å gjøre.

Fra eksamen i mai 2013.



Oppsummering – Kapittel 3 (F2, F3, Øv1)



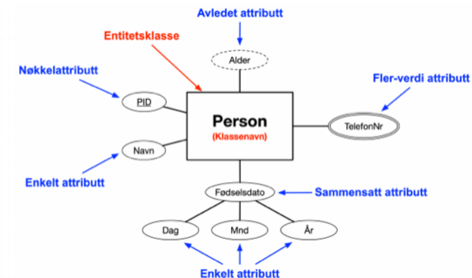
Databasedesign

Design av database består av følgende steg:

1. **Samle og analysere systemkrav** for å lage objekter, sammenhenger og regler for databasen
2. Lage **konseptuell datamodell**
3. Lage **logisk datamodell** (= dataskjema)
4. Lage **fysisk datamodell** (= bestemmer lagringsstruktur, osv.)

ER-modeller

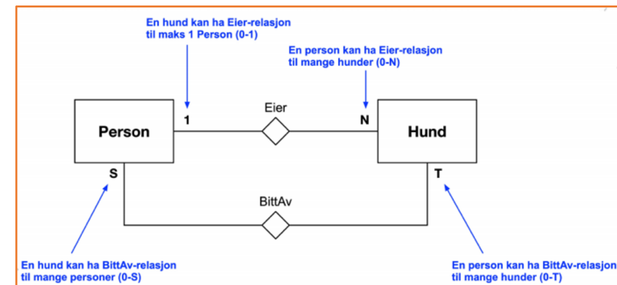
Viktige konsepter i ER-modellen:

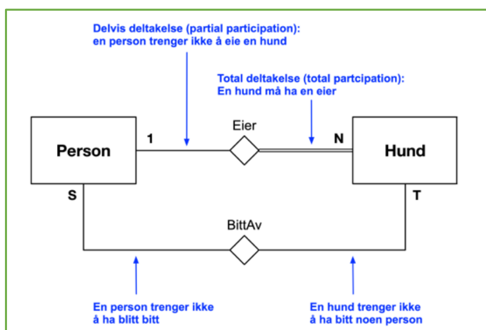


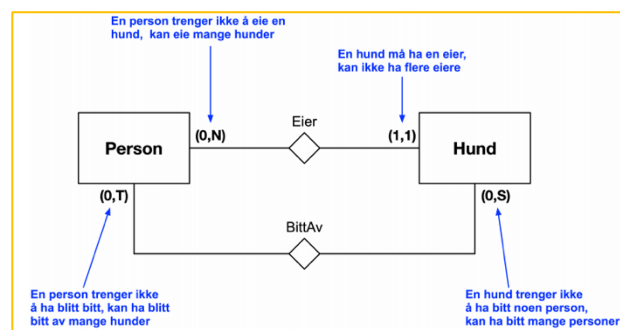
- **Entitet** = et objekt eller «noe» som eksisterer i miniverden. De kan ses på som ting i virkeligheten med en uavhengig eksistens.
- **Attributt** = beskriver egenskaper ved entiteter og henter sine mulige verdier fra et domene. Ulike typer er enkle eller sammensatte, en- eller fler-verdi, avledet (vha. regel) og nøkkelattributt (entydig ID).
- **Entitetsklasse (-type)** = beskriver samlingen av entiteter som har de samme egenskapene De har en eller flere attributter som har unike verdier for hver enkelt entitet. Et slik attributt kalles en nøkkelattributt, og brukes for å identifisere hver entitet
- **Relasjon** = sammenheng (assosiasjon mellom to eller flere entiteter). Kan ha egenskaper som modelleres som attributter. **Graden** til relasjonen er antall entiteter som inngår. Relasjoner vil ikke eksistere uten de entitetene som inngår.
- **Relasjonsklasser** = beskriver samlingen av relasjoner mellom samme entitetsklasser

Relasjonsklasser har tre hovedrestriksjoner:

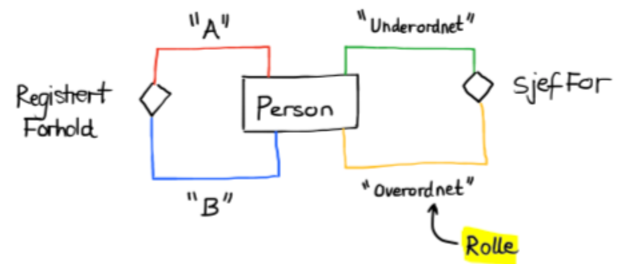
1. Kardinalitetsforhold – begrenser hvor mange entiteter en entitet kan ha samme relasjon til.
2. Deltakelseskrav – kan spesifisere at en entitet må ha en relasjon til minst en entitet.
3. Strukturelle restriksjoner (min, maks) – spesifiserer både kardinalitetsforhold og deltakelseskrav i et min-maks par.







Dette er alternative måter å spesifisere det samme, og vi må forstå begge! Legg merke til at maks bytter side!



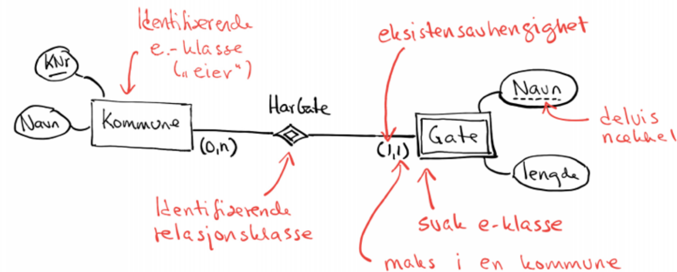
Rekursive relasjonsklasser

Rekursive relasjonsklasser er relasjonsklasser der samme entitetsklasse deltar flere ganger. På figuren ser vi at vi har to rekursive relasjonsklasser RegForhold og SjefFor, siden de beskriver relasjoner mellom to Person entiteter. Det er viktig å legge til rollenavn slik at vi kan bestemme hvilken rolle en entitet har i relasjonen (eks: er personen underordnet eller overordnet den andre personen).

Svake entitetsklasser

En entitetsklasse er sterk hvis den har en nøkkel. **En svak entitetsklasse mangler en «naturlig» nøkkel, altså en unik identifikator**. Det er to måter å håndtere dette:

1. **Legge til en nøkkel** (unik identifikator) og modeller klassen som en ordinær entitetsklasse. Dette kan være krevende, fordi vi må passe på at den blir holdt unik
2. **Identifisere entitetsklassen gjennom en relasjon til en annen entitetsklasse**. Dette gjøres når vi ikke har noen attributter som kan brukes som nøkkel.



Vi bruker svake klasser når klassen ikke har noen entydig identifikator eller når klassen har total deltakelse til en annen klasse som fungerer som tilhørende relasjonsklasse (dvs. relasjonen kan ikke være null). På figuren er kommune en sterk klasse, mens Gate er en svak klasse som identifiseres gjennom relasjonen til kommune. For å identifisere en gate må vi se på KNr + Navn (i Gate). Viktige begrep:

- **Identifiserende entitetsklasse (eier)** = den sterke klassen som brukes for å identifisere svake entiteter.
- **Svak entitetsklasse** = mangler nøkkel, og må h a en delvis nøkkel
- **Identifiserende relasjonsklasse** – relasjonen mellom svak klasse og eierklasse.
- **Eksistensavhengighet** – svak klasse må ha minst én relasjon til identifiserende entitetsklasse for at den skal kunne identifiseres. Hvis kardinaliteten må kunne være 0 må vi legge til en kunstig identifikator slik at klassen blir sterk.
- **Delvis nøkkel** = en eller flere attributter som brukes for å skille svake entiteter som er relatert til samme eier-entitet. For eksempel vil gatenavn brukes for å skille ulike gater i samme kommune. Svake entiteter i ulike relasjoner kan ha samme delvis nøkkel, for eksempel er det flere kommuner som har en gate med navn Kongens gate.

Kapittel 4 – EER modellen

ER modellen er tilstrekkelig for å representere database skjemaer for mange tradisjonelle database applikasjoner. Etterhvert ble det utviklet databaser med mer komplekse krav, og da kom behovet for database skjemaer som mer nøyaktig reflekterer egenskaper og begrensninger ved dataen. Dette førte til utviklingen av semantiske datamodell konsepter som ble innarbeidet i ER modellen. Vi skal se på konsepter som har blitt foreslått for semantiske datamodeller og hvordan ER modellen kan forbedres (*enhanced*) ved å inkludere disse konseptene og gi **EER modellen**.

4.1 Subklasser, superklasser og arving

EER modellen inkluderer alle konseptene til ER modellen (kapittel 3), og i tillegg:

1. **Spesialisering/generalisering** gir subklasser og superklasse
2. **Kategori (union)** som representerer en samling av objekter (entiteter), altså en union av objekter fra ulike entitetsklasser. Det lages entitetsklasser med entiteter fra ulike entitetsklasser
3. **Arving** av attributter og relasjoner

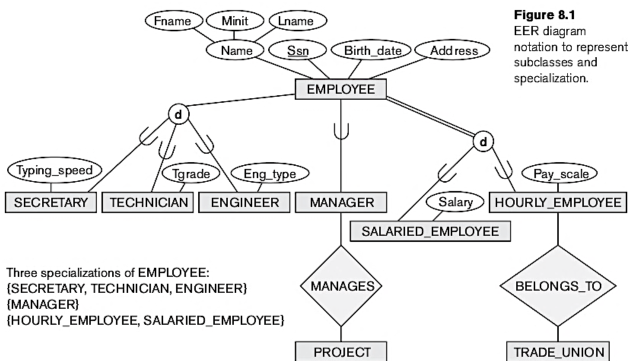


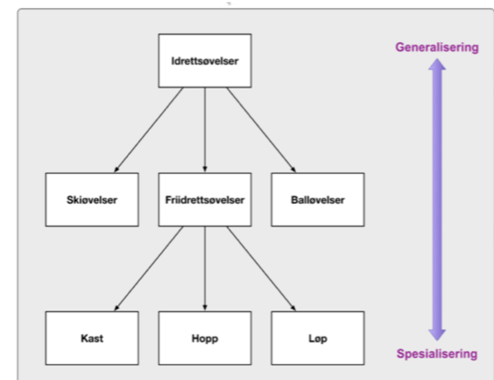
Figure 8.1 EER diagram notation to represent subclasses and specialization.

I mange tilfeller vil en entitetsklasse ha flere delgrupper som må representeres fordi de er signifikante til database applikasjonen. For eksempel kan EMPLOYEE entiteten deles inn i SECRETARY, ENGINEER, MANAGER, osv. Alle entitetene som er medlem av disse subsettene vil også være medlem av EMPLOYEE entitetssettet. Her vil delgruppene være **subklasser**, mens EMPLOYEE entitetsklassen er **superklassen**. Figuren viser notasjonen for disse konseptene i EER diagram.

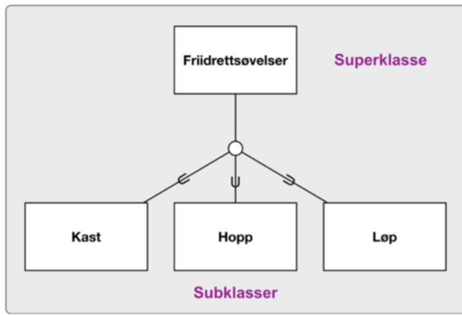
Relasjonen mellom en superklasse og en av dens subklasser kalles **superklasse/subklasse relasjon** eller IS-A-relasjon (eks: EMPLOYEE/SECRETARY). Subsett symbolet indikerer retningen på superklasse/subklasse relasjonen. **En entitet fra subklassen representerer samme virkelige-verden entitet som den fra superklassen**. For eksempel vil SECRETARY entiteten 'Hans Pettersen' også være EMPLOYEE entiteten 'Hans Pettersen'. **En entitet i en subklasse er alltid en entitet i superklassen**, **men har en spesiell rolle.** Entiteter i superklassen må ikke være med i en subklasse, men det kan være et krav. **Entiteter i subklasser vil arve alle attributtene og relasjonene til superklassen, og subklassene kan i tillegg ha egne attributter og relasjonsklasser**. Disse attributtene kalles **spesifikke (lokale) attributter** til subklassen og er festet til rektanglene som representerer subklassen. Subklassen kan også delta i **spesifikke relasjonsklasser**, for eksempel HOURLY_EMPLOYEE som deltar i BELONGS_TO relasjonen.

4.2 Spesialisering og generalisering

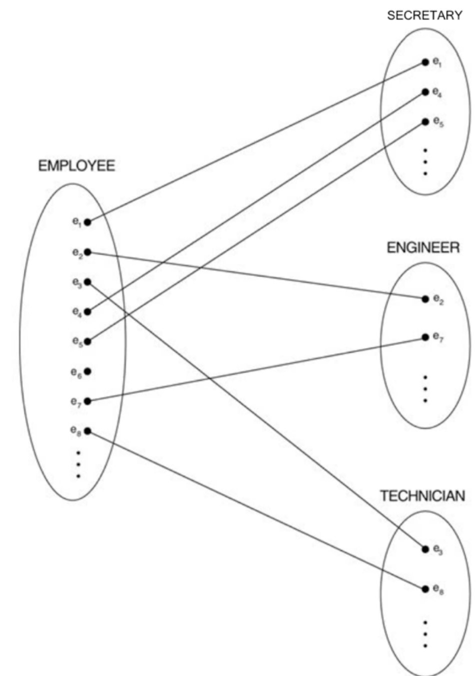
Spesialisering går ut på å definere en mengde subklasser for en entitetsklasse (superklassen). Vi finner spesialiseringene ved å bestemme egenskaper som skiller entitetene i superklassen fra hverandre. For eksempel vil {Kast, Hopp, Løp} være et sett av



Spesialisering = lager subklasser fra superklasse
Generalisering = lager superklasse fra subklasser



subklasser som er en spesialisering av Friidrettsøvelser, der øvelsene deles opp basert på type aktivitet. **I EER diagrammet vil subklassene som definerer en spesialisering være koblet til en sirkel som igjen er koblet til superklassen** (figur til venstre).

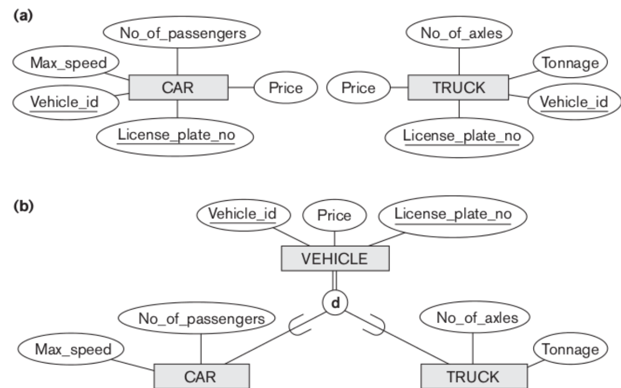


Figuren til høyre viser entiteter som hører til subklassene i {SECRETARY, ENGINEER, TECHNICIAN} spesialiseringen av EMPLOYEE. Legg merke til at entiteten i subklassen representerer den samme virkelige-verden entiteten i superklassen. Entiteten i subklassen har en spesialisert rolle, for eksempel en EMPLOYEE spesialisert i rollen som SECRETARY.

Grunnen til at vi bruker superklasse/subklasse er at **bestemte attributter og/eller relasjoner kan gjelde for noen, men ikke alle entitetene i superklassen. Felles egenskaper og relasjoner modelleres på superklassen, mens det som er unikt for en subklasse kan modelleres på subklassen.** For eksempel vil SECRETARY ha det spesifikke attributtet Typing_speed.

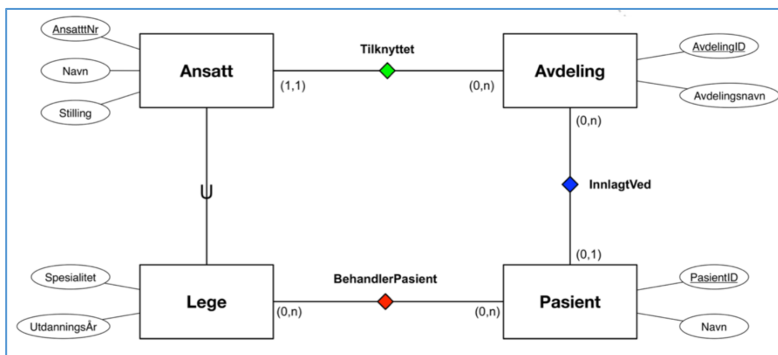
Generalisering – lager superklasse fra subklasser

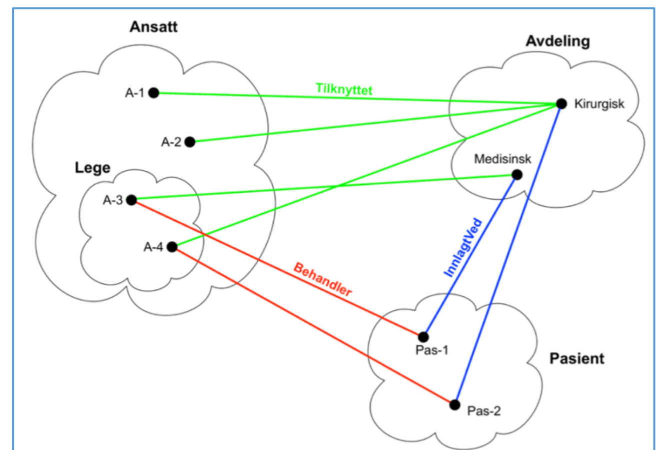
Generalisering går ut på å samle entitetsklasser med felles egenskaper under en felles superklasse (også kalt overklasse). De originale entitetsklassen blir dermed subklasser. For eksempel kan vi se på entitetsklassene CAR og TRUCK (figur a), som har mange felles attributter og derfor kan generaliseres til entitetsklassen VEHICLE (figur b). CAR og TRUCK er nå subklasser til den generaliserte superklassen VEHICLE. **{CAR, TRUCK} er en spesialisering av VEHICLE, mens VEHICLE er en generalisering av CAR og TRUCK.**



Eksempel – sykehus

Et sykehus vil ha en Ansatt entitetsklasse og blant disse vil det være en mengde leger som er en delmengde av mengden ansatte. Ansatt vil være superklassen, mens Lege er subklassen. Figuren til venstre er et EER diagram, mens figuren til høyre er et forekomstdiagram.





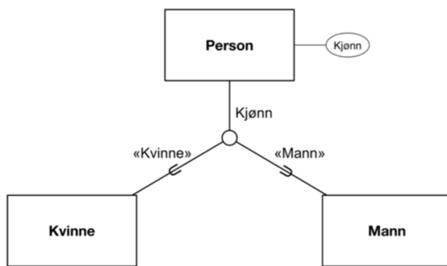
4.3 Restriksjoner og egenskaper ved spesialisering og generalisering

Restriksjoner på spesialisering og generalisering

En entitetsklasse kan ha flere spesialiseringer, og i slike tilfeller kan entiteter høre til flere subklasser. En spesialisering kan også bestå av kun én subklasse (bruker ikke sirkel notasjon. Eks: MANAGER, s. 21). Det finnes flere typer restriksjoner på spesialisering (og generalisering):

Regelbasert eller brukerstyrt deltakelse i subklasse

Vi kan bestemme hvilke entiteter som blir medlem av en subklasse ved å plassere betingelser på verdien til en attributt i superklassen. Slike subklasser kalles **predikat-definerte subklasser**. For eksempel dersom superklassen Person har et attributt Kjønn, kan vi kreve at Kjønn = 'Kvinne' for at en entitet skal kunne være medlem i subklassen Kvinne. Denne betingelsen kalles **definerende predikat**, og det er en restriksjon som bestemmer hvilke entiteter som hører til de ulike subklassene. I EER diagram vil restriksjonen skrives ved linjen som kobler subklassen til spesialiseringssirkelen. Ved en **attributtdefinert spesialisering** vil alle subklassene ha betingelse på samme attributt i superklassen. Dette attributtet kalles **definerende attributt** og blir merket med å skrive attributtnavnet på linjen mellom superklasse og spesialiseringssirkel (eks: Kjønn på figur). Entiteter med samme verdi for definerende attributt vil høre til samme subklasse.



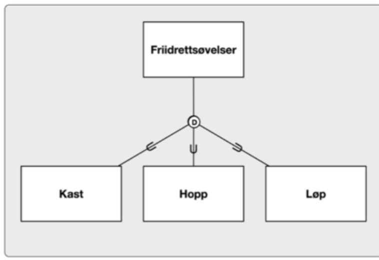
Hvis det ikke er noen betingelse som kan bestemme medlemskap i subklasser, sier vi at subklassen er brukerstyrt. Brukeren bestemmer hvilken subklasse entiteten skal legges til i.Medlemskap blir spesifisert individuelt for hver entitet istedenfor at det går automatisk vha betingelser.

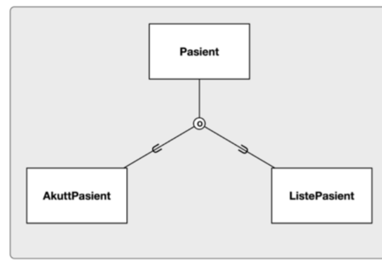
Disjunkte eller overlappende subklasser

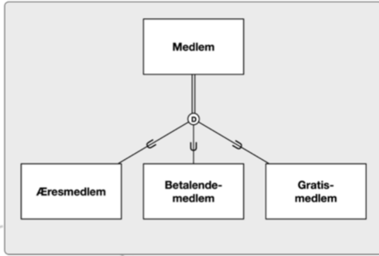
Disjunkt restriksjon gir at subklassene i spesialiseringen må være disjunkte sett. Dvs. en entitet kan være medlem i maksimalt én subklasse. Dette vil være tilfellet for attributt-definert spesialisering dersom attributtet har enkeltverdi. I EER modellen blir **disjunkte subklasser markert med en d** (= disjunkt) i spesialiseringssirkelen. **Ved overlappende subklasser kan samme entiteten være medlem i flere subklasser ved spesialiseringen**. I EER modellen blir **overlappende subklasser markert med en o** (= overlappende) i spesialiseringssirkelen.

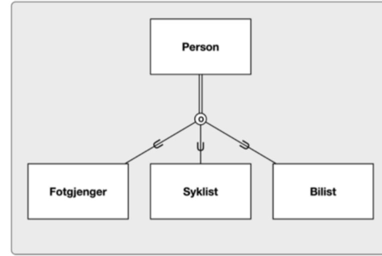
Delvis eller total spesialisering – fullstendighetsrestriksjon

Total spesialisering krever at alle entiteter i superklassen er medlem av minst en subklasse i spesialiseringen. I EER modellen blir **total spesialisering markert med dobbel linje** mellom superklassen og spesialiseringssirkelen. **Ved delvis spesialisering kan det være entiteter i superklassen som ikke hører til noen subklasser.** I EER modellen blir **delvis spesialisering markert med enkel linje** mellom superklassen og spesialiseringssirkelen.









Entitet deltar i	Disjunkt	Overlappende
Delvis	0-1 subklasser	0-n subklasser
Total	1 subklasse	1-n subklasser

På figuren kan vi se at spesialiseringene:

- Friidrettsøvelser: disjunkt og delvis
- Pasient: overlappende og delvis
- Medlem: disjunkt og total
- Person: overlappende og total

Det er altså fire mulige restriksjoner på spesialisering (se tabell over)

Følgende er regler på innsetting og sletting ved spesialisering (og generalisering):

1. Sletting av entitet fra superklassen betyr at det blir automatisk slettet fra alle subklassene
2. Innsetting av entitet i superklassen betyr at det blir automatisk innsatt i alle predikat-definerte subklasser basert på verdien til definerende predikat
3. Innsetting av entitet i superklassen med total spesialisering betyr at entiteten må settes inn i minst en av subklassene til spesialiseringen

Spesialisering og generalisering – hierarki og gitter

En subklasse kan selv ha flere subklasser, slik at det dannes et hierarki eller gitter av spesialiseringer. For eksempel vil ENGINEER være en subklasse av EMPLOYEE og en superklasse for ENGINEERING_MANAGER (se figur). To typer spesialisering:

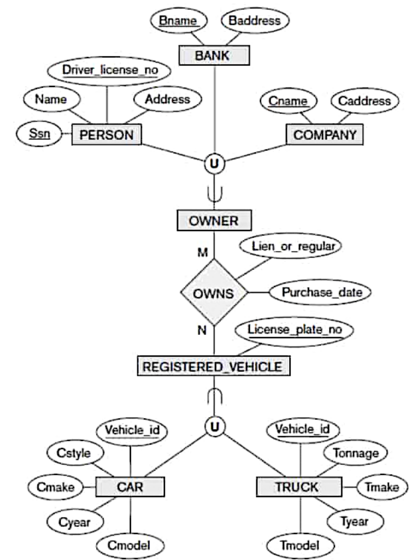
- **Spesialiseringshierarki** – **subklasser har kun én superklasse, altså kun én forelder i en trestruktur.** Subklasser vil arve fra sine forgjengere hele veien til roten, og det er kun én vei til roten, noe som kalles **enkel arv**.
- **Spesialiseringsgitter – subklassen kan ha flere superklasser.** Subklasser vil arve fra sine forgjengere hele veien til roten, men de kan ha flere superklasser og dermed flere veier til roten. Vi sier at subklassen er delt og vi har **multiple arv**, siden den delte subklassen kan arve attributter og relasjoner fra flere superklasser. Dersom en attributt eller relasjon er arvet fra samme superklasse via ulike baner, skal den inkluderes kun én gang i den delte subklassen.

Dette gjelder også for generalisering!

Bruk av spesialisering og generalisering for å raffinere konseptuelle skjema

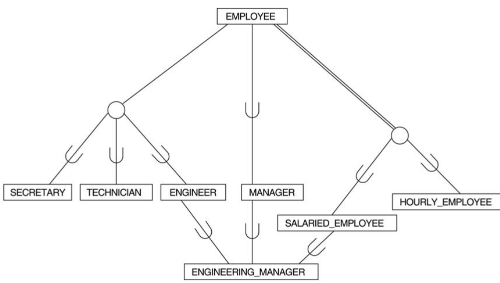
Spesialisering og generalisering brukes for å raffinere (dvs. fjerne uønskede elementer) fra konseptuelle skjemaer ila database design. Ved spesialisering vil designerne starte med en entitetsklasse og deretter gjentatt definere subklasser til miniverden er beskrevet. Dette kalles **top-down konseptuell raffinering**. Ved generalisering vil designerne finne entitetsklassene ved bladene og deretter gjentatt definere superklasser til miniverden er beskrevet. Dette kalles **bottom-up konseptuell syntese**. Det endelige designet kan bli identisk ved begge metoder. Som regel blir en kombinasjon av begge metodene brukt.

4.4 Union typer og kategorier



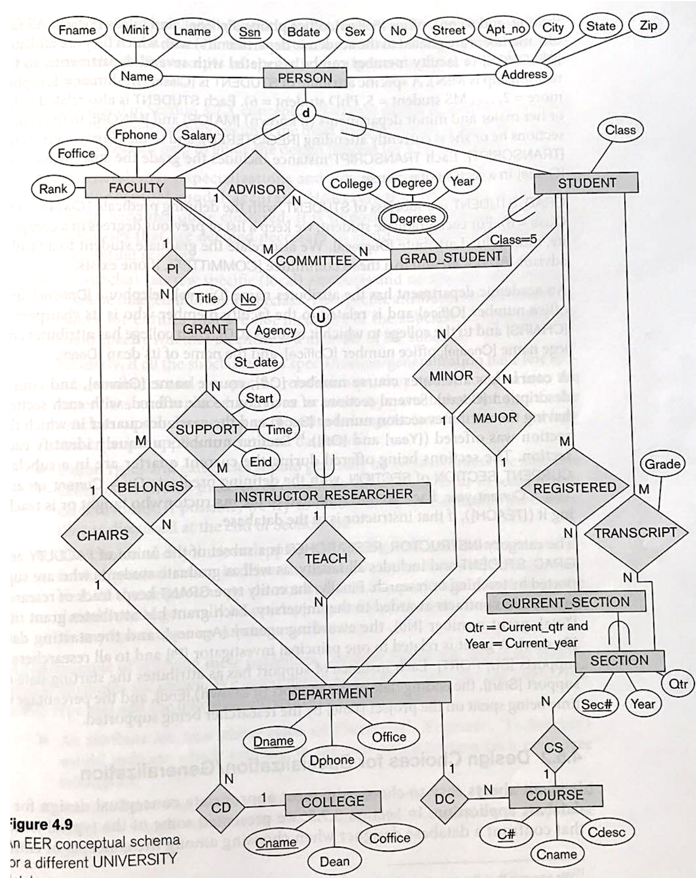
Kategori (union type) er en subklasse som representerer en samling av entiteter fra ulike entitetsklasser. **Denne subklassen er et subsett til UNION av entitetsklassene**. For eksempel kan vi ha entitetstypene PERSON, BANK og COMPANY, og vi ønsker å lage en database for registrering av kjøretøy, der eieren kan være en person, bank eller selskap. Da må vi lage en subklasse av kjøretøyeiere med entiteter fra disse tre entitetsklassene. Vi lager kategorien OWNER, som er en subklasse (subsett) til unionen av PERSON, BANK og COMPANY. I EER diagrammet vil entitetsklassene være koblet til en sirkel som igjen er koblet til subklassen via en linje markert med U symbol (sett union operasjon). På figuren har vi to kategorier: OWNER og REGISTERED_VEHICLE.

En kategori har to eller flere superklasser som kan representere en samling av entiteter fra ulike entitetsklasser, mens andre superklasse/subklasse relasjoner har alltid én superklasse. **En kategori er et subsett til unionen av superklassene, mens en delt subklasse er et subsett til snittet av superklassene**. Et medlem av kategorien OWNER må eksistere i kun én av superklassene, mens et medlem i delt subklasse ENGINEERING_MANAGER må eksistere i alle



tre superklassene. Medlemmet i OWNER må være et selskap, en bank eller en person, mens medlemmet i ENGINEERING_MANAGER må være en ingeniør, en manager og en lønnet ansatt. Vi bruker kategori når en entitet skal være i én av superklassene og delt subklasse når en entitet skal være i alle superklassene. **En kategori entitet vil arve attributter fra superklassen som entiteten hører til, mens en delt subklasse vil arve attributtene til alle superklassene.**

En total kategori er unionen av alle entitetene i superklassene, mens en delvis kategori er et subsett av unionen. Total kategori representeres med dobbel linje mellom kategorien og sirkelen, mens delvis kategori representeres av enkel linje. En total kategori kan representeres som en total spesialisering eller total generalisering (foretrekkes hvis klassene deler attributter og har felles nøkkelattributt, som ved REGISTERED_VEHICLE).



4.5 EER skjema for UNIVERSITY

Figuren viser et EER konseptuelt skjema for UNIVERSITY databasen. For hver person vil databasen inneholde informasjon om navn, personnummer, adresse, sex og bursdagsdato. PERSON entitetsklassen har to subklasser: FACULTY med attributter for rang, kontor, mobilnummer og lønn, og STUDENT med attributt

for klasse. Fakultetsmedlemmer er relatert til de akademiske avdelingen(e) som de hører til og studenter er relatert til akademisk avdeling, emnene de er registrert i og emnene de har fullført (transkript, inkluderer karakter). GRAD_STUDENT er en subklasse av STUDENT med definerende predikat CLASS = 5. Hver avgangselev har en sammensatt attributt for tidligere grader og er relatert til en rådgiver og en komité ved et fakultet.

En avdeling har attributter for navn, telefon og kontornummer, og den er relatert til fakultetsmedlemmet som er formann (CHAIRS) og til universitetet som det hører til (CD). Hvert universitet har attributter for navn, kontornummer og rektor. Et kurs vil ha attributter for kursnummer, kursnavn og kursbeskrivelse. Hvert kurs vil ha flere seksjoner som har attributter for seksjonsnummer, år og kvartal. CURRENT_SECTION er en subklasse av SECTION og representerer seksjonene som blir gitt i nåværende kvartal. Hver seksjon er relatert til instruktør som kan være et fakultetsmedlem eller en avgangselev (dvs. det er en kategori). Entitetsklassen GRANT gir informasjon om forskningsstipender og har attributter for stipendtittel, stipendnummer, tildelt byrå og startdato. En grant er relatert til hoved (PI) og til alle forskerne den støtter (SUPPORT).

Design valg for spesialisering/generalisering

Design av konseptuell database bør ses på som en iterativ raffineringsprosess, helt til man får det mest passende designet. Følgende er veiledningspunkter:

- Det er viktig å kun inkludere subklassene som er nødvendig, slik at man unngår rot i det konseptuelle skjemaet
- Hvis en subklasse har få attributter og ingen spesifikke relasjoner kan den fusjoneres med superklassen. De spesifikke attributtene vil være NULL for entiteter som ikke er medlem av subklassen. En type attributt brukes for å spesifisere om en entitet er medlem av subklassen.
- Union typer og kategorier bør unngås hvis det er mulig. Bruk heller spesialisering/generalisering
- Valget for disjunkt/overlappende og total/delvis bestemmes av reglene i miniverden. Hvis kravene ikke gir bestemte begrensninger, bruk overlappende og delvis som standard.

Formelle definisjoner for EER modellen

En **subklasse (S)** er en klasse der entitetene må være et subsett av entitetene i en annen klasse, kalt **superklassen (C)** til superklasse/subklasse relasjonen (IS-A). Vi bruker notasjonen S/C, der $S \subseteq C$.

En **spesialisering** $\boldsymbol{Z} = \{\boldsymbol{S_1}, \boldsymbol{S_2}, \dots, \boldsymbol{S_n}\}$ er et sett av subklasser som har samme superklasse G, og vi bruker notasjonen G/S_i. G kalles en **generalisert entitetsklasse**. Z er **total** hvis $\bigcup_{i=1}^{n} S_i = G$, altså alle entitetene i superklassen må være i minst en subklasse. Hvis ikke vil Z være **delvis**. Z er **disjunkt** hvis $S_i \cap S_j = \emptyset$ for $i \neq j$, altså en entitet vil kun høre til én subklasse. Hvis ikke vil Z være **overlappende**.

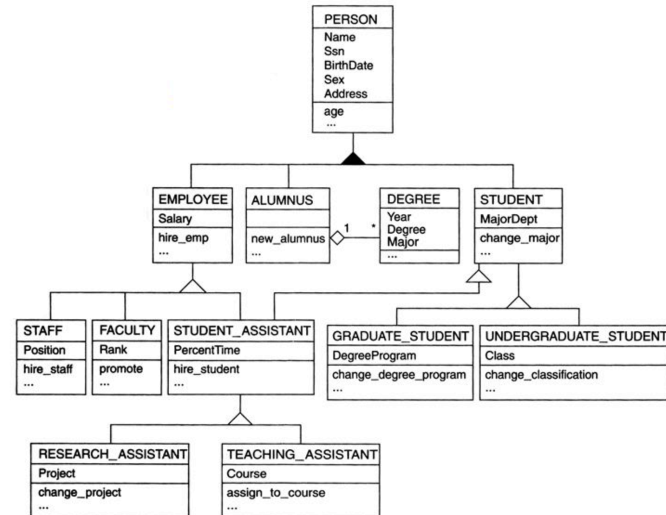
En subklasse S er **predikatdefinert** hvis et predikat p på attributtene til C brukes for å bestemme hvilke entiteter i C som er medlem av S. Vi bruker notasjonen $S = C[p]$, der $C[p]$ er entitetene i C som tilfredsstiller p. En subklasse som ikke defineres av et predikat kalles **brukerdefinert**.

En spesialisering er **attributtdefinert** hvis et predikat ($A = c_i$, der A er superklasse attributt og c_i er konstant verdi) brukes for å spesifisere medlemskap i subklassene.

En **kategori** T er en klasse som er subsettet til unionen av n superklasser $D_1, D_2, \ldots, D_n$, altså $T \subseteq (D_1 \cup D_2 \cup \ldots \cup D_n)$. Et predikat p_i på attributtene til D_i kan brukes for å bestemme medlemmene i superklassene: $T = (D_1[p_1] \cup D_2[p_2] \cup \ldots \cup D_n[p_n])$.

4.6 UML klassediagram

Figuren viser et UML klassediagram for UNIVERSITY databasen. Ved spesialisering/generalisering blir subklassene koblet med vertikale linjer til en horisontal linje som igjen er koblet til superklassen via en triangel. En tom triangel representerer disjunkt begrensning, mens en fylt triangel representerer en overlappende begrensning. Superklassen ved roten kalles baseklassen, mens subklassene kalles bladklasser.



Oppsummering – Kapittel 4 (F4, Øv1)

Definisjon av EER (Enhanced ER)

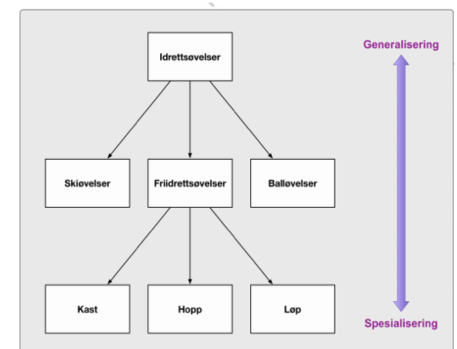
EER modellen er lik standard ER, pluss:

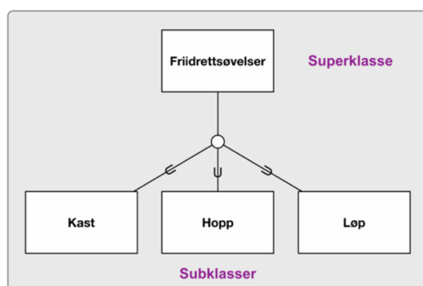
1. **Spesialisering/generalisering** – gir subklasser og superklasse
2. **Kategorier (union)** – representerer en samling av entiteter fra ulike entitetsklasser.
3. **Arving** av attributter og relasjoner

Spesialisering og generalisering

Skiller mellom:

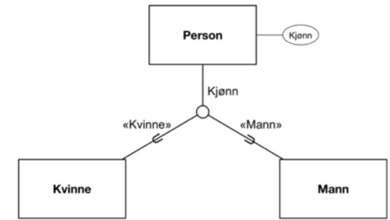
- **Spesialisering** = definerer en mengde subklasser (underklasser) for en entitetsklasse kalt superklassen.
- **Generalisering** = samler entitetsklasser med felles egenskaper som subklasser under en felles superklasse (overklasse).





Felles egenskaper modelleres på superklassen, mens det som er unikt for en subklasse modelleres på subklassen.

En entitet i subklassen er alltid en entitet i superklassen, men har en spesiell rolle. Entiteter i subklasser arver alle attributter og relasjoner til superklassen. Subklassene kan også ha egne attributter og relasjonsklasser. Entiteter må ikke være med i en subklasse, men det kan være et krav.



Restriksjoner ved spesialisering/generalisering:

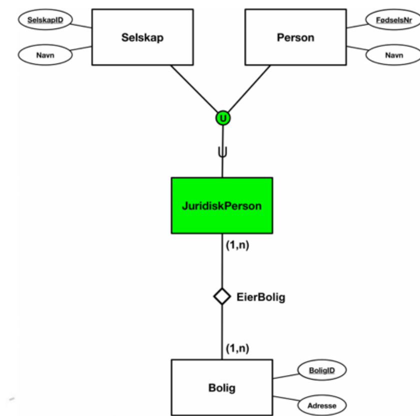
1. **Regelbasert eller brukerstyrt deltakelse i subklasse** – hvilke entiteter som blir medlem av en subklasse bestemmes av å plassere betingelser på verdien til en attributt i superklassen. Subklassene kalles predikatdefinerte subklasser og betingelsen kalles definerende predikat.
2. **Disjunkte eller overlappende subklasser** – ved disjunkte subklasser kan entiteter være i maksimalt én subklasse, mens ved overlappende kan de være i flere
3. **Delvis eller total spesialisering** – ved total spesialisering må alle entiteter være medlem av minst én subklasse, mens delvis spesialisering tillater at entiteter ikke tilhører noen av subklassene.

Tabellen til høyre viser en oppsummering av de ulike mulighetene.

Entitet deltar i	Disjunkt	Overlappende
Delvis	0-1 subklasser	0-n subklasser
Total	1 subklasse	1-n subklasser

Kategorier (F12)

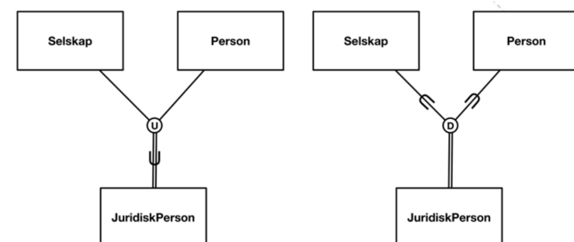
Kategorier er subklasser med flere superklasser, slik at mengden entiteter i en kategori er en delmengde av entitetene i superklassene. Dette er en UNION-klasse og betegnes derfor med **U i sirkelen**. For eksempel kan vi se på JuridiskPerson som er enten et selskap eller en person. Kategorier har selektiv arving, slik at entiteter vil arve egenskapene til superklassen som de stammer fra. For eksempel hvis JuridiskPerson er et Selskap vil den arve selskapsegenskapene, mens hvis den er en Person vil den arve personegenskapene.



Her bruker vi ikke spesialisering, fordi hvis vi sier at JuridiskPerson er superklassen og Selskap og Person er subklassene, vil vi tvinge Selskap og Person til å være JuridiskPerson. Vi opphøyer JuridiskPerson til å være viktigere enn Selskap og Person, noe som kan bli feil. Vi unngår dette ved å bruke en kategori.

Kategorier kan ha restriksjoner:

- **Delvis kategori** – Personer og Selskap trenger ikke å være med i JuridiskPerson (enkel strek).
- **Total kategori** – alle Personer og Selskap må være med i JuridiskPerson (venstre på figur). I dette tilfellet kunne vi ha brukt en spesialisering (høyre på figur).



Del 3 – Relasjonsmodell og SQL

Kapittel 5 – Relasjonsmodell og begrensninger

Relasjonsmodellen bruker matematisk relasjoner som ligner en verditabell, som grunnleggende byggeblokker. Vi skal se på grunnleggende egenskaper og begrensninger ved modellen.

5.1 Relasjonsmodell konsepter

Relasjonsmodellen representerer databasen som en samling av relasjoner, som ser ut som en tabell med verdier eller en flat fil (headeren). Når man ser på en relasjon som en verditabell, vil hver rad representere en samling av relaterte dataverdier. Raden vil ofte korrespondere til en entitet eller relasjon i den virkelige verden. Navnet på raden gir hvordan verdiene skal tolkes, for eksempel Name i Student relasjonen. Terminologi:

- **Tuple** = en rad i tabellen
- **Attributt** = en kolonne header
- **Relasjon** = selve tabellen
- **Domene** = datatypen til verdiene i en kolonne

5.1.1 Domene, attributter, tupler og relasjoner

Et domene *D* er et sett med atomiske verdier (dvs. hver verdi i domenet er udelelig). Det er vanlig å spesifisere domenet ved å gi datatypen som verdiene i domenet trekkes fra. Det kan være nyttig å gi et navn til domenet. Noen eksempler på domener er:

- Social_security_number– et sett med ni nummer (0-9) som unikt identifiserer en person
- Name – et sett med karakterer som representerer navnet til personer
- Employee_age – et heltall mellom 15 og 80

Dette er logiske definisjoner av domener. **Datatype** eller **format** blir også spesifisert for hvert domene. For eksempel for domenet Social_security_number vil datatypen være en karakterstring på formen dddddd-ddddd, der hver d er et tall og de første 6 tallene er en gyldig fødselsdato. For domenet Employee_age vil datatypen være et heltall mellom 15 og 80. Man kan også legge til mer informasjon som gjør det lettere å tolke verdiene. For eksempel for Person_weights kan man legge til måleenhet (eks: kg). **Et domene har altså et navn, data, type og format**.

Et relasjonsskjema R betegnes som $R(A_1, A_2, \dots, A_n)$ og består av relasjonsnavnet R og en liste av attributtene $(A_1, A_2, \dots, A_n)$. A_i er navnet på domenet, og domenet D til attributtet A_i betegnes som $\text{dom}(A_i)$. **Graden til en relasjon er antall attributter (n)**, for eksempel vil STUDENT(Name, Ssn, Age) ha grad 3. Dette kan også skrives med datatypen til hvert attributt: STUDENT(Name: string, Ssn: string, Age: integer). En **relasjon $r(R)$** i relasjonsskjemaet $R(A_1, \dots, A_n)$ er et sett med n-tupler: $r = \{t_1, \dots, t_m\}$ som hver har n verdier (en for hver attributt). i-ende verdi i tuple t korresponderer til attributt A_i og betegnes som $t[A_i]$ eller $t.A_i$.

Figuren viser et eksempel på en STUDENT relasjon der hver tuple representerer en

Relation Name
Attributes
STUDENT
Tuples

Name	Ssn	Home_phone	Address	Office_phone	Age	Gpa
Benjamin Bayer	305-61-2435	373-1616	2918 Bluebonnet Lane	NULL	19	3.21
Chung-cha Kim	381-62-1245	375-4409	125 Kirby Road	NULL	18	2.89
Dick Davidson	422-11-2320	NULL	3452 Elgin Road	749-1253	25	3.53
Rohan Panchal	489-22-1100	376-9821	265 Lark Lane	749-6492	28	3.93
Barbara Benson	533-69-1238	839-8461	7384 Fontana Lane	NULL	19	3.25

Merk: dette er en relasjon siden det er et sett med fem 7-tupler. Relasjonsskjemaet er STUDENT(Name, Ssn, Home_phone, Address, Office_phone, Age, Gpa), altså navnet på relasjonen og attributtene.

student entitet (eller objekt). NULL verdier representerer attributter der verdien er ukjent for en bestemt STUDENT tuple.

En relasjonstilstand består av tupler som representerer en bestemt tilstand i den virkelige verden. Når denne tilstanden endres (eks: en student er ferdig med studiene) vil også relasjonstilstandene endres. Relasjonsskjemaet R er derimot uendret og statisk (antall kolonner blir ikke endret).

5.1.2 Egenskaper ved relasjoner

Følgende er egenskaper ved relasjoner:

- **Ingen ordning av tupler i en relasjon** – en relasjon er et sett av tupler og disse har ingen bestemt rekkefølge. Når relasjonen blir vist som en tabell vil radene vises i en bestemt rekkefølge, men i databasen har de ingen bestemt ordning. Det er mulig å spesifisere en orden basert på en av attributtene (eks: alfabetisk etter Name), men definisjonen av en relasjon spesifiserer ingen rekkefølge.
- **Ordning av verdier i en tuple** – rekkefølgen til attributtene og deres verdier er ikke så viktig, så lenge korrespondansen mellom attributter og deres verdier er bevart. Definisjonen av en relasjon er likevel at attributtene er ordnet i en liste, fordi dette forenkler notasjonen. Alternativt kan vi bruke selvbeskrivende data der både verdien og attributtnavnet inkluderes i tuplen, for eksempel $t = < (\text{Name}, \text{Ola}), (\text{Ssn}, 05049854672), (\text{Age}, 25) >$.
- **Verdier og NULL i tupler** – verdiene i en tuple er atomiske, altså ikke delelige (dvs. ikke tillatt med sammensatte eller flerverdi attributter). Denne antagelsen kalles første normale form. NULL verdier brukes for å representere verdien til attributter som er ukjent eller ikke er gyldig for en bestemt tuple. NULL kan ha flere betydninger, for eksempel *value unknown*, *value exist but is not available* eller *value undefined to this tuple*. NULL verdier kan bli problematisk ved aritmetikk operasjoner og sammenligninger (eks: A og B har begge adresse NULL, men det betyr ikke at de har samme adresse).
- **Tolkning av en relasjon** – relasjonsskjemaet kan ses på som en type påstand. For eksempel påstår STUDENT relasjonen at hver student entitet har et navn, personnummer, osv. Hver tuple i relasjonen vil da være en fakta for påstanden. Relasjoner kan representere fakta om entiteter eller relasjoner, og man må av og til gjette hvilken som er representert hvis kun relasjonen er gitt. Relasjonsskjema kan også ses på som et predikat der verdiene i hver tuple tilfredsstiller predikatet.

5.2 Begrensninger ved relasjonsmodellen

En relasjonsdatabase vil ofte bestå av mange relasjoner der tupler er relatert på ulike måter. Det er mange begrensninger på verdiene i en database tilstand og disse er hentet fra reglene i miniverden som databasen representerer. Disse kan deles inn i tre hovedkategorier:

1. **Inneboende modell-baserte begrensninger (implisitte)** er inneboende i datamodellen. For eksempel at en relasjon ikke kan ha duplikate tupler.
2. **Skjema-baserte begrensninger (eksplisitte)** kan direkte spesifiseres i skjemaet til datamodellen.
3. **Applikasjonsbaserte begrensninger** kan ikke spesifiseres i skjemaet og må derfor uttrykkes i applikasjonen. De gjelder ofte betydningen og oppførselen til attributter.

Skjema-baserte begrensninger inkluderer begrensinger for domene, nøkler, NULL verdier og entitets- og referensiell integritet. Vi skal se nærmere på disse.

5.1.2 Domene begrensninger

Domene begrensninger sier at innenfor hver tuple må verdien til hvert attributt A være en atomisk verdi fra domenet til attributtet dom(A). Datatyper assosiert med domener inkluderer heltall, reelle tall, karakterer, boolean, stringer, dato, tid, osv.

5.2.2 Nøkkelbegrensninger og begrensninger på NULL verdier

En relasjon er et sett med tupler, der alle tuplene er ulike. To tupler i en relasjon kan derfor ikke ha samme kombinasjon av attributtverdier. En relasjon vil ha et subsett av attributter (SK) som ikke kan være lik for ulike tupler:

$$t_1[SK] \neq t_2[SK]$$

Dette kalles en supernøkkel, og SK gir en unikhetsbegrensning om at to ulike tupler ikke kan ha samme verdi for SK. Dersom ikke noe annet er oppgitt vil SK være alle attributtene. Supernøkkelen kan ha overflødige attributter, så ofte brukes heller en nøkkel k som er en supernøkkel der fjerning av en attributt vil gjøre at det ikke lenger er en supernøkkel (dvs. inkluderer kun nødvendige attributter). Nøkkelen har to egenskaper:

1. To ulike tupler i en relasjon **kan ikke ha identiske verdier for alle attributtene i nøkkelen**. Dette er også et krav for supernøkkel.
2. **Det er en minimal supernøkkel**, dvs. dersom vi fjerner en attributt vil ikke unikhetsbegrensningen gjelde lenger. Dette er ikke et krav for supernøkkel.

En nøkkel vil være en supernøkkel, men ikke nødvendigvis omvendt. For STUDENT relasjonen vil {Ssn} være en nøkkel og supernøkkel, mens for eksempel {Ssn, Name, Age} er en supernøkkel, men ikke en nøkkel (den er ikke minimal!). Supernøkler med en attributt vil alltid være en nøkkel.

CAR	LicenseNumber	EngineSerialNumber	Make	Model	Year
	Texas ABC-739	A69352	Ford	Mustang	96
	Florida TVP-347	B43696	Oldsmobile	Cutlass	99
	New York MPO-22	X83554	Oldsmobile	Delta	95
	California 432-TFY	C43742	Mercedes	190-D	93
	California RSK-629	Y82935	Toyota	Camry	98
	Texas RSK-629	U028365	Jaguar	XJS	98

Dersom relasjonsskjemaet har flere nøkler, kaller vi hver nøkkel for kandidatnøkkel. For eksempel på figuren ser vi at CAR relasjonen har to kandidatnøkler: Licence_number og Engine_serial_number. Det er vanlig å sette en av kandidatnøklene som primærnøkkelen til relasjonen (vilkårlig valg). **Verdien til primærnøkkelen brukes for å identifisere tuplene i relasjonen** og disse blir understreket i relasjonsskjemaet (se figur). De andre nøklene blir betegnet som unike nøkler og blir ikke understreket.

Merk: både supernøkkel og kandidatnøkkel kan identifisere en bestemt tuple, men kandidatnøkkelen må i tillegg være minimal

Dersom en attributt ikke kan være NULL kan dette spesifiseres ved å inkludere NOT NULL etter attributtet. En nøkkel vil ha denne begrensningen.

5.2.3 Relasjonsdatabaser

En relasjonsdatabase består som regel av mange relasjoner som er relatert til hverandre. **Et relasjonsdatabase skjema S er et sett av relasjonsskjema: $S = \{R_1, \ldots, R_m\}$ og et sett av integritetsbegrensninger IC**. En databasetilstand må tilfredsstille integritetsbegrensningene for å være gyldig.

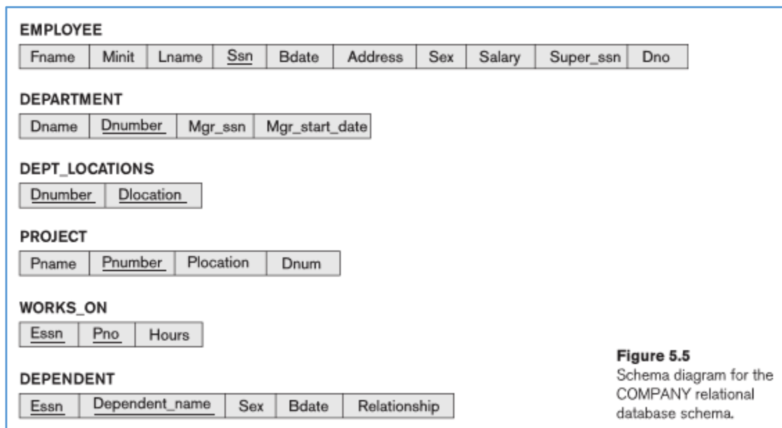


Figure 5.5
Schema diagram for the COMPANY relational database schema.

Figure 5.6
One possible database state for the COMPANY relational database schema.

EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Alicia	J	Zelaya	999887777	1968-01-19	3321 Castle, Spring, TX	F	25000	987654321	4
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5
Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000	987654321	4
James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000	NULL	1

DEPARTMENT

Dname	Dnumber	Mgr_ssn	Mgr_start_date
Research	5	333445555	1988-05-22
Administration	4	987654321	1995-01-01
Headquarters	1	888665555	1981-06-19

DEPT_LOCATIONS

Dnumber	Dlocation
1	Houston
4	Stafford
5	Bellaire
5	Sugarland
5	Houston

WORKS_ON

Essn	Pno	Hours
123456789	1	32.5
123456789	2	7.5
666884444	3	40.0
453453453	1	20.0
453453453	2	20.0
333445555	2	10.0
333445555	3	10.0
333445555	10	10.0
333445555	20	10.0
999887777	30	30.0
999887777	10	10.0
987987987	10	35.0
987987987	30	5.0
987654321	30	20.0
987654321	20	15.0
888665555	20	NULL

PROJECT

Pname	Pnumber	Plocation	Dnum
ProductX	1	Bellaire	5
ProductY	2	Sugarland	5
ProductZ	3	Houston	5
Computerization	10	Stafford	4
Reorganization	20	Houston	1
Newbenefits	30	Stafford	4

DEPENDENT

Essn	Dependent_name	Sex	Bdate	Relationship
333445555	Alice	F	1986-04-05	Daughter
333445555	Theodore	M	1983-10-25	Son
333445555	Joy	F	1958-05-03	Spouse
987654321	Abner	M	1942-02-28	Spouse
123456789	Michael	M	1988-01-04	Son
123456789	Alice	F	1988-12-30	Daughter
123456789	Elizabeth	F	1967-05-05	Spouse

Figuren over viser skjemaet til relasjonsdatabasen COMPANY, mens figuren til høyre viser en tilstanden til databasen. Dnumber i DEPARTMENT og DEPT_LOCATION representerer tallet som er gitt til avdelingen, og dette er kalt Dno i EMPLOYEE og Dnum i PROJECT. Attributter som representerer samme konsept fra den virkelige verden kan ha like eller ulike navn i ulike relasjoner. Attributter som representerer ulike ting kan ha samme navn i ulike relasjoner.

5.2.4 Entitetsintegritet, referensiell integritet og fremmednøkler

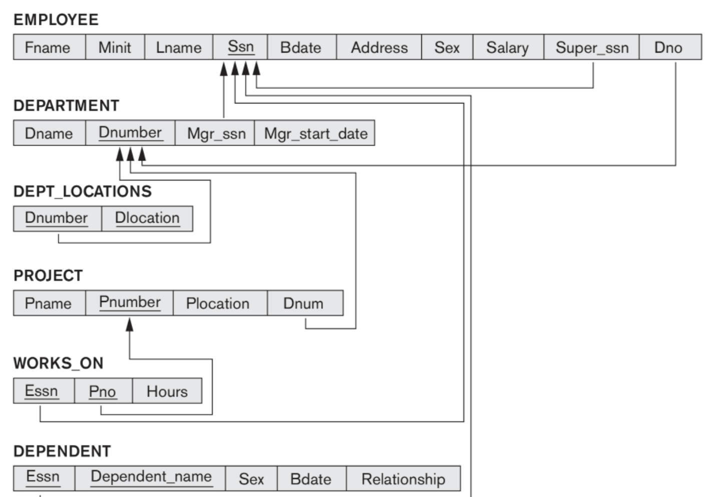
To viktige begrensninger i relasjonsdatabase

- **Entitetsintegritet begrensning – den primære nøkkelverdien kan ikke være NULL, fordi den brukes til å identifisere individuelle tupler i en relasjon**. Dersom to eller flere tupler har NULL som primærnøkkel, vil vi ikke kunne skille mellom disse.
- **Referensiell integritetsbegrensning** – spesifiseres mellom tupler i to ulike relasjoner og brukes for å opprettholde riktig forhold mellom disse. Hvis tuple A i relasjon R_1 refererer til tuple B i relasjon R_2, må tuple B eksistere. Dette forholdet blir oppnådd vha fremmednøkkel og primærnøkkel. **Tuple A vil ha en fremmednøkkel som har samme verdi som primærnøkkelen til tuple B: $t_A[FK] = t_B[PK]$. Fremmednøkkelen skal ha verdi NULL dersom tuple A ikke refererer til noen tupler i den andre relasjonen.** Hvis disse to betingelsene er oppfylt sier vi at den referensielle integritetsbegrensningen fra R_1 til R_2 er oppfylt.

For å undersøke om begrensningene er oppfylt må vi se på rollen til attributtene i skjemaet. **De primære nøklene er understreket og vi tegner piler som peker fra fremmednøkkel til primærnøkkel**. På figuren kan vi se at Dno er en fremmednøkkel i EMPLOYEE som refererer til DEPARTMENT relasjonen. Verdien til Dno må derfor matche verdien til Dnumber eller være NULL.

Merk: **fremmednøkler kan også referere til sin egen relasjon**, for eksempel Super_ssn i EMPLOYEE.

Figure 5.7
Referential integrity constraints displayed on the COMPANY relational database schema.



5.2.5 Andre typer begrensninger

Begrensningene på forrige side kalles tilstandsbegrensninger (bestemmer gyldig tilstand) og de er definert i DDL (data definition language) fordi de er til stede i de fleste databaseapplikasjonene. Generelle begrensninger, som kalles semantisk integritetsbegrensninger, blir spesifisert i applikasjonsprogrammene som oppdaterer databasen. For eksempel at maksimal antall timer en ansatt kan arbeide per prosjekt er 56 timer i uken. De kan også spesifiseres vha triggers og assertions i SQL, men det kan være mer komplisert. Transisjonsbegrensninger håndterer endringer i databasetilstanden (eks: lønnen kan kun øke), og de blir som regel håndhevet i applikasjonsprogrammene.

5.3 Oppdatering, transaksjoner og håndtering av brutte begrensninger

Operasjoner på relasjonsmodeller kan deles inn i uttak (*retrivals*, kapittel 8) og oppdateringer. Vi ser på operasjoner for oppdatering (modifisering), og det er tre grunnleggende operasjoner som kan endre tilstanden til relasjoner i en database: Insert, Delete og Update/Modify.

5.3.1 Insert operasjonen

Insert operasjonen gir en liste med attributtverdier for en ny tuple *t* som skal settes inn i relasjonen *R*, og den kan bryte alle de de oppgitte tilstandsbegrensningene:

- Domene begrensningen kan brytes hvis en gitt attributtverdi ikke hører til riktig domene eller har feil datatype
- Nøkkel begrensningen kan brytes hvis tuplen har samme nøkkelverdi(er) som en annen tuple som allerede er i relasjonen
- Entitetsintegritet begrensningen kan brytes hvis tuplen har NULL som primærnøkkel
- Referensiell integritetsbegrensning kan brytes hvis fremmednøkkelen ikke refererer til noen eksisterende tuple i den andre relasjonen

Hvis en eller flere av disse begrensningene brytes, vil innsettingen avslås.

5.3.2 Delete operasjonen

Delete operasjonen fjerner en tuple *t* fra relasjonen *R*, og den kan kun bryte referensiell integritetsbegrensning ved at den sletter en tuple som blir referert av fremmednøkler i andre tupler i databasen. Det er ulike måter å håndtere dette:

- ***Restrict***– delete operasjonen blir avslått
- ***Cascade*** – slettingen blir forplantet ved å slette tupler som refererer til tuplen
- ***Set null/set default*** – det refererende attributtet som forårsaker feilen blir endret ved at det settes lik NULL eller lik primærnøkkelen til en annen standard tuple som er gyldig. **OBS: hvis attributtet er en del av primærnøkkelen kan den ikke settes lik NULL!**

5.3.3 Update operasjonen

Update operasjonen brukes for å endre verdien til en eller flere attributter i en tuple, og den kan bryte alle de oppgitte tilstandsbegrensningene (se Insert).

5.3.4 Transaksjonskonseptet

En transaksjon er et utførende program som inkluderer databaseoperasjoner som lesing, innsetting, sletting og oppdatering av databasen. En transaksjon må etterlate databasen i en gyldig tilstand som tilfredsstiller begrensningene spesifisert i databaseskjemaet.

Oppsummering – Kapittel 5 (F4, Øv2)

Relasjonsmodellen representerer databasen som en samling av **relasjoner**, som ser ut som en tabell med verdier. Hver rad representere en entitet eller relasjon i den virkelige verden. **Primærnøkkelen** er en entydig identifikator, mens **fremmednøkkelen** er en verdi som viser til en rad i en annen tabell (evt. samme tabell).

Navn
Primær nøkkel
kolonne /attributt
Tabell/ relasjon
rad/tuple
verdi fra domenet
Fremmed nøkkel

Hund

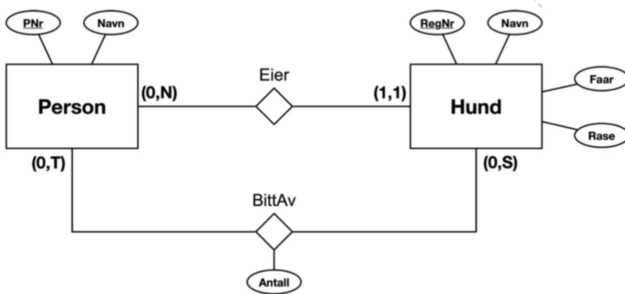
RegNr	Navn	FødselsÅr	EierPnr
200	Pluto	2018	1
400	Bolivar	2017	2
300	Fant	NULL	1

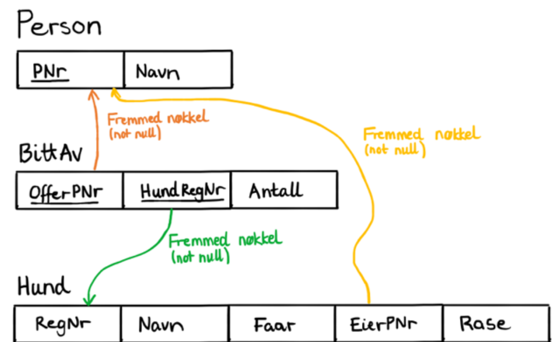
Standard relasjonsdatabaser har atomiske verdier i domenene og en verdi for hvert attributt, noe som gir oss flate og 2-dimensjonale tabeller. **Ikke-standard relasjonsdatabaser** kan ha sammensatte attributter og repeterende grupper (dvs. flerverdig attributter).

To viktige begrensninger i relasjonsmodellen:

1. **Entitetsintegritet** – en relasjon kan ikke ha to like rader, og tillater derfor ikke NULL-verdier i primærnøkkelen, selv om den er flere attributter.
2. **Referanseintegritet** – fremmednøkler må referere til en rad som finnes eller ha NULL-verdi

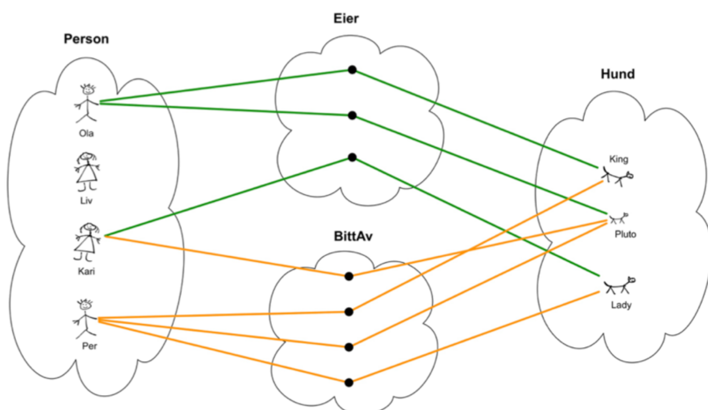
Figuren under viser overgangen fra ER modell til relasjonstabell





Legg merke til at BittAv får en egen relasjonstabell siden den er en N-N relasjon (mer på neste side). Da vil fremmednøklene OfferPNr og HundRegNr tilsammen være primærnøkkelen til relasjonen. Eier er en 1-N relasjon, og får derfor ikke en egen relasjonstabell.

Figuren under viser overgangen fra forekomstdiagram til relasjonstilstander:



Person

PNr	Navn
1	Ola
2	Liv
3	Kari
4	Per

Hund

RegNr	Navn	Faar	Rase	EierPNr
1	King	NULL	NULL	1
2	Pluto	NULL	NULL	1
3	Lady	NULL	NULL	3

BittAv

OfferPNr	HundRegNr
4	1
4	2
4	3
3	2

Legg merke til at primærnøklene ikke kan være NULL. Fremmednøkkelen kan være NULL dersom tuplen ikke har noen relasjon til en annen tuple.

Kapittel 6 – Grunnleggende SQL

SQL språket er en standard for relasjonsdatabaser. Hvis en bruker har et relasjonelt DBMS produkt og ønsker å omdanne det til et annet relasjonelt DBMS produkt, vil det ikke være dyrt eller tidskrevende, siden begge systemene følger samme språkstandard. Brukere kan også skrive påstander i et database applikasjonsprogram som kan hente data lagret i to eller flere relasjonelle DBMS uten å endre språket. Dette kapitlet handler om den praktiske relasjonsmodellen som er basert på SQL standarden, mens kapittel 8 handler om relasjonsalgebra som brukes for å forstå forespørselstypene som kan brukes på en relasjonsdatabase. Et spørsmål i relasjonsalgebra gir en sekvens av operasjoner som vil produsere et resultat når de utføres, og brukeren må spesifiserer rekkefølgen til disse. For mange DBMS vil relasjonsalgebra være ved et for lavt nivå. SQL språket spesifiserer kun hva resultatet skal bli, så hvordan spørsmålet skal utføres blir avgjort av DBMS. **SQL er et omfattende databasespråk som har påstander for datadefinisjon, spørsmål og oppdateringer**. Derfor er SQL både en DDL og en DML (s. 8). Det har også tjenester for å definere visninger på databasen, spesifisere sikkerhet og autentisering, osv. SQL påstander kan introduseres i generelle programmeringsspråk som Java og C++.

6.1 SQL datadefinisjon og datatyper

SQL bruker begrepene tabell, rad og kolonne for hhv. relasjon, tuple og attributt.

6.1.1 Skjema og katalog konsepter i SQL

SQL skjema brukes for å samle tabeller og andre konstruksjoner som hører til samme database applikasjon. Et SQL skjema identifiseres av et skjemanavn og inkluderer en autoriserings-identifikasjon for å indikere brukeren som eier skjemaet. Skjemaet inkluderer også beskrivelser av alle skjemaelementene, for eksempel tabeller, typer, begrensninger, osv. For å lage skjemaet brukes påstanden CREATE SCHEMA. For eksempel kan vi lage skjemaet kalt COMPANY eid av brukeren 'JSMITH' vha:

CREATE SCHEMA COMPANY **AUTHORIZATION** 'Jsmit';

SQL bruker en **katalog**, som er en samling av skjemaer og inneholder et informasjonsskjema som gir informasjon om alle skjemaene i katalogen og elementene i disse. Integritetsbegrensninger kan kun defineres mellom relasjoner som er i skjema innenfor samme katalog. Skjemaer i samme katalog kan også dele bestemte elementer, som for eksempel definisjon av type og domene.

6.1.2 CREATE TABLE

CREATE TABLE brukes for å lage en ny relasjon ved å gi den et navn og spesifisere attributter og initiale begrensninger. Hvert attributt får et navn, en datatype som spesifiserer domene og kanskje en begrensning (eks: NOT NULL). Figuren viser påstandene for å definere tabellene i COMPANY skjemaet.

```
CREATE TABLE EMPLOYEE
        ( Fname             VARCHAR(15)       NOT NULL,
          Minit             CHAR,
          Lname             VARCHAR(15)       NOT NULL,
          Ssn               CHAR(9)           NOT NULL,
          Bdate             DATE,
          Address           VARCHAR(30),
          Sex               CHAR,
          Salary            DECIMAL(10,2),
          Super_ssn         CHAR(9),
          Dno               INT               NOT NULL,
        PRIMARY KEY (Ssn),
CREATE TABLE DEPARTMENT
        ( Dname             VARCHAR(15)       NOT NULL,
          Dnumber           INT               NOT NULL,
          Mgr_ssn           CHAR(9)           NOT NULL,
          Mgr_start_date    DATE,
        PRIMARY KEY (Dnumber),
        UNIQUE (Dname),
        FOREIGN KEY (Mgr_ssn) REFERENCES EMPLOYEE(Ssn) );
CREATE TABLE DEPT_LOCATIONS
        ( Dnumber           INT               NOT NULL,
          Dlocation         VARCHAR(15)       NOT NULL,
        PRIMARY KEY (Dnumber, Dlocation),
        FOREIGN KEY (Dnumber) REFERENCES DEPARTMENT(Dnumber) );
CREATE TABLE PROJECT
        ( Pname             VARCHAR(15)       NOT NULL,
          Pnumber           INT               NOT NULL,
          Plocation         VARCHAR(15),
          Dnum              INT               NOT NULL,
        PRIMARY KEY (Pnumber),
        UNIQUE (Pname),
        FOREIGN KEY (Dnum) REFERENCES DEPARTMENT(Dnumber) );
CREATE TABLE WORKS_ON
        ( Essn              CHAR(9)           NOT NULL,
          Pno               INT               NOT NULL,
          Hours             DECIMAL(3,1)      NOT NULL,
        PRIMARY KEY (Essn, Pno),
        FOREIGN KEY (Essn) REFERENCES EMPLOYEE(Ssn),
        FOREIGN KEY (Pno) REFERENCES PROJECT(Pnumber) );
CREATE TABLE DEPENDENT
        ( Essn              CHAR(9)           NOT NULL,
          Dependent_name    VARCHAR(15)       NOT NULL,
          Sex               CHAR,
          Bdate             DATE,
          Relationship      VARCHAR(8),
        PRIMARY KEY (Essn, Dependent_name),
        FOREIGN KEY (Essn) REFERENCES EMPLOYEE(Ssn) );
```

SQL skjemaet blir som regel indirekte spesifisert i miljøet der CREATE TABLE påstandene utføres. Alternativ kan vi direkte feste skjemanavnet til relasjonsnavnet ved å skrive skjemanavn.relasjonsnavn. For eksempel kan vi la EMPLOYEE tabellen bli en del av COMPANY skjemaet ved å skrive:

CREATE TABLE COMPANY. EMPLOYEE;

I SQL vil kolonnene være ordnet slik de blir spesifisert, mens radene er ikke ordnet. Det er viktig at du ikke legger til en fremmednøkkel som peker mot en relasjon som enda ikke har blitt laget. Du må sikre at relasjonen som det pekes mot blir laget før fremmednøkkelen refererer til den. For eksempel er det viktig å lage DEPARTMENT før DEPT_LOCATION på figuren.

```
CREATE TABLE DEPARTMENT
        ( Dname             VARCHAR(15)        NOT NULL,
          Dnumber           INT                NOT NULL,
          Mgr_ssn           CHAR(9)            NOT NULL,
          Mgr_start_date    DATE,
        PRIMARY KEY (Dnumber),
        UNIQUE (Dname),
        FOREIGN KEY (Mgr_ssn) REFERENCES EMPLOYEE(Ssn) );
CREATE TABLE DEPT_LOCATIONS
        ( Dnumber           INT                NOT NULL,
          Dlocation         VARCHAR(15)        NOT NULL,
        PRIMARY KEY (Dnumber, Dlocation),
        FOREIGN KEY (Dnumber) REFERENCES DEPARTMENT(Dnumber) );
```

6.1.3 Datatyper og domener i SQL

De grunnleggende datatypene som er tilgjengelig for attributtene i SQL er:

- **Numeriske** – heltall av ulike størrelser (INTEGER, INT, SMALLINT) og flyttall med ulik presisjon (FLOAT, REAL, DOUBLE PRECISION, DEC(i, j), osv.).
- **Karakter-streng** – har fast lengde CHAR(n) eller varierende lengde VARCHAR(n). For strenger med fast lengde vil blanke karakterer legges til ord som er for korte. CLOB spesifiserer lange tekstkolonner (eks: CLOB(20M) er maksimalt 20 megabytes). Verdier er lagret som '…'
- **Bit-streng** – har fast lengde BIT(n) eller varierende lengde BIT VARYING(n). BLOB spesifiserer lange binære kolonner. Verdier er lagret som B'…'
- **Boolean** – har verdien TRUE, FALSE eller UNKNOWN (SQL tillater NULL verdier).
- **Dato** – datatypen DATE har komponentene YEAR, MONTH og DAY lagret på formen DATE 'YYYY-MM-DD'. Det er kun gyldige verdier som er tillatt.
- **Tid** – datatypen TIME har komponentene HOUR, MINUTE og SECOND lagret på formen TIME 'HH:MM:SS'. Det er kun gyldige verdier som er tillatt
- **Timestamp** – inkluderer DATE og TIME felt, og har i tillegg seks sekunddesimaler. Verdier er lagret som TIMESTAMP 'YYYY-MM-DD HH:MM:SS.SSSSSS'
- **Intervall** – datatypen INTERVAL spesifiserer en relativ verdi som brukes for å øke eller minke en absolutt verdi til en dato, tid eller timestamp.

6.2 Spesifisering av begrensninger i SQL

Vi skal se på grunnleggende begrensninger som kan spesifiseres i SQL som en del av tabellen som lages.

Spesifisering av begrensninger og standarder for attributter

Følgende er begrensninger på attributter:

- **NOT NULL** – SQL tillater NULL som attributtverdi, så derfor kan NOT NULL begrensningen kan brukes for å bestemme at attributter ikke kan være NULL. NULL. For eksempel blir det brukt for attributtene som er en del av primærnøkkelen til relasjonen.
- **DEFAULT ⟨value⟩** - brukes for å definere en standardverdi til en attributt, som brukes dersom ingen annen verdi blir gitt i en tuple. NULL er standardverdi for attributter som har ingen NOT NULL begrensning.

```
CREATE TABLE EMPLOYEE
    ( … ,
      Dno        INT          NOT NULL      DEFAULT 1,
    CONSTRAINT EMPPK
      PRIMARY KEY (Ssn),
```

- **CHECK** (...) - brukes for å begrense verdier til attributter eller domener. For eksempel kan vi sørge for at avdelingsnummeret er mellom 1 og 20 ved å se på:

  Dnum INT **NOT NULL CHECK** (Dnum > 0 **AND** Dnum < 21)

  Vi kan også legge denne begrensningen til domenet, som så kan brukes som attributtype for alle attributtene som refererer til avdelingsnummer (eks: Dnum i PROJECT, Dno i EMPLOYEE, osv.)

CREATE DOMAIN D_NUM **AS** INTEGER **CHECK** (D_NUM > 0 **AND** D_NUM < 21)

Dermed kan vi bruke Dnum D_NUM for å lage Dnum attributtet.

```
CREATE TABLE Film
    (FilmID INT NOT NULL PRIMARY KEY
    );

CREATE TABLE Film
    (FilmID INT NOT NULL,
    SjangerID INT NOT NULL,
    PRIMARY KEY(FilmID, SjangerID)
    );
```

Spesifisering av begrensninger for nøkkel og referensiell integritet

Nøkkel og referensiell integritetsbegrensninger er svært viktige:

- **PRIMARY KEY** – brukes for å **spesifisere en eller flere attributter som utgjør primærnøkkelen** til relasjonen. Hvis det er en attributt kan det legges til etter attributtet, mens hvis det er flere må det legges til som en egen setning (se figur).
- **UNIQUE** – brukes for å **spesifisere kandidatnøkler** i tilfeller der relasjonen har flere attributter som skal være unike for hver tuple. Hvis det er en attributt kan det legges til etter attributtet, mens hvis det er flere må det legges til som en egen setning (se figur).

Husk: kandidatnøkler er når relasjonen har flere unike attributter, og en eller flere settes som primærnøkkel

```
CREATE TABLE Film
    (FilmID INT NOT NULL PRIMARY KEY,
    SjangerID INT UNIQUE
    );

CREATE TABLE Film
    (FilmID INT NOT NULL PRIMARY KEY,
    SjangerID INT,
    FilmNavn CHAR,
    UNIQUE(SjangerID, FilmNavn)
    );
```

- **FOREIGN KEY** – brukes for å spesifisere referensiell integritet og må skrives i en egen setning (se figur). Denne begrensningen kan brytes når tupler blir satt inn, slettet eller når fremmednøkkel eller primærnøkkel blir oppdatert (kapittel 5). Standard handling for SQL er å **avslå endringen**, noe som kalles RESTRICT. Alternative handlinger kan spesifiseres ved å legge til **referensielle utløste handlinger**. Disse spesifiseres etter **ON DELETE** eller **ON UPDATE**, som representerer når tuplen fremmednøkkelen referer til blir hhv. slettet eller oppdatert. De ulike handlingene er:
  - **SET NULL** – når tuplen blir slettet eller oppdatert vil fremmednøkkelen settes lik NULL.
  - **CASCADE** – hvis primærnøkkelen til tuplen det pekes mot blir endret til en ny verdi, vil denne verdien også overføres til alle refererende fremmednøkler. Ved **ON DELETE CASCADE** vil alle tupler som refererer til den slettede tuplen også bli slettet. **CASCADE** er nyttig for relasjoner som representerer relasjonsklasser, flerverdi attributter og svake entitetsklasser.
  - **SET DEFAULT** - når tuplen blir slettet eller oppdatert vil fremmednøkkelen settes lik standardverdien (NULL hvis ikke noe annet er spesifisert).

```
CREATE TABLE Film
    (FilmID INT NOT NULL,
    SjangerID INT,
    SkuespillerID CHAR,

    PRIMARY KEY (FilmID),
    FOREIGN KEY (SjangerID) REFERENCES Sjanger(SjangerID)
        ON DELETE SET NULL
        ON UPDATE CASCADE,
    Foreign key(SkuespillerID) References Skuespiller(SkuespillerID)
        ON DELETE SET NULL
        ON UPDATE CASCADE
    );
```

Legg merke til at fremmednøkkelen defineres ved å gi navnet til attributtet som skal være fremmednøkkel og deretter gi hvilken relasjon og attributt den skal referere til

Det er vanlig å bruke **ON DELETE SET NULL** og **ON UPDATE CASCADE**.

```
CREATE TABLE EMPLOYEE
    ( ... ,
      Dno           INT            NOT NULL       DEFAULT 1,
    CONSTRAINT EMPPK
      PRIMARY KEY (Ssn),
    CONSTRAINT EMPSUPERFK
      FOREIGN KEY (Super_ssn) REFERENCES EMPLOYEE(Ssn)
                    ON DELETE SET NULL        ON UPDATE CASCADE,
    CONSTRAINT EMPDEPTFK
      FOREIGN KEY(Dno) REFERENCES DEPARTMENT(Dnumber)
                    ON DELETE SET DEFAULT     ON UPDATE CASCADE);
CREATE TABLE DEPARTMENT
    ( ... ,
      Mgr_ssn CHAR(9)          NOT NULL       DEFAULT '888665555',
      ... ,
    CONSTRAINT DEPTPK
      PRIMARY KEY(Dnumber),
    CONSTRAINT DEPTSK
      UNIQUE (Dname),
    CONSTRAINT DEPTMGRFK
      FOREIGN KEY (Mgr_ssn) REFERENCES EMPLOYEE(Ssn)
                    ON DELETE SET DEFAULT     ON UPDATE CASCADE);
CREATE TABLE DEPT_LOCATIONS
    ( ... ,
    PRIMARY KEY (Dnumber, Dlocation),
    FOREIGN KEY (Dnumber) REFERENCES DEPARTMENT(Dnumber)
                  ON DELETE CASCADE           ON UPDATE CASCADE);
```

Gi navn til begrensninger

Figuren viser hvordan man kan gi begrensninger navn etter **CONSTRAINT**, for å identifisere bestemte begrensninger i tilfeller der man ønsker å bruke dem eller erstatte dem med andre begrensninger. Innenfor et skjema må alle begrensningsnavn være unike. Det er valgfritt å bruke begrensningsnavn.

Spesifisering av begrensninger vha CHECK

Andre tabellbegrensninger kan spesifiseres vha en CHECK kommando etter CREATE TABLE. Disse begrensningene kalles **radbaserte begrensninger**, fordi de brukes på hver rad individuelt og blir sjekket hver gang en rad blir lagt til eller endret. For eksempel dersom vi ønsker å sikre at avdelingen ble dannet før manageren startet i jobben sin, kan vi legge til:

CREATE TABLE DEPARTMENT **CHECK** (DepCreateDate <= MgrStartDate)

6.3 Grunnleggende hentespørringer i SQL

SELECT er en grunnleggende påstand i SQL som brukes for å hente informasjon fra en database. Det er ikke det samme som SELECT operasjonen i relasjonsalgebra (kapittel 8). Vi vil se på eksempler som bruker databasen på figuren til høyre. I kapittel 7 vil vi se på mer komplekse spørringer. Det er viktig å være klar over at SQL tabeller kan inneholde to eller flere identiske tupler, hvis det ikke er oppgitt at alle må være unike (eks: oppgitt primærnøkkel).

EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Alicia	J	Zelaya	999887777	1968-01-19	3321 Castle, Spring, TX	F	25000	987654321	4
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5
Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000	987654321	4
James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000	NULL	1

DEPARTMENT

Dname	Dnumber	Mgr_ssn	Mgr_start_date
Research	5	333445555	1988-05-22
Administration	4	987654321	1995-01-01
Headquarters	1	888665555	1981-06-19

DEPT_LOCATIONS

Dnumber	Dlocation
1	Houston
4	Stafford
5	Bellaire
5	Sugarland
5	Houston

WORKS_ON

Essn	Pno	Hours
123456789	1	32.5
123456789	2	7.5
666884444	3	40.0
453453453	1	20.0
453453453	2	20.0
333445555	2	10.0
333445555	3	10.0
333445555	10	10.0
333445555	20	10.0
999887777	30	30.0
999887777	10	10.0
987987987	10	35.0
987987987	30	5.0
987654321	30	20.0
987654321	20	15.0
888665555	20	NULL

PROJECT

Pname	Pnumber	Plocation	Dnum
ProductX	1	Bellaire	5
ProductY	2	Sugarland	5
ProductZ	3	Houston	5
Computerization	10	Stafford	4
Reorganization	20	Houston	1
Newbenefits	30	Stafford	4

DEPENDENT

Essn	Dependent_name	Sex	Bdate	Relationship
333445555	Alice	F	1986-04-05	Daughter
333445555	Theodore	M	1983-10-25	Son
333445555	Joy	F	1958-05-03	Spouse
987654321	Abner	M	1942-02-28	Spouse
123456789	Michael	M	1988-01-04	Son
123456789	Alice	F	1988-12-30	Daughter
123456789	Elizabeth	F	1967-05-05	Spouse

SQL – enkle spørringer: seleksjon VIKTIG

Den grunnleggende formen til SELECT påstanden kalles mapping eller select-from-where blokk, og har følgende form:

```
SELECT    <attribute list>
FROM      <table list>
WHERE     <condition>;
```

Her er ⟨attribute list⟩ attributtnavnene som man ønsker å hente verdien til, ⟨table list⟩ er relasjonsnavnene som trengs for å behandle spørringen og ⟨condition⟩ er en betingelse som identifiserer tuplene som skal hentes i spørringen. I SQL blir =, >=, >, <, <= og <> (≠) brukt for grunnleggende sammenligning. Figurene under viser noen eksempler.

```
Q0:   SELECT   Bdate, Address
      FROM     EMPLOYEE
      WHERE    Fname='John' AND Minit='B' AND Lname='Smith';
```

Henter fødselsdato og adresse til ansatte som heter «John B. Smith»

```
Q1:   SELECT   Fname, Lname, Address
      FROM     EMPLOYEE, DEPARTMENT
      WHERE    Dname='Research' AND Dnumber=Dno;
```

Henter fornavn, etternavn og adresse til ansatte som jobber ved «Research» avdelingen

```
Q2:   SELECT   Pnumber, Dnum, Lname, Address, Bdate
      FROM     PROJECT, DEPARTMENT, EMPLOYEE
      WHERE    Dnum=Dnumber AND Mgr_ssn=Ssn AND
               Plocation='Stafford';
```

For alle prosjekt i «Stafford» skal prosjektnummer, avdelingsnummer og etternavn og fødselsdato til manageren hentes

Avsnitt som er merket med SQL - ... VIKTIG er det som er fremhevet som viktige SQL spørringer i forelesning.

Bdate	Address
1965-01-09	731Fondren, Houston, TX

Fname	Lname	Address
John	Smith	731 Fondren, Houston, TX
Franklin	Wong	638 Voss, Houston, TX
Ramesh	Narayan	975 Fire Oak, Humble, TX
Joyce	English	5631 Rice, Houston, TX

Pnumber	Dnum	Lname	Address	Bdate
10	4	Wallace	291Berry, Bellaire, TX	1941-06-20
30	4	Wallace	291Berry, Bellaire, TX	1941-06-20

Figurene over viser resultatet på de tre spørringene på forrige side.

Attributtene vi ønsker verdiene til spesifiseres i SELECT og kalles **projeksjonsattributter** i relasjonsalgebra. Betingelsene som må tilfredsstilles for at tupler skal hentes ut spesifiseres i WHERE og kalles **seleksjonsbetingelser** i relasjonsalgebra. Vi kan se på det som en iterator som går over alle tuplene og henter ut de oppgitte attributtene til tuplene som tilfredsstiller betingelsene. I Q1 på forrige side ser vi at den ene betingelsen er Dnumber=Dno, noe som kalles en **join betingelse** siden den kombinerer to tupler som tilfredsstiller betingelsen. Derfor må vi ha med både EMPLOYEE og DEPARTMENT i FROM klausulen.

SQL – enkle spørringer: omdøping (alias) VIKTIG

I SQL kan attributter ha samme navn, så lenge de er i ulike tabeller. Hvis en multitabell spørring refererer til to eller flere attributter med samme navn, må vi gi attributtnavnet sammen med relasjonsnavnet for å hindre tvetydigheter. Som vi kan se på figurene under blir dette gjort ved å skrive **Relasjonsnavn. attributtnavn**. Denne notasjonen kan også brukes i tilfeller der det ikke er tvetydigheter.

```
Q1A:  SELECT   Fname, EMPLOYEE.Name, Address
      FROM     EMPLOYEE, DEPARTMENT
      WHERE    DEPARTMENT.Name='Research' AND
               DEPARTMENT.Dnumber=EMPLOYEE.Dnumber;
```

Vi kan bruke alias for tabellnavn slik at vi unngår mye gjentakelse av lange navn. Som vi kan se på figuren under blir dette gjort ved å skrive **Tabellnavn AS Alias.** På figuren kan vi se at vi bruker alias for å referere til to ulike entiteter fra samme relasjon (ansatt og supervisor). I dette tilfellet må vi gi ulike alias, også kalt tuple variabler, for EMPLOYEE relasjonen. Vi kan se på dette som to ulike kopier av relasjonen, der E representerer en ansatt og S representerer en leder. E.Super_ssn=S.Ssn vil så slå sammen tuplene som tilfredsstiller denne betingelsen.

```
SELECT  E.Fname, E.Lname, S.Fname, S.Lname
FROM      EMPLOYEE AS E, EMPLOYEE AS S
WHERE E.Super_ssn=S.Ssn;
```

Uspesifisert WHERE og bruk av *

Dersom WHERE mangler betyr det at det er ingen betingelse på seleksjonen av tupler, og dermed vil alle tuplene i relasjonen gitt i FROM klausulen hentes. Hvis det er flere relasjoner i FROM klausulen vil spørringen hente kryssproduktet til relasjonene, altså alle mulige tuple kombinasjoner. For eksempel vil Q9 hente personnummer til alle ansatte, mens Q10 vil hente alle kombinasjoner av personnummer til ansatte og avdelingsnavn til en avdeling, uansett om den ansatte jobber for avdelingen eller ikke.

```
Q9:   SELECT   Ssn
      FROM     EMPLOYEE;

Q10:  SELECT   Ssn, Dname
      FROM     EMPLOYEE, DEPARTMENT;
```

For å hente alle attributtverdiene til valgte tupler, bruker vi en asterisk (*). Dermed slipper vi å skrive alle attributtnavnene. Dersom vi ønsker å hente alle attributtene i en bestemt relasjon kan vi bruke **Relasjonsnavn.***, for eksempel vil EMPLOYEE.* referere til alle attributtene i

```
Q1C:   SELECT   *
       FROM     EMPLOYEE
       WHERE    Dno=5;

Q1D:   SELECT   *
       FROM     EMPLOYEE, DEPARTMENT
       WHERE    Dname='Research' AND Dno=Dnumber;

Q10A:  SELECT   *
       FROM     EMPLOYEE, DEPARTMENT;
```

EMPLOYEE tabellen. På figuren vil Q1C hente alle attributtene til ansatte ved avdeling 5, mens Q1D vil hente alle attributtene til «Research» avdelingen og attributtene til ansatte som jobber der. Q10A vil hente kryssproduktet til EMPLOYEE og DEPARTMENT relasjonene (dvs. alle kombinasjoner).

SQL – enkle spørringer: DISTINCT VIKTIG

SQL behandler en tabell som et multisett, siden det tillater duplikate tupler (et sett har ingen duplikater). SQL vil ikke automatisk slette duplikate tupler fordi det er dyrt, aggregatfunksjoner brukes gjerne på duplikater (kap. 7) og det kan hende brukeren ønsker en løsning med duplikater.

```
Q11:   SELECT   ALL Salary
       FROM     EMPLOYEE;

Q11A:  SELECT   DISTINCT Salary
       FROM     EMPLOYEE;
```

En SQL tabell med en nøkkel må være et sett, siden nøkkelen må være unik for hver tuple. Hvis vi ønsker å eliminere duplikate tupler fra resultatet i spørringen kan vi bruke **DISTINCT** i SELECT klausulen, fordi dette betyr at kun distinkte tupler blir valgt. Vi skiller mellom:

- **SELECT ALL** ... – vil ikke eliminere duplikate tupler. Dette er standard hvis ikke noe annet oppgis.
- **SELECT DISTINCT ...** – vil eliminere duplikate tupler.

Q11

Salary
30 000
40 000
25 000
43 000
38 000
25 000
25 000
55 000

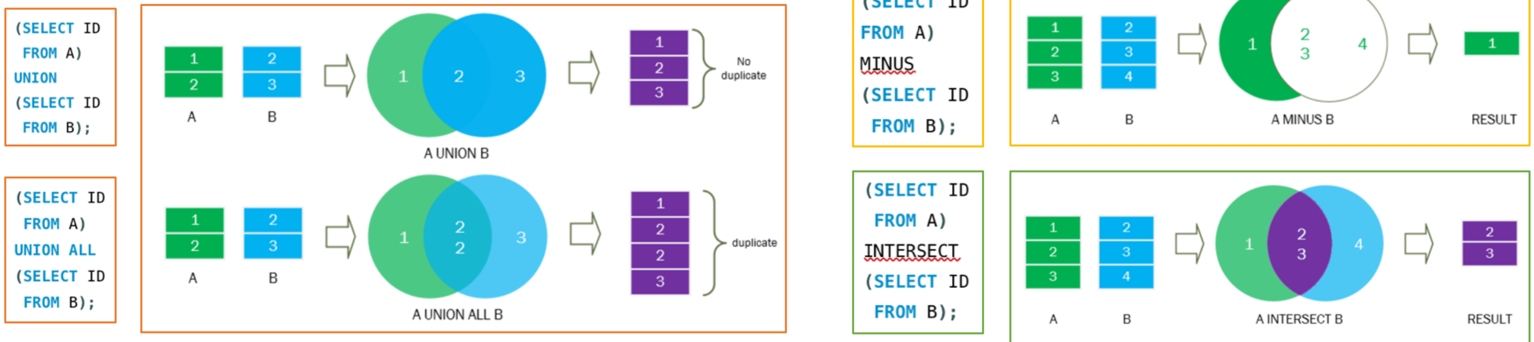
Q11A

Salary
30 000
40 000
25 000
43 000
38 000
55 000

SQL – Mengde-spørringer VIKTIG

SQL tillater flere settoperasjoner og resultatet fra disse vil være et sett av tupler, siden duplikate tupler blir eliminert. **For å kunne bruke operasjonene på to relasjoner, må de ha samme attributter i samme rekkefølge.** Operasjonene er:

- **UNION** – lar oss kombinere resultatet fra flere hentespørringer for å gi én tabell. For å ikke fjerne duplikater må vi bruke UNION ALL.
- **EXCEPT/MINUS** – regner ut differansen mellom to sett ($S_1 - S_2$), slik at resultatet vil inneholde de tuplene som er i S_1 og ikke i S_2. Brukes ofte for å sjekke om $S_1 = S_2$, fordi da vil differansen bli NULL.
- **INTERSECT** – regner ut snittet mellom to sett ($S_1 \cap S_2$), slik at resultatet vil innehold e de tuplene som er i S_1 og S_2.



Feilmeldingene skyldes at MySQL er en DBMS som ikke bruker EXCEPT/MINUS eller INTERSECT på denne måten. Se side 55 for alternativ bruk av EXCEPT.

SQL – tekstspørringer (LIKE) VIKTIG

SQL lar oss bruke LIKE operatoren for å sammenligne attributtverdien med deler av en karakterstreng, i noe som kalles streng mønstermatching. Denne metoden bruker % for å representere et vilkårlig antall (0 eller flere) karakterer og _ for å representere én karakter.

```sql
SELECT Fname, Lname
FROM EMPLOYEE
WHERE Address LIKE '%Houston, TX%'
```

```sql
SELECT Fname, Lname
FROM EMPLOYEE
WHERE Bdate LIKE '------7-'
```

Spørringen til venstre vil hente navnet til alle ansatte som har adresse i Houston, Texas. Legg merke til at % foran og etter representerer resten av adressen. Spørringen til høyre vil hente navnet til alle ansatte som er født på 70-tallet. Dette krever at 7'eren har en bestemt plass i karakterstrengen, så derfor må vi bruke _. Hvis vi skal bruke % og _ som en del av strengen, må vi bruke 'ESCAPE'. For eksempel vil 'AB_C%' skrives som 'AB_C\%'ESCAPE'\', fordi \ defineres som escape karakteren. For å inkludere en apostrofe ' bruker vi dobbeltapostrofe ''.

SQL lar oss bruke standard aritmetiske operatorer som addisjon (+), subtraksjon (-), multiplikasjon (*) og divisjon (/) på numeriske verdier eller attributter med numeriske domener. Spørringen under henter lønnen til alle ansatte som jobber med Produkt X, dersom de får en 10% lønnsøkning. Legg merke til at vi gir det resulterende lønnsattributtet et navn vha. AS i SELECT.

```sql
SELECT E.Fname, E.Lname, 1.1*E.Salary AS Increased_sal
FROM EMPLOYEE AS E, WORKS_ON AS W, PROJECT AS P
WHERE E.Ssn = W.Essn AND W.Ono = P.P.number AND P.Pname = 'ProductX'
```

Nyttige triks i SQL:

- || kan brukes for å slå sammen to strengverdier
- + og – kan brukes for å hhv. øke og minke dato, tid, timestamp og intervall datatyper.
- BETWEEN operatoren kan brukes for å lage betingelser som krever at en verdi er mellom en min og maks (se figur).

```sql
SELECT *
FROM EMPLOYEE
WHERE Dno=5 AND (Salary BETWEEN 30000 AND 40000)
```

SQL – Enkle spørringer: sortering (ORDER BY) VIKTIG

Vi bruker ORDER BY klausulen for å ordne rekkefølgen til tuplene i resultatet etter spørringen basert på verdiene til en eller flere attributter som er i resultatet. Spørringen under vil hente en liste over ansatte som er ordnet etter hvilken avdeling de jobber for og innenfor hver avdeling er det ordnet etter fornavn og deretter etternavn. Legg merke til at rekkefølgen til attributtene i ORDER BY bestemmer hva det blir sortert på først.

```sql
SELECT D.Dname, E.Lname, E.Fname, P.Pname
FROM DEPARTMENT AS D, EMPLOYEE AS E, WORKS_ON AS W, PROJECT AS P
WHERE D.Dnumber=E.Dno AND E.Ssn=W.Essn AND W.Pno=P.Pnumber
ORDER BY D.Dname, E.Lname, E.Fname
```

OBS: ORDER BY skal alltid plasseres sist, fordi den utføres til slutt

ASC brukes for å ordne etter stigende verdi (eks: 'A' over 'B'), og dette er standarden. DESC brukes for å ordne etter synkende verdi. For eksempel kan vi bruke:

```sql
ORDER BY D.Dname DESC, E.Lname ASC, E.Fname ASC
```

Oppsummering av grunnleggende hentespørring i SQL

En enkel hentespørring i SQL kan bestå av opptil fire klausuler, der de to første (SELECT og FROM) er obligatoriske. SELECT gir attributtene som skal hentes, mens FROM gir relasjonene som attributtene hentes fra. Andre klausuler som kan være med er WHERE, ORDER BY, GROUP BY og HAVING.

6.4 INSERT, DELETE og UPDATE i SQL

I SQL blir INSERT INTO, DELETE og UPDATE brukt for å endre på databasen.

INSERT INTO

Den enkleste formen for INSERT INTO brukes for å sette en tuple inn i en relasjon, altså legge til en rad i tabellen. Vi må gi relasjonsnavnet og en liste av verdier for tuplen. Verdiene må gis i samme orden som korresponderende attributter ble spesifisert i CREATE TABLE kommandoen. Figuren til høyre viser hvordan vi kan legge til en ny ansatt i EMPLOYEE relasjonen.

```
INSERT INTO EMPLOYEE
VALUES ('Richard', 'K', 'Marini', '653298653', '1962-12-30',
    '98 Oak Forest, Katy, TX', 'M', 37000, '653298653', 4);
```

```
INSERT INTO EMPLOYEE(Fname, Lname, Dno, Ssn)
VALUES ('Richard', 'Marini', 4, '653298653');
```

På figuren til venstre ser vi en INSERT form som lar brukeren **spesifisere hvilke attributter de gitte verdiene tilhører**. Dette er nyttig for relasjoner som har mange attributter der få blir tildelt verdi i en ny tuple. Attributter som kan være NULL eller har spesifisert en standardverdi, kan utelukkes. Verdiene må gis med samme rekkefølge som attributtene spesifisert etter INSERT INTO kommandoen. Hvis en innsetting bryter integritetsbegrensninger vil DBMS avslå operasjonen, for eksempel hvis verdien til en primærnøkkel ikke er oppgitt (kan ikke være NULL).

```
CREATE TABLE WORKS_ON_INFO
(Emp_name VARCHAR(15),
Proj_name VARCHAR(15),
Hours_per_week DECIMAL(3,1));

INSERT INTO WORKS_ON_INFO(Emp_name, Proj_name, Hours_per_week)
SELECT E.Lname, P.Pname, W.Hours
FROM EMPLOYEE as A, PROJECT as P, WORKS_ON as W
WHERE P.Pnumber=W.Pno AND W.Essn=E.Ssn
```

Figuren til høyre viser en variasjon av INSERT der vi **lager en relasjon og laster den med resultatet av en spørring**. Vi lager en midlertidig tabell WORKS_ON_INFO som skal ha tre attributter. Deretter har vi en SELECT spørring som vil hente etternavnet til den ansatte, prosjektnavnet og arbeidstimer. Siden vi begynner med en INSERT INTO kommando med spesifiserte attributter vil disse verdiene bli satt inn i tabellen. Videre kan vi lage spørringer for WORKS_ON_INFO tabellen og når vi ikke trenger den lenger, kan vi fjerne den vha DROP TABLE kommandoen. Informasjonen i denne tabellen kan være utdatert.

De fleste DBMS har bulklasteverktøy, som lar brukeren laste formatert data fra en fil inn i en tabell uten å må skrive et stort antall INSERT kommandoer. Figuren viser hvordan vi kan lage en relasjon som har samme form som en annen relasjon og fylles med dataen som oppfyller et krav. I dette tilfellet blir D5EMPS lastet med alle ansattetupler ved avdeling 5. LIKE gjør at det blir laget en lignende relasjon, mens WITH DATA gjør at den fylles med spesifisert data.

```
CREATE TABLE D5EMPS LIKE EMPLOYEE
(SELECT E.*
FROM EMPLOYEE AS E
WHERE E.Dno=5)WITH DATA;
```

DELETE

DELETE kommandoen vil fjerne tupler fra en relasjon, og WHERE klausulen brukes for å spesifisere tuplen som skal slettes. En kommando vil kun slette tupler fra én relasjon om gangen, men hvis referensiell trigget handlinger er spesifisert kan det føre til at slettingen forplanter seg til andre relasjoner. Basert på det som er gitt i WHERE klausulen kan én DELETE kommando slette null, en eller flere tupler om gangen. Hvis ikke noe er oppgitt vil alle tuplene slettes og relasjonen vil være en tom tabell i databasen. For å fjerne tabellen helt må vi bruke DROP TABLE.

```
DELETE FROM EMPLOYEE
WHERE Lname = 'Brown';
DELETE FROM EMPLOYEE
WHERE Lname = Dno=5;
DELETE FROM EMPLOYEE
```

```
UPDATE PROJECT
SET Plocation='Bellaire', Dnum=5
WHERE Pnumber=10;

UPDATE EMPLOYEE
SET Salary=Salary*1.1
WHERE Dno=5;
```

UPDATE

UPDATE kommandoen brukes for å endre attributtverdier til en eller flere valgte tupler, og WHERE klausulen brukes for å velge tuplene som skal endres. En kommando vil kun endre tupler i en relasjon om gangen, men hvis en primærnøkkel blir endret kan dette forplantes til andre relasjoner. En SET klausul brukes for å gi attributtene som skal endres og deres nye verdi. Den nye verdien kan også være NULL eller DEFAULT.

Oppsummering – Kapittel 6 (F8)

SQL er ikke like mengdeorientert som relasjonsalgebra, så resultattabeller kan ha like tupler. Hvis vi ønsker at resultatet ikke skal inneholde duplikater, må vi be SQL om å fjerne disse.

SQL som DBMS språk

SQL er et DBMS språk, og som vi så på side 9 er det en kombinasjon av DDL (Data Definition Language), VDL (View Definition Language) og DML (Data Manipulation Language).

DDL (Data Definition Language)

DDL brukes for å definere skjema i databasen, altså hvilke tabeller som skal lages og hvilke attributter disse tabellene skal ha. I SQL blir dette gjort vha kommandoene:

- **CREATE TABLE** (**SCHEMA/DOMAIN**) – lager skjemaet
- **ALTER TABLE** – kan brukes for å endre skjemaet underveis, for eksempel legge til eller fjerne kolonne.
- **DROP TABLE** (**SCHEMA/DOMAIN**)– sletter skjemaet

```
CREATE TABLE Person (
   Pnr    INTEGER NOT NULL,
   Navn   VARCHAR(20),
   CONSTRAINT Person_PK PRIMARY KEY (Pnr));

CREATE TABLE Hund (
   Regnr    INTEGER NOT NULL,
   Navn     VARCHAR(30),
   Faar     INTEGER,
   Rase     VARCHAR(30),
   EierPnr INTEGER NOT NULL,
   CONSTRAINT Hund_PK PRIMARY KEY (Regnr),
   CONSTRAINT Hund_FK FOREIGN KEY (EierPnr) REFERENCES Person(Pnr)
                                            ON UPDATE CASCADE
                                            ON DELETE NO ACTION,

   CONSTRAINT Alderssjekk CHECK (Faar > 1980));

CREATE TABLE BittAv (
   Pnr    INTEGER NOT NULL,
   Regnr  INTEGER NOT NULL,
   Antall INTEGER DEFAULT 0,
   CONSTRAINT BittAv_PK PRIMARY Key (Pnr, Regnr),
   CONSTRAINT BittAV_FK1 FOREIGN KEY (Pnr) REFERENCES Person(Pnr)
                                           ON UPDATE CASCADE
                                           ON DELETE CASCADE,
   CONSTRAINT BittAV_FK2 FOREIGN KEY (Regnr) REFERENCES Hund(Regnr)
                                             ON UPDATE CASCADE
                                             ON DELETE CASCADE,
   CONSTRAINT AntallSjekk CHECK (Antall >= 0));
```

Figuren viser et eksempel på hvordan vi kan lage skjemaene i hundedatabasen. En viktig del av DDL er **definering av restriksjoner**, slik som primærnøkkel, fremmednøkkel og egendefinerte restriksjoner (vha CHECK, se Aldersjekk på figur).

Fremmednøkkelrestriksjon

I en **fremmednøkkelrestriksjon** må man oppgi attributtet som skal være fremmednøkkel, samt gi relasjonen og attributtet i denne relasjonen som det refereres til. Man må til slutt definere hva som skjer når attributtet det refereres til blir oppdatert og slettet. Det er flere muligheter for hva som skjer med fremmednøkkelen (FK) når primærnøkkelen det refereres til blir endret:

- **UPDATE** – FK blir endret til noe annet
- **DELETE** – FK blir slettet
- **NO ACTION/RESTRICT** – endringene blir avslått
- **SET NULL** – FK blir satt lik NULL
- **CASCADE** – FK blir endret til samme verdi som primærnøkkelen ble endret til

```
update:
    update Person
    set Navn = 'William'
    where Navn = 'Bill'

delete:
    delete from Hund
    where Navn = 'Tarzan'

select:
    select Navn
    from Hund
    where Rase = 'Puddel'
```

DML (Data Manipulation Language)

DML brukes for å manipulere databasen, altså for å sette inn, slette eller endre data. I SQL blir dette gjort vha kommandoene:

- **INSERT** – sette inn tupler
- **UPDATE** – endre på attributter i tupler.
- **DELETE** – fjerne tupler
- **SELECT** – spørringer for å hente data

Merk at vi kan endre, slette og hente flere tupler samtidig. Dersom vi bruker primærnøkkelen i WHERE klausulen vil vi sikre at maks én tuple blir manipulert.

DML – spørringer

```
SELECT < attributt-liste >
FROM < tabell-liste >
WHERE < logisk betingelse >
```

SELECT er ikke det samme som seleksjon i relasjonsalgebra, fordi den kan seleksjon, men også mye annet. Den består i minimum av:

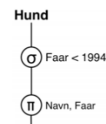
- **SELECT** – hva som hentes. Denne lager «skjemaet» til resultattabellen.
- **FROM** – hvor det hentes fra. Denne sier hvilke tabeller som er involvert og sammenstillingen av disse (eks: JOIN vil slå sammen flere tabeller)

Setningen kan også bygges ut med flere deler, for eksempel WHERE som gir en betingelse som alle rader i resultattabellen må oppfylle. Husk at SQL ikke eliminerer duplikater. Figuren viser et eksempel med både SQL og relasjonsalgebra (mer senere).

Hund-tabellen

REGNR	NAVN	FAAR	RASE	EIER_PNR
1	King	1992	Rottweiler	1
2	Tarzan	1993	Puddel	3
3	Troll	1995	Collie	7
4	King	1995	Schæfer	6
5	Varg	1995	Newfoundland	1
6	Prins	1994	Schæfer	1
7	King	1996	Puddel	4
8	King	1993	Rottweiler	4
9	Tarzan	1995	Doberman	3
10	Troll	1996	Puddel	1

```
SELECT Navn, Faar
FROM Hund
WHERE Faar < 1994
```

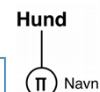


Navn	Faar
King	1992
Tarzan	1993
King	1993

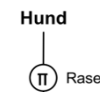
Viktige fremgangsmåter:

- **SELECT DISTINCT** fjerner duplikater fra resultattabellen. Merk: SQL fjerner ikke duplikater uten DISTINCT, mens relasjonsalgebra fjerner duplikater automatisk.
- **ORDER BY** brukes for å ordne tuplene i resultattabellen i en bestemt rekkefølge basert på det gitte attributtet. ASC gir stigende orden, mens DESC gir synkende. Denne klausulen er alltid til *slutt* i SELECT-setningen.

```
SELECT DISTINCT Navn
FROM Hund
```




```
SELECT DISTINCT Rase
FROM Hund
ORDER BY Rase ASC
```



SELECT DISTINCT Navn, Rase
FROM Hund

RegNr	Navn	Rase
1	Rex	Puddel
2	Rex	Collie
3	Rex	Collie
4	King	Collie

→

Navn	Rase
Rex	Puddel
Rex	Collie
King	Collie

På figuren kan vi se at SELECT DISTINCT vil slette duplikater, men hele tuplen må være identisk for at det skal være en duplikat. Som vi kan se kan Navn være lik for flere tupler, så lenge det er ulik Rase (og motsatt). For at resultatet ikke skal inneholde hunder med like navn, må vi ha kun SELECT DISTINCT Navn.

Spørring om data fra flere tabeller

Hvis vi ønsker å **hente data fra flere tabeller, må vi slå sammen disse tabellene** og deretter hente dataen fra én samlet tabell som har alle de ønskede attributtene (og mer). To måter:

1. **Klassisk SQL** – tabellene tas inn i FROM og vi legger til en join-betingelse i WHERE. Tupler fra de ulike tabellene vil slås sammen hvis de oppfyller denne betingelsen. På figuren vil det dannes en ny tabell med tupler der Eier_Pnr = Pnr. Fra denne tabellen vil SELECT hente de gitte attributtene.

```
SELECT RegNr, Hund.Navn, Person.Navn
FROM Hund, Person
WHERE Eier_Pnr = Pnr
```

Merk: INNER JOIN og JOIN gjør det samme

```
SELECT RegNr, Hund.Navn, Person.Navn
FROM Hund JOIN Person ON Eier_Pnr = Pnr
```

2. **Moderne SQL** (bruk denne!) – vi slår sammen tabellene i FROM klausulen vha JOIN. Legg merke til at vi må oppgi hva sammenslåingen skal baseres på etter ON.

Moderne SQL spesifiserer altså sammenstillingen i FROM-delen, og det finnes flere JOIN-typer:

- **JOIN/INNER JOIN** – filtrerende join basert på attributter etter ON
- **NATURAL JOIN** – join med implisitt join-betingelse (like attributtnavn i tabellene)
- **LEFT OUTER JOIN**
- **RIGHT OUTER JOIN**
- **FULL OUTER JOIN**
- **CROSS JOIN** – vi ønsker et kartesisk produkt.

Uten betingelse vil alle join generere et kartesisk produkt, som vil si at alle mulige sammenslåinger blir inkludert i den nye tabellen. JOIN betingelsen kan skrives på flere måter:

- **NATURAL JOIN** – sammenslåing av tupler som har lik verdi for attributter med samme navn i tabellene. Eks: Hund og Bittav har REGNR, så tupler vil slås sammen basert på disse.
- **ON <logiske betingelse>** - sammenslåing av tupler som oppfyller betingelsen
- **USING(<attributtliste>)** – sammenslåing av tupler som har alle attributtene i listen

Eksempler

Figuren til høyre viser relasjonene i databasen.

Person-tabellen

PNR	NAVN
1	Olav
2	Kari
3	Anne
4	Lisbeth
5	Harald
6	Liv
7	Trude
8	Per
9	Kristin
10	Christina
11	Petter
12	Liv
13	Merete

Bittav-tabellen

PNR	REGNR	ANTALL
2	2	5
3	9	1
4	9	1
5	2	3
5	4	2
6	9	2
8	5	2
9	4	4
11	5	1
12	4	3

Hund-tabellen

REGNR	NAVN	FAAR	RASE	EIER_PNR
1	King	1992	Rottweiler	1
2	Tarzan	1993	Puddel	3
3	Troll	1995	Collie	7
4	King	1995	Schæfer	6
5	Varg	1995	Newfoundland	1
6	Prins	1994	Schæfer	1
7	King	1996	Puddel	4
8	King	1993	Rottweiler	4
9	Tarzan	1995	Doberman	3
10	Troll	1996	Puddel	1

Q1: RegNr og navn for pudler som har bitt noen

```
SELECT Hund.RegNr, Navn
FROM Hund INNER JOIN BittAv ON
     (Hund.RegNr = BittAv.RegNr)
WHERE Rase = 'Puddel'
```

Alternativ med naturlig join

```
SELECT Hund.RegNr, Navn
FROM Hund NATURAL JOIN BittAv
WHERE Rase = 'Puddel'
```

RegNr	Navn
2	Tarzan

Merk: Rase = 'Puddel' krever 100% likhet. Dersom vi vil gi noe rom for ulikhet kan vi bruke LIKE. Da vil % representere 0-n tegn, mens _ er ett tegn. For eksempel kan vi velge alle raser som starter på P ved å bruke Rase LIKE 'P%'.

Q2: Offers navn, gjerningshundens navn og eiers navn, ordnet på offers navn

```
SELECT O.Navn AS Offer, H.Navn AS Gjerningshund,
       E.Navn AS Eier
FROM ((Person AS O INNER JOIN BittAV AS BA
      ON (O.Pnr = BA.Pnr))
      INNER JOIN Hund AS H ON (BA.RegNr = H.RegNr))
     INNER JOIN Person AS E ON (E.Pnr = H.Eier_Pnr)
ORDER BY O.Navn ASC
```

Offer	Gjerningshund	Eier
Anne	Tarzan	Anne
Harald	Tarzan	Anne
Harald	King	Liv
...	...	...

Merk: her deltar Person tabellen to ganger i sammenslåingen, fordi den inngår i to ulike «roller» (eier og offer). I dette tilfellet må vi gi tabellen alias, for å skille mellom de to ulike «rollene». Legg også merke til at vi gir attributtene i resultatet navn ved å bruke alias i SELECT klausulen.

Q3: RegNr, hundenavn, eiers Pnr og eiers Navn for hunder som har bitt sin eier

```
SELECT H.RegNr, H.Navn, E.Navn
FROM  Hund AS H JOIN BittAv AS BA ON (H.RegNr = BA.RegNr)
      JOIN Person AS E on (H.EierPnr = E.Pnr)
WHERE BA.Pnr = E.Pnr
```

RegNr	Navn	Navn
9	Tarzan	Anne

Kapittel 7 – Mer kompleks SQL

Dette kapittelet ser på mer avanserte SQL setninger.

7.1 Mer komplekse SQL hentespørringer

I kapittel 6 har vi sett på grunnleggende hentespørringer, og vi skal nå se at SQL også lar brukeren spesifisere mer komplekse hentinger fra databasen.

Sammenligninger med NULL og tre-verdi logikk

SQL har flere regler for å håndtere NULL verdier. NULL brukes for å representere en manglende verdi, og har ulike tolkninger (s. 33):

- ***value unknown*** – verdien eksisterer, men er ikke kjent eller er det ikke kjent om verdien eksisterer. For eksempel kan telefonnummeret til en person være ukjent.
- ***value not available*** – verdien eksisterer, men er med vilje holdt tilbake. For eksempel kan personen ha hustelefon, men ønsker ikke at nummeret skal være listet.
- ***value not applicable*** – attributtet er udefinert for denne tuplen. For eksempel kan personen ikke ha noen hustelefon, og dermed ingen telefonnummer.

SQL skiller ikke mellom de ulike tolkningene, fordi det er ofte ikke mulig. **Hver NULL verdi blir sett på som ulik andre NULL verdier**. Når en tuple med NULL i en av attributtene er involvert i en sammenligning, vil resultatet bli UNKOWN (kan være true og false). **SQL bruker en tre-verdi logikk med verdier TRUE, FALSE og UNKNOWN**, istedenfor enkel boolean logikk. Tabellen viser resultatet for AND, OR og NOT for denne logikken. Legg merke til at NOT UNKOWN = UNKOWN. Dette er resultatet av sammenligningsbetingelser og vil som regel være i WHERE klausulen. Den generelle regelen i hentespørringer er at kombinasjonen av tupler som gir TRUE blir hentet.

(a)

AND	TRUE	FALSE	UNKNOWN
TRUE	TRUE	FALSE	UNKNOWN
FALSE	FALSE	FALSE	FALSE
UNKNOWN	UNKNOWN	FALSE	UNKNOWN

(b)

OR	TRUE	FALSE	UNKNOWN
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	UNKNOWN
UNKNOWN	TRUE	UNKNOWN	UNKNOWN

(c)

NOT	
TRUE	FALSE
FALSE	TRUE
UNKNOWN	UNKNOWN

For å sjekke om en attributtverdi er NULL kan vi bruke IS NULL eller IS NOT NULL (se figur). Siden hver NULL er ulik ,vil ikke ekvivalenssammenligning kunne brukes (=). Hvis en join betingelse er spesifisert vil ikke tupler med NULL verdi for join attributtene bli inkludert i resultatet.

```
SELECT Fname, Lname
FROM EMPLOYEE
WHERE Super_ssn IS NULL;
```

SQL – Nøstede spørringer VIKTIG

Nøstede spørringer er fullstendige select-from-where blokker som blir plassert i en annen SQL spørring, og de henter verdier som kan brukes i en sammenligningsbetingelse. Den andre spørringen kalles **ytre spørring**. De nøstet spørringene vil være i SELECT, FROM, WHERE eller en annen klausul. Figuren viser et eksempel på nøstet spørringer. Operatoren **IN** brukes for å sammenligne verdien v med et sett av verdier V, og vil returnere TRUE dersom v er en av elementene i V.

```
SELECT DISTINCT Pnumber
FROM PROJECT
WHERE Pnumber IN
      (SELECT Pnumber
      FROM PROJECT, DEPARTMENT, EMPLOYEE
      WHERE Dnum = Dnumber AND Mgr_ssn = SSn AND Lname = 'Smith')

      OR
      Pnumber in
      (SELECT Pno
      FROM WORKS_ON, EMPLOYEE
      WHERE Essn = Ssn AND Lname = 'Smith');
```

Første nøstet spørring henter prosjektnummeret til prosjektene som har hatt Smith som manager, mens andre nøstet spørring vil hente prosjektnummer til prosjektene der Smith har vært ansatt. Den ytre spørringen bruker OR for å hente alle prosjekttupler der prosjektnummeret er i resultatet til en av de nøstede spørringene.

```sql
SELECT DISTINCT Essn
FROM WORKS_ON
WHERE (Pno, Hours) IN (SELECT Pno, Hours
                       FROM WORKS_ON
                       WHERE Essn = '123456789');
```

For å sammenligne flere verdier samtidig må vi plassere de i en tuple før IN (se figur). Denne spørringen vil hente personnummeret til alle ansatte som har samme (prosjekt, time) kombinasjon som den ansatte med personnummer 123456789, altså alle som har arbeidet like lenge på samme prosjekt. Her vil (Pno, Hours) i alle WORKS_ON tupler sammenlignes med settet av tupler som produseres av den nøstede spørringen.

Det er flere operatorer som kan brukes for sammenligning:

- **IN** – returnerer TRUE hvis v er en av elementene i V
- **NOT IN** – returnerer TRUE hvis v ikke er en av elementene i V
- **... ANY** – kan kombineres med =, >, >=, <, <= og <>. For eksempel vil =ANY returnere TRUE hvis v er lik en av elementene i V (samme som IN), mens >ANY vil returnere TRUE hvis v er større enn en av elementene i V.
- **... ALL** – kan kombineres med =, >, >=, <, <= og <>. For eksempel vil =ALL returnere TRUE hvis v er lik alle elementene i V, mens <>ALL vil returnere TRUE hvis v er ulik alle elementene i V. Spørringen på figuren henter navnet til ansatte som har større lønn enn de ved avdeling 5.

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

Ved nøstede spørringer må vi bruke alias i tilfeller der det brukes attributter med samme navn fra ulike relasjoner. Hvis en attributt ikke har prefiks vil regelen være at den refererer til relasjonen deklarert i FROM til den innerste nøstede spørringen. Det beste er å bruke alias for å sikre at riktige attributter blir brukt. Spørringen på figuren henter navnet til alle ansatte som har en kontaktperson med samme fornavn og kjønn. Hvis vi ikke hadde brukt aliasene ville begge sex-attributtene i den nøstede spørringen referert til sex i DEPENDENT.

```sql
SELECT E.Fname, E.Lname
FROM EMPLOYEE AS E
WHERE E.Ssn IN (SELECT D.Essn
                FROM DEPENDENT AS D
                WHERE E.Fname = D.Dpenedent_name AND E.Sex = D.Sex);
```

Korrelerte nøstede spørringer

Nøstet og ytre spørring er korrelerte når WHERE klausulen til den nøstede spørringen refererer til en attributt i en relasjon som blir deklarert i den ytre spørringen. Den nøstede spørringen blir evaluert en gang for hver tuple i den ytre spørringen. Spørringen over til venstre er et eksempel på en korrelert nøstet spørring. For hver EMPLOYEE tuple vil den nøstede spørringen hente Essn verdien til alle DEPENDENT tupler som har samme kjønn og fornavn som denne EMPLOYEE tuplen. Hvis Ssn verdien til denne EMPLOYEE tuplen er i resultatet til den nøstede spørringen, vil navnet til denne tuplen hentes.

SQL – Nøstede spørringer: EXISTS og UNIQUE VIKTIG

EXISTS og UNIQUE er boolean funksjoner som returnerer TRUE eller FALSE og brukes i WHERE klausulen.

EXISTS brukes for å sjekke om resultatet til en nøstet spørring er tomt, **og den returnerer TRUE dersom resultatet inneholder minst en tuple.** For hver EMPLOYEE tuple på figuren vil den nøstede spørringen hente alle DEPENDENT tupler med samme personnummer, kjønn og fornavn. Hvis dette resultatet ikke er tomt vil navnet til EMPLOYEE tuplen returneres.

```sql
SELECT E.Fname, E.Lname
FROM EMPLOYEE AS E
WHERE EXISTS (SELECT *
              FROM DEPENDENT AS D
              WHERE E.Ssn = D.Essn AND E.sex = D.sex AND E.Fname = D.Dependent_name);
```

Merk: EXISTS returnerer TRUE når det eksisterer minst én tuple i resultatet til nøstet spørring, mens NOT EXISTS returnerer TRUE når resultatet er tomt.

```
SELECT E.Fname, E.Lname
FROM EMPLOYEE AS E
WHERE EXISTS (SELECT *
              FROM DEPENDENT AS D1
              WHERE E.SSn = D1.Essn)
      AND
      EXISTS (SELECT *
              FROM DEPARTMENT AS D2
              WHERE E.SSn = D2.Mgr_ssn)
```

Spørringen på figuren henter navnet til alle managerne som har minst en dependent. For at EMPLOYEE tuplen skal bli hentet må den ha en dependent og den må være en manager, fordi den må eksistere i resultatet til begge nøstet spørringene.

```
SELECT E.Fname, E.Lname
FROM EMPLOYEE AS E
WHERE NOT EXISTS ((SELECT P.Pnumber
                   FROM PROJECT AS P
                   WHERE P.Dnum = 5)
                   EXCEPT (SELECT W.Pno
                           FROM WORKS_ON AS W
                           WHERE E.Ssn = W.Essn));
```

Dersom vi ønsker å hente alle prosjektene som utføres på avdeling 5 og deretter sjekke om en ansatt jobber på alle disse prosjektene, kan vi bruke EXCEPT kombinert med NOT EXISTS (se figur). Den første nøstede spørringen vil hente alle prosjektnumrene ved avdeling 5 og den andre vil hente prosjektnumrene som denne EMPLOYEE tuplen deltar i. Når vi bruker EXCEPT vil vi ta det ene settet minus det andre, og dersom dette er tomt betyr det at EMPLOYEE tuplen arbeider på alle prosjektene i avdeling 5. NOT EXISTS vil returnere TRUE (settet er tomt) og EMPLOYEE tuplen blir hentet. Figuren under viser en alternativ fremgangsmåte. Den vil hente EMPLOYEE tupler der det ikke eksisterer et prosjekt i avdeling 5 som den ansatte ikke arbeider ved.

```
SELECT E.Fname, E.Lname
FROM EMPLOYEE AS E
WHERE NOT EXISTS (SELECT *
                  FROM WORKS_ON AS W1
                  WHERE (W1.Pno IN (SELECT P.Pnumer
                                    FROM PROJECT AS P
                                    WHERE P.Dnum = 5)
                  AND
                  NOT EXISTS (SELECT *
                              FROM WORKS_ON AS W2
                              WHERE W2.Essn = E.Ssn AND W2.Pno = W1.Pno)));
```

UNIQUE returnerer TRUE dersom det ikke er noen duplikate tupler i resultatet til en spørring. Kan brukes for å teste om resultatet til en nøstet spørring er et sett (ingen duplikater).

Eksplisitte sett og nye navn

SQL lar oss bruke et eksplisitt sett med verdier i WHERE klausulen, og dette settet må skrives i parentes. Spørringen på figuren vil hente alle ulike personnummer til ansatte ved prosjekt 1, 2 og 3.

```
SELECT DISTINCT Essn
FROM WORKS_ON
WHERE Pno IN (1, 2, 3);
```

```
SELECT E.LName AS Employee_name, S.Lname AS Supervisor_name
FROM EMPLOYEE AS E, EMPLOYEE AS S
WHERE E.Super_ssn = S.Ssn
```

Attributter i SELECT klausulen kan gis nye navn vha AS operatoren etterfulgt av det nye navnet (se figur). De nye navnene vil representere kolonneheadere i resultattabellen.

SQL – JOIN spørringer VIKTIG

JOIN ble introdusert til SQL for å la brukerne sammenslå tabeller i FROM klausulen istedenfor WHERE. Når vi bruker JOIN vil vi lage en enkelt sammenslått tabell, som inneholder alle attributtene fra begge relasjonene. Vi bruker ON for å gi join betingelsen, og på figuren ser vi at tupler som slås sammen må oppfylle Dnumber = Dno.

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

SELECT Fname, Lname, Address
FROM EMPLOYEE JOIN DEPARTMENT ON Dnumber = Dno
WHERE Dname = 'Research';
```

I en NATURAL JOIN av to relasjoner S og R, vil ikke join betingelsen være spesifisert. I stedet vil det lages en ekvivalensbetingelse for hvert attributtpar med samme navn fra S og R. Hvert attributtpar blir inkludert kun én gang i den resulterende relasjonen. Vi kan gi attributter nye navn for at de skal matche i en NATURAL JOIN. Vi bruker AS for å gi relasjonen og dens attributter nye navn i FROM klausulen. På figuren endrer vi attributtnavnet til Dno, slik at join betingelsen til NATURAL JOIN skal bli EMPLOYEE.Dno = DEPT.Dno.

```
SELECT Fname, Lname, Address
FROM (EMPLOYEE NATURAL JOIN (DEPARTMENT AS DEPT (Dname, Dno, Mssn, Msdate)))
WHERE Dname = 'Research';
```

Det finnes flere typer JOIN:

- **INNER JOIN (JOIN)** – resultattabellen vil kun inneholde tupler som har matchet fra de to relasjonene, altså oppfylt join betingelsen.
- **OUTER JOIN** – resultattabellen vil inneholde alle tupler fra en eller begge relasjonene. Hvis join betingelsen ikke er oppfylt vil attributtene til den andre relasjonen settes til NULL. Det finnes flere varianter:
  - **LEFT OUTER JOIN** – alle tupler i venstre tabell må inkluderes i resultatet.
  - **RIGHT OUTER JOIN** – alle tupler i høyre tabell må inkluderes i resultatet.
  - **FULL OUTER JOIN** – alle tupler i begge tabellen må inkluderes i resultatet.
- **CROSS JOIN** – genererer alle kombinasjoner av tupler fra de to relasjonene

EMPLOYEE

Ssn	Fname	Lname	Super_Ssn
123123	Per	Olsen	NULL
456456	Kari	Bremnes	123123
789789	Ola	Bakken	NULL

```sql
SELECT E.LName AS Employee_name, S.Lname AS Supervisor_name
FROM EMPLOYEE AS E INNER JOIN EMPLOYEE AS S ON E.Super_ssn = S.Ssn;
```

INNER JOIN:

E.Ssn	E.Fname	E.Lname	E.Super_Ssn = S.Ssn	S.Fname	S.Lname	S.Super_Ssn
456456	Kari	Bremnes	123123	Per	Olsen	NULL

```sql
SELECT E.LName AS Employee_name, S.Lname AS Supervisor_name
FROM EMPLOYEE AS E LEFT OUTER JOIN EMPLOYEE AS S ON E.Super_ssn = S.Ssn;
```

LEFT OUTER JOIN:

E.Ssn	E.Fname	E.Lname	E.Super_Ssn = S.Ssn	S.Fname	S.Lname	S.Super_Ssn
123123	Per	Olsen	NULL	NULL	NULL	NULL
456456	Kari	Bremnes	123123	Per	Olsen	NULL
789789	Ola	Bakken	NULL	NULL	NULL	NULL

Vi kan også **nøste JOIN spesifikasjoner**, altså slå sammen tre eller flere tabeller, noe som kalles en **multiway join** (se figur).

```sql
SELECT Pnumber, Dnum, Lname, Address, Bdate
FROM PROJECT JOIN DEPARTMENT ON Dnum = Dnumber JOIN EMPLOYEE ON Mgr_ssn = SSn
WHERE Plocation = 'Stafford';
```

SQL – Gruppering: aggregatfunksjoner VIKTIG

Aggregatfunksjoner brukes for å summere informasjon fra flere tupler i en enkel-tuple summering (dvs. en tabell med en rad). Noen aggregatfunksjoner er:

- **COUNT** – gir antall tupler eller verdier som spesifiseres i spørringen
- **SUM** – gir summen til et sett av numeriske verdier
- **MAX** – gir største verdi i et sett av numeriske verdier
- **MIN** – gir minste verdi i et sett av numeriske verdier
- **AVG** – gir gjennomsnittet til et sett av numeriske verdier
- **SOME** og **ALL** – brukes på boolean verdier. SOME returnerer TRUE hvis minst ett element i samlingen er TRUE, mens ALL returnerer TRUE hvis alle elementene er TRUE.

Disse kan brukes i SELECT eller HAVING (mer senere). MAX og MIN kan også brukes med attributter som ikke er numeriske, men har en total ordning (eks: Time og Date).

```sql
SELECT SUM(Salary), MAX(Salary), Min(Salary), AVG(Salary)
FROM EMPLOYEE;

SELECT SUM(Salary) AS Total_Sal, MAX(Salary) AS Highest_Sal
FROM EMPLOYEE;

SELECT SUM(Salary), MAX(Salary), Min(Salary), AVG(Salary)
FROM EMPLOYEE
WHERE Dname = 'Research';

SELECT SUM(Salary), MAX(Salary), Min(Salary), AVG(Salary)
FROM EMPLOYEE JOIN DEPARTMENT ON Dno = Dnumber
WHERE Dname = 'Research';
```

Den første spørringen vil returnere en enkelt summerende rad for alle EMPLOYEE tuplene. Den andre spørringen illustrerer at vi kan gi kolonnene i denne raden nye navn. Den tredje spørringen viser at vi kan legge til en betingelse i WHERE klausulen, mens den fjerde spørringen viser at vi kan bruke JOIN.

Her kan vi se at COUNT kan brukes for å telle antall tupler i EMPLOYEE tabellen eller antall ansatte i Research avdelingen. (*) refererer til radene i tabellen, så COUNT(*) vil gi antall rader.

```
SELECT COUNT(*)
FROM EMPLOYEE;

SELECT COUNT(*)
FROM EMPLOYEE JOIN DEPARTMENT ON Dno = Dnumber
WHERE Dname = 'Research';
```

```
SELECT COUNT(DISTINCT Salary)
FROM EMPLOYEE;
```

Denne spørringen vil returnere antall ulike lønnsverdier i EMPLOYEE. Hvis vi bruker COUNT(Salary) vil ikke duplikate verdier bli eliminert. I begge tilfeller vil NULL-verdier ignoreres, fordi **aggregatfunksjoner overser NULL-verdier når de brukes på enkeltattributter.** Ved COUNT(*) vil NULL-verdier inkluderes, fordi den teller antall tupler og ikke antall verdier. Når en aggregatfunksjon brukes på en samling av verdier vil den fjerne NULL-verdiene før utregningen. Hvis samlingen blir tom vil funksjonen returnere NULL (COUNT returnerer 0).

Aggregatfunksjoner kan også brukes på nøstede spørringer. Spørringen på figuren vil hente navnet til ansatte som har mer enn en dependent. Legg merke til at betingelsen i WHERE er at `COUNT(*) >= 2`

```
SELECT Fname, Lname
FROM EMPLOYEE
WHERE (SELECT COUNT(*)
       FROM DEPENDENT
       WHERE Ssn = Essn) >= 2
```

SQL – Gruppering: GROUP BY og HAVING VIKTIG

I mange tilfeller ønsker vi å bruke aggregatfunksjoner på grupper av tupler i en relasjon, der gruppen er basert på noen attributtverdier. For eksempel kan vi ønske å finne gjennomsnittslønn til ansatte ved hver avdeling. I slike tilfeller må vi dele relasjonen inn i ikke-overlappende grupper, som vi så kan bruke aggregatfunksjonen på. **Hver gruppe består av tupler som har samme verdi for grupperende attributt(er), som blir spesifisert i GROUP BY klausulen.** Grupperende attributt(er) bør også gis i SELECT, slik at resultatet til aggregatfunksjonen gis sammen med verdien til grupperende attributt(er). **En separat gruppe lages for tuplene som har NULL i grupperende attributt(er).** Spørringen på figuren vil gruppere ansatte basert på hvilken avdeling de arbeider ved, og den vil for hver av gruppene telle antall ansatte og regne ut gjennomsnittslønnen.

EMPLOYEE databasen

Ssn	Fname	Lname	Dno	Salary
123123	Ola	Bakken	1	50000
134134	Bente	Bakken	2	60000
234234	Kari	Bakken	1	40000
345345	Per	Olsen	1	43000
456456	Unni	Storm	3	32000
567567	Lise	Hansen	3	21000
678678	Ina	Perkli	2	94000
789789	Fredrik	Trondvik	2	87000
890890	Veronika	Bakken	1	56000

```
SELECT Dno, COUNT(*), AVG(Salary)
FROM EMPLOYEE
GROUP BY Dno;
```

Dno	COUNT(*)	AVG(Salary)
1	4	47250.0000
2	3	80333.3333
3	2	26500.0000

```
SELECT Pnumber, Pname, COUNT(*)
FROM PROJECT JOIN WORKS_ON ON Pnumber = Pno
GROUP BY Pnumber, Pname;
```

Denne spørringen viser hvordan vi kan kombinere JOIN og GROUP BY. I dette tilfellet vil de to relasjonene slås sammen til en tabell og deretter vil grupperingen og aggregatfunksjonen brukes. Spørringen vil telle antall ansatte som jobber på prosjekt med samme nummer og samme navn.

HAVING klausulen brukes for å hente verdien til aggregatfunksjoner på grupper som tilfredsstiller bestemte betingelser. HAVING kan brukes på gruppene av tupler som lages av GROUP BY, slik at kun gruppene som tilfredsstiller betingelsen blir en del av resultatet til spørringen. Figuren viser spørringen som grupperer ansatte basert på avdelingen de arbeider ved, men i dette tilfellet blir HAVING brukt for å kun inkludere avdelinger som har 3 eller flere ansatte (dvs. avdeling 3 blir ikke inkludert).

```
SELECT Dno, COUNT(*), AVG(Salary)
FROM EMPLOYEE
GROUP BY Dno
HAVING COUNT(*) >= 3;
```

Dno	COUNT(*)	AVG(Salary)
1	4	47250.0000
2	3	80333.3333

Både WHERE og HAVING brukes for å begrense utvalget, men de fungerer på ulike måter! WHERE vil begrense hvilke tupler som aggregatfunksjonen brukes på, mens HAVING vil begrense hele grupper. Det er viktig å huske at **WHERE blir utført først i det vanlige oppsettet**! Vi ønsker å lage grupper for avdelinger som har 3 eller flere ansatte, og for hver av disse vil vi hente antall ansatte som har lønn større enn 45000. Spørringen på figuren til høyre er feil, fordi WHERE klausulen vil utføres først og dermed fjerne alle ansatte som har lønn mindre enn 45000 før grupperingen. Dermed vil flere avdelinger som egentlig skal være med bli droppet av HAVING. Vi ønsker å gruppere først og deretter hente de som oppfyller WHERE betingelsen, og dette kan vi gjøre vha en nøstet spørring (figur til venstre).

```
SELECT Dno, COUNT(*)
FROM EMPLOYEE
WHERE Salary > 45000
GROUP BY Dno
HAVING COUNT(*) >= 3;
```

Dno	COUNT(*)
2	3

```
SELECT Dno, COUNT(*)
FROM EMPLOYEE
WHERE Salary > 45000 AND Dno IN (SELECT Dno
                                 FROM EMPLOYEE
                                 GROUP BY Dno
                                 HAVING COUNT(*) >= 3)
GROUP BY Dno;
```

Dno	COUNT(*)
1	2
2	3

WITH og CASE klausuler

WITH klausulen lar brukeren definere en tabell som kun brukes i en bestemt spørring, og ligner dermed en view som brukes i en spørring og blir deretter droppet (mer senere). For eksempel kan spørringen over lages med WITH istedenfor den nøstede spørringen. Her vil vi lage en midlertidig tabell med grupperingen som vi så bruker i resten av spørringen. Feilmeldingen kan skyldes at WITH ikke godtas av alle SQL-baserte DBMS. Når spørringen er utført vil BIGSALARY tabellen droppes.

```
WITH BIGSALARY(Dno) AS (SELECT Dno
                        FROM EMPLOYEE
                        GROUP BY Dno
                        HAVING COUNT(*) >= 3)
SELECT Dno, COUNT(*)
FROM EMPLOYEE
WHERE Salary > 45000 AND Dno IN BIGSALARY
GROUP BY Dno;
```

```
UPDATE EMPLOYEE
SET Salary =
CASE    WHEN Dno = 3 THEN Salary + 2000
        WHEN Dno = 2 THEN Salary + 1500
        WHEN Dno = 1 THEN Salary + 3000
        ELSE Salary + 0
        END;
```

CASE klausulen brukes når verdier kan være ulike basert på bestemte betingelser. Dette kan brukes i enhver del av SQL der en verdi forventes, inkludert spørring, innsetting og oppdatering av tupler. På figuren ser vi hvordan vi kan gi ansatte ulike lønnsøkninger basert på hvilken avdeling de arbeider ved.

Rekursive spørringer

Et rekursivt forhold er et forhold mellom tupler fra samme relasjon, for eksempel forholdet mellom ansatt og veileder som begge er EMPLOYEE tupler. Dette forholdet blir gitt vha fremmednøkkelen Super_ssn som kan koble to EMPLOYEE tupler sammen eller være NULL. Et eksempel på en rekursiv operasjon er å hente alle ansatte som en veileder har ansvar for, både direkte og indirekte (dvs. veileder for en veileder for en veileder, osv.). I spørringen under definerer vi en view kalt SUP_EMP som holder resultatet til den rekursive spørringen. Den første delen av den rekursive spørringen kalles **basespørringen** og den vil hente veileder-veiledet forholdet ved dette nivået. UNION brukes for å koble basespørringen til etterfølgende nivå som representeres av den andre delen. Når vi kommer til den andre delen vil spørringen gjentas for neste nivå (= rekursiv spørring). Dette gjentas helt til vi når et fast punkt der ingen flere tupler blir lagt til. Ved dette punktet vil SUP_EMP være fylt med personnummer til veileder og veiledet i alle nivåer av veileder-veiledet forhold.

SUP_EMP vil være tom i begynnelsen. Deretter vil den fylles med det første nivået av veiledere og veiledet ansatte. Videre vil den fylles med andre nivå av veileder-veiledet. Denne prosessen gjentas helt til den når nivået der det er ingen veiledere. Deretter vil spørringen hente alle tuplene i SUP_EMP.

```
WITH RECURSIVE SUP_EMP(SupSsn, EmpSsn) AS (SELECT SupervisorSsn, Ssn
                                           FROM EMPLOYEE
                                           UNION
                                           SELECT E.Ssn, S.SupSsn
                                           FROM EMPLOYEE AS E JOIN SUP_EMP AS S ON E.SupervisorSsn = S.EmpSsn)
SELECT *
FROM SUP_EMP;
```

7.3 Views (virtuelle tabeller) i SQL

En view er en tabell som utledes fra andre tabeller, som kan være basetabeller eller tidligere definerte views. En view blir sett på som en **virtuell tabell**, som vil si at den som regel ikke har tupler som er fysisk lagret i databasen, slik som i en basetabell. Dette begrenser oppdateringen av views, men gir ingen begrensning på spørringen. **Views blir brukt for å spesifisere tabeller som vi ofte refererer til, men som kanskje ikke fysisk eksisterer.** For eksempel ved COMPANY databasen kan det hende vi ofte ønsker å hente navnet til den ansatte og prosjektet som den ansatte arbeider ved. I stedet for å spesifisere join for EMPLOYEE, WORKS_ON og PROJECT for hver spørring, kan vi definere en view som representerer resultatet av denne sammenslåingen. Deretter kan vi lage enkelt-tabell spørringer på dette viewet. EMPLOYEE, WORKS_ON og PROJECT kalles **definerende tabeller** for dette viewet.

Spesifisering av views i SQL

CREATE VIEW brukes for å spesifisere en view i SQL. Viewet får et tabellnavn, en liste av attributtnavn og en spørring som spesifiserer innholdet. Hvis ingen av attributtene er resultatet av funksjoner eller aritmetiske operasjoner, trenger vi ikke å gi nye attributtnavn, siden attributtene i viewet vil være de samme som i de definerende tabellene.

```
CREATE VIEW WORKS_ON1
AS SELECT Fname, Lname, Pname, Hours
FROM EMPLOYEE JOIN PROJECT ON Pno = Pnumber JOIN WORKS_ON on Ssn = Essn;
```

WORKS_ON1

FNAME	LNAME	PNAME	HOURS

```
CREATE VIEW DEP_INFO(Dep_name, No_of_emps, Total_sal)
AS SELECT Dname, COUNT(*), SUM(Salary)
FROM DEPARTMENT JOIN EMPLOYEE ON Dnumber = Dno
GROUP BY Dname;
```

DEPT_INFO

DEPT_NAME	NO_OF_EMPS	TOTAL_SAL

På figurene over kan vi se at WORKS_ON1 bruker attributtnavnene til de definerende tabellene, mens DEP_INFO har aggregatfunksjoner og må dermed definere nye attributtnavn (merk: disse må ha samme rekkefølge som attributtene i SELECT). **Spørringer på et view blir laget på samme måte som for basetabeller.** Figuren viser en spørring der vi henter fornavn og etternavn til ansatte som arbeider på ProductX. Uten view hadde denne spørringen krevd to JOIN.

```
SELECT Fname, Lname
FROM WORKS_ON1
WHERE Pname = 'ProductX';
```

Views brukes for å forenkle bestemte spørringer eller som mekanismer i sikkerhet og autorisering. En view skal alltid være oppdatert, så hvis vi endrer tupler i definerende tabeller må viewet automatisk reflektere disse endringene. Viewet må derfor realiseres når vi spesifiserer en spørring på viewet, og dette er ansvaret til DBMS og ikke brukeren. Vi bruker DROP VIEW for å fjerne et view som vi ikke trenger lenger.

View implementasjon , view oppdatering og in-line views

Det er to hovedtilnærminger for hvordan en DBMS kan implementere en view for spørring:

1. **Spørring modifisering** – view spørringen blir endret til en spørring på underliggende basetabeller. Dette blir gjort av DBMS og figuren viser et eksempel. Ulempen med denne tilnærmingen er at det er ineffektivt for views som er definert via komplekse spørringer som er tidkrevende å utføre.

```
SELECT Fname, Lname
FROM WORKS_ON1
WHERE Pname = 'ProductX';
```

DBMS

```
SELECT Fname, Lname, Pname, Hours
FROM EMPLOYEE JOIN PROJECT ON Pno = Pnumber JOIN WORKS_ON on Ssn = Essn
WHERE Pname = 'ProductX';
```

2. **View materialisering** – det lages en midlertidig eller permanent, fysisk view tabell når viewet blir laget eller brukt i en spørring. Denne blir tatt vare på med utgangspunkt i at den kan brukes i etterfølgende spørringer. DBMS må i dette tilfellet ha en effektiv teknikk for å oppdatere view tabellen når basetabellene endres, for eksempel inkrementell oppdatering. Hvis viewet ikke blir brukt i en spørring ila en bestemt tid vil den automatisk fjernes av systemet. Det er ulike strategier for når viewet skal oppdateres:
   a. **Umiddelbar oppdatering** – viewet blir oppdatert med en gang basetabellene endres
   b. **Lat oppdatering** – viewet oppdateres når det trengs i en spørring
   c. **Periodisk oppdatering** – viewet blir oppdatert ved jevne mellomrom (kan føre til views som ikke er oppdatert ved spørring!)

Som regel kan man ikke bruke INSERT, DELETE eller UPDATE på en view tabell. Noen observasjoner for oppdatering av views:

- En view kan oppdateres hvis den har én definerende tabell og viewet inneholder primærnøkkelen og alle attributter med NOT NULL begrensning uten spesifisert standardverdi.
- Views som er definert på flere tabeller kan generelt ikke oppdateres
- Views som er definert vha gruppering og aggregatfunksjoner kan generelt ikke oppdateres

Dersom brukeren skal kunne endre viewet vha INSERT, DELETE og UPDATE må vi legge til WITH CHECK OPTION i enden av view definisjonen. Dette lar systemet avslå operasjoner som bryter SQL reglene for oppdateringer. Dersom vi definerer view tabellen i FROM klausulen kalles det et **in-line view**, siden viewet defineres inni spørringen.

View som autoriserende mekanisme

Views kan brukes for å gjemme bestemte attributter eller tupler fra uautoriserte brukere.
Vi kan gi tilgang til bestemte tupler ved å lage en view tabell som vi kun fyller med tupler som oppfyller den gitte WHERE betingelsen. For eksempel kan vi ha en bruker som kun har lov til å se informasjon hos ansatte som arbeider ved avdeling 5. Da kan vi lage viewet DEPT5EMP og gi brukeren tillatelse til å lage spørringer for dette viewet, men ikke til basetabellen EMPLOYEE.

```
CREATE VIEW DEPT5EMP AS
SELECT *
FROM EMPLOYEE
WHERE Dno = 5;
```

```
CREATE VIEW EMP_DATA AS
SELECT Fname, Lname, Address
FROM EMPLOYEE;
```

Vi kan gi tilgang til bestemte attributter ved å lage en view tabell som vi kun fyller med attributtene gitt i SELECT. For eksempel kan vi ha en bruker som kun har lov til å se fornavn, etternavn og adresse til de ansatte. Vi lager EMP_DATA og gir brukeren tilgang til dette viewet.

Ved å lage passende view og gi bestemte brukere tilgang til viewet og ikke basetabellen, har vi begrenset deres tilgang til databasen.

7.4 Endring av skjemaet

Vi skal nå se på DDL (Data Definition Language) delen av SQL som innebærer kommandoer som definerer skjemaet i databasen (s. 49). Skjema evolusjonskommandoer brukes for å endre et skjema ved å legge til eller fjerne tabeller, attributter, begrensninger eller andre skjemaelementer.

DROP kommandoen

DROP kommandoen brukes for å droppe navngitte skjemaelementer, som tabeller, domener, typer eller begrensninger. Vi bruker **DROP SCHEMA** for å fjerne hele skjemaet (eks: COMPANY) og **DROP TABLE** for å fjerne en bestemt tabell (eks: DEPENDENT). Det er to typer dropp oppførsel:

1. **CASCADE – vil fjerne alt.** DROP SCHEMA … CASCADE vil fjerne hele skjemaet og alle dens tabeller, domener og andre elementer, mens DROP TABLE … CASCADE vil fjerne hele tabellen og alle referanser til denne tabellen.
2. **RESTRICT – vil kun fjerne dersom de er tomme**. DROP SCHEMA … RESTRICT vil fjerne skjemaet hvis det ikke har noen elementer, mens DROP TABLE … RESTRICT vil fjerne tabellen hvis den ikke blir referert til i noen begrensninger (eks: fremmednøkkel), views eller andre elementer.

Legg merke til at DROP TABLE vil slette alle tuplene og tabelldefinisjonene fra katalogen. Hvis vi ønsker å ta vare på tabelldefinisjonene må vi bruker DELTE.

ALTER kommandoen

ALTER kommandoen kan brukes for å endre tabeller eller andre navngitte skjemaelementer underveis. Når vi skal endre på en tabell bruker vi ALTER TABLE etterfulgt av skjema.relasjon. Deretter kan vi bruke en av følgende ALTER kommandoer:

1. **Legge til en kolonne (attributt)** – vi bruker ADD COLUMN etterfulgt av det nye attributtnavnet. Vi må deretter legge til en verdi for det nye attributtet for hver individuelle tuple, enten ved å spesifisere en standard klausul eller vha UPDATE for hver tuple. Hvis ingen standard er spesifisert vil det nye attributtet settes til NULL når kommandoen utføres, så NOT NULL begrensning er ikke tillatt i dette tilfellet.
2. **Droppe en kolonne (attributt)** – vi bruker DROP COLUMN etterfulgt av attributtet som skal fjernes. Vi må velge mellom CASCADE (alle referanser fjernes) eller RESTRICT (blir kun utført hvis det er ingen referanser).
3. **Endre kolonnedefinisjon** – vi bruker ALTER COLUMN etterfulgt av attributtet som skal endres. For å fjerne standarden til dette attributtet bruker vi DROP DEFAULT, mens for å lage en ny standard bruker vi SET DEFAULT etterfulgt av ny standardverdi.
4. **Endre begrensninger** – vi bruker DROP CONSTRAINT … for å fjerne den navngitte *(!)* begrensningen og ADD CONSTRAINT … for å legge til en ny begrensning.

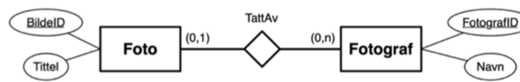
```sql
ALTER TABLE COMPANY.EMPLOYEE
ADD COLUMN Job VARCHAR(12);

ALTER TABLE COMPANY.EMPLOYEE
DROP COLUMN Address CASCADE;

ALTER TABLE COMPANY.DEPARTMENT
ALTER COLUMN Mgr_ssn DROP DEFAULT;

ALTER TABLE COMPANY.DEPARTMENT
ALTER COLUMN Mgr_ssn SET DEFAULT '333445555';

ALTER TABLE COMPANY.EMPLOYEE
DROP CONSTRAINT EMPSUPERFK CASCADE;
```



Foto

BildeID	Tittel	FotografID
1	Geilo	1
2	Beach	2
3	Korgen	1
4	Nidaros	NULL
5	Central Park	2

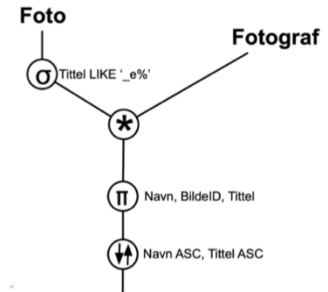
Fotograf

FotografID	Navn
1	Wilse
2	Erwitt
3	Leibovitz

Oppsummering – Kapittel 7 (F9)

Q4: Navn, BildeID og Tittel for fotografier som har «e» som andre tegn i Tittel

Vi ønsker å hente attributter fra Foto og Fotograf og må derfor slå i sammen disse. Siden begge har FotografID kan vi bruke en naturlig join. Vi bruker LIKE med ′_e%′ for å hente tuplene med Tittel som har «e» i posisjon to. Vi sorterer etter Navn og deretter Tittel ved å bruke ORDER BY der vi oppgir Navn først og deretter Tittel.

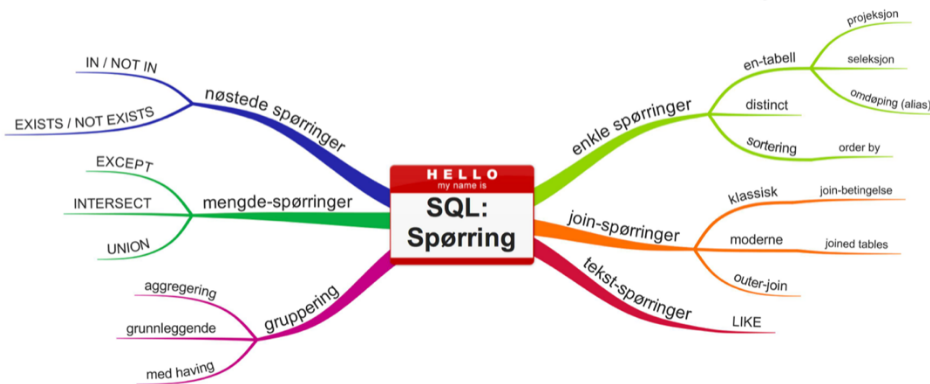



```
SELECT Navn, BildeID, Tittel
FROM Foto NATURAL INNER JOIN Fotograf
WHERE Tittel LIKE '_e%'
ORDER BY Navn ASC, Tittel ASC
```

Navn	BildeID	Tittel
Erwitt	2	Beach
Erwitt	5	Central Park
Wilse	1	Geilo

Spørringer i SQL

Figuren under viser en oversikt over spørringer i SQL.



Nøstede spørringer

Nøstede spørringer er fullstendige select-from-where blokker som plasseres i en annen spørring.

IN/NOT IN

Vi bruker operatoren IN eller NOT IN når vi ønsker å sjekke om en attributtverdi er i ett verdisett. For eksempel kan vi sjekke om en hund har bitt noen eller ikke (se figur). Disse spørringene vil se på RegNr til hver Hund tuple og sammenligne denne med RegNr til hunder som har bitt noen (resultatet av indre spørring). IN returnerer TRUE hvis RegNr er i settet, mens NOT IN returnerer TRUE hvis RegNr ikke er i settet.

```
/*Henter hunder som har bitt noen*/
SELECT *
FROM Hund
WHERE RegNr IN (SELECT RegNr
                FROM BittAv);

/*Henter hunder som ikke har bitt noen*/
SELECT *
FROM Hund
WHERE RegNr NOT IN (SELECT RegNr
                    FROM BittAv);
```

EXISTS/NOT EXISTS

Vi bruker operatoren EXISTS eller NOT EXISTS for å sjekke om resultatet av indre spørring er tomt eller ikke (dvs. inneholder tupler). For eksempel kan vi sjekke om en hund har bitt sin eier eller ikke (se figur). Disse spørringene vil for hver hund se på resultatet fra den indre spørringen, som vil være tom dersom denne hunden ikke har bitt sin eier. EXISTS returnerer TRUE hvis den indre spørringen ikke er tom, mens NOT EXISTS returnerer TRUE hvis den er tom. Dette er en **korrelert delspørring**, siden vi sender inn data fra den ytre spørringen, noe som gjør at den indre spørringen må utføres en gang for hver rad i Hund tabellen.

```
/*Henter hunder som har bitt sin egen eier*/
SELECT *
FROM Hund AS H
WHERE EXISTS (SELECT *
              FROM BittAv AS BA
              WHERE BA.RegNr = H.RegNr AND Ba.Pnr = H.Eier_Pnr);

/*Henter hunder som ikke har bitt sin egen eier*/
SELECT *
FROM Hund AS H
WHERE NOT EXISTS (SELECT *
                  FROM BittAv AS BA
                  WHERE BA.RegNr = H.RegNr AND Ba.Pnr = H.Eier_Pnr);
```

Korrelert delspørring er mye dyrere enn ikke-korrelert, fordi den krever at den indre spørringen må gjennomføres en gang for hver rad i den ytre tabellen.

Q5 Pnr og navn for personer som ikke eier hunder

```
SELECT Pnr, Navn
FROM Person
WHERE Pnr NOT IN (SELECT Eier_PNR
                  FROM Hund);
```

Pnr	Navn
1	King
1	Varg
1	Prins
1	Troll
3	Tarzan
3	Tarzan
4	King
4	King
6	King
7	Troll

Person-tabellen

PNR	NAVN
1	Olav
2	Kari
3	Anne
4	Lisbeth
5	Harald
6	Liv
7	Trude
8	Per
9	Kristin
10	Christina
11	Petter
12	Liv
13	Merete

Bittav-tabellen

PNR	REGNR	ANTALL
2	2	5
3	9	1
4	9	1
5	2	3
5	4	2
6	9	2
8	5	2
9	4	4
11	5	1
12	4	3

Hund-tabellen

REGNR	NAVN	FAAR	RASE	EIER_PNR
1	King	1992	Rottweiler	1
2	Tarzan	1993	Puddel	3
3	Troll	1995	Collie	7
4	King	1995	Schæfer	6
5	Varg	1995	Newfoundland	1
6	Prins	1994	Schæfer	1
7	King	1996	Puddel	4
8	King	1993	Rottweiler	4
9	Tarzan	1995	Doberman	3
10	Troll	1996	Puddel	1

Legg merke til at Eier_Pnr i Hund er fremmednøkkelen som refererer til Pnr i Person. Ved å sjekke om Pnr er lik en av fremmednøklene vil vi dermed sjekke om personen eier en hund.

Mengeoperatorer

Mengdeoperatorer **fjerner duplikater** og vi har tre ulike typer:

- **UNION** – union. Bruker UNION ALL for å lage multisett (dvs. inkludere duplikater).
- **INTERSECT** – snitt. Bruker INTERSECT ALL for å lage multisett
- **EXCEPT** – minus. Bruker EXCEPT ALL for å lage multisett

```
select < attributtliste (*) >
from < tabell-spesifikasjon >
where …

union | intersect | except

select < attributtliste (*) >
from < tabell-spesifikasjon >
where …
```

(*) union-kompatible

Det er viktig at settene er kompatible, altså at de har samme attributter i samme rekkefølge. Figuren til høyre viser hvordan disse operatorene brukes, og som vi kan se blir de plassert mellom to SELECT-FROM-WHERE blokker.

Q6: Pnr og navn for personer som har blitt bitt av minst en puddel og som eier pudler

Vi bruker INTERSECT for å finne personer som både eier puddel og har blitt bitt av puddel vha snittet til to SELECT setninger. Hver SELECT setning vil gi en resulterende tabell, og i dette tilfellet vil snittet være tomt siden tabellene ikke inneholder noen felles tupler. Det finnes altså ingen personer som både eier og har blitt bitt av en puddel.

```
SELECT P.Pnr, P.Navn
FROM Person AS P
     INNER JOIN Hund AS H ON (P.Pnr = H.EierPnr)
WHERE H.Rase = 'Puddel'

INTERSECT

SELECT P.Pnr, P.Navn
FROM Person AS P
INNER JOIN BittAv AS BA ON ( P.Pnr = BA.Pnr )
INNER JOIN Hund AS H ON ( H.Regnr = BA.Regnr )
WHERE H.Rase = 'Puddel'
```

Pnr	Navn
3	Anne
4	Lisbeth
1	Olav

Pnr	Navn
2	Kari
5	Harald

```
SELECT P.Pnr, P.Navn
FROM Person AS P
     INNER JOIN Hund AS H ON (P.Pnr = H.EierPnr)
WHERE H.Rase = 'Puddel'
  AND P.Pnr IN (SELECT BA.Pnr
                FROM BittAv AS BA
                INNER JOIN Hund AS H ON(BA.Regnr = H.Regnr)
                WHERE H.Rase = 'Puddel')
```

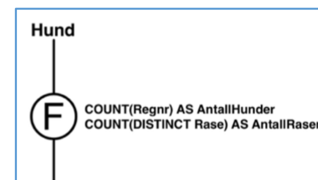
Siden MySQL ikke støtter INTERSECT enda, må vi finne alternative løsninger og ofte vha nøstede spørringer. På figuren vil ytre spørring finne alle puddeleierne og hente Pnr og Navn hvis de har blitt bitt av en puddel. Vi kunne også gjort det motsatt vei.

Innebygde funksjoner

Aggregatfunksjoner brukes for å summere informasjon fra flere tupler, og inkluderer COUNT, SUM, MIN, MAX, AVG, osv. For eksempel kan vi bruke COUNT(∗) for å telle antall hunder og antall ulike rader. Legg merke til at **DISTINCT gjør at funksjonen opererer på unike verdier**. Aggregatfunksjonene vil ignorere NULL-verdier, bortsett fra COUNT(∗) som teller antall rader selv om rader har attributter som er NULL.

```
SELECT COUNT(*) AS AntallHunder,
       COUNT(DISTINCT Rase) AS AntallRaser
FROM Hund
```

AntallHunder	AntallRaser
10	6



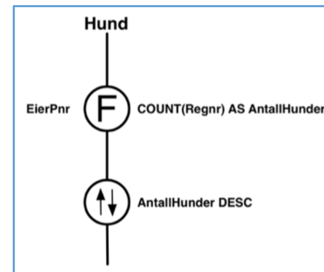
Gruppering

Aggregering er et nyttig verktøy, spesielt når vi kan gruppere tuplene før vi summerer dem. For å partisjonere tabellen inn i grupper av tupler bruker vi GROUP BY⟨ attributtliste ⟩. Her vil attributtliste være attributtene som definerer partisjoneringen, altså avgjør hvilke tupler som kommer i ulike grupper.

Q7: Antall hunder per eier, de med flest hunder først

```
SELECT EierPnr, COUNT(Regnr) AS AntallHunder
FROM Hund
GROUP BY EierPnr
ORDER BY AntallHunder DESC
```

EierPnr	AntallHunder
1	4
3	2
4	2
7	1
6	1



Legg merke til at vi bruker AS i SELECT for å gi navn til antall hunder i resultattabellen. Når vi har med GROUP BY EierPnr vil COUNT(Regnr) telle antall ulike Regnr for gruppene av tupler med samme EierPnr. Hvis vi ikke hadde laget gruppene, ville totalt antall hunder blitt telt. De som ikke eier noen hunder blir ikke med, siden COUNT av en attributt ignorerer NULL-verdier.

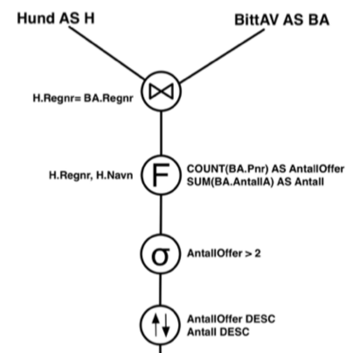
HAVING

HAVING⟨betingelse⟩ brukes for å gi en betingelse <u>etter</u> grupperingen, slik at grupper som ikke oppfyller betingelsen droppes. **WHERE brukes for å begrense rader, mens HAVING brukes for å begrense grupper**

Q8: Regnr, navn, antall offer og sum av Antall for hunder som har bitt mer enn to personer

```
SELECT H.Regnr, Navn, COUNT(*) AS AntallOffer,
       SUM(Antall) AS Antall
FROM Hund AS H INNER JOIN BittAv AS BA ON (H.Regnr = BA.Regnr)
GROUP BY H.Regnr, Navn
HAVING AntallOffer > 2
ORDER BY AntallOffer DESC, Antall DESC
```

Regnr	Navn	AntallOffer	Antall
4	King	3	9
9	Tarzan	3	4



Her vil vi gruppere basert på Regnr og navn, altså vil hver gruppe inneholde en Hund tuple. Det er kun grupper der AntallOffer er større enn to som blir inkludert og gruppene blir sortert først etter synkende AntallOffer og deretter etter synkende Antall. Dersom vi for eksempel kun ønsker å inkludere rasen pudler, kan vi legge til en WHERE $H.\text{rase} = \text{'puddel'}$.

Views – virtuelle tabeller

I SQL kan man lage **virtuelle tabeller som kalles views som er avledet fra andre basetabeller**. Views brukes for å forenkle spørringer (en-tabell spørring og slipper å gjenta JOIN) og for å oppnå større sikkerhet (kan gi brukere tilgang til viewet, men ikke basetabeller). DBMS kan implementere views på to ulike måter: *query modification* (spørring på view omdannes til spørring på basetabell) og *view materialization* (view blir fysisk lagret). Views kan bedre ytelsen veldig, men utfordringen er å holde de oppdatert, siden endring i basetabeller må føre til tilsvarende endringer i views. Flere tabeller og aggregering gir utfordringer i oppdateringen.

```
CREATE VIEW Hundeeier(Pnr, Navn, Hund)
AS SELECT Pnr, P.NAvn, H.Navn
   FROM Person AS P INNER JOIN Hund AS H ON (P.Pnr = H.EierPnr)
```

Pnr	Navn	Hund
1	Olav	King
3	Anne	Tarzan
7	Trude	Troll
6	Liv	King
1	Olav	Varg
1	Olav	Prins
4	Lisbeth	King
4	Lisbeth	King
3	Anne	Tarzan
1	Olav	Troll

Kapittel 8 – Relasjonsalgebra

Relasjonsmodellen har to formelle språk: **relasjonsalgebra** og **relasjonskalkulus**. I dette emnet er det kun relasjonsalgebra som er pensum. SQL er i sammenligning et praktisk språk, som ble utviklet etter relasjonsalgebraen. I kapittel 2 så vi at en datamodell må inkludere et sett av operatorer for å manipulere databasen og konsepter for å definere strukturen og begrensninger til databasen. Relasjonsalgebra representerer settet av operatorer som lar brukeren utføre grunnleggende hentespørringer. Resultatet av en spørring vil være en ny relasjon, som kan videre manipuleres. En sekvens av relasjonsalgebraoperatorer danner et **relasjonsalgebra uttrykk**, som gir en ny relasjon som representerer resultatet til en databasespørring.

Relasjonsalgebra er viktig fordi det gir et formelt grunnlag for operatorer i relasjonsmodellen og det brukes som et grunnlag for å implementere og optimere spørringer i spørrebehandling og optimeringsmoduler som er en integrert del i RDBMSs (relasjonell DBMS, kapittel 18). Noen konsepter i relasjonsalgebra er også blitt en del av SQL språket hos RDBMS. De fleste RDBMS gir ikke brukergrensesnitt for relasjonsalgebra spørringer, men kjerneoperasjoner og funksjoner i interne moduler i de fleste relasjonssystemer er basert på operatorer i relasjonsalgebra.

Operatorene i relasjonsalgebra kan deles inn i to grupper:

- **Mengeoperatorer** – disse kan brukes på relasjoner, siden hver relasjon er definert som et sett med tupler. Det inkluderer union, snitt, mengdedifferanse og kartesisk produkt
- **Spesielle operatorer –** dette er operatorer som ble utviklet spesifikt for relasjonsdatabaser. Det inkluderer seleksjon, projeksjon, join-operatorer, osv. SELECT og PROJECT er ensartet (*unary*) operatorer, siden de opererer på en tabell, mens JOIN er binære operatorer, siden den opererer på to tabeller.

For å gi videre funksjonalitet ble **aggregatfunksjoner** laget, og disse kan summere data. Det ble også utviklet flere typer JOIN og UNION operatorer, for eksempel OUTER JOIN og OUTER UNION. Dette er operatorer som ble lagt til fordi de er viktige for mange databaseapplikasjoner.

8.1 Ensartet relasjonsoperatorer – SELECT og PROJECT

SELECT operatoren

SELECT operasjonen brukes på en relasjon for å velge et subsett av tupler som tilfredsstiller seleksjonsbetingelsen. Det kan ses på som et filter som kun slipper gjennom tupler som tilfredsstiller en betingelse. For eksempel kan vi velge ut EMPLOYEE tupler som arbeider ved avdeling 4:

$$\sigma_{\text{Dno}=4}(\text{EMPLOYEE})$$

Eller vi kan velge ut EMPLOYEE tupler som tjener mer enn 30 000:

$$\sigma_{\text{Salary}>30000}(\text{EMPLOYEE})$$

Generelt vil SELECT operatoren ha formen:

$$\sigma_{\langle\text{Seleksjonsbetingelse}\rangle}(\text{Relasjon})$$

Symbolet σ (sigma) brukes for å betegne SELECT operatoren og seleksjonsbetingelsen er et boolean uttrykk for attributtene til relasjonen R. **Resultatet til SELECT vil være en ny relasjon med samme attributter som *R*.** Seleksjonsbetingelsen kan sammenligne en attributt med en verdi eller en annen attributt. For å gjøre dette brukes som regel en av operatorene $=, >, \geq, <, \leq, \neq$. Flere betingelser kan kobles sammen av AND, OR eller NOT, for eksempel kan vi velge ut EMPLOYEE tupler som enten arbeider ved avdeling 4 og tjener mer enn 25 000 eller arbeider ved avdeling 5 og tjener mer enn 30 000:

$$\sigma_{(\text{Dno}=4 \text{ AND Salary}>25000) \text{ OR } (\text{Dno}=5 \text{ AND Salary}>30000)}(\text{EMPLOYEE})$$

Figuren under viser resultatet av seleksjonen på databasen til høyre

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5

EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Alicia	J	Zelaya	999887777	1968-01-19	3321 Castle, Spring, TX	F	25000	987654321	4
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5
Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000	987654321	4
James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000	NULL	1

DEPARTMENT

Dname	Dnumber	Mgr_ssn	Mgr_start_date
Research	5	333445555	1988-05-22
Administration	4	987654321	1995-01-01
Headquarters	1	888665555	1981-06-19

DEPT_LOCATIONS

Dnumber	Dlocation
1	Houston
4	Stafford
5	Bellaire
5	Sugarland
5	Houston

WORKS_ON

Essn	Pno	Hours
123456789	1	32.5
123456789	2	7.5
666884444	3	40.0
453453453	1	20.0
453453453	2	20.0
333445555	2	10.0
333445555	3	10.0
333445555	10	10.0
333445555	20	10.0
999887777	30	30.0
999887777	10	10.0
987987987	10	35.0
987987987	30	5.0
987654321	30	20.0
987654321	20	15.0
888665555	20	NULL

PROJECT

Pname	Pnumber	Plocation	Dnum
ProductX	1	Bellaire	5
ProductY	2	Sugarland	5
ProductZ	3	Houston	5
Computerization	10	Stafford	4
Reorganization	20	Houston	1
Newbenefits	30	Stafford	4

DEPENDENT

Essn	Dependent_name	Sex	Bdate	Relationship
333445555	Alice	F	1986-04-05	Daughter
333445555	Theodore	M	1983-10-25	Son
333445555	Joy	F	1958-05-03	Spouse
987654321	Abner	M	1942-02-28	Spouse
123456789	Michael	M	1988-01-04	Son
123456789	Alice	F	1988-12-30	Daughter
123456789	Elizabeth	F	1967-05-05	Spouse

Vi kan bruke $=, >, \geq, <, \leq, \neq$ når domenet er ordnet, for eksempel nummer, dato eller strenger (alfabetisk). Hvis domenet ikke har noe orden (eks: Color) kan vi kun bruke $=$ og $\neq$. Noen domener har også flere typer sammenligningsoperatorer, for eksempel for strenger kan vi bruke A SUBSTRING_OF B. **Seleksjonsbetingelsen blir uavhengig påført hver individuelle tuple i relasjonen.** Attributtet til hver tuple blir sjekket opp mot betingelsen og dersom den gir TRUE blir tuplen valgt ut. SELECT operatoren er ensartet, som vil si at den **brukes på én tabell om gangen**. Seleksjonen blir brukt på en individuell tuple, så derfor **kan ikke betingelsen involvere mer enn én tuple**. Resulterende tabell etter seleksjonen vil ha samme attributter som relasjonen (dvs. graden er den samme) og like mange eller færre tupler. Selektiviteten til seleksjonsbetingelsen er fraksjonen av tupler som blir valgt ut.

SELECT operatoren er kumulativ, som vil si at rekkefølgen til en sekvens av seleksjoner er vilkårlig.

$$\sigma_{\langle bet1 \rangle}\left(\sigma_{\langle bet2 \rangle}(R)\right) = \sigma_{\langle bet2 \rangle}\left(\sigma_{\langle bet1 \rangle}(R)\right)$$

En sekvens kan også slås sammen til en enkelt seleksjon vha AND:

$$\sigma_{\langle bet1 \rangle}\left(\sigma_{\langle bet2 \rangle}(R)\right) = \sigma_{\langle bet1 \rangle \text{ AND } \langle bet2 \rangle}(R)$$

I SQL vil seleksjonsbetingelsen som regel gis i WHERE klausulen:

$$\sigma_{\text{Dno}=4 \text{ AND Salary}>25000}(\text{EMPLOYEE})$$

```
SELECT *
FROM EMPLOYEE
WHERE Dno = 4 AND Salary > 25000;
```

PROJECT operatoren

SELECT operatoren velger ut rader som skal inkluderes i resultatet, mens **PROJECT operatoren velger ut kolonner (dvs. attributter) som skal inkluderes i resultatet**. Hvis vi er interessert i enkelte attributter i relasjonen bruker vi PROJECT for å hente ut disse. Det kan ses på som en vertikal partisjonering inn i to

EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Alicia	J	Zelaya	999887777	1968-01-19	3321 Castle, Spring, TX	F	25000	987654321	4
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5
Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000	987654321	4
James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000	NULL	1

Lname	Fname	Salary
Smith	John	30000
Wong	Franklin	40000
Zelaya	Alicia	25000
Wallace	Jennifer	43000
Narayan	Ramesh	38000
English	Joyce	25000
Jabbar	Ahmad	25000
Borg	James	55000

deler, der den ene har de ønskede kolonnene (resultatet av projeksjonen) og den andre har resten av kolonnene. For eksempel kan vi lage en liste over navnet og lønnen til alle ansatte:

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

Generelt vil PROJECT operatoren ha formen:

$$\pi_{\langle\text{attributtliste}\rangle}(\text{Relasjon})$$

Symbolet π (pi) brukes for å betegne PROJECT operatoren og attributtlisten gir attributtene som vi ønsker å hente fra relasjonen R. **Attributtene i resulterende tabell vil ha samme rekkefølge som i attributtlisten**. Graden til resultatet er derfor antall attributter i listen. Hvis attributtlisten kun består av attributter som ikke er nøkler, vil det sannsynligvis være mange duplikater. **PROJECT operatoren fjerner duplikate tupler, slik at resultatet er et sett av distinkte tupler og dermed en gyldig relasjon.** Dette kalles **duplikat eliminasjon**. For eksempel kan vi se på:

$$\pi_{\text{Sex,Salary}}(\text{EMPLOYEE})$$

Sex	Salary
M	30000
M	40000
F	25000
F	43000
M	38000
M	25000
M	55000

På tabellen kan vi se at tuplen $\langle 'F', 25000\rangle$ kun er inkludert én gang selv om EMPLOYEE har flere tupler med denne kombinasjonen. Eliminasjon av duplikater er kostbart, men nødvendig for at det skal lages et sett, slik formell relasjonsmodell krever (merk: SQL tillater multisett).

Hvis attributtlisten inneholder en supernøkkel vil resulterende relasjon ha samme antall tupler som R (siden nøkler må være unike = ingen duplikater). **Projeksjon er ikke kumulativ, dvs. rekkefølgen er viktig**. Hvis list2 inneholder alle attributtene i list1, vil:

$$\pi_{\langle list1\rangle}\left(\pi_{\langle list2\rangle}(R)\right) = \pi_{\langle list1\rangle}(R)$$

I SQL vil projeksjonen som regel gis i SELECT klausulen:

$$\pi_{\text{Sex,Salary}}(\text{EMPLOYEE})$$

```
SELECT DISTINCT Sex, Salary
FROM EMPLOYEE;
```

Legg merke til at vi må inkludere DISTINCT for at SQL skal eliminere duplikater og lage et sett, siden SQL tillater multisett.

Operatorsekvens og RENAME operatoren

De fleste spørringer krever at flere operatorer blir brukt etter hverandre. For å oppnå dette må vi enten lage ett enkelt uttrykk vha nøstede operatorer eller utføre operatorene en etter en og lagre resultatet i midlertidige relasjoner. I det siste tilfellet må vi **gi navn til relasjonene som skal holde det midlertidige resultatet**.

Vi ønsker å hente fornavn, etternavn og lønn til alle ansatte som arbeider ved avdeling fem. Første fremgangsmåte er et enkelt uttrykk med nøstede operatorer (kalles **in-line uttrykk**):

$$\pi_{\text{Fname,Lname,Salary}}\left(\sigma_{\text{Dno}=5}(\text{EMPLOYEE})\right)$$

Den andre fremgangsmåten er å bruker operatorene en etter en og lagre resultatet i midlertidige relasjoner som blir navngitt vha tildelingsoperatoren (←):

$$\text{DEP5_EMPS} \leftarrow \sigma_{\text{Dno}=5}(\text{EMPLOYEE})$$
$$\text{RESULT} \leftarrow \pi_{\text{Fname,Lname,Salary}}(\text{DEP5_EMPS})$$

Vi kan også bruke tildelingsoperatoren for å omdøpe attributter i midlertidige og resulterende relasjoner. Dette kan brukes for å legge til rette for UNION eller JOIN (mer senere). De nye navnene gis i parentes på venstre side:

$$\text{RESULT(FirstName, LastName, Salary)} \leftarrow \pi_{\text{Fname,Lname,Salary}}(\text{DEP5_EMPS})$$

Hvis det ikke oppgis noen attributtnavn vil den resulterende relasjonen bruke samme navn som den originale relasjonen. Vi kan også bruke RENAME operatoren ρ:

- $\rho_{S(B_1,B_2,\ldots,B_n)}(R)$ – gir nytt relasjonsnavn S og nye attributtnavn $B_1, B_2, \ldots, B_n$
- $\rho_S(R)$ – gir nytt relasjonsnavn S
- $\rho_{(B_1,B_2,\ldots,B_n)}(R)$ – gir nye attributtnavn $B_1, B_2, \ldots, B_n$

I SQL blir omdøping gjort vha AS og aliasing.

8.2 Mengdeoperatorer

UNION, INTERSECTION og MINUS operatorer

Settoperatorer brukes for å kombinere elementer fra to sett på ulike måter, og det inkluderer UNION, INTERSECTION og SET DIFFERENCE (eller MINUS/EXCEPT). Dette er binære operasjoner siden de opererer på to sett. **For at settoperasjoner skal kunne brukes på to relasjoner, må relasjonene ha samme type tupler, noe som kalles union-kompatible**. De to relasjonene må ha samme antall attributter og hvert attributtpar må ha samme domenet. Vi kan definere de tre operasjonene på to union-kompatible sett R og S, som følger:

- **UNION** ($R \cup S$) – gir en relasjon som inneholder alle tupler som enten er i R, S eller i begge. Duplikater blir eliminert.
- **INTERSECTION** ($\text{R} \cap \text{S}$) – gir en relasjon som inneholder alle tupler som er i R og S.
- **SET DIFFERENCE** ($\text{R} - \text{S}$) – gir en relasjon som inneholder alle tupler som er i R, men ikke i S.

For eksempel kan vi bruke UNION for å hente personnummer til alle ansatte som jobber ved avdeling 5 eller er direkte veileder for en ansatt som arbeider ved avdeling 5:

$$\text{DEP5_EMP} \leftarrow \sigma_{\text{Dno}=5}(\text{EMPLOYEE})$$
$$\text{RES1} \leftarrow \pi_{\text{Ssn}}(\text{EMPLOYEE})$$
$$\text{RES2(Ssn)} \leftarrow \pi_{\text{Super_ssn}}\,(\text{DEP5_EMP})$$
$$\text{RESULT} \leftarrow \text{RES1} \cup \text{RES2}$$

RESULT1

Ssn
123456789
333445555
666884444
453453453

RESULT2

Ssn
333445555
888665555

RESULT

Ssn
123456789
333445555
666884444
453453453
888665555

For mengeoperatorer vil resulterende relasjon bruke samme attributtnavn som den første relasjonen i operatoren (eks: R for $R \cup S$). Vi kan også omdøpe attributtene som på forrige side.

Figuren illustrerer bruken av mengeoperatorer:

a) To union-kompatible relasjoner (merk: attributtnavnene må ikke være like, men antall attributter og domenet må samsvare).
b) STUDENT ∪ INSTRUCTOR (union)
c) STUDENT ∩ INSTRUCTOR (snitt)
d) STUDENT − INSTRUCTOR (minus)
e) INSTRUCTOR − STUDENT (minus)

(a) STUDENT

Fn	Ln
Susan	Yao
Ramesh	Shah
Johnny	Kohler
Barbara	Jones
Amy	Ford
Jimmy	Wang
Ernest	Gilbert

INSTRUCTOR

Fname	Lname
John	Smith
Ricardo	Browne
Susan	Yao
Francis	Johnson
Ramesh	Shah

(b)

Fn	Ln
Susan	Yao
Ramesh	Shah
Johnny	Kohler
Barbara	Jones
Amy	Ford
Jimmy	Wang
Ernest	Gilbert
John	Smith
Ricardo	Browne
Francis	Johnson

(c)

Fn	Ln
Susan	Yao
Ramesh	Shah

(d)

Fn	Ln
Johnny	Kohler
Barbara	Jones
Amy	Ford
Jimmy	Wang
Ernest	Gilbert

(e)

Fname	Lname
John	Smith
Ricardo	Browne
Francis	Johnson

Husk at union og snitt er kumulativ og assosiativ:

$$R \cup S = S \cup R \qquad og \qquad R \cap S = S \cap R$$
$$R \cup (S \cup T) = (R \cup S) \cup T \qquad og \qquad R \cap (S \cap T) = (R \cap S) \cap T$$

Kartesisk produkt (kryssoperator)

Kartesisk produkt kalles også kryssprodukt og betegnes med ×. Det er en binær operator og **de to relasjonene behøver ikke å være union-kompatible.** Denne mengdeoperatoren vil lage en ny relasjon ved å kombinere alle tupler i den ene relasjonen med alle tupler i den andre relasjonen. For $R(A_1, A_2, \ldots, A_n) \times S(B_1, B_2, \ldots, B_m)$ vil resultatet være $Q(A_1, A_2, \ldots, A_n, B_1, B_2, \ldots B_m)$. Hvis R har n_R tupler og S har n_s tupler vil Q ha $n_R * n_s$ tupler.

Det kartesiske produktet er unyttig alene, men kan bli nyttig hvis det kombineres med en seleksjon. For eksempel kan det brukes for å hente navnet til dependents hos alle kvinnelige ansatte:

$$\text{FEMALE_EMPS} \leftarrow \sigma_{\text{Sex}='\text{F}'}(\text{EMPLOYEE})$$
$$\text{EMPNAMES} \leftarrow \pi_{\text{Fname,Lname,Ssn}}(\text{FEMALE_EMPS})$$
$$\text{EMP_DEPENDENTS} \leftarrow \text{EMPNAMES} \times \text{DEPENDENT}$$
$$\text{ACTUAL_DEPENTENTS} \leftarrow \sigma_{\text{Ssn=Essn}}(\text{EMP_DEPENDENTS})$$
$$\text{RESULT} \leftarrow \pi_{\text{Fname,Lname,Dependent_name}}(\text{ACTUAL_DEPENTENTS})$$

Figuren viser de ulike relasjonene som blir laget. Først velger vi ut EMPLOYEE tuplene som er kvinner og deretter henter vi fornavn, etternavn og personnummer til disse. EMP_DEPENDENTS er resultatet av kryssproduktet, som vil kombinere de kvinnelige ansatte med alle dependents i selskapet. Vi ønsker å kombinere en kvinnelig ansatt med kun hennes dependents, og gjør dette ved å velge ut tuplene der personnummer til kvinnen er lik personnummeret til den ansatte som dependent har et forhold til (Ssn = Essn). Dermed vil vi få en relasjon som inneholder kvinnelige ansatte og deres dependent, og vi kan bruke denne for å hente ut fornavn, etternavn og navnet til dependent.

FEMALE_EMPS

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
Alicia	J	Zelaya	999887777	1968-07-19	3321Castle, Spring, TX	F	25000	987654321	4
Jennifer	S	Wallace	987654321	1941-06-20	291Berry, Bellaire, TX	F	43000	888665555	4
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5

EMPNAMES

Fname	Lname	Ssn
Alicia	Zelaya	999887777
Jennifer	Wallace	987654321
Joyce	English	453453453

EMP_DEPENDENTS

Fname	Lname	Ssn	Essn	Dependent_name	Sex	Bdate	...
Alicia	Zelaya	999887777	333445555	Alice	F	1986-04-05	...
Alicia	Zelaya	999887777	333445555	Theodore	M	1983-10-25	...
Alicia	Zelaya	999887777	333445555	Joy	F	1958-05-03	...
Alicia	Zelaya	999887777	987654321	Abner	M	1942-02-28	...
Alicia	Zelaya	999887777	123456789	Michael	M	1988-01-04	...
Alicia	Zelaya	999887777	123456789	Alice	F	1988-12-30	...
Alicia	Zelaya	999887777	123456789	Elizabeth	F	1967-05-05	...
Jennifer	Wallace	987654321	333445555	Alice	F	1986-04-05	...
Jennifer	Wallace	987654321	333445555	Theodore	M	1983-10-25	...
Jennifer	Wallace	987654321	333445555	Joy	F	1958-05-03	...
Jennifer	Wallace	987654321	987654321	Abner	M	1942-02-28	...
Jennifer	Wallace	987654321	123456789	Michael	M	1988-01-04	...
Jennifer	Wallace	987654321	123456789	Alice	F	1988-12-30	...
Jennifer	Wallace	987654321	123456789	Elizabeth	F	1967-05-05	...
Joyce	English	453453453	333445555	Alice	F	1986-04-05	...
Joyce	English	453453453	333445555	Theodore	M	1983-10-25	...
Joyce	English	453453453	333445555	Joy	F	1958-05-03	...
Joyce	English	453453453	987654321	Abner	M	1942-02-28	...
Joyce	English	453453453	123456789	Michael	M	1988-01-04	...
Joyce	English	453453453	123456789	Alice	F	1988-12-30	...
Joyce	English	453453453	123456789	Elizabeth	F	1967-05-05	...

ACTUAL_DEPENDENTS

Fname	Lname	Ssn	Essn	Dependent_name	Sex	Bdate	...
Jennifer	Wallace	987654321	987654321	Abner	M	1942-02-28	...

RESULT

Fname	Lname	Dependent_name
Jennifer	Wallace	Abner

Det kartesiske produktet i seg selv gir ingen menig, siden det kombinerer kvinner med alle dependents, men når vi klarer å hente ut de tuplene som har vesentlig informasjon, blir kryssproduktet nyttig. Vi vil se at JOIN operatoren gjør dette på en mer oversiktlig måte.

8.3 Binære operatorer – JOIN og DIVISION

JOIN operatoren

JOIN operatoren betegnes med ⋈ og brukes for å kombinere relaterte tupler fra to relasjoner til enkelte «lengre» tupler. Den brukes for å kombinere data fra to relasjoner, slik at relatert informasjon kan presenteres i en enkelt tabell. For eksempel kan vi hente navnet til manageren ved hver avdeling:

$$\text{DEPT_MGR} \leftarrow \text{DEPARTMENT} \bowtie_{\text{Mgr_ssn=Ssn}} \text{EMPLOYEE}$$
$$\text{RESULT} \leftarrow \pi_{\text{Dname,Lname,Fname}}(\text{DEPT_MGR})$$

DEPT_MGR

Dname	Dnumber	Mgr_ssn	...	Fname	Minit	Lname	Ssn	...
Research	5	333445555	...	Franklin	T	Wong	333445555	...
Administration	4	987654321	...	Jennifer	S	Wallace	987654321	...
Headquarters	1	888665555	...	James	E	Borg	888665555	...

For å få navnet til manageren må vi slå sammen hver DEPARTMENT tuple med EMPLOYEE tuplen som har en Ssn verdi som matcher Mgr_ssn. Som vi kan se på figuren vil hver resulterende tuple representere en avdeling og den ansatte som er manageren for denne avdelingen. Legg merke til at Mgr_ssn i DEPARTMENT er en fremmednøkkel som referer til Ssn som er primærnøkkelen til EMPLOYEE. Dette gjør at vi vil ha matchende tupler i de to relasjonene.

Det kartesiske produktet og seleksjonen:

$$\text{EMP_DEPENDENTS} \leftarrow \text{EMPNAMES} \times \text{DEPENDENT}$$
$$\text{ACTUAL_DEPENTENTS} \leftarrow \sigma_{\text{Ssn=Essn}}(\text{EMP_DEPENDENTS})$$

Kan erstattes med en enkel JOIN operator:

$$\text{ACTUAL_DEPENTENTS} \leftarrow \text{EMPNAMES} \bowtie_{\text{Ssn=Essn}} \text{DEPENDENT}$$

Generelt vil JOIN operatoren ha formen:

$$R \bowtie_{\langle \text{join betingelse} \rangle} S$$

For $R(A_1, A_2, \ldots, A_n) \times S(B_1, B_2, \ldots, B_m)$ vil resultatet være $Q(A_1, A_2, \ldots, A_n, B_1, B_2, \ldots B_m)$. Hvis R har n_R attributter og S har n_s attributter vil Q ha $n_R + n_s$ attributter. **Q har en tuple for hver kombinasjon som tilfredsstiller join betingelsen** (husk: kryssprodukt har alle kombinasjoner). Join betingelsen er spesifisert på attributter fra de to relasjonene og blir evaluert for hver kombinasjon av tupler. Det er kun kombinasjoner som gir TRUE som blir inkludert i resultatet. Tupler der join attributtene er NULL blir ikke inkludert.

Variasjoner av JOIN – EQUIJOIN og NATURAL JOIN

EQUIJOIN er en sammenslåing av tupler der join betingelsen er en ekvivalens-sammenligning (=). Dette er den vanligste typen JOIN, og i resultatet vil en eller flere attributtpar ha identiske verdier. For eksempel i DEPT_MGR på forrige side vil Mgr_ssn og Ssn være identiske i alle tupler, fordi dette er kravet til join betingelsen.

NATURAL JOIN betegnes med $*$ og er en sammenslåing av tupler der join betingelsen ikke blir spesifisert, fordi tuplene blir slått sammen basert på attributter med samme navn i de to relasjonene. Denne varianten av JOIN krever at de to attributtene i alle join attributtpar må ha samme navn, så hvis det ikke er tilfellet må vi begynne med en omdøping av attributter. For eksempel kan vi kombinere hver PROJECT tuple med DEPARTMENT tuplen som kontrollerer prosjektet. Attributtparet er Dnumber i DEPARTMENT og Dnum i PROJECT. Siden de ikke har samme navn, må vi omdøpe det ene attributtet vha RENAME operatoren (ρ). Deretter kan vi kombinere tuplene vha en naturlig join ($*$):

$$\text{PROJ_DEPT} \leftarrow \text{PROJECT} * \rho_{(\text{Dname},\text{Dnum},\text{Mgr_ssn},\text{Mgr_start_date})}(\text{DEPARTMENT})$$

Her vil Dnum være join attributtet, siden det er det eneste attributtet med samme navn i begge relasjonene. To tupler slås sammen hvis de har samme verdi for dette attributtet.

Hvis ingen kombinasjoner av tupler tilfredsstiller join betingelsen, vil resultatet være en tom relasjon med ingen tupler. Join selektiviteten til join betingelsen er forventet størrelse til join resultatet delt på maksimal størrelse $n_R * n_S$. **Hvis det ikke blir gitt noen join betingelse og ingen attributter har samme navn, vil JOIN gi et kartesisk produkt**. INNER JOIN er en match-og-kombiner operator som er definert som en blanding mellom kartesisk produkt og seleksjon (standard JOIN). Ved OUTER JOIN vil alle tupler fra en eller begge relasjonene være i resultatet.

Naturlig join og equijoin kan brukes på flere tabeller for å gi n-way join, for eksempel:

$$(\text{PROJECT} \bowtie_{\text{Dnum}=\text{Dnumber}} \text{DEPARTMENT}) \bowtie_{\text{Mgr_ssn}=\text{Ssn}} \text{EMPLOYEE}$$

Hver prosjekt tuple blir kombinert med sin kontrollerende avdeling, og den resulterende tuplen blir så kombinert med den ansatte som er manageren til avdelingen.

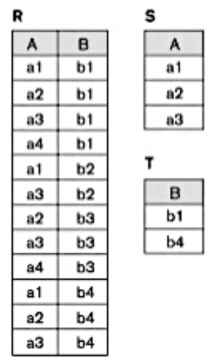
Fullstendig sett av relasjonsalgebra operasjoner

Settet av relasjonsalgebra operatorer: $\{\sigma, \pi, \cup, \rho, -, \times\}$ er et fullstendig sett, altså kan enhver annen operator uttrykkes som en sekvens av operatorene fra dette settet. For eksempel kan INTERSECTION og JOIN uttrykkes som følgende:

$$R \cap S \equiv (R \cup S) - \big((R - S) \cup (S - R)\big)$$
$$R \bowtie_{\langle\text{betingelse}\rangle} S \equiv \sigma_{\langle\text{betingelse}\rangle}(R \times S)$$

Samtidig er det ineffektivt å spesifisere disse komplekse uttrykkene hver gang vi ønsker å bruke snittet eller join, så derfor blir flere operatorer inkludert som separate operatorer.

Merk: DIVISION operatoren representeres med % i diagram



R

A	B
a1	b1
a2	b1
a3	b1
a4	b1
a1	b2
a3	b2
a2	b3
a3	b3
a4	b3
a1	b4
a2	b4
a3	b4

S

A
a1
a2
a3

T

B
b1
b4

DIVISION operatoren

DIVISION operatoren betegnes med ÷ og brukes for å finne verdiene i den ene relasjonen som er kombinert med alle verdiene i den andre relasjonen. Figuren til høyre viser $T \leftarrow R \div S$. Her kan vi se at resultatet er b_1 og b_4, fordi disse verdiene i R er kombinert med alle verdiene i S (dvs. vi har a_1b_1, a_2b_1, a_3b_1 og a_1b_4, a_2b_4, a_3b_4).

Dette er nyttig når vi ønsker å hente ut tupler som oppfyller *alle* betingelsene som tuplen det sammenlignes med gjør. For eksempel kan vi hente navnet til ansatte som arbeider på *alle* prosjektene som 'Jim Smith' arbeider på. Vi må hente ut personnummer og prosjektnummer til alle ansatte og prosjektnummer til 'Jim Smith'. Deretter må vi se hvilke personnummer som har en relasjon til alle prosjektnumrene til 'Jim Smith', fordi det betyr at denne ansatte arbeider ved samme prosjekt:

$$\text{SSN_PNOS} \leftarrow \pi_{\text{Essn,Pno}}(\text{WORKS_ON})$$

$$\text{SMITH} \leftarrow \sigma_{\text{Fname='John' AND Lname='Smith'}}(\text{EMPLOYEE})$$
$$\text{SMITH_PNOS} \leftarrow \pi_{\text{Pno}}(\text{WORKS_ON} \bowtie_{\text{Essn=Ssn}} \text{SMITH})$$

$$\text{SSNS(Ssn)} \leftarrow \text{SSN_PNOS} \div \text{SMITH_PNOS}$$
$$\text{RESULT} \leftarrow \pi_{\text{Fname,Lname}}(\text{SSN} * \text{EMPLOYEE})$$

SSN_PNOS

Essn	Pno
123456789	1
123456789	2
666884444	3
453453453	1
453453453	2
333445555	2
333445555	3
333445555	10
333445555	20
999887777	30
999887777	10
987987987	10
987987987	30
987654321	30
987654321	20
888665555	20

SMITH_PNOS

Pno
1
2

SSNS

Ssn
123456789
453453453

Som vi kan se på figuren vil SSNS som er resultatet av divisjonen inneholde 123456789 og 453453453 fordi i SSN_PNOS er det de eneste Essn verdiene som har relasjon til både Pno 1 og 2. Vi gir attributtet navnet Ssn, slik at vi kan bruke en naturlig join med EMPLOYEE for å hente navnene som hører til disse personnumrene. DIVISION operatoren blir generelt brukt på to relasjoner $R(X) \div S(Y)$, der attributtene til S er et subsett av attributtene til R $(Y \subseteq X)$. Resultatet vil inneholde settet av attributter i R som ikke er i S, altså: $Z = X - Y$. En tuple vil være i resultatet dersom den har verdier i R som er kombinert med *alle* verdiene i S.

Tabellen viser en oppsummering av operatorene vi har sett på

Operator	Mål	Notasjon
SELECT	Velger ut tupler som tilfredsstiller seleksjonsbetingelsen fra en relasjon R	$\sigma_{\langle \text{seleksjonsbetingelse} \rangle}(R)$
PROJECT	Lager en relasjon med kun bestemte attributter fra R og fjerner duplikater	$\pi_{\langle \text{attributtliste} \rangle}(R)$
THETA JOIN	Produserer alle kombinasjoner av tupler fra R_1 og R_2 som tilfredsstiller join betingelsen	$R_1 \bowtie_{\langle \text{join betingelse} \rangle} R_2$
EQUIJOIN	Join betingelsen er en ekvivalenssammenligning	$R_1 \bowtie_{\langle \text{join betingelse} \rangle} R_2$
NATURAL JOIN	Sammenslåingen er basert på attributter med samme navn	$R_1 * R_2$
UNION	Lager en relasjon med alle tupler i R_1, R_2 eller begge. Relasjonene må være union kompatible	$R_1 \cup R_2$
INTERSECTION	Lager en relasjon med tupler som er i R_1 og R_2. Relasjonene må være union kompatible	$R_1 \cap R_2$
DIFFERENCE	Lager en relasjon med tupler som er i R_1 og ikke i R_2. Relasjonene må være union kompatible	$R_1 - R_2$
CARTESIAN PRODUCT	Produserer en relasjon som har attributtene til R_1 og R_2, og inkluderer alle kombinasjoner av tupler	$R_1 \times R_2$
DIVISION	Lager en relasjon $R(X)$ med alle tupler som er i R_1 i kombinasjon med alle tupler fra R_2.	$R_1 \div R_2$

Notasjon for spørretrær
Har valgt å ikke inkludere denne delen av boka, siden forelesning bruker en annerledes og mer intuitiv bruk av spørretrær (se oppsummering kapittel 8).

8.4 Spesielle operatorer
Noen vanlige databaseforespørsler kan ikke utføres av de originale relasjonsalgebra operatorene. For å uttrykke disse forespørslene har det blitt utviklet flere spesielle operatorer.

Generalisert projeksjon
Generalisert projeksjon utvider projeksjonsoperatoren ved å la funksjoner på attributter inkluderes i projeksjonslisten. Det uttrykkes som:

$$\pi_{F1,F2,\ldots,F_n}(R)$$

der $F_1, F_2, \ldots, F_n$ er funksjonene som kan involvere aritmetiske operasjoner og konstante verdier. Dette er nyttig når vi ønsker at utregnede verdier skal være kolonner i resultatet. For eksempel for relasjonen: $\text{EMPLOYEE}(\text{Ssn}, \text{Salary}, \text{Deduction}, \text{Years_service})$, kan det hende vi ønsker å at projeksjonen skal inkludere:

$$\text{Net Salary} = \text{Salary} - \text{Deduction}$$
$$\text{Bonus} = 2000 * \text{Years_service}$$
$$\text{Tax} = 0.25 * \text{Salary}$$

Dette gjør vi vha følgende generaliserte projeksjon:

$$\text{RESULT} \leftarrow \rho_{\text{Ssn,Net_Salary,Bonus,Tax}}(\pi_{Ssn,(\,Salary-Deduction),(2000*\text{Years_service}),(0.25*\text{Salary})}(\text{EMPLOYEE})$$

Aggregatfunksjoner og gruppering
Aggregatfunksjoner brukes for å summere informasjon fra flere tupler i en enkel-tuple summering. Eksempler på slike funksjoner er å hente gjennomsnittet eller total lønn for alle ansatte eller hente totalt antall ansatte tupler. SUM, AVERAGE, MAXIMUM og MINIMUM brukes på samlinger av numeriske verdier, mens COUNT brukes for å telle tupler eller verdier.

Et vanlig ønske er å gruppere tupler i en relasjon etter verdien til noen av deres attributter og deretter bruke en aggregatfunksjon på hver gruppe. For eksempel kan vi gruppere EMPLOYEE tupler etter Dno, slik at hver gruppe vil inneholde ansatte som arbeider ved samme avdeling. Deretter kan vi liste hver Dno verdi sammen med for eksempel gjennomsnittlig lønn ved denne avdelingen. Den generelle formen for aggregatfunksjonen og grupperingen er:

$$_{\langle\text{grupperende attributt}\rangle}\gamma_{\langle\text{funksjonsliste}\rangle}(R)$$

I tekst bruker vi γ som operator, mens på grafene bruker vi F.

Her vil grupperende attributt være en liste av attributter som bestemmer hvilke grupper ulike tupler plasseres i. Funksjonslisten er en liste av (⟨funksjon⟩ ⟨attributt⟩) par som gir aggregatfunksjonen som skal brukes på bestemte attributter i gruppene. Resultatet vil ha kolonner for grupperende attributter pluss én kolonne for hvert element i funksjonslisten.

For eksempel kan vi hente avdelingsnummer, antall ansatte i avdelingen og gjennomsnittlig lønn, samt omdøpe resulterende attributter ved å se på:

$$\rho_{\text{Dno,No_of_employees,Average_sal}}\left({}_{\text{Dno}}\gamma_{\text{Count Ssn,Average Salary}}(\text{EMPLOYEE})\right)$$

Merk: for å telle alle tupler i en relasjon kan vi bruke Count(Primærnøkkel). Figur a viser resultatet av denne operasjonen. Hvis vi ikke bruker RENAME operatoren vil de resulterende attributtene få navn på formen ⟨funksjon⟩_⟨attributt⟩, for eksempel vil resultatet av Count Ssn få navnet Count_Ssn (figur b).

(a)

	DNO	NO_OF_EMPLOYEES	AVERAGE_SAL
	5	4	33250
	4	3	31000
	1	1	55000

Hvis ingen grupperende attributt blir gitt, vil funksjonene brukes på alle tuplene i relasjonen, slik at resultatet får én enkelt tuple. Figur c viser resultatet av:

$$\gamma_{\text{Count Ssn,Average Salary}}(\text{EMPLOYEE})$$

(b)

DNO	COUNT_SSN	AVERAGE_SALARY
5	4	33250
4	3	31000
1	1	55000

(c)

COUNT_SSN	AVERAGE_SALARY
8	35125

Noen viktige merknader:

- **Duplikater blir som regel ikke eliminert når en aggregatfunksjon blir brukt**, slik at vi regner ut den normale tolkningen av funksjoner som sum og gjennomsnitt.
- **NULL verdier blir ikke tatt med i aggregering**.
- **Resultatet til en aggregatfunksjon vil være en relasjon**, og ikke et skalart nummer.

OUTER JOIN operatoren

INNER JOIN vil eliminere tupler som ikke matcher eller har NULL i join attributtet, og dette er standard for JOIN (dvs. brukes hvis ikke noe annet oppgis). **Et sett av operasjoner, kalt OUTER JOIN ble utviklet for tilfeller der brukeren ønsker å ta våre på alle tupler i en eller begge relasjonene, selv om de ikke har noen matchende tupler i den andre relasjonen.** Tupler fra de to tabellene kan kombineres ved å matche korresponderende rader og tupler som mangler matchende verdier blir ikke tapt. Det finnes tre typer OUTER JOIN:

- **LEFT OUTER JOIN** (⟕)– tar vare på alle tupler hos relasjonen til venstre i uttrykket
- **RIGHT OUTER JOIN** (⟖)– tar vare på alle tupler hos relasjonen til høyre i uttrykket
- **FULL OUTER JOIN** (⟗)– tar vare på alle tupler hos begge relasjonene

For eksempel kan det hende vi ønsker navnet til alle ansatte i tillegg til avdelingsnavnet dersom den ansatte er manager ved en avdeling. Dersom de ikke er manager kan dette indikeres med en NULL verdi. Vi kan gjøre dette vha en LEFT OUTER JOIN:

$$\text{TEMP} \leftarrow \left(\text{EMPLOYEE} ⟕_{\text{Ssn=Mgrssn}} \text{DEPARTMENT}\right)$$
$$\text{RESULT} \leftarrow \pi_{\text{Fname,Minit,Lname,Dname}}(\text{TEMP})$$

RESULT

Fname	Minit	Lname	Dname
John	B	Smith	NULL
Franklin	T	Wong	Research
Alicia	J	Zelaya	NULL
Jennifer	S	Wallace	Administration
Ramesh	K	Narayan	NULL
Joyce	A	English	NULL
Ahmad	V	Jabbar	NULL
James	E	Borg	Headquarters

Som vi kan se på figuren vil attributter til den andre relasjonen fylles med NULL når det er ingen matchende tupler.

Oppsummering – Kapittel 8 (F6, F7)

Relasjonsalgebra

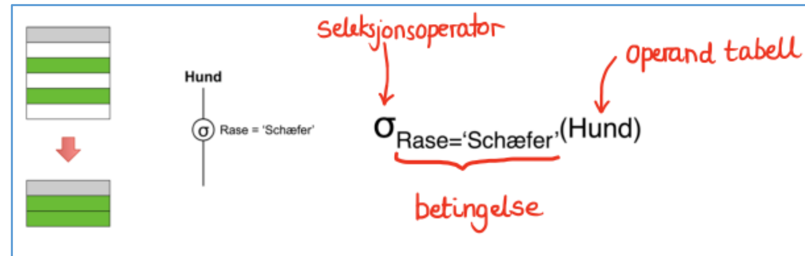
Relasjonsalgebra beskriver operatorer som brukes for å manipulere tabeller, og vi ser kun på spørrefunksjonalitet og ikke innsetting, endring eller sletting. Dette er et grunnlag for å forstå hva man kan gjøre med tabeller, noe som er viktig når vi senere skal lære om normaliseringsteorien, spørreoptimalisering og spørreutføring. Husk at tabellforekomster er mengder av tupler.

Operatorene er lukket over tabeller, noe som vil si at operanden(e) og resultatet er tabeller. Vi skal se at operatorene kan deles inn i:

- **Mengdeoperatorer** - union, snitt, mengdedifferanse, kartesisk produkt, osv.
- **Spesielle operatorer** – seleksjon, projeksjon, join, osv.

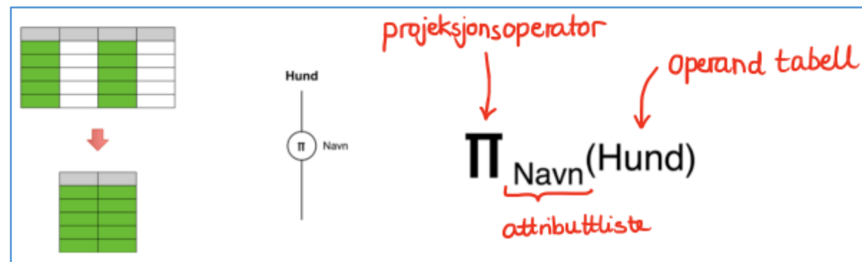
Spesielle operatorer – seleksjon

SELECT operatoren betegnes med *σ* og vil velge ut tupler som oppfyller en logisk betingelse. Resultattabellen vil ha samme skjema som operandtabellen. Legg merke til hvordan relasjonsalgebra-diagrammet tegnes. Operanden plasseres ved toppen og operatorene plasseres i sirkler. Betingelsen til operatoren blir skrevet ved siden av sirkelen.



Spesielle operatorer – projeksjon

PROJECT operatoren betegnes med *π* og vil lage en relasjon som kun består av kolonnene (attributtene) som gis i attributtlisten. Resultattabellen kan derfor få et nytt skjema og **duplikater tupler blir fjernet**.



Kombinere operatorer

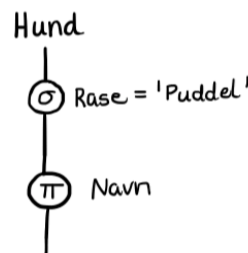
Vi kan lage et relasjonsalgebra uttrykk som bruker flere operatorer etter hverandre. For eksempel kan vi hente navnet til alle hunder som er av rasen Puddel, ved å se på:

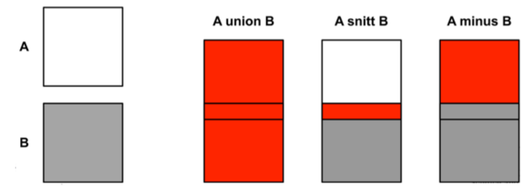
$$\pi_{Navn}(\sigma_{Navn='Puddel'}(Hund))$$

Hund-tabellen

REGNR	NAVN	FAAR	RASE	EIER_PNR
1	King	1992	Rottweiler	1
2	Tarzan	1993	Puddel	3
3	Troll	1995	Collie	7
4	King	1995	Schæfer	6
5	Varg	1995	Newfoundland	1
6	Prins	1994	Schæfer	1
7	King	1996	Puddel	4
8	King	1993	Rottweiler	4
9	Tarzan	1995	Doberman	3
10	Troll	1996	Puddel	1

Navn
Tarzan
King
Troll

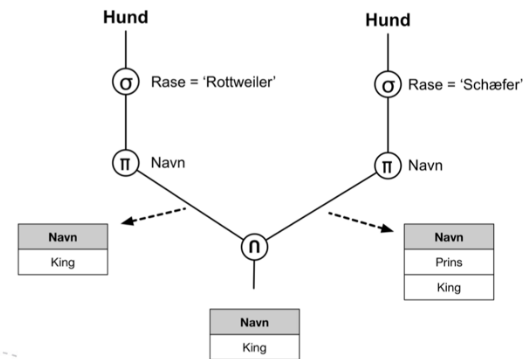




Mengdeoperatorer – Union, snitt og differanse

Mengdeoperatorene UNION (∪), INTERSECTION (∩) og SET DIFFERENCE (−) kan kun brukes på to union-kompatible tabeller, som vil si at de har samme grad (like mange attributter) og korresponderende kolonner har samme domenet. Figuren viser hvordan operatorene fungerer.

For eksempel kan vi hente ut navn som har blitt brukt på både Rottweilere og Schæfere. Vi lager to separate seleksjoner og projeksjoner som henter ut navnene til Rottweilere og navnene til Schæferne, og deretter tar vi snittet av disse tabellene:



Her kan vi bruke snitt, siden de to relasjonene har like mange attributter med samme domene = de er union-kompatible

$$\pi_{\text{Navn}}(\sigma_{\text{Rase}='\text{Rottweiler}'}) \cap \pi_{\text{Navn}}(\sigma_{\text{Rase}='\text{Schæfer}'})$$

A

a	b
a_1	b_1
a_1	b_2
a_2	b_3

} 3 rader

B

c	d
c_1	d_1
c_2	d_2

} 2 rader

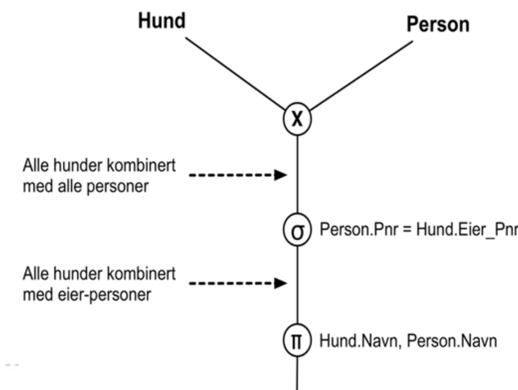
A × B

a	b	c	d
a_1	b_1	c_1	d_1
a_1	b_1	c_2	d_2
a_1	b_2	c_1	d_1
a_1	b_2	c_2	d_2
a_2	b_3	c_1	d_1
a_2	b_3	c_2	d_2

} $3 \cdot 2 = 6$ rader

Mengdeoperatorer – Kartesisk produkt

Kartesisk produkt betegnes med × og vil kombinere alle tupler i den ene tabellen med alle tupler i den andre tabellen. Resultattabellen vil dermed få alle kolonner fra den første og den andre tabellen. Hvis operandtabellene har hhv. r og s tupler, vil resultattabellen ha $r * s$ tupler (figur til venstre).

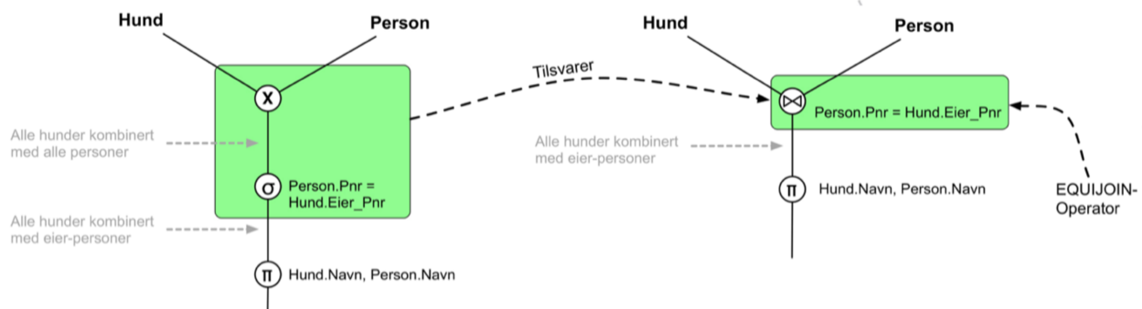


Det kartesiske produktet er ikke nyttig alene, men dersom det kombineres med en seleksjon vil det bli mer nyttig. For eksempel kan vi bruke dette til å hente navnet til alle hunder og deres eiere:

$$\pi_{\text{Hund.navn,Person.Navn}}\left(\sigma_{\text{Person.Pnr=Hund=Eier}_{\text{Pnr}}}(\text{Hund} \times \text{Person})\right)$$

Spesielle operatorer – join

Kombinasjonen av kartesisk produkt og seleksjon ble svært mye brukt i databaseapplikasjoner, så derfor ble det utviklet en egen operator, kalt JOIN som tilsvarer denne kombinasjonen. Kartesisk produkt er ukritisk til sammenstillingen, siden alle rader kombineres med alle andre rader, og dette gir store resultattabeller. **JOIN-operatoren kombinerer relaterte rader, slik at radene i resultattabellen oppfyller en join-betingelse**. Dersom denne betingelsen er et likhetskrav, kalles operatoren for en EQUIJOIN.



Kartesisk produkt og seleksjon vil gi samme resultat som JOIN, men effektiviteten vil variere. Som regel vil JOIN være mer effektivt, siden vi unngår store resultattabeller som vi så må utføre en seleksjon på.

Foto

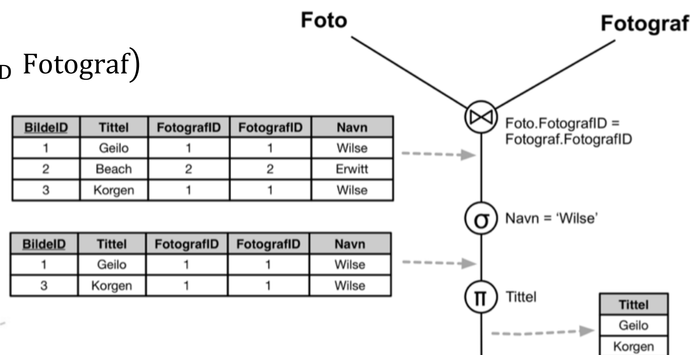
BildeID	Tittel	FotografID
1	Geilo	1
2	Beach	2
3	Korgen	1
4	Nidaros	NULL

Fotograf

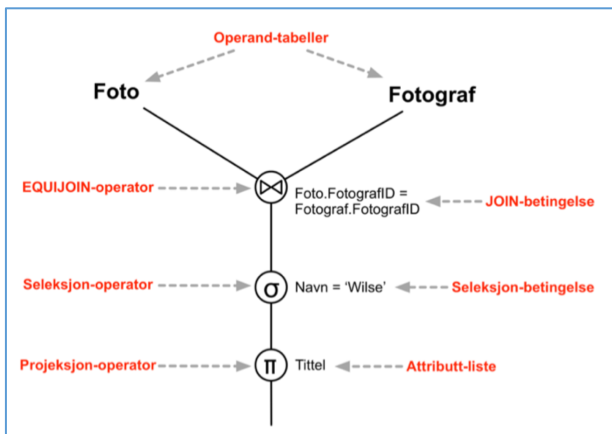
FotografID	Navn
1	Wilse
2	Erwitt
3	Leibovitz

For eksempel kan vi hente alle bilder som har blitt tatt av fotografen Wilse:

$$\pi_{\text{Tittel}}(\sigma_{\text{Navn}='\text{Wilse}'}(\text{Foto} \bowtie_{\text{FotografID}=\text{FotografID}} \text{Fotograf})$$



På figuren kan vi se de ulike stegene i uttrykket. Her kunne vi ha brukt en naturlig join siden join attributtene har samme navn.



EQUIJOIN

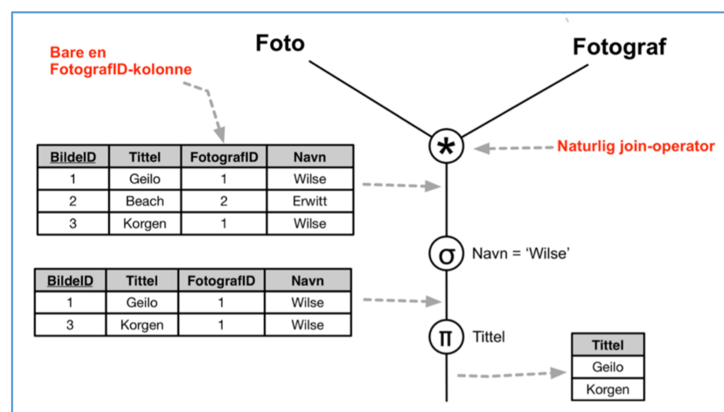
Ved EQUIJOIN vil rader kombineres basert på likhet og rader som ikke matcher vil filtreres bort. For eksempel vil Foto 4 (Nidaros) som har ukjent fotograf og Fotograf 3 (Leibovitz) som ikke har noen foto bli droppet. Join betingelsen vil altså være en ekvivalenssammenligning av attributter, og det kan være ett eller flere kolonner fra hver sin operandtabell. Som vi kan se på figuren over vil EQUIJOIN gi duplikatkolonner i resultatet (to FotografID kolonner).

NATURAL JOIN

NATURAL JOIN betegnes med $*$ og har en implisitt join-betingelse som er likhet i alle kolonnepar med like navn. Altså hvis et attributtpar fra de to relasjonene har samme navn vil tupler slås sammen hvis de har samme verdi for korresponderende attributter. I resultatet vil attributtpar representeres som én kolonne, altså **duplikatkolonner fjernes**. Naturlig join tilsvarer kartesisk produkt + seleksjon + projeksjon (fjerner duplikatkolonner). **Risikoen er at join-betingelsen kan bli større enn det man ønsker og ved endringer i operandtabellene kan join-betingelsen endre seg fordi det blir flere eller færre like attributtnavn.**

Siden Foto og Fotograf begge har en attributt som heter FotografID, kan vi bruke naturlig join for å slå sammen disse:

$$\pi_{\text{Tittel}}(\sigma_{\text{Navn}='\text{Wilse}'}(\text{Foto} * \text{Fotograf})$$



Merk at det er ingen endringer bortsett fra at vi bruker $*$ istedenfor $\bowtie_{\text{FotografID}=\text{FotografID}}$. **Her har vi slått sammen to tabeller vha fremmednøkkel-primærnøkkel forholdet**.

Person-tabellen

PNR	NAVN
1	Olav
2	Kari
3	Anne
4	Lisbeth
5	Harald
6	Liv
7	Trude
8	Per
9	Kristin
10	Christina
11	Petter
12	Liv
13	Merete

Bittav-tabellen

PNR	REGNR	ANTALL
2	2	5
3	9	1
4	9	1
5	2	3
5	4	2
6	9	2
8	5	2
9	4	4
11	5	1
12	4	3

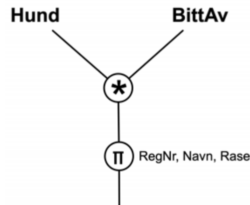
Q1: Finn RegNr, Navn og Rase for alle hunder som har bitt noen

Vi må ha RegNr og Navn for alle Hund tupler som også er registrert i BittAv-tabellen. Derfor må vi slå sammen Hund-. og Bittav-tabellene, og siden begge har attributtet RegNr, kan vi bruke naturlig join:

Hund-tabellen

REGNR	NAVN	FAAR	RASE	EIER_PNR
1	King	1992	Rottweiler	1
2	Tarzan	1993	Puddel	3
3	Troll	1995	Collie	7
4	King	1995	Schæfer	6
5	Varg	1995	Newfoundland	1
6	Prins	1994	Schæfer	1
7	King	1996	Puddel	4
8	King	1993	Rottweiler	4
9	Tarzan	1995	Doberman	3
10	Troll	1996	Puddel	1

$$\pi_{\text{RegNr,Navn,Rase}}(\text{Hund} * \text{Bittav})$$



RegNr	Navn	Rase
2	Tarzan	Puddel
4	King	Schæfer
5	Varg	Newfoundland
9	Tarzan	Doberman

Her har vi utnyttet filtreringseffekten til join-operatoren for å kun hente ut Hund tupler som har bitt noen. For eksempel vil ikke hund 10 fra Hund-tabellen bli med i resultatet til naturlig join, fordi den har ingen matchende tuple i Bittav-tabellen. I dette tilfellet vil naturlig join gi flere tupler for samme hund som har bitt ulike personer. Når vi så gjennomfører en projeksjon der kun Regnr, Navn og rase til hunden velges ut, vil duplikate tupler fjernes.

Q2: Finn Pnr og Navn for alle personer som er bitt av hunder som heter Tarzan

Vi må ha Pnr og Navn for alle Person tupler som har et forhold via Bittav-tabellen til hunder med navn Tarzan. Derfor må vi slå sammen Person-, Bittav- og Hund-tabellen. Vi bruker:

$$\pi_{\text{Person.Pnr,Perosn.Navn}}\left((\sigma_{\text{Navn='Tarzan'}}(\text{Hund}) * \text{BittAv}) \bowtie_{\text{Person.Pnr=Bittav.Pnr}} \text{Person}\right)$$

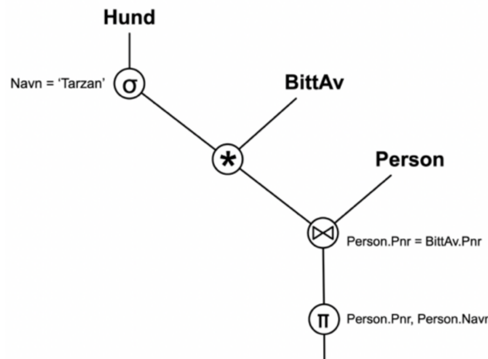
OBS: Her kan vi ikke bruke naturlig join på Person-tabellen, fordi både Person og Hund har et attributt som heter Navn, og dermed ville dette også blitt et join attributtpar. Legg merke til at vi bruker seleksjon i join for å kun slå sammen tupler som oppfyller seleksjonsbetingelsen.

Person-tabellen

PNR	NAVN
1	Olav
2	Kari
3	Anne
4	Lisbeth
5	Harald
6	Liv
7	Trude
8	Per
9	Kristin
10	Christina
11	Petter
12	Liv
13	Merete

Bittav-tabellen

PNR	REGNR	ANTALL
2	2	5
3	9	1
4	9	1
5	2	3
5	4	2
6	9	2
8	5	2
9	4	4
11	5	1
12	4	3



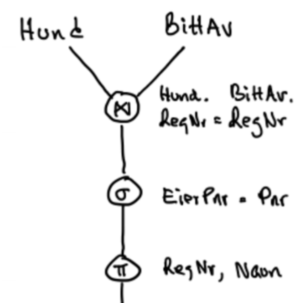
Hund-tabellen

REGNR	NAVN	FAAR	RASE	EIER_PNR
1	King	1992	Rottweiler	1
2	Tarzan	1993	Puddel	3
3	Troll	1995	Collie	7
4	King	1995	Schæfer	6
5	Varg	1995	Newfoundland	1
6	Prins	1994	Schæfer	1
7	King	1996	Puddel	4
8	King	1993	Rottweiler	4
9	Tarzan	1995	Doberman	3
10	Troll	1996	Puddel	1

Q3: Finn RegNr og navn for hunder som har bitt sin egen eier

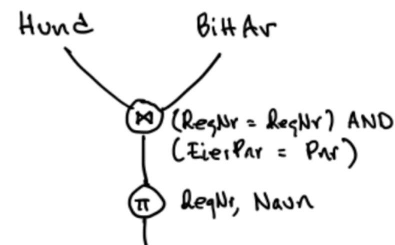
For å finne hunder som har bitt noen må vi slå sammen Hund og Bittav gjennom RegnNr. Deretter må vi velge ut hundene som har bitt sin egen eier vha en seleksjon:

$$\pi_{\text{RegNr,Navn}}\left(\sigma_{(\text{Eier_Pnr=Pnr})}(\text{Hund} * \text{Bittav})\right)$$



OBS: For at vi skal kunne sette sammen Bittav og Hund på en logisk måte, må vi inkludere RegNr, fordi dette er fremmednøkkelen i Bittav som peker på primærnøkkelen i Hund og som dermed beskriver koblingen mellom tupler i de to relasjonene! Du kan bruke JOIN uten nøkler, men som regel blir nøklene brukt for å skape logiske sammenslåinger.

Merk: når vi legger til et krav i join-betingelsen betyr det at vi legger en større begrensning på hvilke to tupler som vil slås sammen. Når vi legger til en JOIN operator betyr det at vi legger til enda en tuple i sammenslåingen.

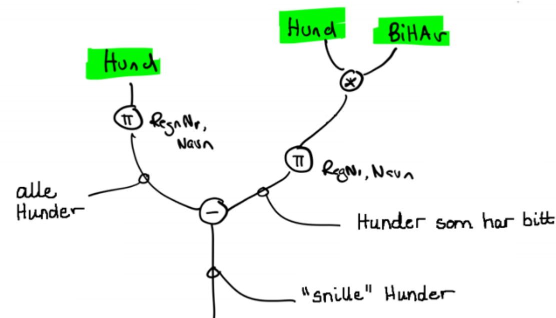


Dette kan også gjøres ved å ha flere join betingelser, slik at vi slipper seleksjonen. I dette tilfellet vil en Hund tuple slås sammen med Bittav tuplen kun hvis RegNr er likt og Eier_Pnr = Pnr. Vi får:

$$\pi_{\text{RegNr,Navn}}(\text{Hund} \bowtie_{\text{RegNr=RegNr AND Eier_Pnr=Pnr}} \text{Bittav})$$

Q4: Finn RegNr og Navn for alle hunder som ikke har bitt noen

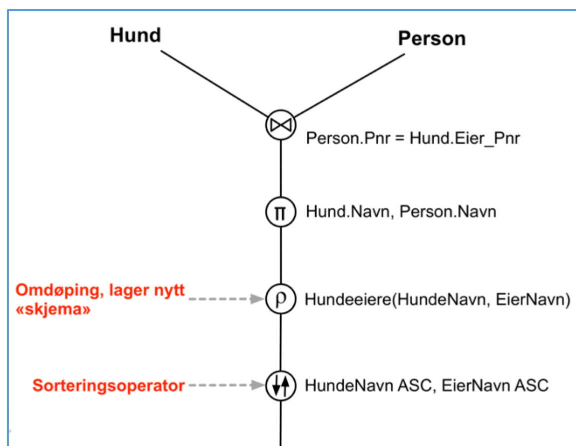
Vi må velge ut alle hunder og deretter fjerne de som har bitt noen. Dette kan vi gjør vha minus operatoren, men den krever at relasjonene er union-kompatible.



$$\pi_{\text{RegNr,Navn}}(\text{Hund}) - \pi_{\text{RegNr,Navn}}(\text{Hund} * \text{Bittav})$$

Her henter vi ut RegNr og Navn først, slik at vi får to union-kompatible relasjoner som vi så kan sende inn i en SET DIFFERENCE operator.

Omdøping og sortering



RENAME operatoren betegnes med ρ og brukes for å gi nye navn på tabeller og kolonner. På figuren bruker vi $\rho_{\text{Hundeeiere(HundeNavn,EierNavn)}}$ som vil sette relasjonsnavnet til Hundeeiere og attributtnavnene til HundeNavn og EierNavn. Denne operatoren lager dermed et nytt skjema.

ORDER BY operatoren betegnes med ↓↑ og brukes for å sortere attributtverdiene i stigende (ASC) eller synkende (DESC) rekkefølge. Verdiene må ha en orden, for eksempel nummer, dato eller strenger (alfabetisk). Dette vil ikke endre på innholdet, men gjør at det blir presentert på en annen måte. Derfor blir denne operatoren utført ved enden av uttrykket.

Aggregering og gruppering

Aggregeringsoperatorer gjør at vi kan regne ut ting basert på dataen vi har i databasen, for eksempel summere, regne ut gjennomsnittet, finne maks og min verdier, telle antall rader, osv. For COUNT operatoren har vi ulike varianter:

- COUNT(⟨attributt⟩) – teller antall verdier, unntatt NULL
- COUNT(DISTINCT ⟨attributt⟩) – teller antall ulike verdier, unntatt NULL
- COUNT(∗) – teller antall rader

Grupperingsattributtene definerer en partisjonering av radene i tabellen (husk: partisjonering vil si at alle tupler vil være i nøyaktig én gruppe). Aggregerings-operatoren vil operere på alle radene i hver partisjon og hver gruppe bidrar til én rad i resultattabellen. Hvis det ikke er gitt grupperingsattributter vil aggregeringen operere på alle radene i tabellen.

I tekst bruker vi γ som operator, mens på grafene bruker vi F.

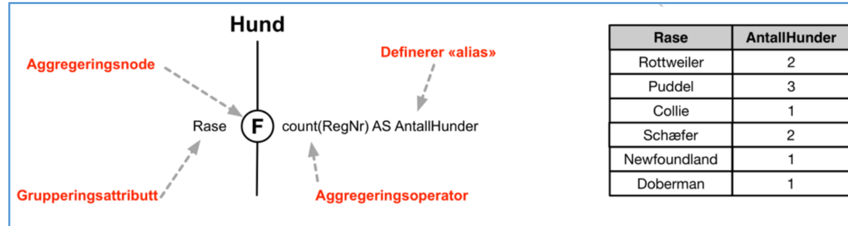
Merk: resultattabellen til en gruppering og aggregering vil inneholde grupperingsattributtene og resultatet av aggregeringen. Den vil ikke bestå av noen flere kolonner!

Q1: Finn antall hunder per rase

Vi må gruppere Hund tuplene etter rase og deretter bruke COUNT på RegNr (telling av primærnøkkel sikrer unike tupler):

$$\rho_{\text{Rase,AntallHunder}}\left({}_{\text{Rase}}\gamma_{\text{Count RegNr}}(\text{Hund})\right)$$

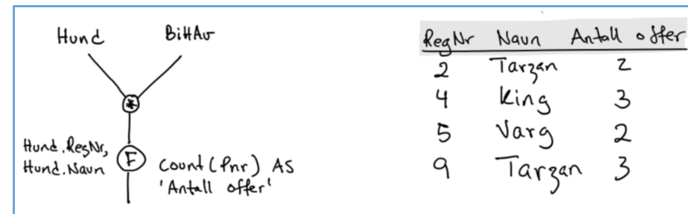
Her bruker vi ρ for å gi navn til attributtet som blir resultatet etter Count. Merk at i diagrammet kan vi gjøre dette vha AS. Grupperingsattributtet gis før F, mens aggregatfunksjonene gis etter. Legg merke til at **grupperingsattributtet blir med i resultatet**!



Rase	AntallHunder
Rottweiler	2
Puddel	3
Collie	1
Schæfer	2
Newfoundland	1
Doberman	1

Q2: Finn RegNr, Navn og antall offer per gjerningshund

Vi må slå sammen Hund og Bittav slik at vi finner hundene som har bitt noen. Deretter må vi gruppere tuplene etter RegNr og Navn, og bruke Count på Pnr for å telle antall personer hver hund har bitt:



RegNr	Navn	Antall offer
2	Tarzan	2
4	King	3
5	Varg	2
9	Tarzan	3

$$\rho_{\text{RegNr,Navn,Antall offer}}\left({}_{\text{RegNr,Navn}}\gamma_{\text{Count Pnr}}(\text{Hund} * \text{Bittav})\right)$$

For at Navn skal «overleve» aggregeringen og grupperingen må den tas med som grupperingsattributt.

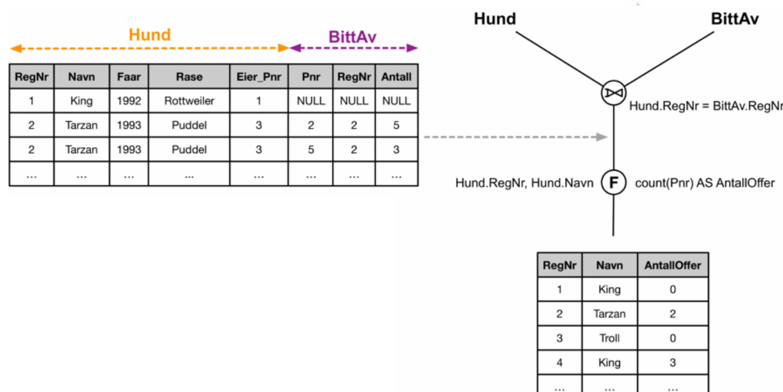
INNER JOIN vs. OUTER JOIN

Ved INNER JOIN er det kun tuplene som matcher som blir tatt med i resultattabellen. Den vil altså filtrere vekk rader som ikke oppfyller join betingelsen. Dersom man ønsker at resultattabellen skal inneholde alle tuplene fra en eller begge relasjonene må man i stedet bruke OUTER JOIN. Tupler som ikke matcher vil inkluderes i resultatet og attributter som «mangler data» blir fylt med NULL-verdier.

Q1: Finn RegNr, Navn og antall offer for *alle* hunder

Siden vi ønsker å finne RegNr, Navn og antall offer for alle hunder må vi bruke en OUTER JOIN som inkluderer alle Hund tupler uansett om de har bitt noen eller ikke:

$$\rho_{\text{RegNr,Navn,Antall offer}}\left({}_{\text{RegNr,Navn}}\gamma_{\text{Count Pnr}}(\text{Hund} ⟕_{\text{Hund.RegNr=Bitav.RegNr}} \text{Bittav})\right)$$



RegNr	Navn	Faar	Rase	Eier_Pnr	Pnr	RegNr	Antall
1	King	1992	Rottweiler	1	NULL	NULL	NULL
2	Tarzan	1993	Puddel	3	2	2	5
2	Tarzan	1993	Puddel	3	5	2	3
...	...	...	...	...	...	...	...

RegNr	Navn	AntallOffer
1	King	0
2	Tarzan	2
3	Troll	0
4	King	3
...	...	...

Restriksjoner i relasjonsalgebra

Det viktigste er at vi kan bruke relasjonsalgebra til spørring. Vi skal nå se at relasjonsalgebra også kan brukes til å uttrykke restriksjoner.

Entitetsintegritet

Entitetsintegritet sier at vi ikke kan ha to like rader i en tabell. Dette betyr at antall tupler med en bestemt verdi for primærnøkkelen må være én. For eksempel kan vi se på Person tabellen:

$$\sigma_{\text{Antall}>1}\left({}_{\text{Pnr}}\gamma_{\text{ Count(Navn) AS Antall}}\ (\text{Person})\right) = \emptyset$$

Her lager vi grupper for personnummer (nøkkel) og teller antall navn per gruppe. Vi kaller dette antallet for Antall. Deretter velger vi ut tuplene som har Antall > 1, og hvis det ikke er et tomt sett betyr det at flere personer har samme personnummer, noe som bryter med entitetsintegriteten (Pnr er primærnøkkel).

Referanseintegritet

Referanseintegritet sier at referansen mellom tupler i to ulike relasjoner må være riktig. Dette betyr at fremmednøklene i en relasjon må være et subsett av primærnøklene til relasjonen de referer til. For eksempel kan vi se på forholdene mellom Person, Bittav og Hund tabellene:

$$\pi_{\text{RegNr}}(\text{Bittav}) \subseteq \pi_{\text{RegNr}}(\text{Hund})$$
$$\pi_{\text{Pnr}}(\text{Bittav}) \subseteq \pi_{\text{Pnr}}(\text{Person})$$
$$\pi_{\text{EierPnr}}(\text{Hund}) \subseteq \pi_{\text{Pne}}(\text{Person})$$

Første uttrykk sier at det ikke kan eksistere et RegNr i Bittav som ikke er likt et RegNr i Hund.

Generelle restriksjoner

Relasjonsalgebra kan også brukes for å uttrykke andre generelle restriksjoner, for eksempel at en hund kan bite maks 10 personer:

$$\sigma_{\text{AntallOffer}>10}\left({}_{\text{RegNr}}\gamma_{\text{ Count(Pnr) AS AntallOffer}}\ (\text{Bittav})\right) = \emptyset$$

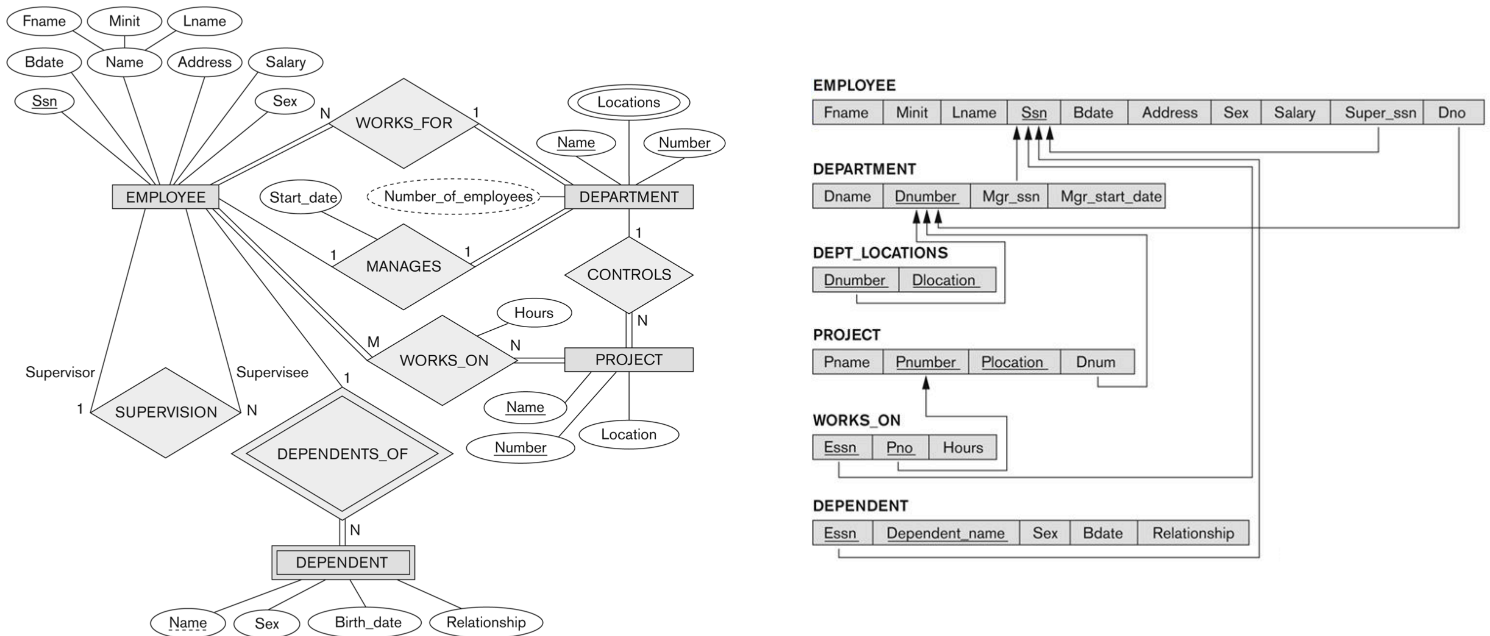
Det er forbudt å ha en databaseforekomst der spørringen over gir et resultat som ikke er tomt. Når vi setter inn en ny Bittav forekomst kan systemet sjekke denne betingelsen og nekte innsettingen dersom den brytes.

Kapittel 9 – Fra ER til Relasjonell database

Dette kapittelet beskriver hvordan et relasjonelt databaseskjema kan designes basert på et konseptuelt skjema (ER). Denne prosessen utgjør steg 3 i design av database (s. 13) og kalles **logisk design** eller **datamodell kartlegging**. ER og EER diagram blir ofte brukt for å grafisk utvikle skjemaet og samle informasjon om datatyper og begrensninger.

9.1 ER-til-relasjonell kartlegging

Denne seksjonen beskriver stegene i en algoritme som kartlegger et ER konseptuelt skjemadiagram til et relasjonelt databaseskjema. Vi bruker COMPANY databasen for å illustrere prosessen. Venstre figur viser ER skjemaet, mens høyre figur viser det relasjonelle databaseskjemaet som er resultatet av kartleggingen.



Steg 1 – Kartlegging av vanlige entitetsklasser

For hver vanlige, sterke entitetsklasse *E* i ER skjemaet, lag en relasjon *R* som har alle simple attributter i *E*. For sammensatte attributter blir kun de simple komponentattributtene inkludert. **Velg en av nøkkelattributtene til *E* som primærnøkkel i *R*.** Hvis den valgte nøkkelen i *E* er sammensatt, vil de simple komponentattributtene sammen utgjøre primærnøkkelen.

I vårt eksempel lager vi relasjonene EMPLOYEE, DEPARTMENT og PROJECT (se figur). Som vi kan se er ikke fremmednøkler, relasjonsattributter eller flerverdi attributter lagt til enda, for eksempel mangler EMPLOYEE Super_ssn og Dno. Ssn, Dnumber og Pnumber blir valgt som primærnøkler, og kunnskapen om at Dname og Pname er unike nøkler blir bevart for mulig senere bruk.

EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary

DEPARTMENT

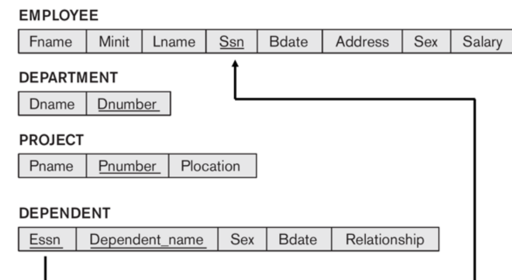
Dname	Dnumber

PROJECT

Pname	Pnumber	Plocation

Steg 2 – Kartlegging av svake entitetsklasser

For hver svake entitetsklasse W, lag en relasjon R som har alle simple attributter til W og fremmednøkkel som er primærnøkkelen(e) til identifiserende entitetsklasse. **Primærnøkkelen til R vil være kombinasjonen av primærnøklene til identifiserende entitetsklasse og delvis nøkkel til den svake entitetsklassen.** Hvis eieren er en svak klasse, må denne kartlegges først for å bestemme primærnøkkelen.

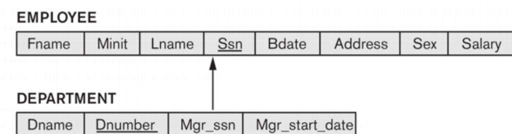


I vårt eksempel lager vi relasjonen DEPENDENT, som vil ha en fremmednøkkel som vi kaller Essn som er lik primærnøkkelen Ssn til EMPLOYEE relasjonen (vi kan også bruke samme navn!). Primærnøkkelen til DEPENDENT er kombinasjonen {Essn, Dependent_name}, siden Dependent_name er delvis nøkkel til den svake klassen.

Steg 3 – Kartlegging av binære 1:1 relasjonsklasser

For hver binære 1:1 relasjonsklasser i ER skjemaet, identifiser relasjonene S og T som deltar i denne relasjonen. Det er tre mulige tilnærminger:

1. **Fremmednøkkel – velg relasjonen S og legg til en fremmednøkkel som er primærnøkkelen til den andre relasjonen T**. Det er vilkårlig hvilken relasjon som har rollen til S, men det er **best om den har total deltagelse i relasjonsklassen** R (mindre NULL-verdier). Hvis relasjonsklassen har attributter blir disse lagt til i S. Vi kartlegger 1:1 relasjonsklassen MANAGER ved å bruke DEPARTMENT som S, siden den har total deltagelse (alle avdelinger har en manager). Derfor legger vi til Mgr_ssn som er primærnøkkelen til EMPLOYEE og Mgr_strt_date som er attributtet Start_date til MANAGER.

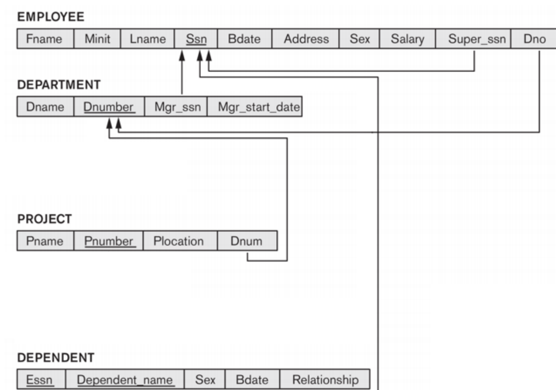


2. **Fusjonert relasjoner – hvis begge deltagelsene i relasjonsklassen er totale, kan vi slå sammen de to entitetsklassene og relasjonen til én enkelt relasjon**.
3. **Relasjon (kryssreferanse) – lag en egen relasjon som representerer relasjonsklassen og har primærnøklene til S og T som attributter**. Hver tuple i relasjonen vil relatere en tuple fra S til en tuple fra T. **Primærnøkkelen til relasjonen vil være en av fremmednøklene**. Ulempen er at vi har en ekstra relasjon og det krever flere JOIN operasjoner for å kombinere relaterte tupler. Brukes når det er få relasjonsinstanser.

Steg 4 – Kartlegging av binære 1:N relasjonsklasser

For hver binære 1:N relasjonsklasser i ER skjemaet, identifiser relasjonene S og T som deltar i denne relasjonen. Det er to mulige tilnærminger:

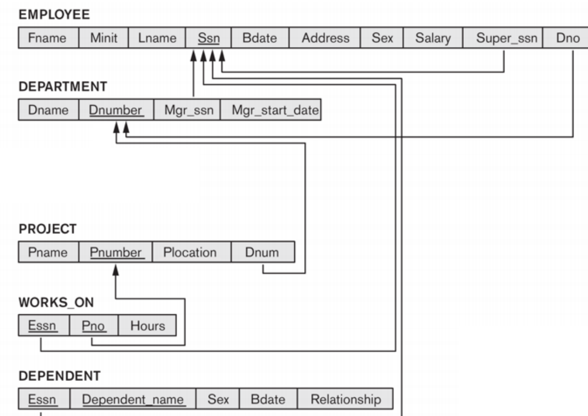
1. **Fremmednøkkel – identifiser relasjonen S som er ved N-siden av relasjonsklassen og legg til fremmednøkkel i S som er primærnøkkelen til T**. Dette blir riktig siden hver entitetsinstans på N-siden er relatert til maks én instans på 1-siden. Simple attributter til relasjonsklassen blir lagt til S. Vi bruker denne metoden for å kartlegge WORKS_FOR, CONTROLS og SUPERVISION. For WORKS_FOR blir Dno lagt til EMPLOYEE, for CONTROLS blir Dno lagt til PROJECT og for SUPERVISOR blir Super_ssn lagt til EMPLOYEE.



2. **Relasjon (kryssreferanse) – lag en egen relasjon som representerer relasjonsklassen og har primærnøklene til S og T som attributter**. Hver tuple i relasjonen vil relatere en tuple fra S til en tuple fra T. **Primærnøkkelen til relasjonen vil er primærnøkkelen til S ved N-siden**. Denne tilnærmingen brukes når få tupler i S deltar i relasjonen, slik at vi unngår mye NULL-verdier.

Steg 5 – Kartlegging av binære M:N relasjonsklasser

For hver binær M:N relasjonsklasse R, lag en ny relasjon S med fremmednøkler som er lik primærnøklene til deltakende entitetsklasser. **Primærnøkkelen til S er kombinasjonen av fremmednøklene**. Simple attributter til relasjonsklassen blir lagt til som attributter i S.



I vårt eksempel må vi lage en egen relasjonsklasse for WORKS_ON som har fremmednøkler Pno og Essn som er lik primærnøklene til hhv. PROJECT og EMPLOYEE. Merk: omdøping av nøkler er ikke et krav, men et design valg! Vi legger også til Hours som er attributtet til relasjonsklassen. Primærnøkkelen til WORKS_ON er kombinasjonen av fremmednøklene {Essn, Pno}.

Fremmednøkkel vs. relasjon (kryssreferanse)

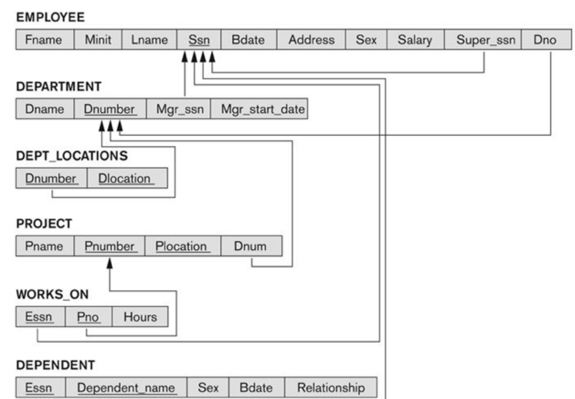
For M:N relasjonsklasser er altså relasjon (kryssreferanse) den eneste tilnærmingen. Som vi har sett kan dette også brukes for 1:1 og 1:N relasjonsklasser. **Dette er kun anbefalt når det eksisterer få relasjonsinstanser, fordi da vil denne tilnærmingen redusere antall fremmednøkler med NULL-verdi**. Hvis dette er tilfellet vil primærnøkkelen til relasjonsklassens relasjon være kun én av fremmednøklene (vilkårlig for 1:1 og entitetsklassen på N-siden for 1:N).

Steg 6 – Kartlegging av flerverdi attributter

For hver flerverdi attributt A, lag en ny relasjon R som har et attributt som korresponderer til A og en fremmednøkkel som er lik primærnøkkelen til entitets- eller relasjonsklassen som A tilhører. **Primærnøkkelen til R er kombinasjonen av A og fremmednøkkelen.** Hvis flerverdi attributtet er sammensatt blir de simple komponentene inkludert.

I vårt eksempel lager vi relasjonen DEPT_LOCATIONS som har attributtet Dlocation som representerer flerverdi attributtet LOCATIONS i DEPARTMENT og Dnumber som er lik primærnøkkelen til DEPARTMENT. Primærnøkkelen til DEPT_LOCATIONS er kombinasjonen av {Dnumber, Dlocation}. Merk: Flere moderne relasjonsmodeller tillater datatyper som er arrays, slik at man slipper separate relasjoner.

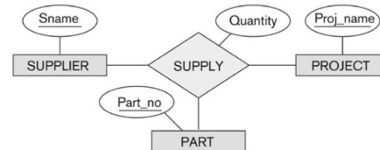
På figuren kan vi se det endelige relasjonelle databaseskjemaet for COMPANY databasen.

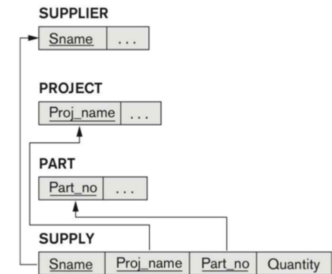


Steg 7 – Kartlegging av N-ære relasjonsklasser

For hver n-ær relasjonsklasse R, der $n > 2$, lag en ny relasjon S som har fremmednøkler som er lik primærnøklene til deltagende entitetsklasser. Simple attributter til relasjonsklassen blir også lagt til i S. Primærnøkkelen til S er en kombinasjon av alle fremmednøklene. Hvis kardinalitet begrensningen til en deltagende entitetsklasse er 1, vil ikke primærnøkkelen til S inkludere fremmednøkkelen som refererer til denne entitetsklassen.

Figurene viser et eksempel på en slik kartlegging. Vi lager en egen relasjon for den ternære relasjonsklassen SUPPLY, og denne relasjonen vil ha fremmednøkler som er lik primærnøklene til SUPLLIER, PROJECT og PART. I tillegg legger vi til Quantity som er en attributt til relasjonsklassen. Siden ingen av deltakende entitetsklasser har kardinalitet 1, vil primærnøkkelen til SUPPLY være kombinasjonen av {Sname, Part_no, Proj_name}.





Diskusjon og summering av kartlegging

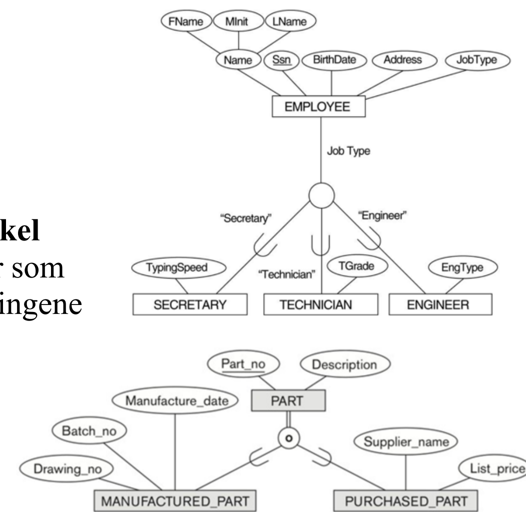
Tabellen under summerer kartleggingen fra ER til relasjonsmodell:

ER-modell	Relasjonsmodell
Entitetsklasse	Entitetsrelasjon
1:1 eller 1:N relasjonsklasse	Fremmednøkkel (eller relasjon)
M:N relasjonsklasse	Relasjon og to fremmednøkler
N-ær relasjonsklasse	Relasjon og n fremmednøkler
Simpel attributt	Attributt
Sammensatt attributt	Sett av simple komponentattributter
Flerverdi attributt	Relasjon
Nøkkelattributt	Primærnøkkel

En av de viktigste forskjellene er at relasjonsklasser i ER-modellen blir ikke representert eksplisitt i relasjonsmodellen. I stedet blir de representert vha to attributter, der en er primærnøkkelen og den andre er fremmednøkkelen i hver sin relasjon. To tupler er relaterte når de har samme verdi for disse to attributtene. Når fremmednøkler blir brukt kan vi kombinere relaterte tupler med én JOIN-operator, mens ved relasjon (kryssreferanse) må vi bruke to.

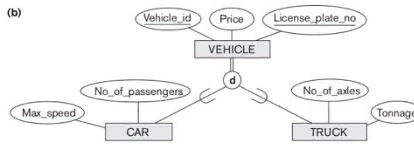
Det er viktig å være klar over fremmednøklene, slik at de kan brukes riktig når relaterte tupler skal kombineres fra to eller flere relasjoner. Dette er ulempen ved relasjonsmodellen, fordi fremmednøkkel/primærnøkkel korrespondensen er ikke alltid like åpenbar når man ser på databaseskjemaet. Hvis man bruker EQUIJOIN på to attributter som ikke har et nøkkelforhold, vil resultatet ofte bli meningsløst og kan føre til misledende data. **En annen ulempe ved relasjonsmodellen er at flerverdi attributter må spesifiseres i en egen relasjon**. Når denne relasjonen slås sammen med «eier» relasjonen vil vi få mye duplikat data, siden de andre attributtene i «eier» relasjonen må gjentas for hver verdi av flerverdi attributtet.

9.2 EER-til-relasjonell kartlegging



Kartlegging av spesialisering eller generalisering

De to vanligste måtene å kartlegge en spesialisering er å lage en enkel tabell eller flere tabeller. Innenfor begge disse er det flere variasjoner som avhenger av begrensningene til spesialiseringen. De vanligste kartleggingene beskrives i steg 8 av algoritmen, og vi bruker spesialiseringen på figurene for å illustrere de ulike fremgangsmåtene.



Steg 8 – Kartlegging av spesialisering og generalisering

For hver spesialisering med subklasser og superklasse, bruk en av følgende metoder:

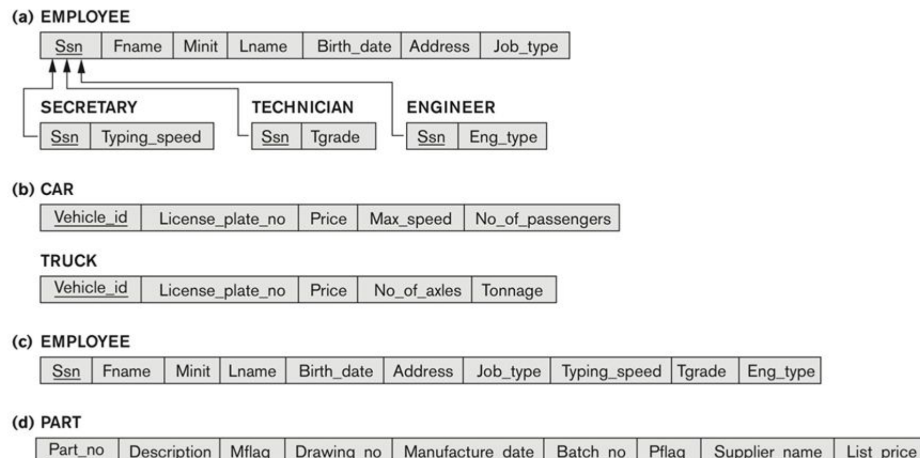
a) **Flere relasjoner for superklasse og subklasse** – lag en relasjon for superklassen med dens attributter og bruk nøkkelen (k) som primærnøkkel. Lag også en relasjon for hver av subklassene, der hver relasjon vil ha attributter som er spesifikke for subklassen og nøkkelattributtet (k) til superklassen. Primærnøkkelen til subklassene er k. **Brukes ved alle typer spesialisering** (total, delvis, overlappende og disjunkt).

b) **Flere relasjoner for subklasser** – lag en relasjon for hver av subklassene, der hver relasjon vil ha alle attributtene til superklassen, attributtene som er spesifikke for subklassen og nøkkelattributtet (k) til superklassen. Primærnøkkelen til subklassene er k. **Brukes ved total og disjunkt spesialisering** (overlapping = duplikate tupler).

c) **Enkel relasjon med en typeattributt** – lag én relasjon som har attributtene til superklassen og alle subklassene. Vi legger til **én typeattributt** t som har en verdi (eks: mellom 1 og m) som indikerer hvilken subklasse hver tuple hører til (hvis ingen, er den NULL). Brukes ved disjunkt spesialisering, og kan generere mange NULL-verdier.

d) **Enkel relasjon med flere typeattributter** – lag én relasjon som har attributtene til superklassen og alle subklassene. Vi legger til flere **typeattributter** $\{t_1, t_2, \dots, t_m\}$ som er boolean attributter som indikerer om en tuple hører til subklasse i eller ikke. Brukes ved overlappende spesialisering (fungerer også for disjunkt).

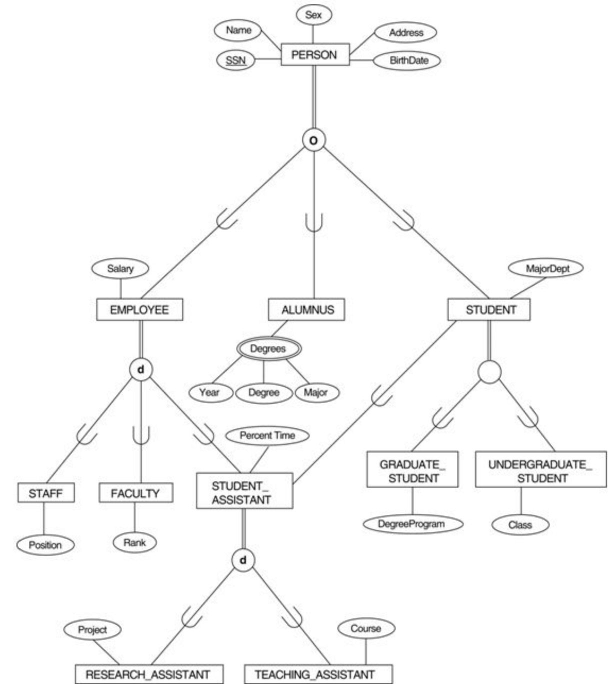
Legg merke til at for metode c og d vil entiteter som ikke hører til bestemt subklasser ha NULL-verdi for attributter som er spesifikke for disse subklassene. **Metode c og d er derfor ikke anbefalt dersom subklassene har mange spesifikke attributter**. Fordelen med disse er at de ikke krever JOIN operatorer og kan derfor være mer effektive i spørringer. Hvis spesialiseringen er attributtdefinert, vil dette attributtet fungere som typeattributt (for eksempel Job_type på figur c). **Dersom vi har en spesialisering med flere nivåer, kan vi bruke ulike metoder for de ulike nivåene.**

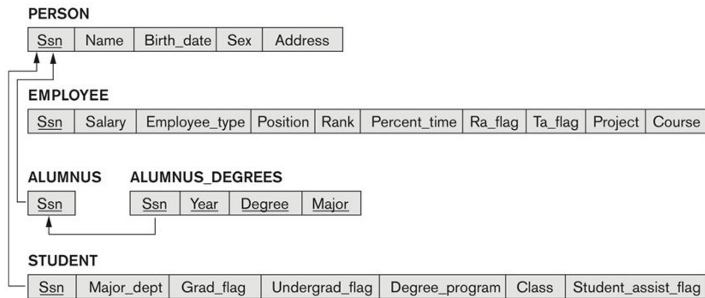


Kartlegging av delte subklasser

En delt subklasse (eks: STUDENT-ASSISTANT) er en subklasse som arver fra flere superklasser. Disse klassene må ha samme nøkkelattributt, for ellers vil den delte subklassen modelleres som en kategori (union type).

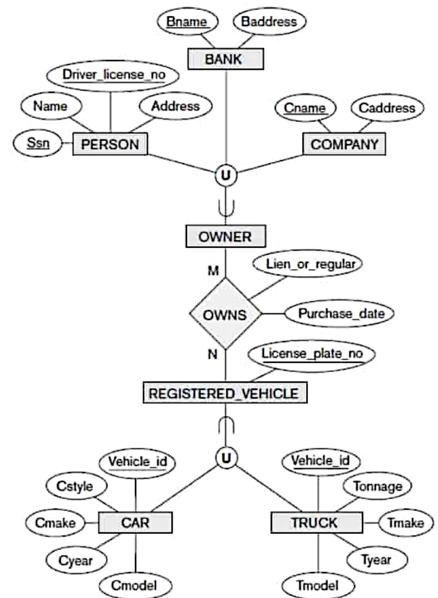
Vi kan bruke alle metodene diskutert på forrige side for å kartlegge en delt subklasse. Figuren under viser en mulig kartlegging som bruker metode c på EMPLOYEE relasjonen og metode d på STUDENT relasjonen:





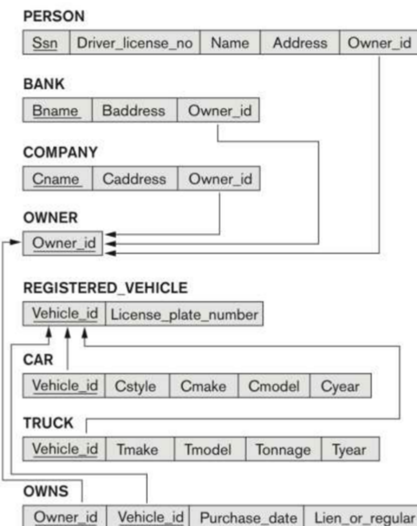
Kartlegging av kategorier (union typer)

En kategori (union type) er en subklasse til unionen av to eller flere superklasser som kan ha ulike nøkler siden de kan være av ulike entitetsklasser. Figuren til høyre viser to kategorier: OWNER og REGISTERED_VEHICLE. Vi kan se at OWNER er et subsett av unionen til tre entitetsklasser: PERSON, BANK og COMPANY som alle har ulike nøkler.



Steg 9 – Kartlegging av kategorier (union typer)

Når vi skal kartlegge en kategori, må vi skille mellom når definerende superklasser har like og ulike nøkler. I begge tilfeller lager vi en egen relasjon for kategorien og fyller den med eventuelle attributter som er spesifikke for kategorien. **Hvis superklassene har like nøkler, legger vi til dette nøkkelattributtet i relasjonen til kategorien** (se på REGISTERED_VEHICLE). Hvis nøklene er ulike, lager vi et nytt nøkkelattributt, kalt **surrogat nøkkel** og legger til denne i relasjonen (se på OWNER). Denne nøkkelen må også legges til som fremmednøkkel i alle definerende superklasser. Disse er NULL dersom en superklasse entitet ikke er medlem i kategorien. Det er også vanlig å legge til et type-attributt (ikke på figuren) som spesifiserer hvilken superklasse kategori entiteten hører til.



Notasjon:

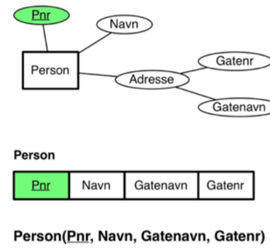
Pnr – Nøkkel-attributt (del av nøkkel)

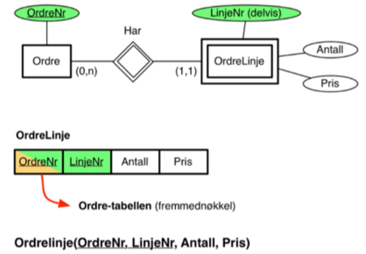
OrdreNr – Fremmednøkkel-attributt (del av fremmednøkkel)

Nasjonsnavn – Både nøkkel-attributt og fremmednøkkel-attributt

Oppsummering – Kapittel 9 (F5)

Figuren til høyre viser notasjonen for reglene

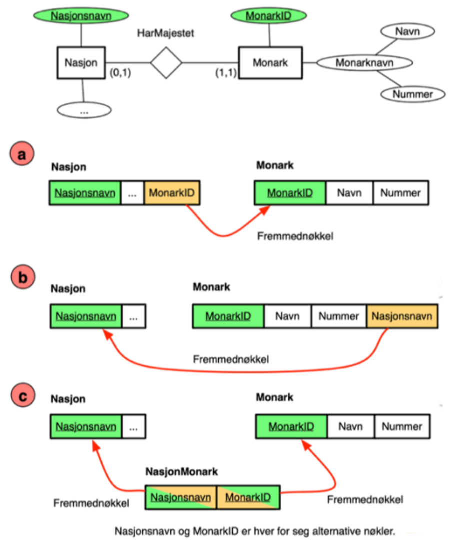




For entitetsklasser er det avgjørende om de er sterke eller svake:

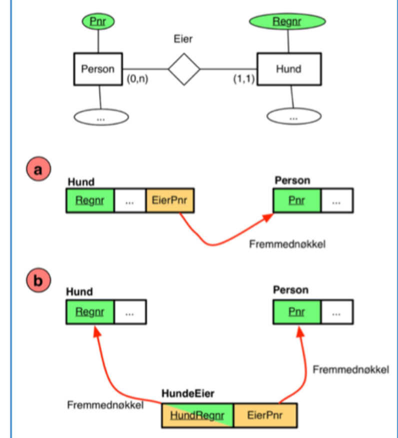
a) **Sterk (regulær) entitetsklasse** – relasjonen inneholder alle en-verdi attributtene til entitetsklassen. For sammensatte attributter blir kun «løv-attributtene» beholdt.

b) **Svak entitetsklasse** – relasjonen til den svake klassen vil inneholde alle en-verdi attributtene, delvis nøkkelattributt og alle nøkkelattributtene til identifiserende entitetsklassen. Nøkkelattributtet til den svake klassen vil være delvis nøkkel og nøkkelattributtene til den sterke klassen, som også vil være en fremmednøkkel.



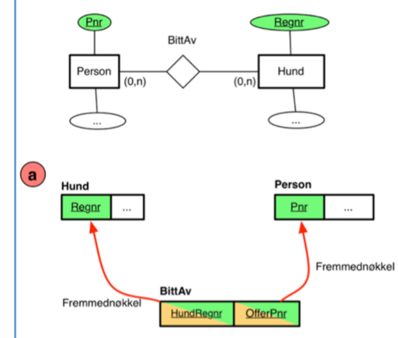
For relasjonsklasser er kardinaliteten avgjørende:

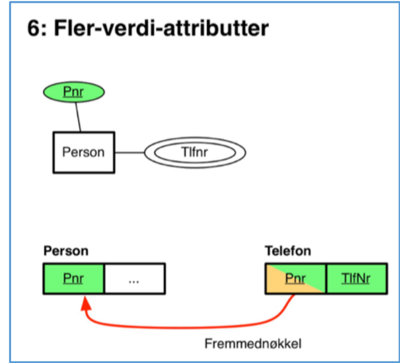
c) **Binære 1:1-relasjonsklasser** – representeres ved å legge til en fremmednøkkel i en av tabellene (alternativ a og b) eller ved å lage en egen koblingstabell (alternativ c). Hvis vi plasserer fremmednøkkelen på (0, 1) siden kan den være NULL, siden det betyr at det er ingen relasjon. Hvis vi plasserer fremmednøkkelen på (1, 1) siden kan den ikke være NULL, fordi det betyr at primærnøkkelen må være NULL. Vi velger den løsningen som er minst kompleks og gir minst NULL-verdier (b på figur). Alternativ c kan være nyttig når begge relasjonene er (0, 1) og det er få relasjoner mellom entitetsklassene, fordi da kan alternativ a og b føre til mange NULL verdier. Hvis relasjonsklassen har attributter, må disse legges til i samme tabell som fremmednøkkelen.



d) **Binære 1:N-relasjonsklasser** – representeres ved å legge til en fremmednøkkel i tabellen på (1, 1)-siden (dvs. N-siden ved 1:N) eller ved å lage en egen koblingstabell. Når vi lager koblingstabellen vil kun fremmednøkkelen til N-siden være primærnøkkel, fordi ellers kan vi få flere like primærnøkler. På figuren kan ikke EierPnr være NULL siden hunden må ha en eier (det er en (1,1) relasjon).

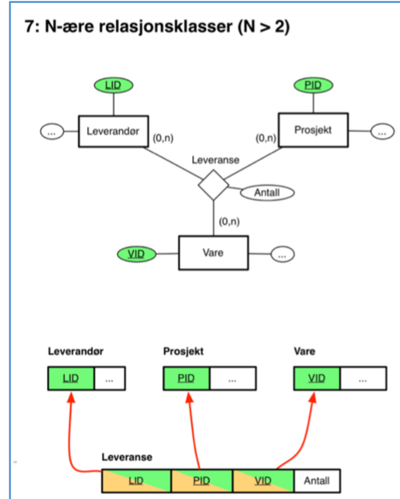
e) **Binære N:M-relasjonsklasser** – representeres ved å lage en koblingstabell. Fremmednøklene utgjør tilsammen primærnøkkelen. Dersom relasjonsklassen har attributter tas disse med i koblingstabellen.





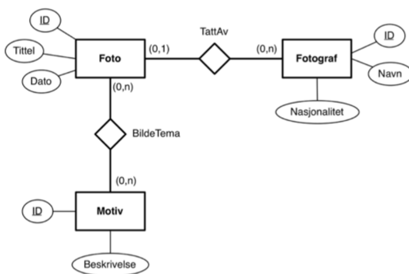
Noen andre regler er:

f) **Flerverdi attributter** – må som regel håndteres i en egen tabell, som har en fremmednøkkel til entitetstabellen. Hvis det er mulig vil vi droppe flerverdi attributter, siden det bruker mer plass. Det kan være flere muligheter for primærnøkkelen, og det avhenger av tilstanden i den virkelige verden. På figuren kan primærnøkkelen være TlfNr hvis det er én person per telefonnummer, mens den må være TlfNr og Pnr hvis flere personer kan dele telefonnummer (må bruke Pnr for å skille mellom personene som har samme TlfNr). Pnr kan ikke være primærnøkkel alene, fordi det betyr at det er kun et telefonnummer per person, noe som ikke stemmer med at TlfNr er flerverdi.



g) **N-ære relasjonsklasser ($N > 2$)** – må lage en koblingstabell som kobler sammen entitetsklassene som inngår i relasjonen. Eventuelle attributter til relasjonen blir også lagt til. Primærnøkkelen blir kombinasjonen av primærnøklene til entitetsklassene i relasjonen. De kan ikke være NULL siden de er primærnøkler i andre tabeller.

Under ser vi en ER modell som blir oversatt til en relasjonsmodell:



Fotograf (ID, Navn, Nasjonalitet)

kan være NULL

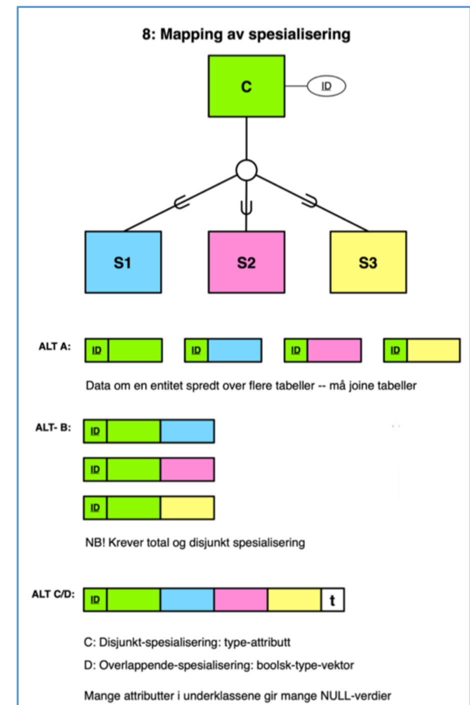
Foto (ID, Tittel, Dato, FotografID)

NOT NULL

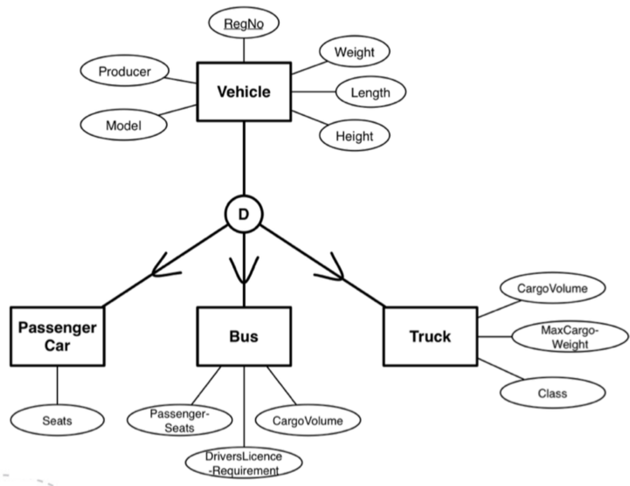
BildeTema (FotoID, MotivID)

NOT NULL

Motiv (ID, Beskrivelse)



h) **Mapping av spesialisering/generalisering** – det finnes flere måter å gjøre det på. Husk at alle entiteter i subklassen er også en entitet i superklassen. Primærnøkkelen til superklassen vil derfor gjelde nedover i hierarkiet (dvs. den brukes som primærnøkkel for subklassene). I alternativ A lager vi relasjoner for alle entitetsklassene. Hver subklasse vil ha primærnøkkelen til superklassen og alle attributtene som er spesifikk for den klassen. I alternativ B lager vi relasjoner for alle subklassene, og legger til superklassens attributter til disse. Superklassen har ingen egen relasjon, og dette krever total og disjunkt spesialisering (*må* være i én subklasse). I alternativ C vil vi lage en relasjon som inneholder alle attributtene til superklassen og subklassene. I tillegg legger vi til en attributt som sier hvilken subklasse entiteten er i (typeattributt ved disjunkt spesialisering og boolsk vektor ved overlappende). Dette alternative er enkelt, men kan føre til mange NULL-verdier.



A)

Vehicle (RegNo, Producer, Model, Weight, Length, Height) – kan legge til type-attributt

Passenger Car (Regno, Seats)

Bus (Regno, Passenger Seats, CargoVolume, Drivers Licence Req)

Truck (Regno, CargoVolume, Max CargoWeight, Class)

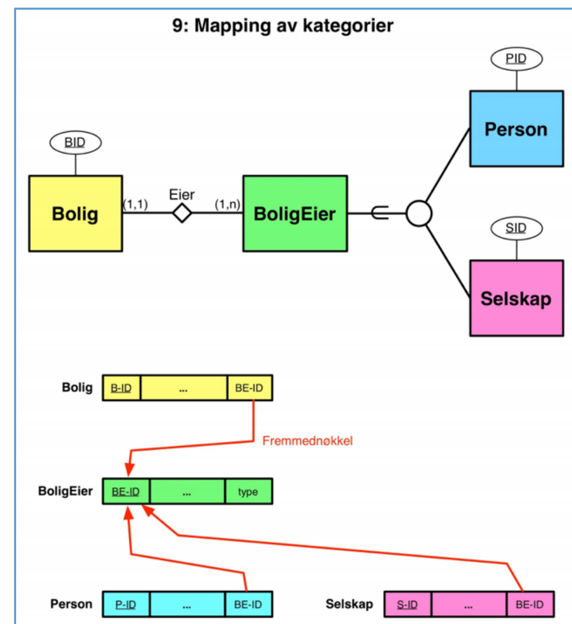
C)

Vehicle (Regno, <alle attributter i alle klasser>, TypeOfVehicle)
└ verdi: Passenger Car, Bus, Truck or NULL.

Her er det kun alternativ A og C som er brukt, siden spesialiseringen ikke er total.

- Alternativ A – fordelen med denne måten er at det blir veldig klart for databasen hvilke egenskaper som gjelder for de ulike subklassene. Ulempen er at vi må hente data fra to tabeller for å få all informasjon om noen kjøretøy. Vi kan legge til et typeattributt til Vehicle, slik at det blir lettere å finne ut hvilken subklasse vi skal hente data fra.
- Alternativ C – fordelen med denne måten er at vi trenger kun én tabell. Ulempen er at det blir uklart hvilke egenskaper som gjelder for de ulike subklassene, så dette må dokumenteres ved siden av (spesialiseringen «forsvinner»). Det vil også føre til mange NULL-verdier, siden spesialiseringen er disjunkt. Legg merke til typeattributtet som kan være NULL dersom kjøretøyet kun er i superklassen.

Kartlegging av kategorier



På figuren ser vi at BoligEier er en kategori av superklassene Person og Selskap. For å kartlegge denne kategorien, lager vi en egen relasjon til BoligEier og fyller den med attributter som er spesifikke for kategorien. Siden superklassene har ulike nøkler må vi lage en **surrogatnøkkel** som må legges til som fremmednøkkel i superklassene. Vi legger også til et typeattributt som forteller om kategori entiteten hører til Person eller Selskap.

Surrogatnøkkel er en nøkkel som vi definerer i relasjonsmodellen, men som ikke trengs i ER-modellen. I ER-modellen vil BoligEier arve PID hvis det er en Person, men for at det skal fungere i relasjonsmodellen må BoligEier ha en egen nøkkel. Derfor må vi lage en nøkkelattributt som ikke er definert i mini-verden.

Del 4 – Databasedesign og Normalisering

Kapittel 14 – Funksjonelle avhengigheter og normalisering

Frem til nå har vi antatt at attributter blir gruppert sammen for å danne et relasjonsskjema ved at databasedesigneren bruker sunn fornuft eller ved kartlegging fra en konseptuelt datamodell som ER eller EER. Vi trenger **en formell måte å analysere hvorfor en gruppering av attributter til et relasjonsskjema kan være bedre enn en annen**. Dette vil beskrive **kvaliteten til designet**, og dette kapittelet vil beskrive teorien som trengs for å evaluere denne kvaliteten. Det er to nivåer der vi kan diskutere hvor «bra» et relasjonsskjema er:

1. **Logisk nivå (konseptuelt)** – hvordan brukere tolker relasjonsskjema og betydningen til deres attributter. Hvis relasjonsskjemaet er bra på dette nivået vil det bli lettere for brukerne å forstå hva dataen i relasjonen betyr og dermed lettere å formulere spørringer.
2. **Implementasjonsnivået (fysisk lagring)** – hvordan tupler i baserelasjonen er lagret og oppdateres. Dette nivået gjelder kun for baserelasjonene som er fysisk lagret som filer (dvs. ikke views).

Database designteorien utviklet i dette kapitlet gjelder først og fremst for baserelasjoner, men noen kriterier brukes også på views.

Databasedesign kan utføres med to tilnærminger:

1. **Bottom-up metode** - grunnleggende forhold mellom individuelle attributter blir sett på som startpunktet og disse brukes for å lage relasjonsskjema. Denne tilnærmingen er ikke populær siden den krever at man må samle et stort antall binære relasjoner i starten, og det er nesten umulig å fange alle forhold mellom attributtpar.
2. **Top-down metode** – databasen blir designet ved å analysere og dekomponere et sett av attributter i en universell relasjon som naturlig eksisterer i mini-verden. Dette blir gjort helt til alle ønskede egenskaper oppnås.

Teorien i dette kapittelet er rettet mot top-down metoden.

Relasjonell databasedesign vil produsere et sett av relasjoner, og **målet ved designet er å bevare informasjon og minimere redundans**. Bevaring av informasjon vil si å ta vare på alle konsepter som blir beskrevet i en EER konseptuell modell, inkludert attributter, entitetsklasser, relasjonsklasser og spesialisering/generalisering. Minimering av redundans innebærer å ha mindre lagring av samme informasjon og redusere behovet for flere oppdateringer, som kreves dersom databasen har flere kopier av samme informasjon.

14.1 Uformelle retningslinjer for design av relasjonsskjema

Fire uformelle retningslinjer som kan brukes å bestemme kvaliteten til et relasjonsskjema design:

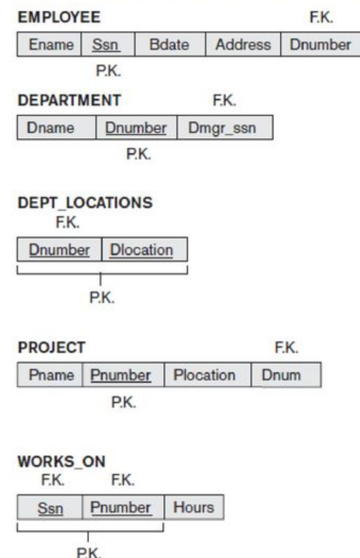
1. Sørge for at semantikken til attributtene er tydelig (semantikk = språkets innhold)
2. Redusere overflødig informasjon i tupler
3. Redusere NULL-verdier i tupler
4. Ikke tillate muligheten til å generere falske tupler

Vi skal nå se nærmere på disse.

1. Tydelig semantikk

Når vi grupperer attributter for å danne et relasjonsskjema, antar vi at attributtene har en logisk tolkning og hører til en relasjon som har en bestemt betydning i den virkelige verden. **Når vi tolker attributtverdiene til en tuple kan vi få en forståelse av meningen til relasjonen, og dette kalles semantikken til relasjonen**. Hvis det konseptuelle designet blir godt gjennomført og kartleggingen blir fulgt systematisk, bør relasjonsskjemaet ha en tydelig mening.

Jo enklere det er å forklare semantikken til relasjonen, altså hva relasjonen betyr og representerer, desto bedre er relasjonsskjemaet. **Hvor lett det er å forklare betydningen til relasjonens attributter er altså et uformelt mål på hvor godt relasjonen er designet.** For eksempel har EMPLOYEE relasjonsskjemaet en enkel betydning, fordi hver tuple representerer en ansatt med verdier for navn, personnummer, bursdagsdato, adresse og avdelingen den arbeider ved. Dnumber attributtet er en fremmednøkkel som representerer et implisitt forhold mellom EMPLOYEE og DEPARTMENT. Det er vanskeligere å forstå betydningen av WORKS_ON, der hver tuple vil gi personnummeret til den ansatte, prosjektnummeret til *en av* prosjektene den ansatte arbeider på og antall timer per uke den ansatte arbeider på dette prosjektet. Dette skjemaet har likevel en veldefinert og entydig tolkning. Alle skjemaene på figuren er lette å forklare og har dermed godt design basert på semantikken.



Retningslinje.1

Design et relasjonsskjema slik at det er lett å forklare dens betydning. Ikke kombiner attributter fra flere entitets- og relasjonsklasser inn i én relasjon, fordi da blir det vanskeligere å forklare meningen til relasjonen.

Figuren viser et relasjonsskjema som bryter denne retningslinjen. Hver tupler representerer en ansatt og gir også avdelingsnavnet og personnummeret til avdelingens manager. Det er ikke noe logisk galt med denne relasjonen, men den bryter retningslinje 1 fordi den blander attributter fra ulike entiteter i den virkelige verden, nemlig ansatt og avdeling. Mht. semantikk har den derfor dårlig design. Den kan brukes som view, men kan forårsake problemer hvis den brukes som baserelasjoner (pga. overflødig lagring).

2. Overflødig informasjon i tupler

Et mål ved skjemadesign er å minimere lagringsplassen som brukes av baserelasjonene. Hvordan attributter grupperes kan ha en signifikant effekt på bruk av lagringsplass. For eksempel kan vi sammenligne EMPLOYEE og DEPARTMENT relasjonene med EMP_DEPT relasjonen. I EMP_DEPT vil verdiene til avdelingsattributtene (Dnum, Dname, Dmgs_ssn) for en bestemt avdeling gjentas for alle ansatte som arbeider ved den avdelingen. I DEPARTMENT vil denne informasjon kun gis én gang og kun Dnum blir gjentatt i EMPLOYEE.

Dette er et eksempel på et problem ved lagring av naturlige joins som baserelasjoner. Et annet problem ved dette er **oppdateringsanomalier**, som deles inn i:

- **Innsettingsanomali** – ved innsetting av tupler i naturlig joins kan det være problematisk å sikre konsistens og at begrensninger ikke brytes. For å sette inn en ny ansatt i

Merk:
Anomali = uregelmessighet

EMP_DEPT må vi inkludere attributtverdiene til avdelingen som den ansatte arbeider ved eller NULL-verdier. Hvis vi setter inn verdiene til en avdeling må vi passe på at disse er konsistent med verdiene i andre tupler som representerer ansatte ved samme avdeling. For EMPLOYEE slipper vi å tenke på dette, siden det er kun Dnum som oppgis og resten av informasjonen ligger i én tuple i DEPARTMENT. For å sette inn en ny avdeling som ikke har noen ansatte enda, må vi plassere NULL i attributtene for den ansatte. Dette bryter entitetsintegriteten som sier at Ssn ikke kan være NULL. Dette er ikke et problem i DEPARTMENT, fordi da vil vi kun lage en ny DEPARTMENT tuple.

- **Slettingsanomali** – sletting av tupler i naturlig joins kan føre til uønsket tap av informasjon. For eksempel hvis vi i EMP_DEPT sletter den siste ansatte som arbeider ved en bestemt avdeling, så vil informasjonen om avdelingen også bli tapt. Dette er ikke et problem i DEPARTMENT, siden disse tuplene lagres separat.
- **Oppdateringssanomali** – oppdatering av attributtverdier i naturlige joins, krever oppdatering av alle relaterte tupler. For eksempel hvis vi endrer manageren ved avdeling 5 i EMP_DEPT må vi oppdatere tuplene til alle ansatte som arbeider ved avdeling 5, fordi ellers vil databasen bli inkonsistent.

Anomalier kan altså føre til overflødig arbeid ved innsetting og modifisering av tupler i en relasjon og til uønsket tap av informasjon ved sletting av tupler

Retningslinje 2

Design skjemaet til baserelasjoner slik at det er ingen anomalier i innsetting, sletting eller modifisering av tupler i relasjonene. Hvis det er anomalier må disse markeres tydelig, slik at programmet som oppdaterer databasen gjør dette på riktig måte.

Det kan hende at disse retningslinjene må brytes for å forbedre ytelsen til bestemte spørringer. Anomali-frie baserelasjoner blir ofte kombinert med views som inkluderer joins for å slå sammen attributter som det ofte spørres om.

3. NULL-verdier i tupler

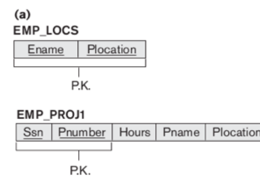
Hvis en relasjon har mange attributter som ikke gjelder for alle tupler, kan den inneholde mange NULL-verdier. Dette kan føre til bortkastet bruk av lagringsrom, problemer med forståelsen av attributtenes betydning og komplisert spesifisering av JOIN operatorer. Et annet problem er hvordan aggregatfunksjoner skal håndtere NULL-verdiene. SELECT og JOIN innebærer sammenligninger som kan bli uforutsigbare når NULL-verdier er tilstede. En NULL-verdi kan også ha flere ulike tolkninger (s. 49), som ikke blir spesifisert.

Retningslinje 3

Unngå å plassere attributter i baserelasjoner dersom de frekvent vil være NULL. Hvis det ikke er mulig å unngå, sørg for at NULL brukes i unntakstilfellet og ikke for de fleste tuplene. Argumenter for å ikke inkludere en kolonne med NULL verdier, er å sikre effektiv bruk av lagringsrom og unngå joins med NULL. For eksempel hvis kun 15% av ansatte har eget kontor, er det **bedre å lage en egen relasjon** EMP_OFFICES som kun inkluderer disse ansatte.

EMP_PROJ

Ssn	Pnumber	Hours	Ename	Pname	Plocation



(b)

EMP_LOCS

Ename	Plocation
Smith, John B.	Bellaire
Smith, John B.	Sugarland
Narayan, Ramesh K.	Houston
English, Joyce A.	Bellaire
English, Joyce A.	Sugarland
Wong, Franklin T.	Sugarland
Wong, Franklin T.	Houston
Wong, Franklin T.	Stafford
Zelaya, Alicia J.	Stafford
Jabbar, Ahmad V.	Stafford
Wallace, Jennifer S.	Stafford
Wallace, Jennifer S.	Houston
Borg, James E.	Houston

EMP_PROJ1

Ssn	Pnumber	Hours	Pname	Plocation
123456789	1	32.5	ProductX	Bellaire
123456789	2	7.5	ProductY	Sugarland
666884444	3	40.0	ProductZ	Houston
453453453	1	20.0	ProductX	Bellaire
453453453	2	20.0	ProductY	Sugarland
333445555	2	10.0	ProductY	Sugarland
333445555	3	10.0	ProductZ	Houston
333445555	10	10.0	Computerization	Stafford
333445555	20	10.0	Reorganization	Houston
999887777	30	30.0	Newbenefits	Stafford
999887777	10	10.0	Computerization	Stafford
987987987	10	35.0	Computerization	Stafford
987987987	30	5.0	Newbenefits	Stafford
987654321	30	20.0	Newbenefits	Stafford
987654321	20	15.0	Reorganization	Houston
888665555	20	NULL	Reorganization	Houston

4. Generering av falske tupler

Det er viktig å unngå generering av falske tupler som representerer falsk informasjon som ikke er gyldig. Figuren viser to relasjonsskjema EMP_LOCS og EMP_PROJ1 som kan brukes istedenfor EMP_PROJ. En tuple i EMP_LOCS gir navnet til den ansatte og lokasjonen til prosjektet den ansatte arbeider på. En tuple i EMP_PROJ1 gir personnummer til den ansatte og prosjektnummer, antall timer, navn og lokasjon til prosjektet den ansatte arbeider på.

Figur b viser en projeksjon av det originale settet av tupler i EMP_PROJ. Det vil være dårlig skjemadesign å bruke EMP_PROJ1 og EMP_LOCS som baserelasjoner, fordi vi kan ikke gjenvinne informasjonen som opprinnelig var i EMP_PROJ fra disse to. Hvis vi bruker en naturlig join, vil vi produsere flere tupler enn det var i det originale settet. Tabellen til venstre viser resultatet til sammenslåingen og * er brukt for å markere falske tupler som gir ugyldig informasjon. **Denne dekomponeringen av EMP_PROJ er altså dårlig design, fordi når vi bruker JOIN av disse vil vi ikke få tilbake den opprinnelige informasjonen**. Dette skyldes at Plocation er det eneste attributtet som kan brukes for å slå sammen EMP_LOCS og EMP_PROJ1, og det er ikke en primærnøkkel eller fremmednøkkel i noen av disse relasjonene. Dermed vil ikke join operatoren fungere riktig.

	Ssn	Pnumber	Hours	Pname	Plocation	Ename
	123456789	1	32.5	ProductX	Bellaire	Smith, John B.
*	123456789	1	32.5	ProductX	Bellaire	English, Joyce A.
	123456789	2	7.5	ProductY	Sugarland	Smith, John B.
*	123456789	2	7.5	ProductY	Sugarland	English, Joyce A.
*	123456789	2	7.5	ProductY	Sugarland	Wong, Franklin T.
	666884444	3	40.0	ProductZ	Houston	Narayan, Ramesh K.
*	666884444	3	40.0	ProductZ	Houston	Wong, Franklin T.
*	453453453	1	20.0	ProductX	Bellaire	Smith, John B.
	453453453	1	20.0	ProductX	Bellaire	English, Joyce A.
*	453453453	2	20.0	ProductY	Sugarland	Smith, John B.
	453453453	2	20.0	ProductY	Sugarland	English, Joyce A.
*	453453453	2	20.0	ProductY	Sugarland	Wong, Franklin T.
*	333445555	2	10.0	ProductY	Sugarland	Smith, John B.
*	333445555	2	10.0	ProductY	Sugarland	English, Joyce A.
	333445555	2	10.0	ProductY	Sugarland	Wong, Franklin T.
*	333445555	3	10.0	ProductZ	Houston	Narayan, Ramesh K.
	333445555	3	10.0	ProductZ	Houston	Wong, Franklin T.
	333445555	10	10.0	Computerization	Stafford	Wong, Franklin T.
*	333445555	20	10.0	Reorganization	Houston	Narayan, Ramesh K.
	333445555	20	10.0	Reorganization	Houston	Wong, Franklin T.

:

Merk: dette er grunnen til at joins som regel er basert på nøkler!

Retningslinje 4

Design relasjonsskjemaer slik at de kan slås sammen vha ekvivalensbetingelser på attributter som er passende relatert (dvs. primærnøkkel, fremmednøkkel), slik at ingen falske tupler blir generert. Unngå relasjoner som inneholder matchende attributter som ikke er fremmednøkkel-primærnøkkel kombinasjoner, fordi JOIN basert på slike attributter kan produsere falske tupler. Vi ser nærmere på denne i seksjon 15.2.

14.2 Funksjonelle avhengigheter

Vi skal nå introdusere et formelt verktøy for å analysere relasjonsskjemaer, slik at vi kan oppdage og beskrive noen av problemene vi så på i forrige seksjon. Funksjonelle avhengigheter er det viktigste konseptet i design av relasjonsskjema.

Definisjon av funksjonell avhengighet

En funksjonell avhengighet er en begrensning mellom to sett med attributter fra databasen. Anta at databasen blir beskrevet av et enkelt universelt relasjonsskjema $R = \{A_1, A_2, \dots, A_n\}$.

Definisjon

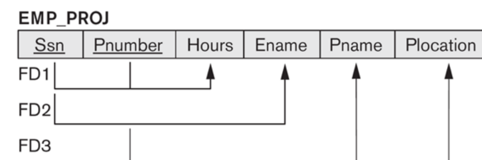
En funksjonell avhengighet, betegnet som $X \rightarrow Y$, mellom to sett med attributter X og Y som er subsett av R, spesifiserer en begrensning på hvilke tupler som kan danne en relasjonstilstand r. Begrensningen er at for tuplene t_1 og t_2 i r som har $t_1[X] = t_2[X]$, må også $t_1[Y] = t_2[Y]$

Verdiene til X komponentene vil unikt (eller funksjonelt) bestemme verdiene til Y komponentene. Vi sier at det er en funksjonell avhengighet fra X til Y, eller at Y er funksjonelt avhengig av X. Forkortelsen for funksjonell avhengighet er FD eller f.d. Settet av attributter X kalles venstre-siden av FD, mens Y kalles høyre-siden. X vil funksjonelt bestemme Y i et relasjonsskjema hvis og bare hvis to tupler med samme X-verdi også har samme Y-verdi. Merk:

- Hvis X er en kandidatnøkkel (dvs. må ha unik verdi) betyr det at $X \rightarrow Y$, der $Y \subseteq R$, siden nøkkelbegrensningen gir at ingen to tupler kan ha samme verdi for X. Hvis vi har to tupler med lik verdi for attributtsettet X, må resten av attributtene også være like.
- Hvis $X \rightarrow Y$ vil det ikke nødvendigvis bety at $Y \rightarrow X$

En funksjonell avhengighet er en egenskap til semantikken eller betydningen til attributtene. Designere bruker deres forståelse av semantikken til R, altså hvordan attributtene henger sammen, for å spesifisere funksjonelle avhengigheter. For eksempel kan vi bruke semantikken til attributtene og relasjonen hos EMP_PROJ får å lage følgende FDs:

1. $\text{Ssn} \rightarrow \text{Ename}$
2. $\text{Pnumber} \rightarrow \{\text{Pname}, \text{Plocation}\}$
3. $\{\text{Ssn}, \text{Pnumber}\} \rightarrow \text{Hours}$



Avhengighet 1 gir at personnummeret til den ansatte vil unikt bestemme navnet til den ansatte. Vi sier at Ename er funksjonelt bestemt av Ssn, så gitt en verdi av Ssn kan vi finne verdien til Ename. Avhengighet 2 gir at prosjektnummeret vil unikt bestemme prosjektnavnet og lokasjonen, og avhengighet 3 gir at kombinasjonen av personnummer og prosjektnummer vil unikt bestemme antall timer den ansatte arbeider på prosjektet per uke. Figuren viser en **diagramnotasjonen for funksjonelle avhengigheter. Hver FD representeres av en horisontal linje, der pilen peker mot høyre-side attributtet.** Legg merke til at pilen kan splittes opp på begge sider. **Settet av funksjonelle avhengigheter betegnes som F**, og skjemadesigneren vil som regel definere FDs som er semantiske åpenbare. Som regel kan man utlede andre FDs fra FDs i F.

En funksjonell avhengighet er en egenskap ved relasjonsskjemaet R og ikke ved en bestemt relasjonstilstand r. Relasjonstilstanden kan derfor ikke brukes for å bestemme funksjonelle avhengigheter, men den kan brukes for å finne mulige forslag. For eksempel for relasjonstilstanden på figuren kan vi ikke si at $\text{Text} \rightarrow \text{Course}$, før vi har bekreftet at det gjelder for alle mulige tilstander av TEACH. Vi kan likevel si at det er mulig at relasjonsskjemaet har en slik FD og senere bekrefte/avkrefte dette ved å se på semantikken til attributtene. **For å avslå en funksjonell avhengighet er det derimot tilstrekkelig å bruke relasjonstilstanden, fordi det holder å finne ett moteksempel**. For eksempel kan vi se at Teacher ikke funksjonelt bestemmer Course, siden vi får samme lærer har ulike kurs.

TEACH

Teacher	Course	Text
Smith	Data Structures	Bartram
Smith	Data Management	Martin
Hall	Compilers	Hoffman
Brown	Data Structures	Horowitz

14.3 Normalformer basert på primærnøkler

Vi skal nå se at funksjonelle avhengigheter kan brukes for å utvikle en formell metode for å teste og forbedre relasjonsskjema. Vi antar at hver relasjon har et sett med FDs og en primærnøkkel. Denne informasjon kombinert med tester for normalformer driver normaliseringsprosessen som vi nå skal se på. Ved databasedesign er det to hovedtilnærminger:

1. En konseptuell modell (eks: ER) brukes for å lage et konseptuelt skjema som så blir kartlagt til et sett med relasjoner

2. Relasjonene blir designet basert på ekstern kunnskap utledet fra en eksisterende implementasjon av filer, former eller rapporter.

Etter designet er det nyttig å evaluere relasjonene og eventuelt dekomponere dem for å oppnå høyere normalformer vha normaliseringsteorien. Vi skal se på de første tre normal formene for relasjonsskjema.

Normalisering av relasjoner

Normaliseringsprosessen tar relasjonsskjemaet gjennom en rekke tester for å sjekke om det tilfredsstiller en bestemt normalform. Prosessen har en top-down tilnærming der hver relasjon blir vurdert mot kriteriet for normalformene og dekomponert om nødvendig. Første, andre, tredje og Boyce-Codd normalform er basert på funksjonelle avhengigheter, mens fjerde og femte normalform er basert på hhv. flerverdi og join avhengigheter.

Ved normalisering av data blir relasjonsskjema analysert basert på deres FDs og primære nøkler for å minimere redundans og anomalier ved innseting, sletting og modifisering. Det kan ses på som en «rensing» som gir designet bedre kvalitet. Et relasjonsskjema som ikke tilfredsstiller betingelsene for normalform vil dekomponeres til mindre relasjonsskjema som består denne testen. Normaliseringsprosessen fungerer som:

- Et **formelt rammeverk for å analysere relasjonsskjema** basert på deres nøkler og FDs
- En rekke normalform tester som kan utføres på individuelle relasjonsskjema, slik at **databasen kan normaliseres** til en ønsket grad*.

* Merk: Normalformen til en relasjon er høyeste normalformtest den har bestått, og det indikerer derfor hvilken grad relasjonen har blitt normalisert.

Normalformene vil ikke alene fungere som en garanti på godt databasedesign. Som regel er det ikke tilstrekkelig å separat sjekke at hvert relasjonsskjema er for eksempel i tredje normalform. **Normaliseringsprosessen må gjennom dekomponeringen også sjekke at relasjonsskjemaet har følgende egenskaper**:

1. **Tapsløs join-egenskapen** – for alle mulige forekomster av den opprinnelige tabellen, kan dataen fordeles på deltabellene og join av deltabellene vil gi nøyaktig den opprinnelige tabellforekomsten. Denne egenskapen vil altså garantere at falske tupler ikke blir generert.
2. **Bevaring av funksjonelle avhengigheter** – alle funksjonelle avhengigheter som gjelder for den opprinnelige tabellen, skal også gjelde for en eller flere av deltabellene eller skal kunne utledes fra de funksjonelle avhengighetene som gjelder for deltabellene.

Tapsløs join er ekstremt kritisk og må oppnås ved enhver kostnad!

Praktisk bruk av normalformer

I praksis vil databasedesign ofte bruke normalisering opp til 3NF, BCNF og 4NF, fordi høyere normalformer er basert på sjeldne begrensninger og de kan være vanskelig for designere og brukere å forstå eller oppdage. Designere kan også velge å ikke normalisere til høyeste mulig normalform. Relasjoner kan etterlates i lavere normalformer (eks: 2NF) pga ytelse, men da må de håndtere anomalier. Denormalisering er når man lagrer join av høyere normalformer som en baserelasjon, som dermed er i en lavere normalform.

Definisjon av nøkler og attributter som deltar i nøkler

En supernøkkel er et sett med attributter som ikke kan være lik for to ulike tupler. En nøkkel er en minimal supernøkkel, som vil si at fjerning av en attributt vil gjøre at den ikke er en supernøkkel lenger. For eksempel for EMPLOYEE vil {Ssn} være en nøkkel, mens {Ssn}, {Ssn, Ename} og {Ssn, Ename, Bdate} er supernøkler (ethvert sett som inkluderer Ssn).
Hvis en relasjon har flere nøkler, kalles de kandidatnøkler. **Vi vil vilkårlig velge en av kandidatnøklene til å være primærnøkkelen, mens de andre vil være sekundærnøkler.** Alle relasjonsskjema må ha en primærnøkkel og hvis ingen kandidatnøkkel er kjent, kan hele relasjonen behandles som supernøkkel. **Et nøkkelattributt (*prime*) er medlem av en kandidatnøkkel**, mens et ikke-nøkkelattributt (*nonprime*) er ikke medlem av noen kandidatnøkler. For WORKS_ON vil både Ssn og Pnumber være nøkkelattributt.

Vi skal nå se på normalformene 1NF, 2NF og 3NF. Disse brukes som en sekvens for å oppnå relasjoner i 3NF ved å gå gjennom de mellomliggende tilstandene 1NF og 2NF. Av definisjon vil en 3NF relasjon allerede tilfredsstille 2NF, selv om de ser på ulike typer problemer som skyldes problematiske FDs blant attributter.

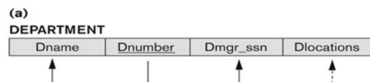
Første normalform – atomiske attributtverdier

Første normalform (1NF) sier at domenet til et attributt kan kun inneholde atomiske (dvs. simple og udelelige) verdier og at attributtverdien i en tuple må være én verdi fra domenet til attributtet. Denne normalformen forbyr flerverdi- og sammensatte attributter. Et sett med verdier, en tuple med verdier eller en kombinasjon av begge kan ikke være en attributtverdi hos en enkelt tuple. 1NF vil forby relasjoner innenfor relasjoner eller relasjoner som attributtverdier hos tupler. **De eneste attributtverdiene som er tillatt av 1NF er atomiske verdier**.

Eksempel – hvordan oppnå 1NF

Anta at vi har DEPARTMENT relasjonsskjemaet og legger til Dlocations attributtet (se figur). Dette skjemaet vil ikke være 1NF fordi Dlocations er ikke en atomisk attributt, som vi kan se av første tuple i figur b. Det er to måter å tolke Dlocations:

- Domenet til Dlocations inneholder atomiske verdier, men noen tupler kan ha et sett av disse verdiene. Da vil ikke Dnumber → Dlocations
- Domenet til Dlocations inneholder sett med verdier, og er dermed ikke-atomisk. Da vil Dnumber → Dlocations, siden hvert sett er et medlem av attributtdomenet.



(b)
DEPARTMENT

Dname	Dnumber	Dmgr_ssn	Dlocations
Research	5	333445555	{Bellaire, Sugarland, Houston}
Administration	4	987654321	{Stafford}
Headquarters	1	888665555	{Houston}

Violates 1NF

(c)
DEPARTMENT

Dname	Dnumber	Dmgr_ssn	Dlocation
Research	5	333445555	Bellaire
Research	5	333445555	Sugarland
Research	5	333445555	Houston
Administration	4	987654321	Stafford
Headquarters	1	888665555	Houston

1NF (Expand Key) :
But Redundant

I begge tilfellene vil ikke DEPARTMENT være i 1NF, og det er tre hovedteknikker for å oppnå første normalform for en slik relasjon:

1. **Fjern attributtet Dlocations som bryter 1NF og plasser det i en egen relasjon sammen med primærnøkkelen Dnumber til DEPARTMENT.** Primærnøkkelen til DEPT_LOCATION er kombinasjonen {Dnumber, Dlocation}. Én ikke-1NF relasjon blir omdannet til to 1NF relasjoner.

Merk: bilder, dokumenter, videoer og lyd blir lagret som store BLOB eller CLOB datatyper som blir behandlet som atomiske verdier, slik at 1NF statusen opprettholdes.

2. **Utvid nøkkelen til å inkludere Dlocation, slik at det vil være en separat tuple for hver lokasjon i DEPARTMENT** (se figur). Primærnøkkelen blir kombinasjonen {Dnumber, Dlocation}. Denne teknikken blir sjeldent brukt, fordi den introduserer mye overflødig data (resten av attributtverdiene må gjentas).
3. **Hvis det er kjent at hver avdeling har maksimalt tre lokasjoner, kan vi erstatte Dlocations med tre atomiske attributter: Dloc1, Dloc2 og Dloc3**. Ulempen med denne teknikken er at introduserer mange NULL-verdier hvis de fleste avdelingene har færre enn tre lokasjoner. Det vil også introdusere falsk semantikk i ordningen av lokasjonene, fordi en slik ordning var opprinnelig ikke meningen. Spørringen blir også mer komplisert. Det er derfor best å unngå dette alternativet.

Den første tilnærmingen er ofte anbefalt, siden den ikke introduserer redundans og er fullstendig generell (setter ingen maksgrense på antall verdier).

Nøstede relasjoner til 1NF

Første normalform forbyr også nøstede relasjoner der flerverdi attributter er sammensatte, slik at hver tuple kan inneholde en relasjon. Figuren viser hvordan EMP_PROJ relasjonen kan lages hvis nøsting er tillatt. Her kan hver tuple ha en relasjon PROJS:

$$EMP_PROJ(Ssn, Ename, \{PROJS(Pnumber, Hours)\})$$

Ssn er primærnøkkelen til EMP_PROJ, mens Pnumber er delvis nøkkel til den nøstede relasjonen. EMP_PROJ er ikke i 1NF, siden den inneholder en nøstet relasjon. **For å normalisere den til 1NF vil vi fjerne attributtene til den nøstede relasjonen og legge til disse i en ny relasjon, som også inneholder primærnøkkelen til den originale relasjonen**. Primærnøkkelen til den nye relasjonen er kombinasjonen av primær- og delvis nøkkel (her: Ssn og Pnumber). Vi får skjemaene EMP_PROJ1 og EMP_PROJ2.

(a)
EMP_PROJ

		Projs	
Ssn	Ename	Pnumber	Hours

(b)
EMP_PROJ

Ssn	Ename	Pnumber	Hours
123456789	Smith, John B.	1	32.5
		2	7.5
666884444	Narayan, Ramesh K.	3	40.0
453453453	English, Joyce A.	1	20.0
		2	20.0
333445555	Wong, Franklin T.	2	10.0
		3	10.0
		10	10.0
		20	10.0
999887777	Zelaya, AliciaJ.	30	30.0
		10	10.0
987987987	Jabbar, Ahmad V.	10	35.0
		30	5.0
987654321	Wallace, Jennifer S.	30	20.0
		20	15.0
888665555	Borg, James E.	20	NULL

(c)
EMP_PROJ1

Ssn	Ename

EMP_PROJ2

Ssn	Pnumber	Hours

Hvis relasjonen har flere nivåer med nøsting, kan denne metoden brukes rekursivt. For eksempel kan vi ha relasjonen som beskriver data om kandidater som søker på jobb, der jobb- og lønnhistorikken er gitt som nøstede relasjoner:

$$CANDIDATE(\underline{Ssn}, Name, \{JOB_HIST(Company, Highest_position, \{SAL_HIST(Year, Max_sal)\})\})$$

Normaliseringen vil resultere i følgende 1NF relasjoner:

Flerverdi attributter plasseres i {} og sammensatte attributter plasseres i ().

$$CANDIDATE(\underline{Ssn}, Name)$$
$$CANDIDATE_JOB_HIST(\underline{Ssn}, \underline{Company}, Highest_position)$$
$$CANDIDATE_JOB_HIST(\underline{Ssn}, \underline{Year}, Max_sal)$$

Flere flerverdi attributter

Dersom en relasjon har flere flerverdi attributter, må dette håndteres på en forsiktig måte. For eksempel kan vi ha følgende ikke-1NF relasjon som representerer at en person kan ha flere biler og mobiler:

$$\mathrm{PERSON}(\mathrm{Ssn}, \{\mathrm{CarLicNr}\}, \{\mathrm{PhoneNr}\})$$

Hvis vi bruker strategi 2 for normaliseringen vil vi få en relasjon der alle attributtene er nøkler: $\mathrm{PERSON_IN_1NF}(\underline{\mathrm{Ssn}}, \underline{\mathrm{CarLicNr}}, \underline{\mathrm{PhoneNr}})$. Dette introduserer overflødig data i form av mange duplikate verdier, noe som fører til problemer som ofte oppdages ved senere steg i normaliseringsprosessen. Den riktige måten å håndtere dette er vha. strategi 1, der vi lager to separate relasjoner:

$$\mathrm{P1}(\underline{\mathrm{Ssn}}, \underline{\mathrm{CarLicNr}})$$
$$\mathrm{P2}(\underline{\mathrm{Ssn}}, \underline{\mathrm{PhoneNr}})$$

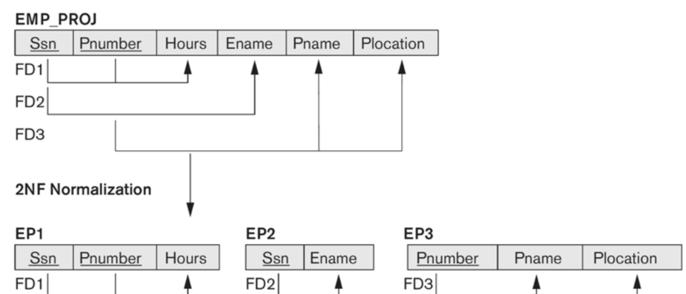
Andre normalform – ingen delvis avhengighet av primærnøkkel

Andre normalform (2NF) er basert på full funksjonell avhengighet ($X \rightarrow Y$), der fjerning av enhver attributt fra X vil gjøre at avhengigheten ikke gjelder lenger. Det motsatte er delvis avhengighet. Hos EMP_PROJ relasjonen ser vi at $\{\mathrm{Ssn}, \mathrm{Pnumber}\} \rightarrow \mathrm{Hours}$ er en full avhengighet, siden verken $\mathrm{Ssn} \rightarrow \mathrm{Hours}$ eller $\mathrm{Pnumber} \rightarrow \mathrm{Hours}$ vil gjelde. $\{\mathrm{Ssn}, \mathrm{Pnumber}\} \rightarrow \mathrm{Ename}$ er derimot en delvis avhengighet, siden $\mathrm{Ssn} \rightarrow \mathrm{Ename}$ stemmer.

Et relasjonsskjema er i 2NF hvis ingen ikke-nøkkelattributt er delvis avhengig av primærnøkkelen. Hvis vi har $\{X_1, X_2, \dots, X_n\} \rightarrow Y$, der Y ikke er en del av en nøkkel og venstre side er primærnøkkelen, må altså venstre side inkludere alle attributtene i primærnøkkelen slik at vi får en full funksjonell avhengighet. **Testen for 2NF må utføres hvis:**

1. **Primærnøkkelen består av flere attributter**
2. **Vi har en FD der primærnøkkelen er på venstre side og attributtet på høyre side er ikke en del av en nøkkel.**

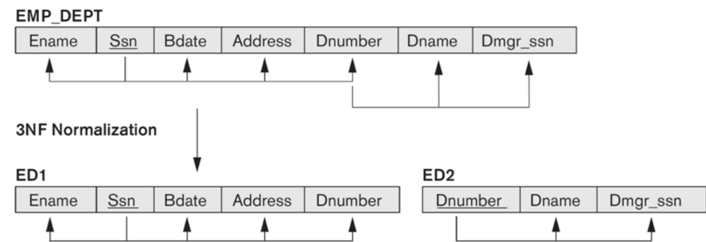
Vi ser altså bort fra FDs der primærnøkkelen ikke deltar eller høyre side er et nøkkelattributt. Hvis primærnøkkelen består av en enkelt attributt, trenger vi heller ikke å utføre testen.



På figur a kan vi se EMP_PROJ som er i 1NF, men ikke i 2NF. Primærnøkkelen er {Ssn, Pnumber}. Ikke-nøkkelattributt Ename bryter med 2NF, fordi den er funksjonelt avhengig av kun Ssn, og ikke Pnumber. Ikke-prim attributtene Pname og Plocation bryter også med 2NF, fordi de er funksjonelt avhengig av kun Pnumber. **EMP_PROJ kan 2NF normaliseres til flere 2NF relasjoner der ikke-nøkkelattributtene er assosiert med kun den delen av primærnøkkelen som de er fult funksjonelt avhengig av**. De funksjonelle avhengighetene FD1, FD2 og FD3 gjør altså at EMP_PROJ må dekomponeres til tre relasjonsskjemaer EP1, EP2 og EP3 som er i 2NF.

Tredje normalform – ingen transitiv avhengighet med ikke-nøkler

Tredje normalform (3NF) er basert på transitiv avhengighet, der vi for en FD X → Y og et sett med attributter Z, som ikke er nøkkelattributt, har både $X \rightarrow Z$ og $Z \rightarrow Y$. For eksempel i EMP_DEPT vil avhengigheten Ssn → Dmgr_ssn være transitiv gjennom Dnumber, siden både Ssn → Dnumber og Dnumber → Dmgr_ssn gjelder, og Dnumber er verken en nøkkel eller et subsett av en nøkkel hos EMP_DEPT. Dette er en uønsket egenskap, siden vi får avhengigheten Dnumber → Dmgr_ssn, der Dnumber ikke er en nøkkel.



Et relasjonsskjema er i 3NF hvis det er i 2NF og ingen ikke-nøkkelattributt er transitiv avhengig av primærnøkkelen. EMP_DEPT på figuren er i 2NF, men den er ikke i 3NF siden avhengigheten Ssn → Dmgr_ssn er transitiv gjennom Dnumber. EMP_DEPT normaliseres ved at den deles inn i to 3NF relasjonsskjema ED1 og ED2, der Dnumber settes som fremmednøkkel i ED1 og primærnøkkel i ED2. Vi kan se at ED1 og ED2 representerer uavhengig fakta om ansatte og avdelinger, og en naturlig join av disse vil gi EMP_DEPT uten å lage falske tupler.

Oppsummering

Funksjonelle avhengigheter der venstre-siden er deler av primærnøkkelen eller ikke en nøkkel vil være problematiske FDs som fjernes av 2NF og 3NF normalisering som deler den originale relasjonen inn i nye relasjoner. Tabellen under er en summering på de tre normalformene, testene som brukes og normaliseringen som utføres for å oppnå disse formene.

Normalform	Test	Normalisering
1NF	Relasjonen skal ikke ha noen flerverdi attributter eller nøstede relasjoner	Lag nye relasjoner for hver flerverdi attributt eller nøstede relasjoner
2NF	For relasjoner der primærnøkkelen inneholder flere attributter, skal ingen ikke-nøkkel attributt være funksjonelt avhengig av kun deler av primærnøkkelen.	Del opp og lag en ny relasjon for hver delvis nøkkel med dens avhengige attributter. Sørg for at en relasjon har den originale primærnøkkelen og eventuelle attributter som er fult avhengig av den.
3NF	Relasjonen skal ikke ha en ikke-nøkkel attributt som er funksjonelt avhengig av en annen ikke-nøkkel attributt. Ingen ikke-nøkkel attributt skal være transitiv avhengig av primærnøkkelen.	Del opp og lag en relasjon som inkluderer ikke-nøkkel attributtene som funksjonelt bestemmer andre ikke-nøkkel attributter.

14.4 Generell definisjon av 2NF og 3NF

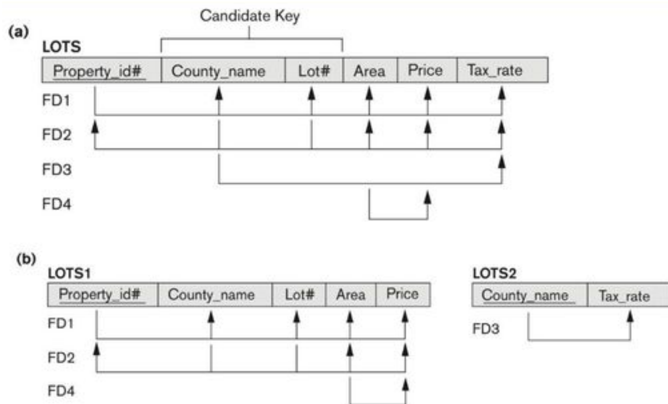
Som regel vil vi designe relasjonsskjemaet slik at det har ingen delvis eller transitive avhengigheter, fordi disse fører til anomalier i oppdateringen. Frem til nå har vi sett på stegene for å oppnå en 3NF ved å forby delvis og transitive avhengigheter på primærnøkkelen. Denne

definisjonen er nyttig når primærnøkkelen allerede er definert, men den tar ikke hensyn til andre kandidatnøkler i relasjonen. **En mer generell definisjon av 2NF og 3NF tar hensyn til alle kandidatnøkler**. Nøkkelattributtet vil da være et attributt som er en del av enhver kandidatnøkkel. 1NF blir ikke påvirket siden den er uavhengig av nøkler og FDs.

For $X \rightarrow Y$ der X er en nøkkel, kan vi ikke fjerne deler av X uten at avhengigheten stopper å gjelde, altså må det være en full funksjonell avhengighet.

Generell definisjon av 2NF

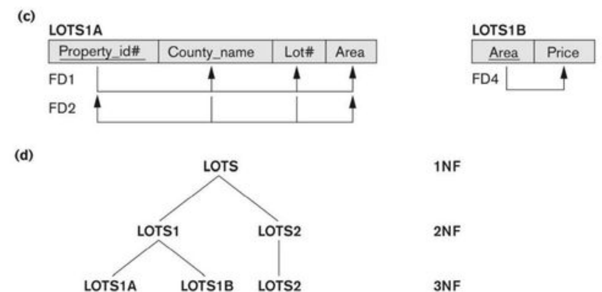
Et relasjonsskjema er i 2NF hvis ingen ikke-nøkkelattributt er delvis avhengig av en nøkkel i relasjonen. Hvis relasjonen har nøkler som består av flere attributter, vil testen gå ut på å sjekke at alle attributtene er representert på venstre side av FDs nøklene deltar i. Hvis nøklene inneholder ett attributt trenger vi ikke å utføre testen.



For eksempel kan vi se på relasjonsskjemaet LOTS som har to kandidatnøkler: Property_id# og {County_name, Lot#}, der vi har valgt at Property_id# er primærnøkkelen (ingen forskjell annet enn understrek på skjema). Basert på de to kandidatnøklene, vil FD1 og FD2 holde. **Merk: 2NF krever ikke at ikke-nøkkelattributtene skal være funksjonelt avhengig av *alle* nøklene**. Så det er greit at FD1 kun har Property_id# på venstre-siden og FD2 kun har {County_name, Lot#} på venstre-siden. **Kravet er at det ikke kan være noen delvis avhengigheter**, **dvs. kun deler av nøkkelen er på venstre-siden**! LOTS er ikke i 2NF, fordi Tax_rate er delvis avhengig av kandidatnøkkelen {County_name, Lot#}, som følger av FD3 (utelater Lot#). For å normalisere LOTS til 2NF må vi dele den inn i to relasjoner LOTS1 og LOTS2. Vi fjerner Tax_rate som bryter 2NF og plasserer den i LOTS2 sammen med County_name som er på venstre side i FD3. Legg merke til at FD4 ikke bryter 2NF, siden den ikke har noen nøkkel-attributt på venstre side. Denne vil derfor være med i LOTS1.

Generell definisjon av 3NF

Et relasjonsskjema er i tredje normalform (3NF) hvis $X \rightarrow A$ (ikke-triviell FD) betyr at (a) X er en supernøkkel *eller* (b) A er en nøkkelattributt. I følge denne definisjonen vil LOTS2 være i 3NF, mens LOTS1 vil ikke være det, siden FD4 bryter definisjonen (Area er ikke en supernøkkel og Price er ikke er nøkkelattributt). For å normalisere LOTS1 må vi dele den inn i LOTS1A og LOTS1B. Vi fjerner Price som bryter 3NF og plasserer den i LOTS1B sammen med Area som er på venstre side i FD4.



Merk at LOTS1 bryter 3NF fordi Price er transitivt avhengig av begge kandidatnøklene via ikke-nøkkelattributtet Area. Vi kan teste om skjemaet er 3NF før 2NF, og **hvis skjemaet er 3NF vil det automatisk også være 2NF**. Hvis vi hadde brukt 3NF testen direkte på LOTS ville vi ha sett at FD3 og FD4 bryter med 3NF (County_name er ikke supernøkkel og Tax_rate er ikke nøkkelattributt). Derfor ville vi ha delt inn LOTS til LOTS1A, LOTS1B og LOTS2 med en gang.

Tolkning av den generelle definisjonen av 3NF

Den første betingelsen i 3NF ($X \rightarrow A$, betyr at X er supernøkkel) vil alene dekke de to problematiske avhengighetene som ligger bak 2NF og 3NF:

- Et ikke-nøkkelattributt bestemmer et annet ikke-nøkkelattributt. Dette er en transitiv avhengighet som bryter med 3NF
- Et subsett av en nøkkel bestemmer et ikke-nøkkelattributt. Dette er en delvis avhengighet som bryter med 2NF

Vi kan derfor definere en generell alternativ definisjon av 3NF. **Et relasjonsskjema er i 3NF hvis alle ikke-nøkkelattributt oppfyller begge disse betingelsene:**

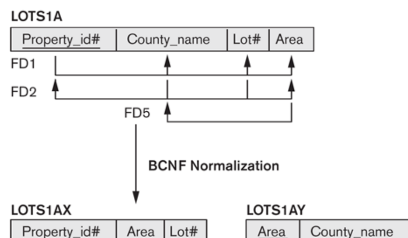
1. **De er fult funksjonelt avhengig av alle nøklene i relasjonen**
2. **De er ikke-transitivt avhengig av alle nøkler i relasjonen**

Den andre betingelsen i 3NF ($X \rightarrow A$, betyr at A er nøkkelattributt) lar bestemte FDs slippe gjennom definisjonen av 3NF, selv om de kan være problematiske. Boyce-Codd normalformen vil fange disse avhengighetene, ved at de ikke er tillatt.

14.5 Boyce-Codd normalform

Et relasjonsskjema er i Boyce-Codd normalform (BCNF) hvis $X \rightarrow A$ (ikke-triviell FD) betyr at X er en supernøkkel. BCNF er en strengere versjon av 3NF. Alle relasjoner som er i BCNF vil også være i 3NF, mens relasjoner i 3NF vil ikke nødvendigvis være i BCNF.

Vi ser på LOTS for å illustrere behovet for en strengere normalform. Anta at relasjonen har tusenvis med tomter (*lot*), men kun fra to fylker (*county*): Trøndelag og Nordland. Anta videre at alle tomtene i Trøndelag er 0.5, ..., 0.9 eller 1.0 dekar, mens alle tomtene i Nordland er 1.1, ..., 1.9 eller 2.0 dekar (*area*). I denne situasjonen vil vi ha $\text{Area} \rightarrow \text{County_name}$. Hvis vi legger til denne betingelsen, vil relasjonen fortsatt være 3NF, siden County_name er et nøkkelattributt (betingelse 2). Hvis vi plasserer FD5 i en egen relasjon kan denne avhengigheten representeres med 16 tupler $R(\text{Area}, \text{County_name})$, siden det er kun 16 mulige Area-verdier. Dermed vil vi **redusere redundansen** fordi mindre informasjon må gjentas i LOTS1A tuplene. For eksempel vil vi slippe å gjenta $\text{County_name} = \text{Trøndelag}$ for alle LOTS tupler som har $\text{Area} = 0.5$. I dette tilfellet med mange tupler i original relasjon og få FDs som tilfredsstiller betingelse 2, vil det lønne seg å bruke BCNF (verdien til A må ikke gjentas for hver tuple).



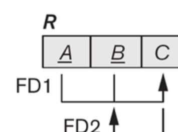
FD5 vil bryte BCNF fordi Area er ikke en supernøkkel. For å normalisere LOTS1A til BCNF må vi dele den inn i LOTS1AX og LOTS1AY. Vi fjerner County_name som bryter BCNF og plasserer den i LOTS1AY sammen med Area som er på venstre side i FD5. **Ved design av en relasjonsdatabase bør man nesten alltid forsøke å oppnå BCNF eller 3NF**.

Dekomponering av relasjoner til BCNF

Tabellen til høyre viser TEACH relasjonen som har følgende avhengigheter:

- $\text{FD1}: \{\text{Student}, \text{Course}\} \rightarrow \text{Instructor}$
- $\text{FD2}: \text{Instructor} \rightarrow \text{Course}$

$\{\text{Student}, \text{Course}\}$ er en kandidatnøkkel og avhengighetene følger mønsteret på figuren under (Student som A, Course som B og Instructor som C).



TEACH

Student	Course	Instructor
Narayan	Database	Mark
Smith	Database	Navathe
Smith	Operating Systems	Ammar
Smith	Theory	Schulman
Wallace	Database	Mark
Wallace	Operating Systems	Ahamad
Wong	Database	Omiecinski
Zelaya	Database	Navathe
Narayan	Operating Systems	Ammar

Når vi skal se på de mulige dekomponeringene, må vi se på alle mulige kombinasjoner av attributtene. For å bestemme hva som er nøkler må vi se på semantikken. For $(\underline{\text{Instructor}}, \underline{\text{Student}})$ og $(\underline{\text{Course}}, \underline{\text{Student}})$ bruker vi begge attributtene som kombinert nøkkel, siden flere instruktører/emner kan ha relasjon til flere studenter, og motsatt. Hvis vi kun bruker Student som nøkkel vil vi ikke få med de ulike instruktørene/emnene en enkelt student kan ha, og motsatt. For $(\text{Course}, \underline{\text{Instructor}})$ holder det å bruke Instructor som nøkkel, siden det er oppgitt av hver instruktør kun underviser ett emne, så gitt instruktør kan vi bestemme emnet. Dette kan vi se av FDs. Vi har FD Instructor → Course, så derfor er det kun Course som blir bestemt av et annet attributt. Vi har ingen FD til de andre relasjonene, så derfor må de være en all-nøkkel relasjon

Denne relasjonen er i 3NF, men ikke i BCNF (Instructor er ikke supernøkkel). Dekomponering av dette relasjonsskjemaet er ikke rett frem, fordi det kan gjøres på følgende måter:

1. $R1(\underline{\text{Instructor}}, \underline{\text{Student}})$ og $R2(\underline{\text{Course}}, \underline{\text{Student}})$
2. $R1(\text{Course}, \underline{\text{Instructor}})$ og $R2(\underline{\text{Course}}, \underline{\text{Student}})$
3. $R1(\text{Course}, \underline{\text{Instructor}})$ og $R2(\underline{\text{Instructor}}, \underline{\text{Student}})$

Vi ønsker å finne oppdelingen som er best. Ved dekomponering ønsker vi å oppnå tapsløs-join egenskapen og bevaring av FDs. Alle tre dekomponeringene over mister FD1, så dermed vil vi ikke kunne oppnå bevaring av FDs. Men vi må oppfylle tapsløs-join egenskapen, og dette kan vi sjekke vha følgende test: **NJB (Nonadditive Join Test for binære dekomponeringer) sier at en dekomponering $D = \{R1, R2\}$ har tapsløs-join egenskapen mht. et sett med FDs hvis en av følgende er oppfylt:**

- **FD $(R1 \cap R2) \rightarrow (R1 - R2)$ er i F^+**
- **FD $(R1 \cap R2) \rightarrow (R2 - R1)$ er i F^+**

F^+ dekker settet av FDs i F. Blir forklart mer i kapittel 15

Det er kun dekomponering 3 som oppfyller denne testen. I dette tilfellet vil $R1 \cap R2$ være Instructor og $R1 - R2$ er Course. Dermed får vi Instructor → Course, som er en FD i F. Denne dekomponeringen har dermed tapsløs-join egenskapen (ikke-additiv), og vil derfor være foretrukket:

$$\text{TEACH1}(\underline{\text{Instructor}}, \text{Course}) \text{ og } \text{TEACH2}(\underline{\text{Instructor}}, \underline{\text{Student}})$$

Følgende er en metode å dekomponere en relasjon slik at den blir BCNF og oppnår tapsløs-join egenskapen. Vi har en relasjon R som ikke er i BCNF og $X \rightarrow A$ er FD som bryter BCNF. Vi deler R inn i to relasjoner:

$$R - A \qquad \text{og} \qquad XA$$

Hvis $R - A$ eller XA ikke er i BCNF må vi gjenta prosessen. I eksempelet med TEACH vil R være $\text{TEACH}(\underline{\text{Student}}, \underline{\text{Course}}, \text{Instructor})$, X er Instructor og A er Course. Derfor vil de to relasjonene bli:

$$R - A = \text{TEACH1}(\underline{\text{Instructor}}, \underline{\text{Student}})$$
$$XA = \text{TEACH2}(\underline{\text{Instructor}}, \text{Course})$$

Legg merke til at nøklene ikke blir bevart, men må bestemmes ved dekomponeringen.

14.6 Flerverdi avhengighet og 4NF

Relasjonen EMP representerer en ansatt som kan arbeide på flere prosjekter og kan ha flere dependents. Prosjekt og dependent er uavhengige av hverandre, så alle kombinasjoner må representeres i separate tupler for å unngå falske forhold mellom dem. For eksempel kan vi se at Smith som arbeider på to prosjekt X og Y og har to dependents John og Anna, må representeres vha fire tupler (figur a). EMP er en **all-nøkkel relasjon**, der alle attributtene er nøkkelen. Det er derfor ingen FDs og relasjonen er i BCNF. Relasjonen har en **åpenbar redundans**, siden informasjonen om hvert prosjekt blir gjentatt for hver dependent, og motsatt. **Denne relasjonen illustrerer at noen relasjoner har begrensninger som ikke kan spesifiseres som FDs og vil dermed ikke bryte BCNF. For å håndtere dette ble flerverdi avhengigheter (MVD) og fjerde normalform definert**. MVDs vil oppstå når relasjonen har mer enn ett uavhengig flerverdi attributt og metode 2 på side 94 blir brukt for å oppnå 1NF.

EMP

Ename	Pname	Dname
Smith	X	John
Smith	Y	Anna
Smith	X	Anna
Smith	Y	John

Merk: en MVD vil oppstå når to uavhengige $1{:}N$ relasjoner $A{:}B$ og $A{:}C$ blir blandet i samme relasjon $R(A, B, C)$.

Formell definisjon av MVD

En flerverdi avhengighet $X \rightarrow Y$ spesifiserer følgende begrensning; hvis to tupler t_1 og t_2 eksisterer slik at $t_1[X] = t_2[X]$, må det også eksistere to tupler t_3 og t_4 med $Z = \big(R - (X \cup Y)\big)$ slik at:

$$t_3[X] = t_4[X] = t_1[X] = t_2[X]$$
$$t_3[Y] = t_1[Y] \text{ og } t_4[Y] = t_2[Y]$$
$$t_3[Z] = t_2[Z] \text{ og } t_4[Z] = t_1[Z]$$

Tuplene t_1, t_2, t_3 og t_4 er ikke nødvendigvis ulike. **En flerverdi avhengighet (MVD) på en relasjon betegnes som $X \twoheadrightarrow Y$. Relasjonsskjemaet har en MVD: $X \twoheadrightarrow Y$ hvis to tupler med like X-verdier kan bytte Y-verdier uten at tabellen endres (dvs. den har samme tupler før og etter). En verdi for X kan altså ha flere verdier for Y og hvilke verdier som er mulig for Y bestemmes av X-verdien**. MVDs kommer ofte i par, så vi skriver $X \twoheadrightarrow Y|Z$. Attributtsettene Y og Z er uavhengige av hverandre og begge har 1:N relasjon til X.

EMP_PROJECTS

Ename	Pname
Smith	X
Smith	Y

EMP_DEPENDENTS

Ename	Dname
Smith	John
Smith	Anna

En MVD $X \twoheadrightarrow Y$ kalles en triviell MVD hvis Y er et subsett av X eller $X \cup Y = R$. For eksempel har EMP_PROJECTS triviell MVD Ename $\twoheadrightarrow$ $Pname$, mens EMP_DEPENDENTS har triviell MVD Ename $\twoheadrightarrow$ $Dname$. En triviell MVD vil ikke gi noen meningsfull begrensning på relasjonen. En MVD som ikke tilfredsstiller betingelsene over kalles en **ikke-triviell MVD**, og **en relasjon med ikke-triviell MVD kan være overflødig** (dvs. verdier gjentas). EMP relasjonen er overflødig, men skjemaet er i BCNF siden det har ingen FDs. Vi trenger derfor å definere en fjerde normalform som er strengere enn BCNF og forbyr relasjoner som EMP. **Relasjoner som inneholder ikke-trivielle MVDs er ofte all-nøkler relasjoner** (dvs. nøkkelen er kombinasjonen av alle attributtene). Slike relasjoner er sjeldne, men det er viktig å gjenkjenne MVDs som en potensiell problematisk avhengighet.

Fjerde normalform (4NF)

Et relasjonsskjema er i fjerde normalform (4NF) hvis alle ikke-trivielle MVDs på formen $X \twoheadrightarrow Y$ i F^+, krever at X er en supernøkkel. Fjerde normalform blir brutt når relasjonen har uønskede MVDs, og den brukes for å identifisere å dekomponere slike relasjoner. For eksempel kan vi se på EMP som er i BCNF, siden det er en all-nøkkel relasjon (ingen FDs). EMP har MVD: Ename $\twoheadrightarrow$ Pname|Dname, siden Pname og Dname er to uavhengige flerverdi attributter og relasjonen har forholdene ansatt-prosjekt og ansatt-dependent. Dette er en ikke-triviell MVD siden den ikke oppfyller de to betingelsene. Siden Ename ikke er en supernøkkel vil EMP derfor ikke være i 4NF, og den må derfor dekomponeres til et sett av relasjoner i 4NF. Denne dekomponeringen vil fjerne redundansen som forårsakes av MVD.

EMP

Ename	Pname	Dname
Smith	X	John
Smith	Y	Anna
Smith	X	Anna
Smith	Y	John

4NF normalisering går ut på å dele opp relasjonen slik at hver MVD blir representert av en separat relasjon der den blir en triviell MVD. EMP er ikke i 4NF på grunn av de ikke-trivielle MVDs Ename $\twoheadrightarrow$ Pname og Ename $\twoheadrightarrow$ Dname, der Ename ikke er en supernøkkel. Derfor vil vi dele EMP inn i EMP_PROJECTS og EMP_DEPENDENTS som begge er i 4NF siden MVD er trivielle.

EMP_PROJECTS

Ename	Pname
Smith	X
Smith	Y

EMP_DEPENDENTS

Ename	Dname
Smith	John
Smith	Anna

Det er viktig å være forsiktig når man bruker disse reglene! For eksempel vil $X \rightarrow A$ og $X \rightarrow B$ gi at $X \rightarrow AB$, mens $X \rightarrow A$ og $Y \rightarrow B$ gir at $XY \rightarrow AB$. $XY \rightarrow A$ vil ikke nødvendigvis bety at $X \rightarrow A$ eller $Y \rightarrow A$.

Kapittel 15 – Designalgoritmer og videre avhengigheter

Kapittel 14 beskrev en top-down relasjonell designteknikk og relaterte konsepter som er mye brukt i databasedesign. I denne prosessen blir en ER/EER konseptuell modell designet og deretter kartlagt til en relasjonsmodell vha flere regler. Primærnøkler blir tildelt basert på kjente funksjonelle avhengigheter. I relasjonsdesign ved analyse vil relasjonene analyseres for å oppdage uønskede FDs. Disse blir fjernet ved normaliseringsprosessen for å oppnå høyere normalform og dermed bedre design. I dette kapittelet skal vi se videre på normalformene og funksjonelle og flerverdi avhengigheter.

15.1 Andre begrep i FDs: Inferensregler, ekvivalens og minimalt dekke

Problematiske funksjonelle avhengigheter kan elimineres av normalisering, der en relasjon blir dekomponert. Vi skal nå se hvordan nye avhengigheter kan utledes fra et gitt sett og diskutere konseptene for lukking og ekvivalens, som vi trenger i design av relasjoner gitt ett sett med FDs.

Inferens = følgeslutning/utledning

Inferensregler for FDs

F er settet av FDs som er spesifisert på et relasjonsskjema R. Skjemadesigneren vil som regel spesifisere FDs som er semantisk åpenbare, og **flere FDs kan utledes fra disse funksjonelle avhengighetene i F**. Disse kalles utledede funksjonelle avhengigheter.

En FD $X \rightarrow Y$ utledes fra et sett av avhengigheter F hvis $X \rightarrow Y$ holder i alle lovlige tilstander r hos R. Altså, hvis r tilfredsstiller alle avhengighetene i F, må $X \rightarrow Y$ gjelde for r for at den skal være en gyldig utledning. Det er umulig å spesifisere alle mulige FDs for en gitt situasjon. For eksempel hvis avdelingen har en manager, slik at avdelingsnummer bestemmer manageren (Dnum → Mgr_ssn) og manageren har et unikt mobilnummer (Mgr_ssn → Mgr_phone), så vil disse avhengighetene gi at Dnum → Mgr_phone. Dette er en utledet FD og den trenger ikke å eksplisitt uttrykkes (holder at de to andre FDs er gitt).

Settet av alle avhengigheter som inkluderer F og alle avhengigheter som kan utledes fra F kalles lukningen (*closure*) til F og betenges som F^+. For eksempel kan vi se på:

$$F = \{\text{Ssn} \rightarrow \{\text{Ename}, \text{Bdate}, \text{Address}, \text{Dnumber}\}, \text{Dnumber} \rightarrow \{\text{Dname}, \text{Dmgr_ssn}\}\}$$

Noen funksjonelle avhengigheter som kan utledes fra F er: $\text{Ssn} \rightarrow \{\text{Dname}, \text{Dmgr_ssn}\}$, $\text{Ssn} \rightarrow \text{Ssn}$ og $\text{Dnumber} \rightarrow \text{Dname}$. **Vi bruker et sett med inferensregler for å bestemme en systematisk måte å utlede nye avhengigheter fra et gitt sett med FDs.** Vi bruker notasjonen $F| = X \rightarrow Y$ når den funksjonelle avhengigheten $X \rightarrow Y$ utledes fra F, og $XYZ \rightarrow UV$ brukes for å representere $\{X, Y, Z\} \rightarrow \{U, V\}$. Tre kjente inferensregler for FDs, som kalles **Armstrongs aksiomer** er:

$$\textbf{IR1 (refleksiv)}: Y \subseteq X, \text{gir } X \rightarrow Y$$
$$\textbf{IR2 (augmentering)}: \{X \rightarrow Y\}| = XZ \rightarrow YZ$$
$$\textbf{IR3(transitiv)}: \{\mathbf{X \rightarrow Y, Y \rightarrow Z}\}| = X \rightarrow Z$$

Disse reglene er gyldige og fullstendige. Med gyldige menes det at enhver avhengighet som kan utledes fra F vha disse reglene vil gjelde for alle relasjonstilstander r av R som tilfredsstiller avhengighetene i F. Med **fullstendige**, menes det at hvis de brukes gjentatte ganger helt til det

Det er viktig å være forsiktig når man bruker disse reglene! For eksempel vil $X \to A$ og $X \to B$ gi at $X \to AB$, mens $X \to A$ og $Y \to B$ gir at $XY \to AB$. $XY \to A$ vil ikke nødvendigvis bety at $X \to A$ eller $Y \to A$.

ikke lenger kan utledes flere FDs, vil resultatet være et fullstendig sett av alle mulige FDs som kan utledes fra F. Disse reglene kan derfor alene brukes for å finne F^+ (lukningen til F). **Refleksiv regel sier at et sett med attributter vil alltid bestemme seg selv eller sine subsett.** IR1 vil alltid generere avhengigheter som er sanne, så disse kalles **trivielle avhengigheter.** En FD $X \to Y$ vil altså være triviell dersom $Y \subseteq X$, og ellers ikke-triviell. **IR2 sier at en ny gyldig avhengighet kan lages ved å legge til samme attributt på begge sider av en FD**, mens **IR3 sier at FDs er transitive**.

Tre andre inferensregler som følger fra IR1, IR2 og IR3 er:

$$\textbf{IR4 (dekomponering)}: \{X \to YZ\} | = X \to Y$$
$$\textbf{IR5 (additiv)}: \{X \to Y, X \to Z\} | = X \to YZ$$
$$\textbf{IR6(pseudotransitiv)}: \{X \to Y, WY \to Z\} | = WX \to Z$$

Dekomponeringsregelen sier at vi kan fjerne attributter fra høyre-siden av FD, så hvis denne regelen gjentas kan den dekomponere en enkel FD $X \to \{A_1, A_2, \dots, A_n\}$ til et sett av avhengigheter $\{X \to A_1, \dots, X \to A_n\}$. **Union regelen lar oss gjøre det motsatte, altså kombinere settet av avhengigheter til en enkel FD**. **Pseudotransitiv regel lar oss erstatte et sett med attributter Y på venstre-siden av FD med et annet sett X som funksjonelt bestemmer Y**. Disse reglene ses på som konsekvenser av Armstrongs inferensregler.

Databasedesignere vil som regel først spesifisere et sett med funksjonelle avhengigheter F basert på semantikken til attributtene i relasjonen. Deretter vil de bruke IR1, IR2 og IR3 for å utlede flere FDs som vil gjelde på R. **En systematisk måte å gjøre dette, er å først bestemme hvert sett med attributter X som er på venstre side av FDs i F og deretter bestemme settet X^+ med alle attributter som bestemmes av X. Vi kaller X^+ for lukningen av X under F**, og det er altså alle attributtene på høyre side av FDs.

Figuren til høyre viser algoritmen som brukes for å regne ut X^+. Algoritmen begynner med å sette X^+ lik alle attributtene i X, siden IR1 gir at alle attributter vil bestemme seg selv. Deretter vil den gå igjennom alle FDs i F. Hvis $Y \to Z$ og Y er i X^+, betyr det at Z skal legges til X^+. Videre kan det hende at en annen attributt blir lagt til i X^+ fordi $Z \subseteq X^+$. Denne algoritmen benytter seg altså av transitiv regel. Prosessen vil gjentas helt til en iterasjon av for-løkken ikke vil legge til noen nye attributter ($X^+ = \text{oldX}^+$).

Input: F (sett med FDs på relasjonen) og X (sett med attributter i relasjonen)

```
X⁺ = X;
repeat
    oldX⁺ = X⁺
    for each Y → Z i F
     if (Y ⊆ X⁺)
          X⁺ = X⁺ ∪ Z
until (X⁺ = oldX⁺)
```

Lukninger kan brukes for å forstå attributter eller sett med attributter i en relasjon. For eksempel kan vi ha følgende relasjon og FDs:

CLASS(ClassID, CourseNr, InstrName, CreditHrs, Text, Publisher, Classrom, Capacity)

ClassID → {CourseNr, InstrName, CreditHrs, Text, Publisher, Classrom, Capacity}
CourseNr → CreditHrs
{CourseNr, InstrName} → {Text, Classroom}
Text → Publisher
Classroom → Capacity

Disse funksjonelle avhengighetene uttrykker bestemt semantikk om dataen i relasjonen, for eksempel vil FD1 gi at hver klasse har en unik ClassID. Hvis vi bruker inferensreglene og definisjonen av lukning, vil vi finne følgende lukninger:

$$\{\text{ClassID}\}^+ = \{\text{ClassID}, \text{CourseNr}, \text{InstrName}, \text{CreditHrs}, \text{Text}, \text{Publisher}, \text{Classrom}, \text{Capacity}\} = \text{CLASS}$$
$$\{\text{CourseNr}\}^+ = \{\text{CourseNr}, \text{CreditHrs}\}$$
$$\{\text{CourseNr}, \text{InstrName}\}^+ = \{\text{CourseNr}, \text{CreditHrs}, \text{Text}, \text{Publisher}, \text{Classrom}, \text{Capacity}\}$$

Hver lukning vil ha en tolkning som sier noe om attributtene på venstre side. For eksempel kan vi se at lukningen av CourseNr kun har CreditHrs i tillegg til seg selv. Emnenummeret vil ikke bestemme InstrName, fordi samme emnet kan undervises av ulike instruktører, og det vil ikke bestemme Text, fordi samme emnet kan bruke ulike tekster. Vi kan også se at lukningen $\{\text{CourseNr}, \text{InstrName}\}$ ikke inkluderer ClassID, noe som betyr at det ikke er en kandidatnøkkel. Dermed kan vi ha ulike klasser med samme emnenummer og instruktørnavn.

Ekvivalens av sett med FDs

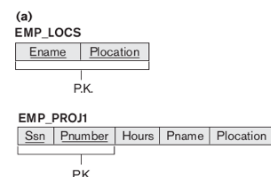
To viktige definisjoner:

- **Et sett med FDs F sies å dekke et annet sett med FDs E, hvis alle FDs i E også er i F^+, altså alle avhengigheter i E kan utledes fra F**. For å sjekke om F dekker E må vi regne ut X^+ med hensyn til F for alle FDs $X \rightarrow Y$ i E, og deretter sjekke om X^+ inkluderer attributtene i Y. Hvis dette er tilfellet for alle FDs i E, kan vi si at F dekker E fordi da vil alle funksjonelle avhengigheter i E være tilfredsstilt.
- **To sett med funksjonelle avhengigheter E og F er ekvivalente hvis $E^+ = F^+$**. Dvs. alle FDs i E kan utledes fra F og alle FDs i F kan utledes fra E. For å sjekke om F og E er ekvivalente må vi altså sjekke at E dekker F og at F dekker E.

15.2 Egenskaper ved dekomponering av relasjoner

I kapittel 14 brukte vi dekomponering av relasjoner for å bli kvitt uønskede avhengigheter og oppnå høyere normalformer. **Normalform vil ikke alene fungere som en garanti på et godt design, fordi relasjonsskjemaet må i tillegg ha andre bestemte egenskaper.** To av disse egenskapene er **bevaring av funksjonelle avhengigheter** og **tapsløs-join egenskapen**.

Dekomponering og utilstrekkeligheten til normalformer



Design av relasjonell database kan starte med et **universelt relasjonsskjema R** som inkluderer alle attributtene til databasen. Universell relasjonsantagelsen sier at alle attributter har unike navn. Designerne vil spesifisere F som inneholder FDs på attributtene i relasjonen. **Disse avhengighetene blir brukt for å dekomponere det universelle relasjonsskjemaet til et sett med relasjonsskjemaer D, som kalles dekomponeringen av R**. Det er viktig at hvert attributt i R vil være i minst ett relasjonsskjema i dekomponeringen slik at ingen attributter blir tapt. Dette kalles **attributtbevaring**.

Et annet mål er at alle individuelle relasjoner i dekomponeringen er i BCNF eller 3NF. **Denne betingelsen er ikke tilstrekkelig for å alene sikre et godt databasedesign.** Vi må se på dekomponeringen som et hele, i tillegg til å se på individuelle relasjoner. For eksempel kan vi se på EMP_LOCS relasjonen som er i BCNF (merk: alle relasjoner med to attributter er i BCNF). Selv om EMP_LOCS er i BCNF vil det lages falske tupler når den slås sammen med

EMP_PROJ1, som ikke er i BCNF. Dette skyldes at Plocation ikke er en primær- eller fremmednøkkel i noen av lokasjonene. Dette illustrerer at det trengs andre kriterier som sammen med betingelsen for 3NF eller BCNF vil hindre slike dårlige design.

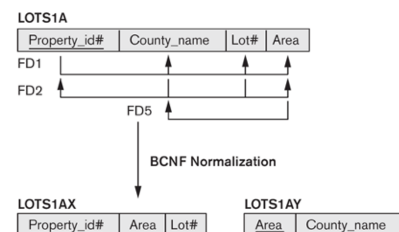
Bevaring av FDs

Hver FD spesifisert i F bør enten være direkte tilstede eller kunne utledes fra en av relasjonsskjemaene i dekomponeringen. Dette kalles **bevaring av funksjonelle avhengigheter** og er ønsket siden hver avhengighet i F representerer en begrensning på databasen. Manglende avhengigheter kan ikke håndteres ved å se på en individuell relasjon, fordi det kan kreve join av flere relasjoner, slik at alle attributtene i avhengigheten blir inkludert.

Avhengighetene etter dekomponeringen trenger ikke å være nøyaktig slik de ble spesifisert i F. **Det er tilstrekkelig at unionen av avhengighetene i de individuelle relasjonene er ekvivalente med F**. Projeksjonen av F på relasjonen R_i betegnes som $\pi_{R_i}(F)$ og er settet av avhengigheter $X \rightarrow Y$, der X og Y er attributter i R_i. Det er altså FDs der relasjonen har attributtene på begge sidene. En dekomponering $D = \{R_1, R_2, \dots, R_m\}$ er avhengighetsbevarende dersom unionen av projeksjonene til F er ekvivalent med F, altså:

$$\left(\pi_{R_1}(F) \cup \dots \cup \pi_{R_m}(F)\right)^+ = F^+$$

Altså hvis alle FDs som er direkte tilstede eller kan utledes er de samme før og etter dekomponeringen, vil avhengighetene være bevart. Hvis en dekomponering ikke er avhengighetsbevarende vil noen avhengigheter bli tapt ved dekomponeringen. For å sjekke om en tapt avhengighet holder må vi ta JOIN på to eller flere relasjoner, slik at vi får en relasjon som har alle attributtene i den tapte avhengigheten. Deretter må vi sjekke om avhengigheten holder. Dette er ikke en praktisk løsning! Et eksempel på en dekomponering som ikke er avhengighetsbevarende er oppdelingen av LOTS1A til LOTS1AX og LOTS1AY. Her blir FD2 tapt i dekomponeringen. **Det er alltid mulig å finne en avhengighetsbevarende dekomponering, der alle relasjonene er i 3NF.**



Tapsløs-join (ikke-additiv) egenskapen

En dekomponering bør ha tapsløs-join egenskapen, som sikrer at ingen falske tupler blir generert når en NATURAL JOIN blir brukt på relasjonene fra dekomponeringen. Dette er en egenskap ved dekomponering av relasjonsskjemaer, så betingelsen for ingen falske tupler skal holde for alle lovlige relasjonstilstander (dvs. tilfredsstiller FDs i F). Tapsløs-join blir derfor definert mht. et spesifikt sett F.

En dekomponering $D = \{R_1, R_2, \dots, R_m\}$ har tapsløs-join egenskapen hvis følgende gjelder for alle relasjonstilstander r som tilfredsstiller FDs i F:

$$\pi_{R_1}(r) * \dots * \pi_{R_m}(r) = r$$

Altså naturlig join av relasjonstilstandene etter dekomponeringen vil gi relasjonstilstanden før dekomponeringen. Ordet tap i tapsløs-join referer til tap av informasjon, siden falske tupler

vil representere ugyldig informasjon. Ikke-additiv er mer beskrivende, siden det betyr at ingen falske tupler blir lagt til (*added*) resulterende relasjon. Denne egenskapen sikrer at **ingen falske tupler blir lagt til resultatet etter bruken av PROJECT eller JOIN operatorer.**

På side 99 så vi at NJB testen kan brukes for å sjekke om en binær dekomponering (dvs. oppdeling i to relasjoner) har tapsløs-join egenskapen. Under ser vi på **en algoritme som kan brukes for å sjekke om en dekomponering inn i n relasjoner har denne egenskapen**:

Input: R (universal relasjon) $D = \{R_1, \dots, R_m\}$ (dekomponering) og F (sett med FDs)

1. Lag **en initial matrise S** med en rad i for hver relasjon R_i i D og en kolonne for hvert attributt A_j i R

2. Sett $\boldsymbol{S(i,j) = b_{ij}}$ for alle matriseenhetene (eks: $S(1,3) = b_{13}$)

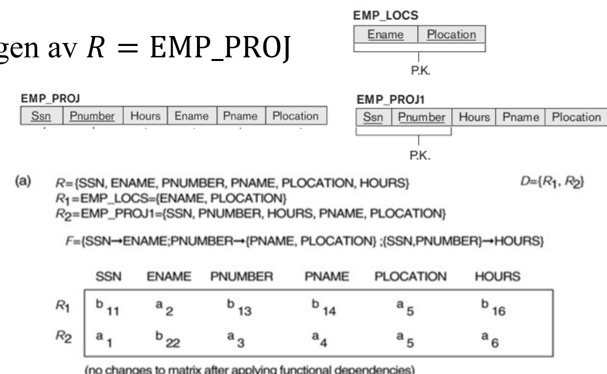
3. For hver rad i som representerer et relasjonsskjema R_i
   For hver kolonne j som representerer et attributt A_j
   Hvis $\boldsymbol{R_i}$ **har** $\boldsymbol{A_j}$ **vil** $\boldsymbol{S(i,j) = a_j}$ (eks: $S(1,3) = a_3$)
   Dette steget vil gjøre at alle relasjonene som har ett bestemt attributt vil få samme a-symbol for dette attributtet.

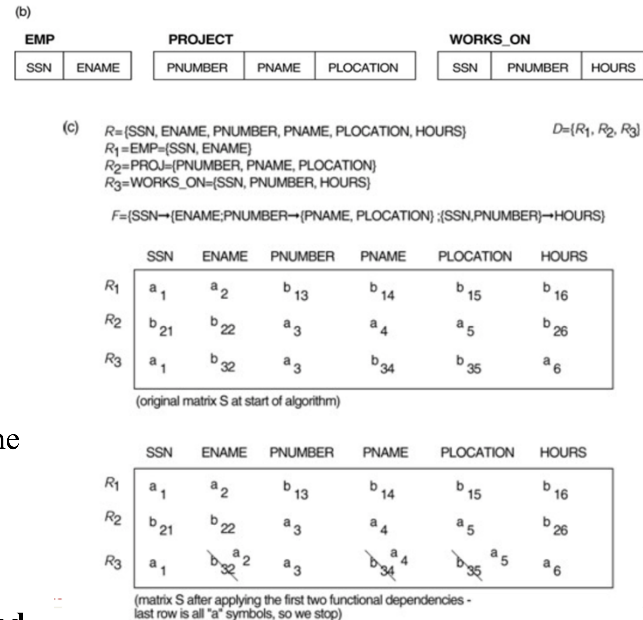
4. Gjenta følgende loop helt til en fullstendig utføring av loopen ikke endrer S:
   For hver FD $X \rightarrow Y$ i F
   For alle radene i S som har *samme symbol* i kolonnene som korresponderer til attributter i X
   For disse radene, skal kolonnen som representerer Y-attributtet bli like. Hvis noen har et a symbol i denne kolonnen, skal denne kolonnen til de andre radene settes til samme a symbol. Hvis ingen kolonner har a symbolet, skal en av b symbolene velges og kolonnen til de andre radene settes til samme b symbol

5. **Hvis en rad har kun a symboler, vil dekomponeringen ha tapsløs-join (ikke-additiv) egenskapen.**

Algoritmen vil begynne med matrisen S, der hver rad representerer en relasjon R_i med a symboler for kolonnene som korresponderer til attributter i R_i og b symboler i resten av kolonnene. Steg 4 vil gå igjennom alle FDs $X \rightarrow Y$ og ser først på X-attributtene. Hvis noen rader har samme verdi for disse attributtene, vil Y-attributtene endres til samme verdi (enten a- eller b-symboler). Hvis noen rader har kun a symboler, vil dekomponeringen ha tapsløs-join egenskapen mht. F. Hvis ingen rader ender opp med kun a symboler vil ikke dekomponeringen ha tapsløs-join egenskapen.

Figur a viser hvordan algoritmen kan brukes på dekomponeringen av $R = \text{EMP_PROJ}$ inn i $R_1 = \text{EMP_LOCS}$ og $R_2 = \text{EMP_PROJ1}$. R_1 har attributtene Ename og Plocation, mens R_2 har alle attributtene bortsett fra Ename, så disse blir endret til a symboler. Det er kun Plocation som har samme symbol i de to radene, men den er kun på høyre side av FD. Derfor vil ikke steg 4 endre matrisen. **Denne dekomponeringen vil ikke ha tapsløs-join egenskapen, siden ingen rader har kun a symboler**.





Figur b viser hvordan algoritmen kan brukes på dekomponeringen av R = EMP_PROJ inn i R_1 = EMP, R_2 = PROJECT og R_3 = WORKS_ON. Etter steg 3 vil vi ha den øverste matrisen, der vi har a-symboler som representerer attributtene i relasjonene. I steg 4 vil matrisen endres:

- R_1 og R_3 har Ssn = a_1, og siden F inneholder Ssn → Ename, vil radene få samme verdi for Ename. Dette vil være et a-symbol siden R_1 har Ename = a_2
- R_2 og R_3 har Pnumber = a_3, og siden F inneholder Pnumber → {Pname, Plocation}, vil radene få samme verdi for disse. Dette vil være et a-symboler siden R_2 har Pname = a_4 og Plocation = a_5

Den nederste matrisen viser resultatet av steg 4. **Denne dekomponeringen vil ha tapsløs-join egenskapen, siden rad 3 består av kun a-symboler.**

NJB-test av dekomponering

Algoritmen over lar oss teste om en dekomponering til n relasjoner har tapsløs-join egenskapen med hensyn til et sett F med FDs. **Binær dekomponering er når relasjonen deles inn i to relasjoner, og da kan vi bruke NJB testen for å se om dekomponeringen har tapsløs-join egenskapen**. Denne testen er enklere enn algoritmen over, men den er altså begrenset til binær dekomponering.

Påfølgende tapsløs-join dekomponering

I kapittel 14 så vi at 2NF og 3NF normalisering involverer påfølgende dekomponering. For å verifisere at disse dekomponeringene er ikke-additive bruker vi følgende påstand:

Bevaring av ikke-additivet i påfølgende dekomponering: Hvis en dekomponering $D = \{R_1, R_2, \dots, R_m\}$ av R har tapsløs-join egenskapen og en dekomponering $D_i = \{Q_1, Q_2, \dots, Q_k\}$ av R_i har tapsløs-join egenskapen, så vil dekomponering $D_2 = \{R_1, R_2, \dots, R_{i-1}, Q_1, Q_2, \dots, Q_k, R_{i+1},$ ..., $R_m\}$ ha tapsløs-join egenskapen.

R

A	B	C
a1	b1	c1
a2	b1	c1

t_2 $t_2[B]$

R(A,B,C) — skjema for tabellen

r(R) — tabellforekomsten

$t_i \in r(R)$ — rad/tuppel i tabellforekomsten

$t_i[A]$ — radens verdi for attributtet A

$X \subseteq R$ — en delmengde av attributtene i R

Oppsummering – kapittel 14 og 15 (F9, F10, F11)

Figuren opp til høyre viser terminologi vi bruker.

Motiverende eksempel (F9)

Tabellen til høyre viser en relasjon kalt Eksamen, der StudNr og EmneNr er nøkkelen. Vi har følgende restriksjoner:

- StudNr bestemmer StudNavn
- EmneNr bestemmer EmneNavn
- StudNr og EmneNr bestemmer Karakter.

Eksamen

StudNr	StudNavn	EmneNr	EmneNavn	Karakter
1	Ole	1	DB	A
2	Kari	1	DB	A
1	Ole	2	OS	B
1	Ole	3	ITGK	B

Siden StudNr og EmneNr er nøkkelen, betyr det at det er bare ett studentnavn per studentnummer og ett emnenavn per emnenummer. Kombinasjonen av StudNr og EmneNr vil ha bare én karakter. Denne relasjonen har flere dårlige egenskaper. For eksempel vil kombinasjonene StudNr-StudNavn og EmneNr-EmneNavn føre til redundans, siden samme informasjon er lagret flere ganger. **Redundans er et problem siden det bruker unødvendig lagringsplass, gjør oppdateringen mer kompleks og åpner en mulighet for at databasen kan bli uenig med seg selv (inkonsistens).** Det fører også til at tabellen er utsatt for tre typer anomalier:

1. **Innsettingsanomali** - du ikke får registrert en student uten at den har et emne (og motsatt)
2. **Oppdateringsanomali** – oppdatering av navnet på for eksempel EmneNr 1 må gjentas for mange tupler
3. **Slettingsanomali** – sletting av StudNr 1 Ole, vil før til tap av informasjonen om at det finnes et emnenummer 3 med navn ITGK

Student

StudNr	StudNavn
1	Ole
2	Kari

Emne

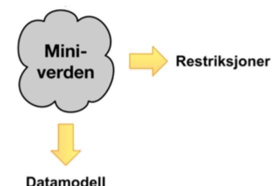
EmneNr	EmneNavn
1	DB
2	OS
3	ITGK

Eksamen

StudNr	EmneNr	Karakter
1	1	A
2	1	A
1	2	B
1	3	B

Et bedre design hadde vært å delt opp relasjonen i tre tabeller: Student, Emne og Eksamen. I Student og Emne vil det ikke være redundans, siden StudNr og EmneNr er entydig, slik at hver student og emne har kun én rad. I Eksamen vil StudNr og EmneNr sammen være nøkkelen, og begge er fremmednøkler som referere til hhv. Student og Emne. Dette designet har kvittet seg med problemene som første tabell hadde. **Dekomponeringen gir et bedre logisk design som er viktig for effektiv innsetting, sletting og oppdatering av data, men spørreytelsen blir potensielt dårligere**. Mindre tabeller unngår redundans og anomalier, og blander ikke «urelatert» data i samme tabell, men flere tabeller betyr flere joins.

Restriksjoner (F10)



Vi har en miniverden som vi bruker for å lage en database. Denne miniverden vil gi opphav til en datamodell med entitetsklasser, relasjonsklasser, attributter, osv. Den vil også gi restriksjoner som er regler som gjelder for miniverden. En databasetilstand er databaseforekomsten ved et gitt tidspunkt, og den er konsistent når alle reglene i miniverden er oppfylt. **Restriksjoner vil altså begrense hvilke data som kan finnes i en konsistent databasetilstand**. Hvis vi ikke følger reglene i miniverden lar vi databasen å modellere en tilstand som egentlig ikke skulle ha vært lovlig.

Vi skiller mellom tre typer begrensninger som må ivaretas for å få en konsistent database (s. 33):

- **Arvede restriksjoner (*inherent*)** – er en del av datamodellen og håndheves av DBMS. For eksempel i relasjonsmodellen kan ikke to rader være like.

- **Eksplisitte restriksjoner** – uttrykkes i datamodellen (skjemaet) og håndheves av DBMS. For eksempel kan vi bestemme primær- og fremmednøkkel i relasjonsmodellen. Vi er nødt til å gi DBMS eksplisitt beskjed om disse restriksjonene i skjemaet.
- **Applikasjonsbaserte restriksjoner** – håndteres utenfor datamodellen som en del av applikasjonsprogrammet. For eksempel at ingen kan tjene mer enn sjefen.

En delmengde av alle restriksjoner kan uttrykkes som funksjonelle avhengigheter (FDs), og disse er grunnlaget for viktige designregler. Et eksempel er StudNr → StudNavn som sier at studentnummer bestemmer studentnavn. Alle rader som har samme verdi for StudNr, må ha samme verdi for StudNavn. Altså, **alle rader som har lik venstre-side attributt må også ha lik høyre-side attributt**. **Dette er en *sterk* sammenheng mellom attributter**. Merk: det kan være flere StudNr som har samme StudNavn, så StudNavn → StudNr gjelder ikke.

$X \rightarrow Y$ der $X, Y \subseteq R$ vil altså uttrykke en restriksjon på alle lovlige tabellforekomster. Alle tupler t_i og t_j som har samme verdier for attributtene i X, dvs. $t_i[X] = t_j[X]$, må ha samme verdier for attributtene i Y, altså $t_i[Y] = t_j[Y]$. X vil ofte være nøkkelattributt, siden de må være unike for ulike tupler og dermed vil verdien til nøkkelattributtet bestemme verdiene til resten av attributtene i tuplen.

Hvilke FDs gjelder for en tabell

A	B	C
a1	b1	c1
a1	b1	c2
a1	b2	c3

Basert på tabellen på figuren kan vi se på følgende funksjonelle avhengigheter:

- $A \rightarrow B$: kan ikke gjelde, siden radene har samme A, men ikke samme B. Vi ser at $t_1[A] = t_3[A]$, men $t_1[B] \neq t_3[B]$
- $B \rightarrow A$: kan kanskje gjelde, siden radene med samme B, har samme A. Vi ser at $t_1[B] = t_2[B]$, og $t_1[A] = t_2[A]$
- $AC \rightarrow B$: kan kanskje gjelde, siden C har ulike verdier i alle radene (ingen moteksempel)
- $A \rightarrow A$: vil alltid gjelde. Denne er triviell ($A \subseteq A$) og representerer ingen restriksjon.

Legg merke til at vi sier «kan kanskje gjelde», siden **relasjonstilstander ikke er tilstrekkelig for å bestemme når en FD gjelder**. De kan brukes for å finne mulige forslag, som deretter må bekreftes ved å se på semantikken. For $A \rightarrow A$ kan vi si «Ja», siden et attributt alltid vil funksjonelt bestemme seg selv. Vi kan si «Nei» fordi det holder å finne ett moteksempel og det kan vi vha en relasjonstilstand.

Utledningsregler

For funksjonelle avhengigheter har vi et sett med utledningsregler (se figur). De tre første reglene kalles Armstrongs aksiomer og disse kan brukes for å utlede alle FDs fra et gitt utgangspunkt. De tre neste er likevel nyttige fordi de gjør det enklere å utlede FDs i praksis.

IR-1 (reflexive):	Hvis Y ⊆ X så X → Y
IR-2 (augmentation):	{ X → Y } gir XZ → YZ
IR-3 (transitive):	{ X → Y, Y → Z} gir X → Z
IR-4 (decomposition):	{ X → YZ } gir X → Y
IR-5 (additive):	{ X → Y, X → Z } gir X → YZ
IR-6 (pseudotransitive):	{ X → Y, WY → Z } gir WX → Z

X, Y, Z, W ⊆ R (mengden av alle attributter)

IR-1 + IR-2 + IR-3 kalles *Armstrongs aksiomer* og er tilstrekkelig for å utlede alle funksjonelle avhengigheter fra et gitt utgangspunkt (en mengde funksjonelle avhengigheter).

Eksempel på utledning av FD

Vi har relasjonen $R(A, B, C)$ med funksjonelle avhengigheter $F = \{A \rightarrow B;\ B \rightarrow C\}$. Vi kan bruke utledningsreglene for å finne FDs:

- $ABC \rightarrow A$, siden $A \subseteq ABC$ (IR1)
- $AC \rightarrow BC$, siden $A \rightarrow B$ (IR2)

- $A \to C$, siden $A \to B$ og $B \to C$ (IR3)
- $AC \to B$, siden $AC \to BC$ (IR4)
- $A \to BC$, siden $A \to B$ og $A \to C$ (IR5)
- $AC \to BC$ siden $A \to B$ og $BC \to BC$ (IR6)
- ...

Tillukning

Hvis vi finner alle funksjonelle avhengigheter som vil gjelde gitt R og F, finner vi F^+ som er tillukningen til F. Dette inkluderer alle avhengighetene som er gitt i F og kan utledes fra disse:

$$F^+ = \{X \to Y | X \to Y \text{ kan utledes fra FDs i } F\}$$

Det er altså en opplisting av alle funksjonelle avhengigheter vi kan finne fra F. **Legg merke til at F og F^+ vil uttrykke akkurat samme restriksjon**. For eksempel ved transitivitet vil $A \to B$ og $B \to C$ i F føre til at F^+ inneholder $A \to C$. Dette er ikke en strengere restriksjon, fordi den har vært der hele tiden. Det blir ikke lagt til noen nye regler, de blir bare formulert på en annen måte. **F^+ vil inneholde mange trivielle FDs**, for eksempel vil ikke $X \to Y$ der $Y \subseteq X$ utrykke en restriksjon. Det er krevende å regne ut lukningen, men heldigvis er det sjeldent interessant.

Tillukning til attributtmengde

Det er vanligere å få bruk av tillukningen til en mengde attributter. Vi har gitt en relasjon R med ett sett funksjonelle avhengigheter F og et sett med attributter $X \subseteq R$. Tillukningen til en mengde attributter er:

$$X^+ = \{Y \in R | X \to Y \in F^+\}$$

```
X⁺ = X;
repeat
   oldX⁺ = X⁺;
   for each Y→Z ∈ F do
      if Y⊆X⁺ then
         X⁺ = X⁺∪Z;
until X+ = oldX+;
```

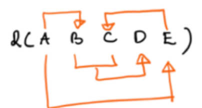
Dvs. X^+ er alle attributter som er funksjonelt avhengig av X, slik at $X \to X^+$. Figuren viser algoritmen for å finne X^+. Vi begynner med å si at $X^+ = X$ (IR1), deretter vil vi starte en løkke. Algoritmen vil gå gjennom alle funksjonelle avhengigheter i F og dersom hele venstre-siden av FD er med i X^+, skal høyre-siden legges til X^+ (IR3). Dette vil gjentas helt til det ikke blir lagt til noen nye attributter i X^+.

Vi kan for eksempel se på relasjonen $R(A, B, C, D, E)$ som har $F = \{A \to B;\ BC \to D;\ E \to C;\ A \to E\}$. Vi ser på tillukningen av noen attributtmengder:

- $A^+ = ABECD = ABCDE = R$
- $B^+ = B \subset R$
- $C^+ = C \subset R$
- $D^+ = D \subset R$
- $E^+ = EC \subset R$
- $BE^+ = BECD = BCDE \subset R$

Disse finner vi ved å bruke algoritmen. For eksempel for A^+ vil vi begynne med å legge til A. Deretter vil første iterasjon av løkken legge til B og E, siden A^+ inneholder A som er på venstre side av $A \to B$ og $A \to E$. Andre iterasjon vil legge til C, siden A^+ inneholder E som er på venstre side av $E \to C$. Tredje iterasjon vil legge til D, siden A^+ inneholder BC som er på venstre side av $BC \to D$. Fjerde iterasjon vil ikke legge til noen nye attributter, så dermed vil løkken termineres. Vi har at $A^+ = ABCDE$ som er alle attributtene i relasjonen R, så $A^+ = R$

På pilfiguren kan vi se at det er ingen piler som peker inn mot A, noe som betyr at eneste mulighet for at A blir med på en tillukning, er at den står på venstre side.



OBS: hele venstre side av FD må være i X^+ for at høyre side skal legges til. For eksempel vil ikke D legges til B^+ fordi den inneholder kun B og FD er $BC \to D$

SpeciesID	SpeciesName	BirdGroup	Prevalence
1	Redwing	Thrushes	Migratory
2	Blackbird	Thrushes	Resident
3	Raven	Crows	Resident
4	Magpie	Crows	Resident
5	Chaffinch	Finches	Resident

Eksamensoppgave

Ta utgangspunkt i tabellen $Birds(SpeciesID, SpeciesName, BirdGroup, Prevalence)$ med tabellforekomst på figuren. Hvilken FDs er det rimelig å anta vil gjelde for denne tabellen? Forklar de forutsetningene du legger til grunn. Du trenger ikke å ta med trivielle FDs eller FDs som kan utledes fra avhengighetene i svaret ditt.

Forutsetningen er at vi antar at SpeciesID er unik for hver fugleart, slik at det blir et nøkkelattributt. Dette vil gjøre at den vil bestemme resterende attributter:

- $\text{SpeciesID} \rightarrow \{\text{SpeciesName}, \text{BirdGroup}, \text{Prevalence}\}$

En annen forutsetning er at SpeciesName også er en entydig identifikator for fuglearter, slik at det er et alternativt nøkkelattributt. Dette vil gjøre at den vil bestemme SpeciesID og dermed resten av attributtene via transitivitet:

- $\text{SpeciesName} \rightarrow \text{SpeciesID}$

Tillukning i definisjon av nøkler

Tillukning brukes for å definere supernøkler og nøkler:

- **Supernøkkel = en mengde attributter $\boldsymbol{S}$, slik at ingen forekomster av tabellen kan ha to tupler $\boldsymbol{t_i}$ og $\boldsymbol{t_j}$ med samme verdier for $\boldsymbol{S}$** (dvs. $t_i[S] \neq t_j[S]$ for $i \neq j$). Supernøkkelen vil altså være en unik identifikator for tabellen**. For at $\boldsymbol{S}$ skal være en supernøkkel, må $\boldsymbol{S^+ = R}$.** For å identifisere en supernøkkel fra gitt R og F, kan vi derfor se om tillukningen inkluderer alle attributter i R. Vi kan legge til et attributt og fortsatt ha en supernøkkel.
- **Nøkkel** = en minimal supernøkkel K, slik at fjerning av ett attributt fra K vil gjøre at vi ikke lenger har en supernøkkel. Alle nøkler er supernøkler, noen supernøkler er nøkler.

R(A, B, C, D) og F = { A → B; B → ACD }

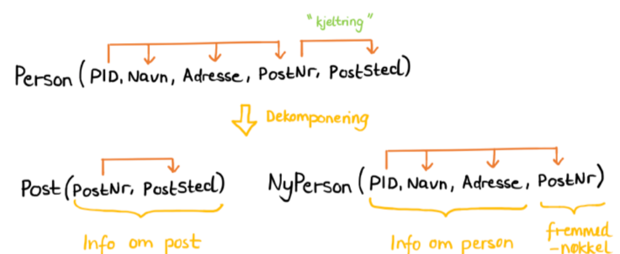
	Supernøkkel?	Nøkkel?
$A^+ = ABCD = R$	Ja	Ja
$B^+ = ABCD = R$	Ja	Ja
$C^+ = C \subset R$	Nei	Nei
$AB^+ = ABCD = R$	Ja	Nei
$AD^+ = ABCD = R$	Ja	Nei
$CD^+ = CD \subset R$	Nei	Nei

Problemet med supernøkler er at de kan være veldig store, siden vi kan legge til attributter og fortsatt ha en supernøkkel. Vi bruker supernøkler for å snakke om alle mulige unike identifikatorer. Nøkler er mer interessante, fordi det er supernøkler som ikke inneholder noe overflødig og er dermed en sterkere restriksjon. Figuren viser noen eksempler. Én-attributt supernøkler (eks: A og B), vil alltid være nøkler. AD vil ikke være en nøkkel, fordi A alene er en supernøkkel, så derfor kan vi fjerne D.

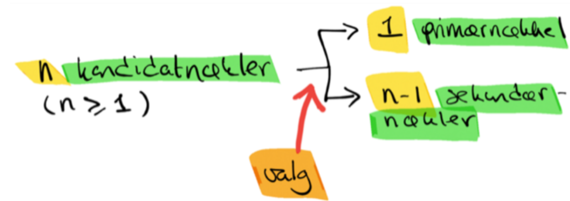
Vi har relasjonen $\text{Person}(\text{PID}, \text{Navn}, \text{Adresse}, \text{PostNr}, \text{PostSted})$ med $F = \{\text{PID} \rightarrow \text{Navn}, \text{Adresse}, \text{PostNr}; \text{PostNr} \rightarrow \text{PostSted}\}$ fra forutsetninger . Vi ser at:

$$\text{PID}^+ = \text{PID}, \text{Navn}, \text{Adresse}, \text{PostNr}, \text{PostSted} = \text{R} \quad \Rightarrow \quad \text{PID er supernøkkel/nøkkel}$$

Vi kan også observere at sammenhengen PostNr og PostSted blir lagret mange ganger, noe som fører til redundans og dermed innsettings- og slettingsanomalier. For å unngå dette bør vi derfor dekomponere Person relasjonen til $\text{Post}(\text{PostNr}, \text{PostSted})$ som kun gir info om post og $\text{NyPerson}(\text{PID}, \text{Navn}, \text{Adresse}, \text{PostNr})$ som kun gir info om person og har fremmednøkkel til mer post-info. Disse tabellene vil ikke ha redundans.



En attributtmengde er supernøkkel hvis den kan unikt identifisere en tuple og den er en kandidatnøkkel hvis det er en minimal supernøkkel

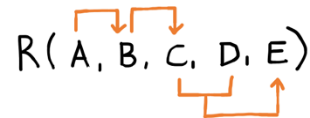


Kandidat-, primær- og sekundærnøkler

Alle tabeller må ha minst én nøkkel. Hvis den har flere nøkler kaller vi dem for **kandidatnøkler**, og **en av disse velges til primærnøkkelen.** De øvrige kandidatnøklene vil utgjøre tabellens **sekundærnøkler**, som er alternative nøkler. Selv om en kandidatnøkkel ikke velges som primærnøkkel, vil restriksjonen om unik verdi fortsatt gjelde! Primærnøkkelen vil brukes i blant annet fremmednøkler, og man bør velge kandidatnøkkelen som oftest har verdi. For eksempel vil PersonNr være bedre enn StudentNr, fordi alle har personnummer, men ikke alle er studenter. Det kan også være bedre å velge et kompakt heltallsfelt, enn for eksempel en tekststreng.

Oppgave (F11)

Gitt $R = \{A, B, C, D, E\}$ *og* $F = \{A \to B, B \to C, CD \to E\}$. *Finn alle kandidatnøklene.*
For at et attributt skal være en kandidatnøkkel må $X^+ = R$ og den må være minimal. Vi må derfor finne tillukningen til attributtene. **Siden *A* og *D* ikke er på noen høyre-sider, må de være med i alle supernøkler.** Derfor ser vi på:



- $A^+ = ABC \subset R$
- $D^+ = D \subset R$
- $AD^+ = ABCDE = R \quad \Rightarrow \quad AD$ er supernøkkel

Merk: eneste måte å nå attributter som ikke er på høyre-side av FDs i F er ved å inkludere de i nøklene!

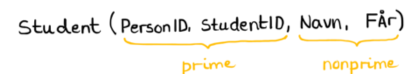
Siden AD er en minimal supernøkkel (kan ikke fjerne A eller D) vil den være eneste nøkkel.

Nøkkel- og ikke-nøkkelattributt

Vi skiller mellom:

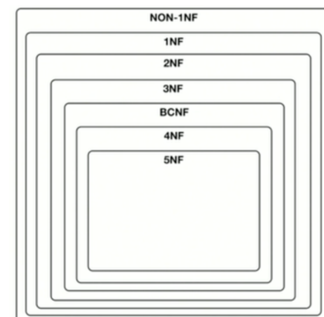
- **Nøkkelattributt (*prime*)** – attributter som inngår i en eller flere kandidatnøkler
- **Ikke-nøkkelattributt (*nonprime*)** – attributter som ikke inngår i noen kandidatnøkler

Dette vil partisjonere attributtene i en tabell i to deler (se figur). Merk at det er kandidatnøkkel og ikke supernøkkel, fordi alle attributtene kan inngå i en supernøkkel.



Normalformer

Normalformer er regler som stiller stadig strengere krav til tabeller for å sikre at tabellene unngår uheldige egenskaper. Alle tabeller som er på 2NF er på 1NF, alle på 3NF er på 2NF, osv. Når tabellen er i 5NF har vi løst problemene.



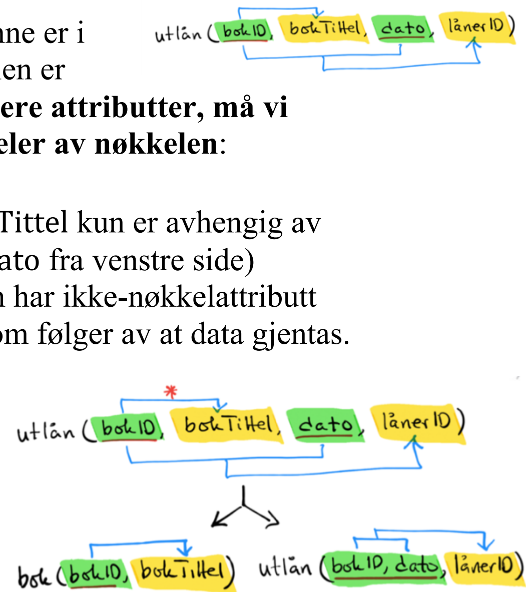
Første normalform (1NF)

For tabeller i første normalform (1NF) vil attributtenes domener inneholde atomiske (udelelige) verdier og verdien til et attributt er en enkelt verdi fra domenet. Dette sikrer flate eller 2-dimensjonale tabeller, siden vi unngår sammensatte attributter, fler-verdi attributter og nøstede tabeller.

Andre normalform (2NF)

For tabeller i andre normalform (2NF) vil ingen ikke-nøkkelattributt være delvis avhengig av en kandidatnøkkel. Ved en full funksjonell avhengighet $X \to Y$ kan vi ikke fjerne et attributt $A \in X$ på venstre side, uten at avhengigheten slutter å gjelde (dvs. venstre side har ingen «overflødige» attributter). Dette er en «sterkere» regel enn delvis funksjonell avhengighet, der vi

kan fjerne et attributt på venstre side. For eksempel vil $\{\text{PersonID}, \text{StudentID}\} \rightarrow$ Navn være en delvis FD, mens StudentID → Navn er en full FD. **Når alle kandidatnøkler har ett attributt vil relasjonen automatisk være i 2NF, siden det ikke kan være noen delvis FD**.



For eksempel kan vi se på utlån relasjonen (se figur). For å se om denne er i 2NF, starter vi med å bruke tillukning for å finne ut at kandidatnøkkelen er kombinasjonen $\{\text{bokID}, \text{dato}\}$. **Siden kandidatnøkkelen består av flere attributter, må vi sjekke at ingen ikke-nøkkelattributt er funksjonelt avhengig av deler av nøkkelen**:

- $\{\text{bokID}, \text{dato}\} \rightarrow$ lånerID – dette er en full FD
- $\{\text{bokID}, \text{dato}\} \rightarrow$ bokTittel – dette er en delvis FD, siden bokTittel kun er avhengig av bokID som følger av bokID → bokTittel (dvs. vi kan fjerne dato fra venstre side)

Denne relasjonen er derfor ikke på 2NF. Det er ikke ønsket at tabellen har ikke-nøkkelattributt som er delvis avhengig av nøkkelen, fordi det vil føre til redundans som følger av at data gjentas.

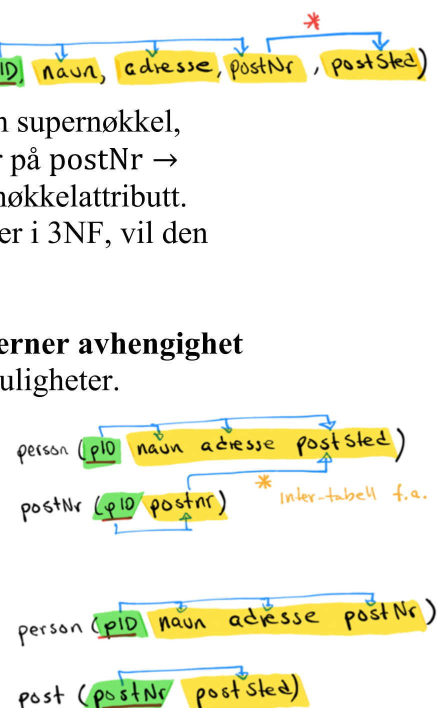
For å oppnå andre normalform må vi dekomponere tabellen, slik at vi fjerner delvis avhengighet av nøkkel som er en kilde til redundans. Figuren viser hvordan vi kan dele opp utlån ved å plassere bokTittel som forårsaker problemet i en egen relasjon med nøkkelattributtet bokID som det avhenger av. Resultatet har ingen redundans, men ulempen er at spørring kan kreve en JOIN.

Tredje normalform (3NF)

For tabeller i tredje normalform (3NF) vil alle funksjonelle avhengigheter på formen $X \rightarrow A$ gjelde hvis :

a) **X er en supernøkkel, eller**
b) **A er et nøkkelattributt**

Husk at $X \rightarrow BCD$ vil utlede $X \rightarrow B; X \rightarrow C; X \rightarrow D$, så hvis vi har FD med flere attributter på høyre side, kan vi dele den opp og deretter bruke definisjonen.



For eksempel kan vi se på Person relasjonen (se figur). For å se om denne er i 3NF, starter vi med å bruke tillukning for å finne ut at pID er en supernøkkel, dermed vil pID → $\{\text{Navn}, \text{adresse}, \text{postNr}\}$ oppfylle definisjonen. Hvis vi ser på postNr → postSted, kan vi se at postNr ikke er en supernøkkel og postSted er ikke et nøkkelattributt. Person-tabellen er derfor ikke på 3NF, men den er på 2NF. Når tabellen ikke er i 3NF, vil den være utsatt for redundans, siden to ikke-nøkkelattributt har et forhold.

For å oppnå tredje normalform må vi dekomponere tabellen, slik at vi fjerner avhengighet mellom ikke-nøkkelattributt som er en kilde til redundans. Vi ser på to muligheter.

- PostNr plasseres i ny relasjon med pID som nøkkel. Begge relasjonene er i 3NF og blir dermed kvitt redundans. Problemet er at vi lager en **inter-tabell FD, som ikke kan sjekkes uten å slå sammen de to tabellene**. Dette kan føre til inkonsistent data.
- PostSted og PostNr plasseres i ny relasjon, med PostNr som nøkkel. Begge relasjonene er i 3NF og blir dermed kvitt redundans. I tillegg unngår vi inter-tabell FD, så dette er et bedre design.

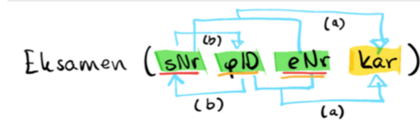
Merk: dekomponering vil ofte gå ut på å plassere attributtene i FD som bryter definisjonen av normalformen i en ny relasjon. Da vil venstre-side attributtet bli nøkkelen.

Boyce-Codd normalform (BCNF)

For tabeller i Boyce-Codd normalform (BCNF) vil alle funksjonelle avhengigheter på formen $X \to A$ gjelde hvis :

a) X er en supernøkkel.

BCNF krever at alle venstre sider i FDs er supernøkler, og den ble foreslått fordi tabeller i 3NF kan fortsatt ha redundansproblemer dersom kandidatnøklene overlapper.

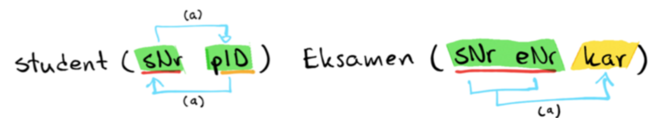


For eksempel kan vi se på Eksamen relasjonen (se figur). For å se om denne er i BCNF, starter vi med å bruke tillukning for å finne ut at kandidatnøklene er $\{\text{sNr}, \text{eNr}\}$ og $\{\text{pID}, \text{eNr}\}$. Relasjonen har følgende FDs:

1. $\text{sNr} \to \text{pID}$
2. $\text{pID} \to \text{sNr}$
3. $\{\text{sNr}, \text{eNr}\} \to \text{kar}$
4. $\{\text{pID}, \text{eNr}\} \to \text{kar}$

Relasjonen vil ikke være i BCNF, fordi FD 1 og 2 oppfyller ikke definisjonen. Relasjonen vil derimot være i 3NF, siden høyre side hos FD 1 og 2 er kandidatnøkler. FD 1 og 2 vil likevel være en kilde til redundans, siden det er en avhengighet blant kandidatnøklene.

For å oppnå Boyce-Codd normalform må vi dekomponere tabellen, slik at vi fjerner avhengighet mellom kandidatnøkler som er en kilde til redundans.



Figuren viser hvordan vi kan dele opp Eksamen ved å plassere kandidatnøklene som forårsaker problemet i en egen relasjon. Her har vi valgt å bruke SNr som primærnøkkel i Student, slik at den blir brukt som fremmednøkkel i Eksamen (kunne også valgt pID).

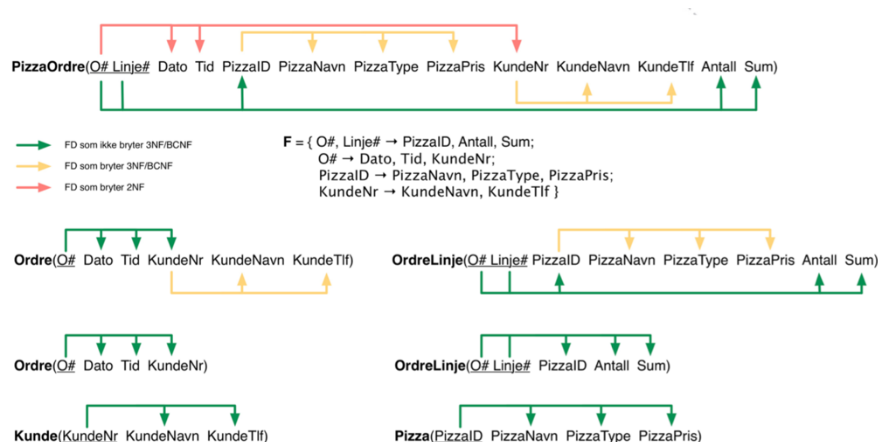
Kilder til redundans

Vi har sett at det er tre kilder til redundans:

1. **Delvis FD av nøkkel** – fjernes i 2NF normalisering
2. **FD mellom ikke-nøkkelattributter** – fjernes i 3NF normalisering
3. **FD mellom kandidatnøkler** – fjernes i BCNF normalisering

Når tabellen er i BCNF vil den ha et redundans-fri design mht. funksjonelle avhengigheter.

Figuren under viser et eksempel på hvilke typer funksjonelle avhengigheter som bryter de ulike normalformene og hvordan dette kan fjernes via dekomponering.



Oppgave

Gitt $R = \{A, B, C, D, E\}$ *og* $F = \{A \to B, B \to C, CD \to E\}$. *Gå ut fra at* R *oppfyller 1NF. Bestem den høyeste normalformen som oppfylles av* R

Vi bruker tillukning for å finne ut at kandidatnøkkelen er AD. Deretter sjekker vi normalformene:

- BCNF – $A \to B$ bryter BCNF, siden A ikke er en supernøkkel
- 3NF – $A \to B$ bryter 3NF, siden B ikke er et nøkkelattributt og A er ikke en supernøkkel
- 2NF – $A \to B$ bryter 2NF, siden B er et ikke-nøkkelattributt som er delvis avhengig av nøkkelen (AD)

Høyeste normalform som oppfylles av R er derfor første normalform.

Relasjonsdatabasedesign

Når vi skal designe en relasjonsdatabase har vi to alternativer:

a) **Analyse (top-down)** – vi lager en universell relasjon som består av alle attributtene fra mini-verden. Deretter bruker vi funksjonelle avhengigheter for å dekomponere relasjonen, helt til den får et design med ønsket normalform.
b) **Syntese (bottom-up)** – vi sender R og F inn i en algoritme som gir en dekomponering. Dette har aldri slått av, fordi man ønsker som regel å designe relasjonene selv.

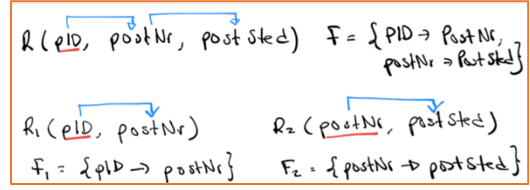
Kriterier for godt design VIKTIG

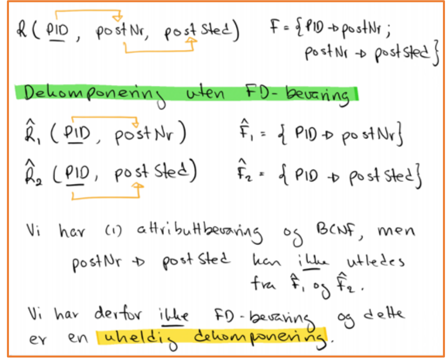
Normalformene alene er ikke tilstrekkelig for å garantere at relasjonsdatabasedesignet er bra. Vi ønsker å starte med R og F og deretter finne en dekomponering D som kan lagre det samme og har bedre egenskaper. Fire kriterier for et godt relasjonsdatabasedesign er:

$D = \{R_1, R_2, \ldots, R_m\}$

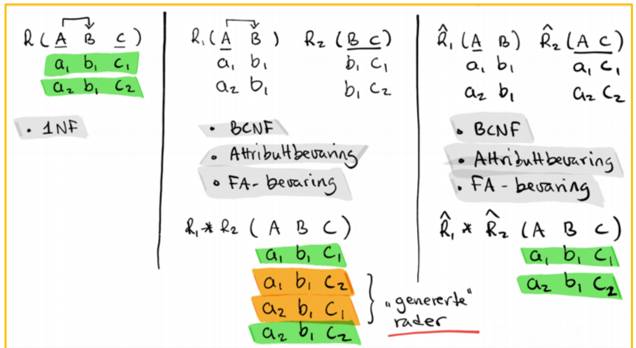
$\bigcup_{i=1}^{m} R_i = R_1 \cup R_2 \cup \ldots \cup R_m = R$

1. **Normalform** – hver enkel tabell separat må ha ønsket normalform
2. **Attributtbevaring** – alle attributter i R må finnes i minst en av tabellene etter dekomponeringen, slik at projeksjonene lagrer de samme dataene.
3. **Bevaring av funksjonelle avhengigheter** – F representerer en restriksjon på alle tabellforekomster av R og den må ivaretas i tabellene etter dekomponeringen. Målet er at avhengighetene skal ivaretas så enkelt som mulig, noe som vil være tilfellet dersom alle FDs finnes i en eller flere R_i eller kan utledes fra FDs i R_i. Hvis ikke vil vi få inter-tabell avhengigheter som krever av vi må slå sammen tabellene for å sjekke. Figuren under viser dekomponering uten bevaring.





4. **Tapsløs-join egenskapen (ikke-additiv)** – hvis vi deler opp en tabell til flere tabeller, må vi kunne komme tilbake til utgangspunktet vha. en naturlig join. Sammenslåingen skal ikke skape «falske data». Dette er svært viktig!. **For å sjekke om** $\boldsymbol{D = \{R_1, R_2\}}$ **har er ikke-additiv kan vi se om** $\boldsymbol{R_1 \cap R_2 \to R_1}$ **eller** $\boldsymbol{R_1 \cap R_2 \to R_2}$**.** Altså, felles attributter i R_1 og R_2 er supernøkkel for en eller begge komponenttabellene. I midterste kolonne er felles attributt B som ikke er supernøkkel for R_1 eller R_2, og denne dekomponeringen har ikke tapsløs join egenskapen. I høyre kolonne er felles attributt A, som er supernøkkel for R_1 og denne dekomponeringen har tapsløs join egenskapen.



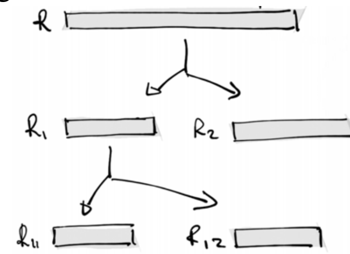
Merk: kravet er at de opprinnelige avhengighetene må være utrykt eller kunne *utledes* fra FDs i dekomponeringstabellene. Ofte vil en FD deles opp i to tabeller og transitivitet sikrer at hele avhengigheten er bevart. For eksempel hvis F hadde inkludert pId → postSted ville denne vært inkludert etter dekomponeringen siden R_1 har pID → postNr og R_2 har postNr → postSted

Sjekk av tapsløs-join egenskapen (F12)

Som vi så på forrige side vil en dekomponering $D = \{R_1, R_2\}$ ha tapsløs-join egenskapen hvis:

$$R_1 \cap R_2 \to R_1 \qquad \text{eller} \qquad R_1 \cap R_2 \to R_2$$

Dette kalles felles-attributt regelen og sier altså at dekomponeringen er ikke-additiv hvis felles attributt i R_1 og R_2 er en supernøkkel i en eller begge tabellene. Denne regelen vil ikke alltid gjelde, men det er sjeldent den gir feil svar. Når vi dekomponerer en relasjon bør vi passe på at denne regelen er oppfylt (dvs. opprette primærnøkkel-fremmednøkkel forhold).



Ved oppsplitting i flere trinn vil sluttresultatet ha T-J egenskapen hvis alle oppsplittingstrinnene har det. Vi kan bruke felles-attributt regelen for å sjekke hver dekomponering og dermed om sluttresultatet har T-J egenskapen.

En alternativ måte å gjøre dette på er vha tabellmetoden (algoritme, s. 105), som alltid vil gjelde. Vi ser på denne vha relasjonen $R = \{A, B, C\}$ og $F = \{A \to B\}$ som dekomponeres til $R_1 = \{A, B\}$ og $R_2 = \{B, C\}$:

1) Lag tabell og fyll med b_{ij}

	A	B	C
R_1	b_{11}	b_{12}	b_{13}
R_2	b_{21}	b_{22}	b_{23}

2) Sett inn a_j for attributtene

	A	B	C
$R_1(A,B)$	a_1	a_2	b_{13}
$R_2(B,C)$	b_{21}	a_2	a_3

3) Ser på $F=\{A \to B\}$. Rader med lik verdi for A settes lik i B

	A	B	C
R_1	a_1	a_2	b_{13}
R_2	b_{21}	a_2	a_3

↳ Ingen rader med bare a-er ⇒ ikke T-J egenskap

1) Lag tabell og fyll med b_{ij}

	A	B	C
R_1	b_{11}	b_{12}	b_{13}
R_2	b_{21}	b_{22}	b_{23}

2) Sett inn a_j for attributtene

	A	B	C
$R_1(A,B)$	a_1	a_2	b_{13}
$R_2(A,C)$	a_1	b_{22}	a_3

3) Ser på $F=\{A \to B\}$. Rader med lik verdi for A settes lik i B

	A	B	C
R_1	a_1	a_2	b_{13}
R_2	a_1	⇒ a_2	a_3

↳ Får rad med bare a-ere ⇒ T-J egenskapen

Oppgave

Gitt $R = \{A, B, C, D, E\}$ *og* $F = \{A \to B, B \to C, CD \to E\}$. *En mulig dekomponering av R er* $R_1(A, B)$, $R_2(B, C)$ *og* $R_3(C, D, E)$. *Er dette en god dekomponering?*
Vi kan bruke tabellmetoden for å finne ut om det er en god dekomponering:

1) Lag tabell og fyll med b_{ij}

	A	B	C	D	E
R_1	b_{11}	b_{12}	b_{13}	b_{14}	b_{15}
R_2	b_{21}	b_{22}	b_{23}	b_{24}	b_{25}
R_3	b_{31}	b_{32}	b_{33}	b_{34}	b_{35}

2) Sett inn a_j for attributtene

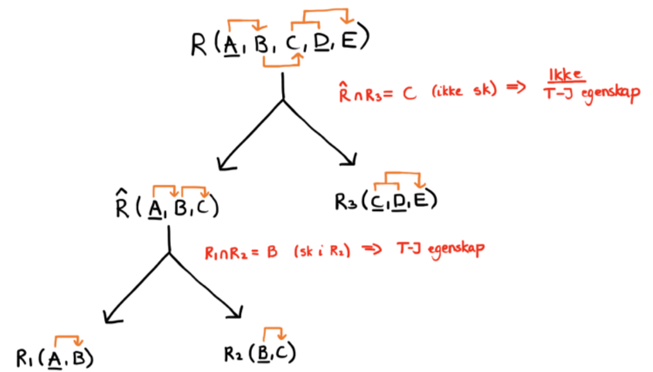
	A	B	C	D	E
$R_1(A,B)$	a_1	a_2	b_{13}	b_{14}	b_{15}
$R_2(B,C)$	b_{21}	a_2	a_3	b_{24}	b_{25}
$R_3(C,D,E)$	b_{31}	b_{32}	a_3	a_4	a_5

3) Ser på $F=\{A \to B, B \to C, CD \to E\}$. Rader som er lik på venstre side settes lik på høyre side

Gjentas helt til det skjer ingen endring

	A	B	C	D	E
$R_1(A,B)$	a_1	a_2	⇒ a_3	b_{14}	b_{15}
$R_2(B,C)$	b_{21}	a_2	a_3	b_{24}	b_{25}
$R_3(C,D,E)$	b_{31}	b_{32}	a_3	a_4	a_5

↳ Ingen rader med kun a-ere ⇒ Ikke T-J egenskap

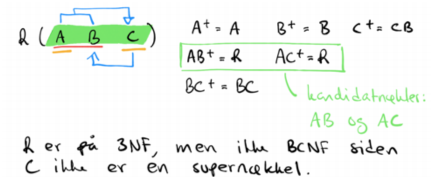


Vi kan også bruke felles-attributt metoden gjentatt for hvert oppsplittingstrinn. For at dekomponeringen skal ha tapsløs-join egenskapen må alle oppsplittingstrinnene ha det. Siden felles attributtet for trinn 1 ikke er en supernøkkel, vil ikke dette trinnet og dermed heller ikke dekomponeringen ha tapsløs-join egenskapen.

For at dekomponeringen skal ha tapsløs-join egenskapen må $\hat{R}(\underline{A}, B, C, \underline{D})$, slik at $\hat{R} \cup R_3 = CD$ som er en supernøkkel i R_3. Videre må vi dele opp $\hat{R}$ i $R_1(\underline{A}, B)$ og $R_2(\underline{A}, \underline{D})$, slik at $R_1 \cup R_2 = A$ som er en supernøkkel i begge dekomponeringstabellene.

BCNF og kriteriene

Vi kan alltid oppnå alle fire kriteriene hvis vi nøyer oss med tabeller på 3NF, mens dersom vi ønsker BCNF kan det hende vi må velge bort noe. Attributtbevaring og tapsløs-join egenskapen kan vi ikke velge bort, fordi da vil vi miste informasjon og få et design som genererer «søppel». Bevaring av funksjonelle avhengigheter kan velges bort, men det blir mer arbeid å ivareta restriksjonene, siden vi får inter-tabell avhengigheter. **Ofte må vi velge mellom relasjoner i 3NF med FD-bevaring eller relasjoner i BCNF uten FD-bevaring**. **BCNF minimerer redundans, men det blir mer arbeid å sjekke at restriksjoner oppfylles**. Hvis det er mye innsetting og oppdatering er det ønskelig å ha lite redundans og derfor vil vi ha relasjoner i BCNF. På figuren kan vi se at BCNF normaliseringen med tapsløs-join egenskapen vil føre til at vi mister avhengigheten $\{A, B\} \rightarrow C$. Denne restriksjonen må derfor sjekkes ved å slå sammen tabellene.



3NF/BCNF og redundansproblemer

Relasjoner som er i 3NF eller BCNF kan også ha store redundansproblemer. For eksempel kan vi se på Ansattinfo, der attributtene Kompetanse og Hobby er uavhengige av hverandre. **Denne relasjonen har ingen funksjonelle avhengigheter, så nøkkelen må være alle attributtene**. Som vi kan se på figuren kan en ansatt ha flere kompetanser og flere hobbyer, og siden disse er uavhengige av hverandre må alle kombinasjoner representeres (eks: DB-Fotball og DB-Foto). Innsetting av <1, Java, Tennis> vil derfor kreve innsetting av <1, Java, Tennis> , <1, Java, Fotball>, <1, Java, Foto> og <1, DB, Tennis>, <1, C++, Tennis>. **Dette er en ekstrem grad av redundans, selv om tabellen er i BCNF (ingen FDs).** Dette skyldes at relasjonen har flerverdi-avhengigheter.

Ansattinfo

AnsattNr	Kompetanse	Hobby
1	DB	Fotball
1	C++	Foto
2	DB	Ski
1	C++	Fotball
1	DB	Foto

Gitt X, Y, Z som er delmengder av R. **En *multi-value dependency* (MVD) $X \twoheadrightarrow Y$ betyr at Y-verdier som er assosiert med en X-verdi bestemmes av X og ingenting annet**. For eksempel vil AnsattNr ↠ Kompetanse, siden hvilke kompetanser en ansatt har kun bestemmes av ansattnummer (hvilken ansatt det er). Det andre eksempelet er AnsattNr ↠ Hobby, siden hvilke hobbyer en ansatt har kun bestemmes av ansattnummer (har ingenting med kompetanse å gjøre).

For en triviell MVD $X \twoheadrightarrow Y$ vil Y være en delmengde av X eller $X \cup Y = R$. Trivielle MVDs gir ingen restriksjon. **Alle FDs er MVDs, men FD er strengere restriksjoner siden det kun tillater én verdi på høyre side, mens MVD tillater flere** (merk: FD tillater flere attributter på

høyre side, men disse må ha én verdi). **MVDs kommer ofte i par, så hvis $X \twoheadrightarrow Y$ vil $X \twoheadrightarrow Z$ der Z er attributtene fra relasjonen som ikke er i X eller Y (dvs. $Z = R - (X \cup Y)$).** For eksempel vil AnsattNr ↠ Kompetanse medføre at AnsattNr ↠ Hobby, siden Hobby «blir igjen» når vi tar ut AnsattNr og Kompetanse.

Fjerde normalform (4NF)

En tabell er på 4NF hvis det for alle ikke-trivielle MVDs, $X \twoheadrightarrow Y$, er slik at X er en supernøkkel.

Fagins teorem gir at hvis $A, B \subseteq R$ og $C = R - (A \cup B)$, så vil projeksjonene $A \cup B$ og $A \cup C$ ha tapsløs-join egenskapen hvis $A \twoheadrightarrow B$ og $A \twoheadrightarrow C$. I eksempelet på forrige side vil vi dele Ansattinfo inn i AnsattKompetanse og AnsattHobbyer (se figur) med hhv. AnsattNr ↠ Kompetanse og AnsattNr ↠ Hobby. Disse tabellene vil være på 4NF, siden AnsattNr er primærnøkkel hos begge. Vi vil også ha tapsløs-join egenskapen mellom komponentene siden de oppfyller Fagins teorem. **Redundansen er fjernet**, siden innsetting av <1, Java, Tennis> kun krever innsetting av <1, Java> i AnsattKompetanse og <1, Tennis> i AnsattHobbyer.

AnsattKompetanse

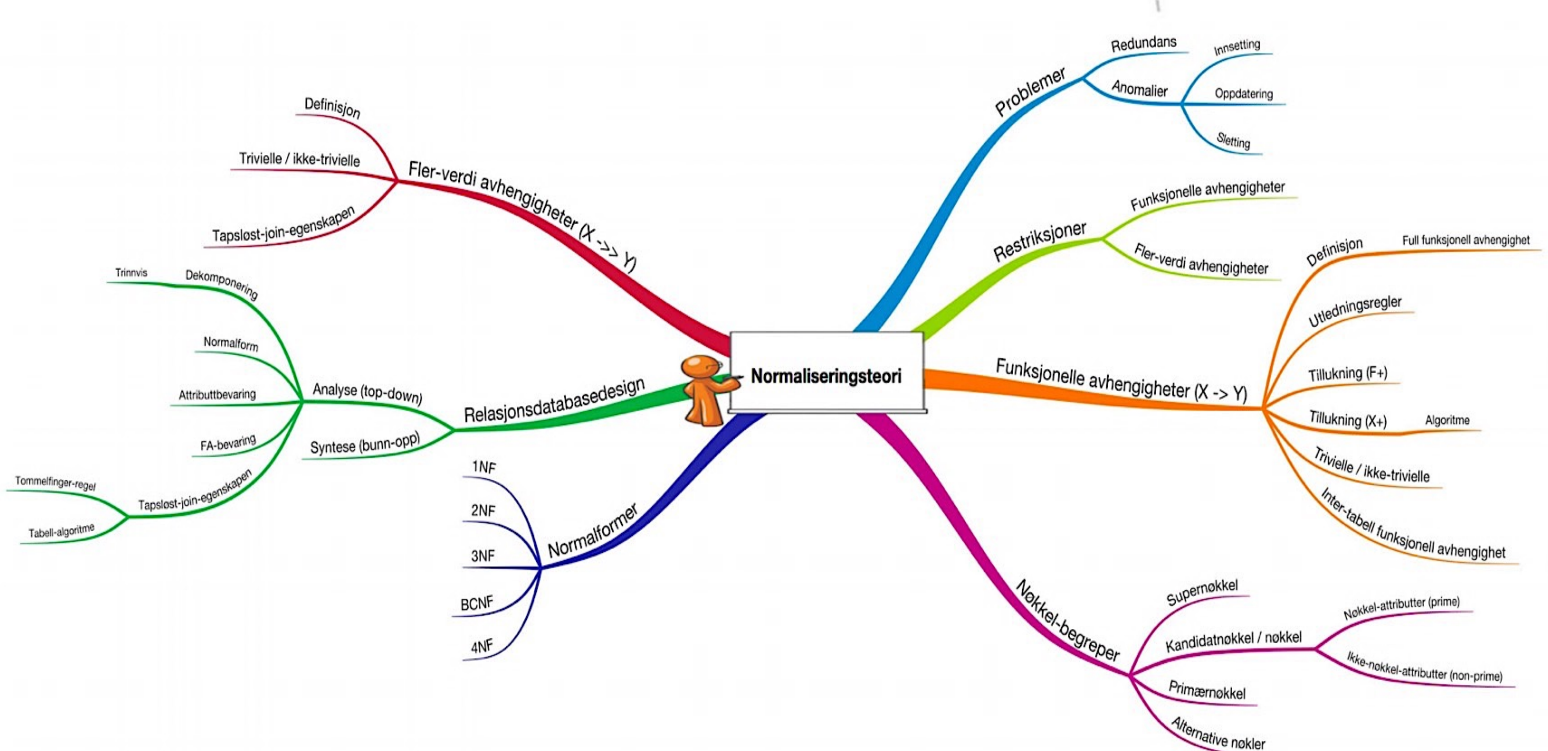
AnsattNr	Kompetanse
1	DB
1	C++
2	DB

AnsattHobbyer

AnsattNr	Hobby
1	Fotball
1	Foto
2	Ski

Oppsummering av normaliseringsteorien

Figuren under viser en oversikt over ulike deler av normaliseringsteorien.



Del 5 er basert på videoforelesningen der utgitte kompendier er brukt for å utfylle informasjonen.

Del 5 – Database internals

Frem til nå har vi sett på hvordan vi kan designe, bruke og programmere en database vha. blant annet SQL, relasjonsalgebra og normalisering. Vi skal nå se på det som er inni selve databasesystemet, slik som indekser, transaksjoner, osv. For å bli en god databasebruker, må man kunne forstå hvordan databaser virker innvendig. Denne delen består av to seksjoner:

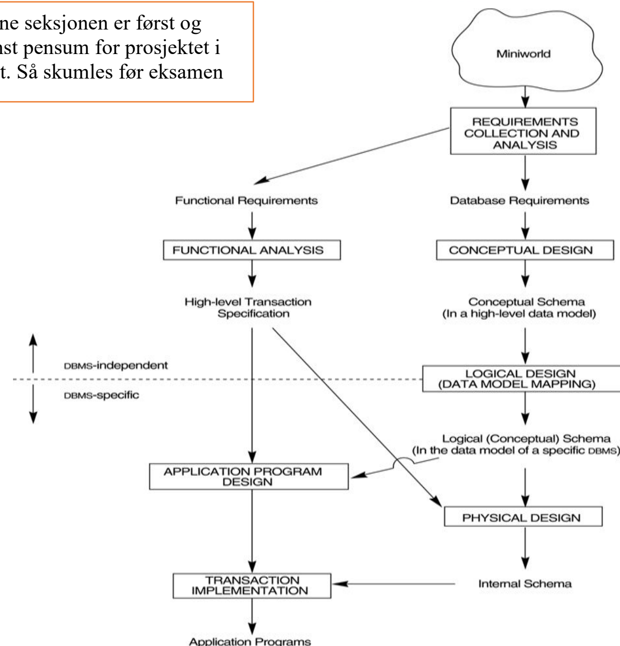
1. **Program- og databasedesign** – handler om enhetlig design der både programvare og data blir designet i samme prosess. Databasen blir designet som en del av prosjektet.
2. **Databaselagring** – handler om lagring, indeksering og behandling av spørringer

Disse er basert på kompendier som er markert som pensum og er skrevet av faglærer.

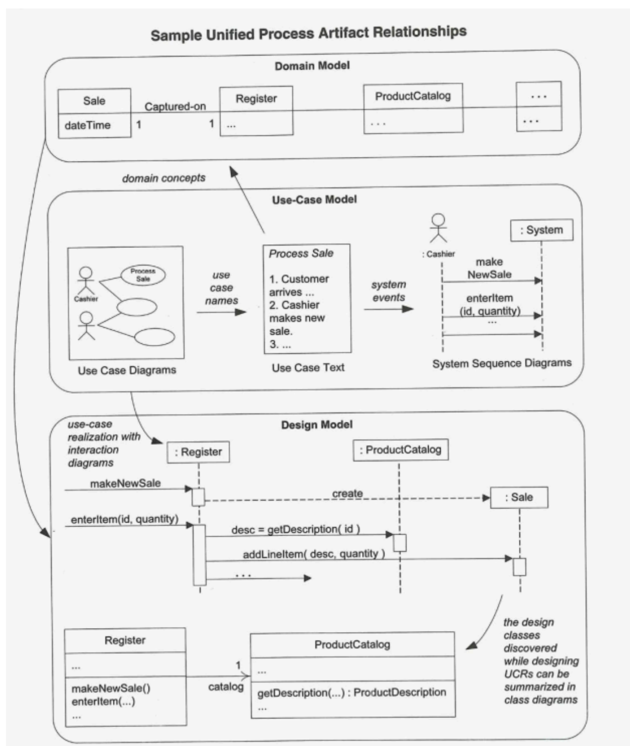
Programvare- og databasedesign

Denne seksjonen er først og fremst pensum for prosjektet i faget. Så skumles før eksamen

Denne seksjonen handler om hvordan databasen kan designes som en del av prosjektet, altså hvordan programvare og data blir designet samtidig.



Figuren viser prosessen fra miniverden til applikasjons-program. Vi begynner med en miniverden som sendes inn i en prosess der krav blir samlet inn og analysert. Resultatet er funksjonelle og database krav som så sendes inn i separate prosesser. Den ene vil lage funksjonene vi trenger og den andre vil lage ER-modellen. Det er altså et skille mellom programmer og data. En kartlegging av konseptuell modell vil lage et logisk skjema som vil definere det fysiske designet av databasen. Det logiske skjemaet vil også brukes sammen med spesifisering av funksjonene for å designe applikasjonsprogrammet.



Figuren viser et eksempel der vi skal designe en elektronisk kasse i en butikk.

Denne seksjonen tar for seg en litt annen objektorientert designmetode, der Unified Process, UML og Patterns er i fokus. Unified Process er et iterativt og inkrementelt rammeverk for programvare-utvikling, og det var forløperen for smidig utvikling. Den iterative designprosessen er basert på Unified Process, og i løpet av denne prosessen vil det lages såkalte *artifacts*. På figuren kan vi se tre av disse som vi skal fokusere på:

- **Domenemodellen** (objekter)
- **Use-Case modellen** (funksjoner)
- **Designmodellen** (basert på de over)

Use-case og domain modellen kan lages uavhengig av hverandre eller samtidig, mens designmodellen lages basert på de to andre. Designmodellen kan deles i flere deler som kan være både statiske og dynamiske, og dette

beskrives vha. ulike UML-diagrammer, slik som klasse-, sekvens- og kommunikasjonsdiagrammer. **Til sammen vil disse gi en forståelse av systemet som skal realiseres**.

Viktige objekter identifiseres i domenemodellen og blir laget i designet for å realisere funksjonene man har funnet i use-case modellen. Dette kalles **ansvarsdrevet design**, siden vi finner hvilke objekter som har ansvar for funksjonene vi trenger. Dette krever tradisjonelle designidealer som liten kobling og høy kohesjon (dvs. liten grad av avhengigheter mellom programvaremoduler = godt design, høy lesbarhet og lett vedlikehold), men vi trenger også en verktøykasse fylt med ferdig design, kalt **design patterns**.

Artifacts

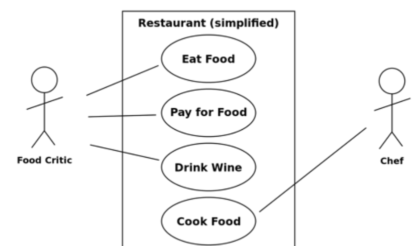
I løpet av designprosessen vil det lages såkalte *artifacts*, som er ulike dokumenter, programvarer eller ting vi lager som en del av prosessen (eks: datamodell som er implementasjonen av database). Disse lages i ulike iterasjoner av prosjektet: Inception, Elaboration, Construction og Transaction. For eksempel vil Use-case modellen starte i Inception og kan revideres/rettes i Elaboration.

Sample Unified Process Artifacts and Timing (s-start; r-refine)

Discipline	Artifact Iteration→	Incep. I1	Elab. E1..En	Const. C1..Cn	Trans. T1..T2
Business Modeling	Domain Model		s		
Requirements	Use-Case Model	s	r		
	Vision	s	r		
	Supplementary Specification	s	r		
	Glossary	s	r		
Design	Design Model		s	r	
	SW Architecture Document		s		
	Data Model		s	r	
Implementation	Implementation Model (code, html, ...)		s	r	r

Use-case modellen

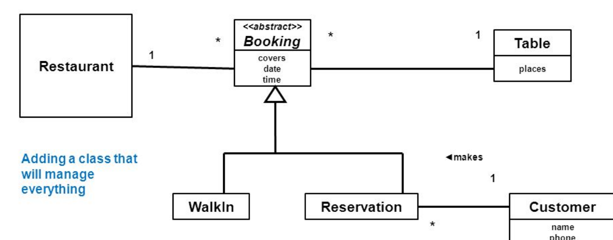
Use-case modellen brukes for å finne funksjonene systemet skal ha, og tanken er å lage funksjonelle krav i en lettlest form (= usecase). Dette bør gjøres ved å lage en oversikt over hvilke usecases systemet har, gjerne i et UML-usecasediagram (se figur). Hver usecase bør deretter utdypes ved å vise hvilke meldinger som går mellom bruker (aktør) og systemet. **Usecasemodellen skal ikke ha med detaljer om brukergrensesnittet kun logisk informasjon som utveksles**.



Et usecase skal oppnå noe for brukeren, og gode mål for systemet vil gi gode krav til funksjonene som systemet har. For eksempel kan målene skrives på formen verbObjekt (eks: Spise mat, Lage mat, osv.). Et usecase vil ofte ha et aktivt verb først i navnet. Vi bør tenke på hva brukeren ønsker å oppnå i dialog med systemet, og vi må identifisere aktørene og finne deres mål med systemet. Når vi lager en dialog er der viktig å ta med hva slags informasjon som går fram og tilbake mellom aktør og systemet. Det er ikke nødvendig å beskrive hvordan systemet internt håndterer meldinger som kommer fra aktøren. Hver melding som går mellom aktøren og systemet kalles en systemmelding, og når vi designer systemet skal vi realisere hver systemmelding.

Domenemodellen (områdemodellen)

Domenemodellen, også kalt områdemodellen, brukes ofte som et diagram som modellerer hele designområdet til problemet, og det kan gjenbrukes ved nye systemer innenfor samme problemområdet. Vi bruker et klassediagram som modellerer meningsfylte konsepter og data som finnes i problemområdet til systemet vi ønsker å utvikle, men vi unngår å modellere typiske programvareobjekter. Dette ligner på et ER-diagram i databasedesign, men det kan ha flere objekter som ikke nødvendigvis lagres i en database. En viktig ide med domenemodellen er å lage objekter som senere kan få ansvar i form av programvare eller metoder.



En domenemodell kan lages på flere måter, og vi skal se på en som kalles **substantivmetoden**, som tar utgangspunkt i en kravspesifikasjon som beskriver hva systemet skal gjøre. Den kan også bruke Use-case modellen hvis den er definert. **Substantivmetoden går ut på å ta alle substantiver fra disse to dokumentene og sette de i entall og ubestemt form. Deretter blir upassende kandidater droppet,** for eksempel systemutviklingsbegreper, operasjoner og hendelser, de som er utenfor systemets rammer og vage/overflødige begreper. Åpenbare attributter blir droppet eller satt som attributter til klassene vi har funnet i substantivanalysen. Resultatet av substantivmetoden er et diagram med en rekke mulige klasser. **Neste steg er å se etter lenker (koblinger) mellom klassene, som ofte er verbsetninger i kravspesifikasjonen**. Dette er meningsfylte sammenhenger mellom objekter som vi må huske. En sammenheng som skjer akkurat nå og sendere blir glemt, trenger vi ikke å modellere som ei lenke.

Det er ikke så farlig om vi glemmer noen klasser i områdemodellen, fordi vi kan legge til flere klasser når vi begynner å designe systemet.

Designmodellen – usecase realisering

Designmodellen lages basert på usecase- og domenemodellen og består av diverse diagrammer som lages parallelt. Kravene fra miniverden blir representert i Domain og Use-case modellene som sammen brukes i **usecase realisering**, der hver use-case blir realisert i design modellen vha samarbeidende objekter. For hver systemmelding i et usecase ser vi på hvilke objekter som tar imot meldingen og hva som skjer med den når den går inn i systemet. Det er viktig å huske at objekter som sender meldinger til hverandre må kjenne hverandre, altså må det være en lenke mellom objektene (gis i domenemodellen). Alternativt kan objektene kjenne hverandre via en parameter i metoden eller objektet kan sende meldingen globalt.

Resultatet av denne prosessen vil være *software artifacts* som inkluderer:

- **Designklassediagram** (konkrete klasser i programmet og koblingen mellom disse)
- **Databaseskjema** (eks: SQL)
- **Sekvensdiagram** (objekter og hvilken rekkefølge de kommuniserer i)
- **Kommunikasjonsdiagram** (objekter og hvordan de snakker sammen)

I dette kompendiet bruker vi sekvensdiagram og designklassediagram. Sekvensdiagram har fokus på sekvens, men kommunikasjonsdiagram har fokus på hvilke objekter som samarbeider (greit for å modellere kjøretidsoppførsel til objektene).

Patterns

Patterns (mønster) er en navngitt beskrivelse av et problem og en løsning som kan brukes i mange sammenhenger. Det er altså en programvare-konstruksjon som løser et kjent problem og har et navn. Dette er nyttig både for den som designer systemet, og for den som senere skal lese eller videreutvikle systemet. Patterns er en god måte å kommunisere løsningen på, siden man skjønner hva løsningen innebærer med en gang man ser navnet til løsningen. Et pattern har som regel en standard beskrivelsesmåte, for eksempel kan det inneholde

General Responsibility Assignment Software Patterns or Principles (GRASP)

IT 2103

Pattern/ Principle	Description
Information Expert	A general principle of object design and responsibility assignment? Assign a responsibility to the information expert—the class that has the information necessary to fulfill the responsibility.
Creator	Who creates? (Note that Factory is a common alternate solution.) Assign class B the responsibility to create an instance of class A if one of these is true: 1. B contains A 2. B aggregates A 3. B has the initializing data for A 4. B records A 5. B closely uses A
Controller	What first object beyond the UI layer receives and coordinates ("controls") a system operation? Assign the responsibility to an object representing one of these choices: 1. Represents the overall "system," a "root object," a device that the software is running within, or a major subsystem (these are all variations of a *facade controller*). 2. Represents a use case scenario within which the system operation occurs (a use-case or *session controller*)

beskrivelser av løsning, problem, eksempel, diskusjon, motindikasjoner, fordeler og relaterte patterns. Noen viktige patterns:

- **Controller:**
  - Løsning – gi ansvaret for å motta eller håndtere systemhendelser til en klasse som representerer hele systemet (FacadeController, én Controller) eller ett usecase (UsecaseController, en Controller per usecase).
  - Problem – hvem skal håndtere systemmeldinger?
  - Diskusjon – input til en Controller kommer ofte fra GUIet, tastatur eller nettet. En Controller er vanligvis tynn, altså vil den delegere ansvaret.
- **Data accessor:**
  - Løsning – gjemmer fysisk databaseaksess i et objekt og tilbyr kun logiske operasjoner. Applikasjonskoden vedlikeholder den underliggende datamodellen, men er frakoblet ansvaret for dataaksess
  - Problem – hvordan frakoble fysisk dataaksess fra applikasjonslogikk? Du ønsker å tilby flere aksessmetoder og vil velge mellom de basert på kjøretid
- **Active domain object:**
  - Løsning – datamodellen og detaljer om dataaksess blir innkapslet i relevante domeneobjekter. Applikasjonskoden slipper å direkte interagere med databasen
  - Problem – hvordan forhindre at applikasjonskoden kjenner til detaljer ved databaseaksessen? Hvordan hindre at endringer i datamodellen forplanter seg hardt til applikasjonskoden?
- **Object/relational map:**
  - Løsning – innkapsler kartleggingen mellom domeneobjekter og relasjonsdata i en enkelt komponent. Applikasjonskoden og domeneobjektene blir frakoblet den underliggende datamodellen og dataaksessdetaljene
  - Problem – du ønsker å gjemme den fysiske datamodellen og kompleksiteten i dataaksessen fra applikasjonene og domeneobjektene.

EKT-eksempel: Design som bruker Controller

Vi ønsker å delvis designe og implementere et elektronisk kvitterings- og tidtakersystem for orienteringsløp. Følgende er en beskrivelse av miniverden:

EKT-system
Et orienteringsløp består av mange klasser, løyper og poster. En løper er påmeldt i en klasse og deltar for en klubb[a]. Løpere påmeldt samme klasse skal løpe samme løype. Flere klasser kan ha samme løype. En løype består av start, mål og en sekvens av poster mellom disse. Flere løyper kan ha noen felles poster. Hver løper har en (EKT-)løperbrikke[a] som har en unik brikkekode[a], dvs. et stort heltall. Hver post har en unik postkode[a], som er et heltall. Start og mål er også poster. Når løperen har kvittert på en post, vil tiden[a] og postkoden lagres på løperbrikken. Løperbrikken vil inneholde en sekvens av (tid, postkode)-data. Når løperen kommer i mål, blir brikkekoden og sekvensen av (tid, postkode)-data registrert i EKT-systemet. EKT-systemet skal også ha startlister, dvs. for hver klasse skal vi vite når de påmeldte løperne starter. Systemet må også vite hvilke løpere som ikke startet når løpet er slutt. De viktigste funksjonene til EKT-systemet er:

- *Kontrollere at løperen har vært på riktige poster i riktig rekkefølge. Løpere som ikke har det blir diskvalifisert*
- *Regne ut løpstiden[a] for løperen, dvs. tiden løperen har brukt mellom start og mål*
- *Lage resultatlister sortert på løpstid. En løper skrives ut med plassering, navn, klubb og løpstid. Diskvalifiserte løpere plasseres til slutt i resultatlisten*
- *Lage strekktidslister, dvs. lister som viser hvor lang tid en løper har brukt mellom hver post*
- *Vise hvorfor en løper er diskvalifisert*

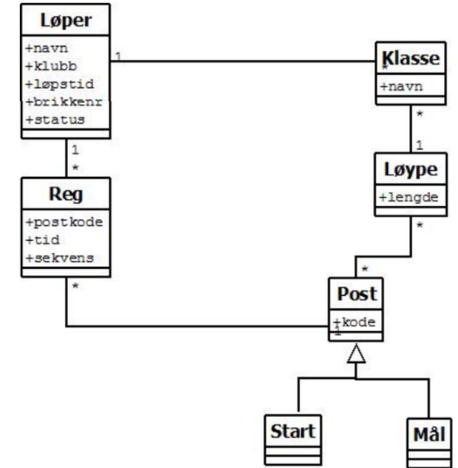
Vi bruker substantivmetoden for å lage domenemodellen, og understreker derfor de relevante substantivene. Åpenbare attributter merkes med ...[a]

Løsning

Domenemodell (områdemodell)

Vi bruker substantivmetoden for å lage domenemodellen. På forrige side har vi markert alle substantivene. Neste steg er å sette disse i entall og ubestemt form, legge til attributter til objektet og deretter fjerne upassende substantiv (blir strøket fra listen):

- ~~EKT-system~~ (metaobjekt)
- ~~Orienteringsløp~~ (databasen er ett orienteringsløp)
- Klasse: navn
- Løype: lengde
- Post: postkode
- Løper: navn, klubb, løpstid, løpebrikke, status (diskvalifisert eller ikke)
- Start
- Mål
- ~~Sekvens av poster~~ (relasjon mellom løype og poster)
- ~~Startliste~~ (lages ved behov)
- ~~Resultatliste~~ (lages ved behov)
- ~~Strekktidsliste~~ (lages ved behov)

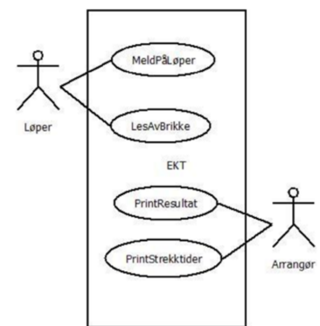


Siden vi ønsker å vite når løperen var på ulike poster, og hvilken rekkefølge løperen var ved ulike poster legger vi til et objekt Reg mellom Løper og Post. Denne vil ha attributtene postkode, tid og sekvens. **Figuren viser hvordan dette kan tegnes som et diagram**. Alle klassene er databaseklasser, som vil si at det er data vi må huske etter programmet stopper.

Videre kartlegger vi domenemodellen til en relasjonsdatabase og forenkler til fire tabeller:

```
Løper(brikkenr,navn,klasse,klubb,starttid,løpstid,status)
Reg(sekvnr,brikkenr,postnr,tid)
Løype(lnr,sekvnr,postnr)
Klasse(klassenavn,lnr)
```

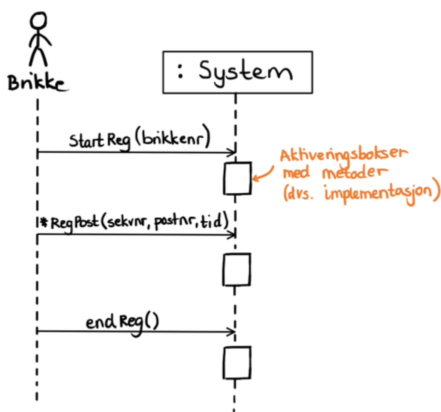
Use-case modell



Vi lager en Use-case modell i form av et UML diagram (figur til høyre). **Her har vi tatt med fire basisfunksjoner og to ulike aktører** (løper og arrangør). Vi kunne også lagt til flere funksjoner, for eksempel «Registrere mål», «Lage resultatliste», osv. Diagrammet viser hvilke funksjoner systemet har og hver usecase er en slags interaksjon med systemet.

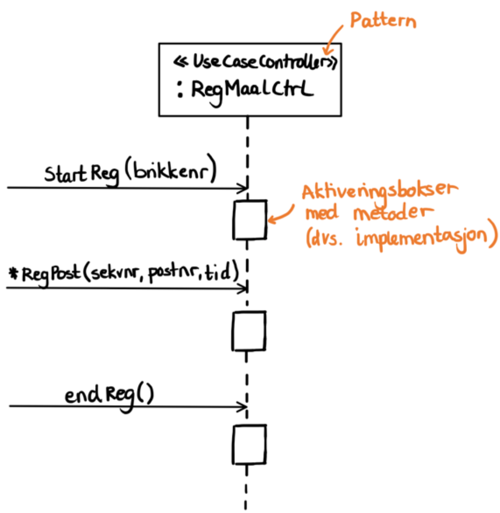
Designmodell – usecase realisering

Videre fokuserer vi på usecase «Løper skal kunne registrere seg etter å ha kommet til mål». **For å realisere dette usecaset, starter vi med å lage et systemsekvensdiagram som viser systemmeldingene som sendes i dette usecaset:**



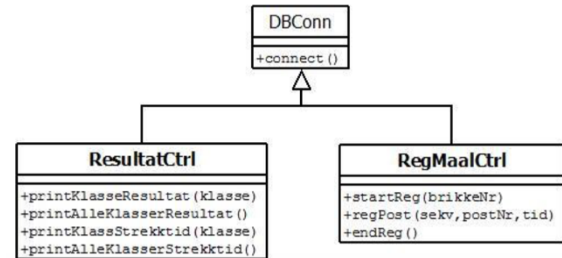
Her vil :System være en instans av systemet. Brikken og systeminstansen har hver sine livslinjer og kommuniserer med hverandre vha systemmeldinger. Kommunikasjonen begynne med startReg meldingen som tar inn brikkenr. Senere kan flere RegPost meldinger sendes med sekvensnummer, postnummer og tid (* markerer at flere meldinger kan sendes). Til slutt vil endReg melding sendes for å avslutte kommunikasjonen. Dette er et **systemusecasediagram.**

Aktiveringsboksene representerer hva systemet gjør når det mottar meldingene. Det er altså implementasjonen av funksjonene.



Vi lager også et usecase sekvensdiagram for Controlleren som tar imot systemmeldinger. Dette er et konkret objekt istedenfor et overordnet system som vi så på i forrige diagram. Dette objektet vil lages hver gang systemet skal snakke dette usecaset, altså hver gang det kommer inn en ny løper og brikkenummer blir registrert. Dette er et **implementasjonsdiagram.** Vi har her valgt å la databasen gjøre jobben, slik at controlleren blir tynn. Dette er naturlig å gjøre hvis objektene som databasen tar hånd om ikke har noe særlig med oppførsel eller logikk innebygd.

Disse usecasediagrammene kan brukes for å lage et designklassediagram som består av klassene vi skal implementere. På figuren kan vi se RegMaalCtrl med funksjonene vi definerte i diagrammene. Vi kan også se at systemet har enda en Controller kalt ResultatCtrl som vil lage et resultat. **For å koble oss til databasen trenger vi objektet DBConn som har metoden connect().** DBConn er en abstrakt klasse som controllerne arver fra.



Merk: klassediagrammet består av Controllere og DBConn

I dette tilfellet har vi valgt å fokusere på to usecase, nemlig å registrere mål og lage resultatet. Det finnes flere usecase, for eksempel påmelding. Neste steg vil være å implementere dette i SQL og Java. I SQL må vi lage tabellene for Løper, Reg, Løype og Klasse. I Java må vi lage klassene DBConn, RegMaalCtrl og ResultatCtrl.

Avtale-eksempelet: Design bruker Active Domain Object

Vi ønsker å designe og delvis implementere ei elektronisk avtalebok for grupper av personer. Følgende er en beskrivelse av miniverden:

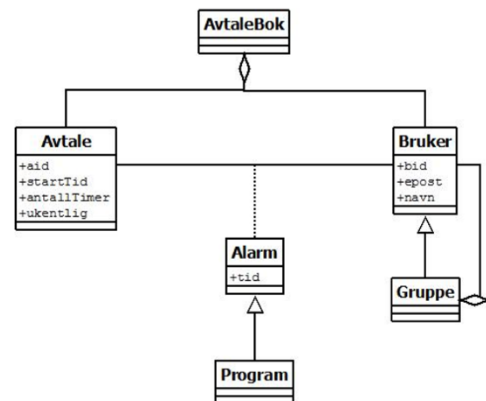
Systemet skal kunne varsle om og avtale møter for brukere av avtaleboka. For at systemet skal virke, må brukerne legge inn avtaler når de ikke er tilgjengelig, som f.eks. på ferie. En avtale kan være mellom personer eller for grupper av personer. Når man skal lage en avtale, oppgir man hvem den gjelder og ønsket tidsrom[a] avtalen skal finne sted i. Systemet vil så presentere mulige avtaletidspunkt[a], som brukeren kan velge blant. Det skal også være mulig å dobbeltbooke møter, men da må tiden settes manuelt. Som en forenkling kan vi anta at avtaler starter hver hele time[a] og varer et antall timer. En avtale vil gi en alarm til brukeren på forhånd. En alarm er vanligvis et lydsignal og en beskjed, men kan også være et brukerdefinert program som startes. Hvilken type alarm som skal brukes, bestemmes av brukeren som lager avtalen. For andre brukere kan man kun lage lydsignal med beskjed. Avtaler kan være faste, ukentlige avtaler[a] eller enkeltavtaler. Avtaler kan kanselleres. Brukere kan legges inn og fjernes fra systemet. Man kan definere, fjerne eller redigere grupper av brukere. En bruker skal kunne se sine avtaler i et gitt tidsrom sortert etter tid.

Løsning

Domenemodell (områdemodell)

Vi bruker substantivmetoden for å lage domenemodellen (områdemodell). Over har vi markert alle substantivene. Neste steg er å sette disse i entall og ubestemt form, legge til attributter til objektet og deretter fjerne upassende substantiv:

- Avtalebok (databasen er en avtalebok)
- Gruppe
- ~~Person~~ (bruker heller Bruker)
- Bruker: bid, epost, navn
- Avtale (møte): aid, startTid, antallTimer, ukentlig
- ~~Ferie~~ (regnes som spesiell avtale)

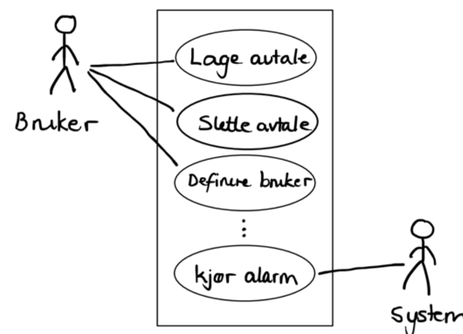


- ~~Tidsrom~~ (attributt)
- ~~Avtaletidspunkt~~ (attributt, start og antall timer)
- Alarm: tid, tekst, lyd
- Program
- ~~Lydsignal~~ (attributt)
- ~~Beskjed~~ (attributt)
- Program
- ~~FastAvtale~~ (attributt)
- ~~EnkeltAvtale~~ (attributt)

Figuren viser hvordan dette kan tegnes som et diagram. Dette er et UML diagram og bruker derfor litt annen notasjon. For eksempel vil notasjonen mellom Gruppe og Bruker bety at en gruppe er en spesiell form for bruker og kan inneholde mange brukere. Den stiplede linjen fra Alarm, betyr at den er en lenkeklasse (*association class*) som kobles til en assosiasjon mellom Avtale og Bruker. Selve alarmen vil være knyttet til brukeren, men slettes hvis avtalen slettes.

Videre kartlegger vi domenemodellen til en relasjonsdatabase og forenkler til fem tabeller:

```
Bruker(bid,epost,navn,bruker_type)
MedlemAv(gruppeId,bid)
Avtale(aid,starttid,timer,avtale_type)
Alarm(alarmId,tid,alarm_type,program,bid,aid)
HarAvtale(bid,aid)
```



Use-Case modell

Vi lager en Use-case modell i form av et UML diagram (figur til høyre). **Her ser vi noen av basisfunksjonene og to ulike aktører** (bruker og system).

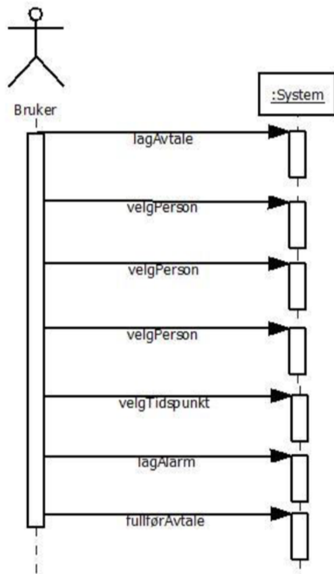
Designmodell – usecase realisering

Videre fokuserer vi på usecase «Registrering av enkelt-avtale for en gruppe som inneholder tre personer, men en alarm som kjører fem minutter før avtalen». Vi begynner med å lage en **usecase tekst** (figur til høyre), som er en formell måte å spesifisere hva usecaset må innebære. Dette skal helst ikke si noe om designet av brukergrensesnittet, slik at det kan brukes for ulike implementasjoner og for å diskutere designet med brukeren.

UseCase : LagAvtale

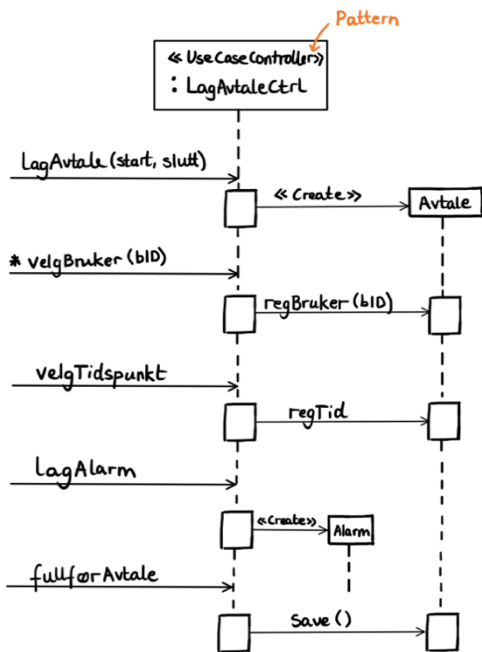
Primær aktør : Bruker av avtaleboka
1. Systemet viser tilgjengelige brukere
2. Brukeren velger brukere/grupper
3. Systemet viser tilgjengelig tidsrom
4. Bruker velger ledig tid og legger inn info

Alternative flows
3a) Det er ingen tilgjengelige tidspunkt scroll dager



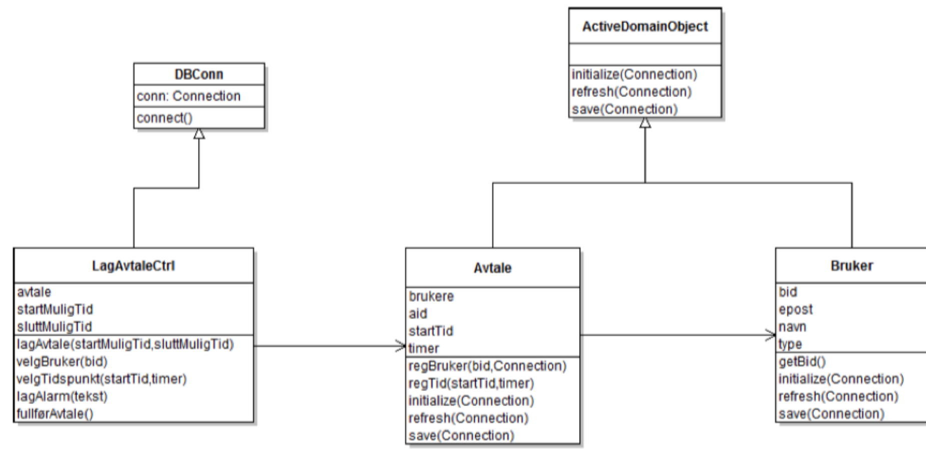
For å realisere dette usecaset, må vi lage et systemsekvensdiagram som viser systemmeldingene som sendes i dette usecaset.

Dette usecaset har syv systemmeldinger. Først vil lagAvtale sendes for å lage en avtale. Deretter vil tre personer velges fra en eller annen presentasjon i GUIet. Hver av deltagerne i avtalen blir registrert vha en egen systemmelding, kalt velgPerson. Når vi har alle deltagerne, kan vi vise alle tilgjengelige tidspunkt, og brukeren kan velge en av disse. Systemmeldingen lagAlarm sørger for at en alarm blir laget og til slutt blir fullførAvtale sendt for å bekrefte at avtalen skal lages. Dette er et **systemusecasediagram.**



Vi lager også et usecase sekvensdiagram for Controlleren som tar imot systemmeldinger. Dette Controller objektet vil lages hver gang systemet skal snakke dette usecaset, altså hver gang en bruker skal lage en avtale med en gruppe. Dette er et **implementasjonsdiagram**.

Merk: ≪ Create ≫ er en standard kommando for å lage et nytt objekt. Siden Avtale er et ActiveDomainObject bruker vi save() for å lagre den nye dataen i databasen. Bruk av ActiveDomainObject-mønsteret (pattern) gjør at de konseptuelle objektene i designet (eks: avtale og bruker) vet hvordan de kan lagres i og leses fra databasen. Dvs. usecaset har lite separat SQL kode, fordi dette blir gjort i Java.



Disse usecasediagrammene kan brukes for å lage et designklassediagram som består av klassene vi skal implementere. På figuren kan vi se LagAvtaleCtrl med funksjonene vi definerte i diagrammene. **Figuren viser også ActiveDomainObject-mønsteret som lar databaseobjektene være i minnet, slik at de kan leses inn og skrives til databasen vha egne metoder: initialize, refresh og save**. **Det er disse tre metodene som inneholder JDBC-kallene og SQL-koden.** Dette gjøres ved at Avtale og Bruker arver fra ActiveDomainObject klassen. Vi kan også se at diagrammet inkluderer DBConn som trengs for å etablere en kobling til databasen. Fordelen med bruk av ActiveDomainObject-mønsteret er at man kan bruke domeneobjektene i programmet uten tanke på hvordan de er realisert i databasen.

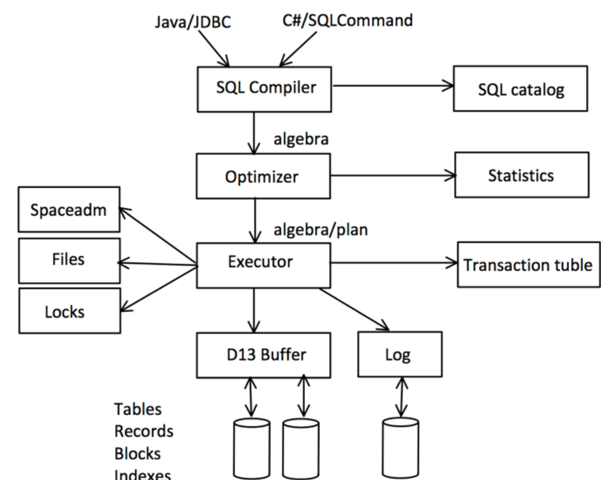
Lagring, indekser og spørringsbehandling

Denne seksjonen handler om lagring og indeksering i databaser og hvordan spørringer blir behandlet. Det er basert på kompendiet som erstatter kapittel 16 og 17 i boka. Det er viktig å lære om innmaten i databaser fordi det lar oss bruke databaser bedre vha indekser, spørringer og transaksjoner. Det er også viktig for å få en komplett forståelse av databaser.

Arkitekturen til en DBMS

Figuren viser **en vanlig programvarearkitektur for en SQL DBMS**. På toppen har vi et grensesnitt mot databasen som ofte går via et progammeringsAPI (eks: Java/JDBC). De ulike komponentene er:

- **SQL Compiler** – tar SQL som input og returnerer tupler/poster til klienten. Vha **SQL kataloger** vil den oversette SQL spørringen til et algebratre.
- **Optimizer** vil omforme algebratreet til et mer optimalisert algebratre som inkluderer en konkret



Merk: i dette kapittelet blir spørring og *query* brukt om hverandre

aksessplan til dataen, laget vha **statistikken** om dataen. Slike *query* planer gir hvordan query skal utføres, altså hvilken rekkefølge de ulike algebraoperasjonene skal gjennomføres, osv. Den lager flere planer og velger den som det er billigst å gjennomføre.
- **Executor** tolker algebratreet og planen og bruker mange moduler i systemet for å utføre spørringen og aksessere data, logge transaksjoner, låse transaksjoner (hindre samtidig utførelse), osv. For å utføre spørringer bruker den flere datastrukturer, for eksempel en transaksjonstabell som holder oversikt over transaksjonene som finnes i det øyeblikket.
- **Databasebufferen** (D13 Buffer) holder kopi av data i minnet og vil periodisk lese og skrive data fra/til disken. Dataen blir lest og skrevet som blokker som inneholder dataen i form av poster som er rader i bestemte tabeller. Dataen kan også indekseres.
- **Loggen** er ansvarlig for å lagre loggposter som brukes i transaksjonsprosessering for at transaksjoner skal være atomiske (alt i transaksjonen blir utført) og holdbare.

Databaselagring

En database må lagres permanent, altså ligge på en disk som enten er HDD (roterende disk) eller SSD (solid-state disk). **Den kan lagres som en fil i filsystemet til operativsystemet eller oppholde seg som *raw devices* som er en partisjon av disken.** Fordelen med *raw devices* er at databasen unngår operativsystemets buffer, og dermed får DBMS full kontroll over oppsettet av dataen og når dataen blir skrevet til eller lest fra disken. **DBMS får full kontroll over hvilken data som ligger i bufferen (hovedminnet), noe som er en fordel siden DBMS har mer kunnskap om hvilken data som bør ligge i bufferen, sammenlignet med operativsystemet**.

Det er vanlig å la filsystemet få ansvar for filhåndtering og bruke operativsystemets støtte for direkte I/O og låsing av data i hovedminnet. Det er flere oppgaver knyttet til filhåndtering:
1. Vi må vite hvilke data som er relatert (hører sammen) ved å ha filer. Data trenger derfor å lenkes sammen
2. Ledig plass på disken må håndteres. DBMS må tildele og allokere data ettersom databasen «lever».

Videre vil vi ikke gjøre noen antagelser om hvilken løsning som er valgt, så vi sier at databasene er lagret i en enhet med identifikasjon enhetID.

Lagringsstrukturer

For selve lagringen av tabeller i databasen, brukes det ulike strukturer:
- **Heapfil** – enkel og rett-frem struktur, der poster blir satt inn som rader etter hverandre. Det er ingen spesiell organisering og hver post har en adresse.
- **B+-trær** – mest brukt struktur som kan brukes for både lagring og indeksering. De er nyttige for områdesøk (mindre, større eller mellom en verdi)
- **Hashfil** – nyttig for søk basert på enkelverdi, som er indeks i tabell (eks: personnummer)
- **LSM-trær** – spesiell måte å lagre data laget for BigData (får data i stor fart). De er nyttige for å raskt sette inn data. Ikke pensum i dette emnet.

Indeksering

En tabell kan lagres i en indeks og en indeks kan lagres i tillegg til tabellen. **Lagring av indeksfiler på attributter brukes for å få rask tilgang til data i tabellene eller for å tvinge gjennom PRIMARY KEY/UNIQUE restriksjoner** (databasen lager en indeks for det unike

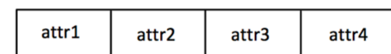
attributtet, fordi det gjør det lettere å finne ut om det finnes en tilsvarende verdi fra før av ved for eksempel innsetting). For indekser brukes flere ulike trær, og de vanligste er hashing, B+-trær og R-trær (flere-dimensjonal indeksering).

Postformat

Tradisjonell DBMS bruker *row store* (radlagring), der hver rad i tabellen lagres som en post i en fil i databasen. **En post vil bestå av flere felter med navn og datatype**, for eksempel integer, long integer, floating point, string (varierende og fast lengde), date/time, Blobs (lange felter), osv. **Det finnes flere mulige postformater, og SQL katalogen vil beskrive formatet til tabellen**, altså hvordan tabellen/posten er lagret (dvs. lengde, datatyper, osv.).

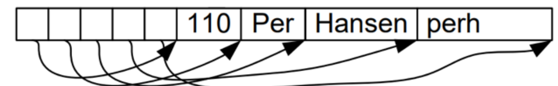
Attributter med fast lengde

Når tabellen har attributter med fast lengde, vil hvert attributt ligge på et fast offset i posten (figur viser post med fire attributter). Når DBMS skal dekode posten vil den derfor vite nøyaktig hvor i posten hvert attributt ligger.



Attributter med varierende lengde

Når tabellen har attributter med varierende lengde (eks: VARCHAR) må posten lagres på en annen måte. For eksempel kan vi bruke **postvektor format (se figur), der hvert felt i posten kan ha varierende lengde og feltene har pekere slik at posten nesten er selvbeskrivende (mangler ofte typeinfo)**. Dette formatet vil ofte ha et spesielt felt som gir antall attributter, antall nøkler, osv. Hver feltpeker kan også kombineres med typeinformasjon, men dette blir ofte gitt i SQL katalogen. **Et annet format som støtter felt med varierende lengde, bruker avgrensningskarakterer som er spesielle verdier som indikere enden av feltet**.



Blokkformat

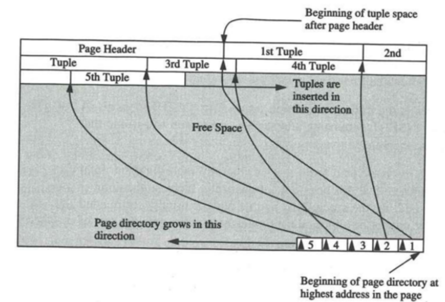
En blokk er en grunnleggende lagringsenhet som brukes av DBMS for å lagre data i databasen. Når DBMS leser eller skriver data til disken, vil det være i blokkenheter. Data vil ofte lagres som blokker både på disken og i minnet, slik at lesingen og skrivingen blir lettere. Høyere nivå DBMS programvare vil likevel ofte se data som samlinger av poster (dvs. som tabeller eller spørringsresultat). **En blokk vil som regel identifiseres vha en blokkID, som består av en enhetsID og en indeks innenfor enheten**. Husk: databasen er lagret i en enhet og blokken er igjen lagret i denne enheten. BlokkID kan være et felt av 4 eller 8 bytes. Med 4 bytes, vil én byte brukes for enhetsID og 3 bytes brukes for indeks inni enheten. Hvis hver blokk er 16 kB, kan hver enhet lagre $2^{3*8 \text{ bits}} = 2^{24}$ blokker med 16 KB, som utgjør 256 GB med data. Med 8 bytes, vil sju bytes kunne brukes for indeks inni enheten og dermed får man mulighet til å lagre store enheter.

En blokk kan ses på som en samling av spor (*slots*), der en post vil identifiseres av (BlokkID, spornummer), som ofte kalles RID (Record Identifier). RID brukes altså for å aksessere bestemte poster og kan brukes for å flytte en post innenfor blokken. **En post kan også aksesseres vha en nøkkel,** for eksempel i B+-trær der poster blir sortert basert på søkenøkkelen til treet.

Merk: en blokk har fast lengde, mens en post kan ha varierende eller fast lengde. Ved postformat ser vi om attributtene har fast eller varierende lengde, siden en post består av flere attributter. ved blokkformat ser vi om postene har fast eller varierende lengde, siden en blokk består av flere poster.

Det er flere ulike blokkformater som brukes i DBMS, og vi skiller mellom poster med:

- **Fast lengde – postene stilles opp etter hverandre. DBMS kan regne ut posisjonen til en bestemt post ved å bruke RID og poststørrelsen som gis i SQL katalogen.**
- **Varierende lengde - Poster med varierende lengde krever mer kompliserte skjema**. En mulighet er å bruke **avgrensningskarakterer** på attributtene og postene for å vise enden av attributtet og posten. Figuren viser en annen mulighet som legger til rette for poster med varierende lengde, lett innsetting og sletting, osv. **I dette formatet blir poster (*tuple* på figur) stilt opp fra starten av blokken og en post vektorpeker (*page directory* på figur) stilles opp fra enden av blokken.** Når disse endene møte, vil blokken være full. Vektorpekerne kan være sorterte på nøklene, slik at de kjenner rekkefølgen postene ble satt inn. Mye sletting resulterer i hull mellom postene og dette kan håndteres ved å kompaktering, der andre poster blir kopiert nedover mot den ene enden og pekerne oppdateres. Det er også vanlig å legge til flip/flop som er stempler ved starten og enden av blokken, som bør matche når man leser en fullstendig blokk. Disse vil oppdateres ved skriving til blokken. Header feltet brukes for identifikasjon, gjenvinning og mulige checksums (feildeteksjon).



Databasebuffer

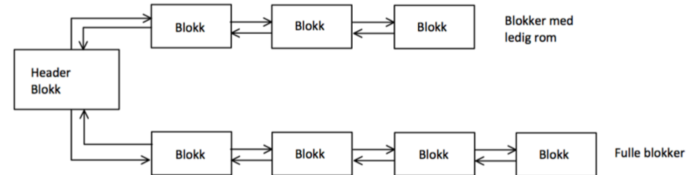
Blokker som skal brukes av DBMS blir lest fra disken inn i databasebuffere, som er buffere som utgjør store deler av hovedminnet (RAM) til datamaskinen. En databuffer er kopier av ofte brukte blokker fra databasen. **BlokkID blir brukt for å aksessere blokker på disken**, ved at de identifiserer enheten eller filen databasen ligger på og deretter indekserer inn i disse for å finne den spesifikke blokken. **Hvis blokkene er i databufferen blir de som regel aksessert vha en hashindeks som er basert på blokkID**. Hashindeksen brukes altså for å finne blokker som identifiseres vha blokkID. Blokker som hører til samme hashindeks vil være lenket sammen.

DBMS foretrekker å bruke sin egen databasebuffer istedenfor det virtuelle minnet til operativsystemet, fordi da får den full kontroll over hvilke blokker som ligger i minnet. Dette blir gjort ved at dataene låses (pinnes) i bufferen, slik at det virtuelle minnet ikke kan kaste dem ut. Det er nyttig at DBMS har egen databasebuffer, siden den vet mer om semantikken til dataen og hvordan den skal brukes. For eksempel kan det støtte *prefetching* av blokker. DBMS må kunne tvinge blokker til disken, som en del av transaksjonsprosesseringen. **Datablokker som endres pga brukertransaksjoner, må altså kunne skrives til disken og ikke bare til bufferen, slik at endringen blir permanent lagret.**

Heapfiler

Heapfiler er den enkleste og mest grunnleggende måten å lagre poster, ved at postene plasseres i filen i rekkefølgen de blir satt inn og nye poster settes inn ved enden av filen. En heapfil er altså en usortert lagring av poster. Tillatte operasjoner er å lage og slette filen, sette inn og slette poster i filen og skanne gjennom postene i filen. **En heapfil vil bestå av flere blokker som igjen består av flere poster.** Det er flere måter å implementere heapfiler, men den vanligste er å bruke lenkede lister med forover og bakover pekere mellom blokkene. Pekerne er blokkID. Noen implementasjon bruker to separate lister der en består av fulle blokker og den andre består av blokker som har ledig rom (se figur). Dette gjør det enklere å finne tilgjengelig rom for nye

poster. For å utvide en heapfil har man to muligheter for å opprette nye blokker. De kan tildeles plass på enheten en om gangen ved behov eller en sekvens av blokker kan tildeles plass samtidig for å oppnå høyere gjennomstrømming.



Innsetting av en ny post er svært effektivt, siden det kun krever at siste blokk i filen kopieres inn i bufferen, den nye posten legges til og blokken blir skrevet tilbake til disken. **Søk etter en post som oppfyller en søkebetingelse (eks: en bestemt attributtverdi) er derimot lite effektivt**, siden det innebærer et lineært (iterativt) søk gjennom blokk etter blokk i filen, helt til posten blir funnet. Hvis det er kun en post som tilfredsstiller søkebetingelsen vil programmet i gjennomsnitt søke gjennom halvparten av blokkene før den finner posten. Hvis ingen poster tilfredsstiller betingelsen må programmet søke gjennom alle blokkene. Ved **sletting av en post** må programmet finne blokken, kopiere den inn i bufferen, slette posten og deretter skrive blokken tilbake til disken. Dette vil føre til ubrukt plass i blokken, så sletting av store antall poster vil føre til bortkastet lagringsplass. Derfor vil filen reorganiseres ved tilstrekkelig mange slettinger, slik at blokkene blir kompakte (dvs. fylt opp).

Vanligvis vil man bruke indekser i tillegg til heapfiler, altså postene har indeks. For å aksessere en bestemt post, kan vi bruke RID, der blokkID brukes for å finne blokken og indeksen brukes for å finne posten innenfor blokken (merk: søk etter post som oppfyller en søkebetingelse og aksess av en bestemt post er to forskjellige handlinger!). **Fordeler med heapfil er at det er lett å sette inn poster, den er god til tabellskanning (ser på all data) og gir bra skriveytelse. Ulempen er at den er dårlig til søk etter attributter og områdesøk.** Heapfil blir derfor brukt når det er mye innsetting og lite søk.

Hashfiler

Hashfiler består av en rekke blokker som fylles med poster basert på deres verdi for en søkenøkkel. **Hashbaserte indekser brukes når vi ønsker direkte aksess basert på en søkenøkkel.** Søkenøkkelen er et felt i postene (dvs. et attributt i tabellen) og dens verdi sendes inn i en hashfunksjon som vil regne ut en hashverdi. Denne verdien vil fungere som en indeks som bestemmer hvilken blokk posten skal plasseres i. Hashverdien vil ofte ha et mye mindre verdiområde enn søkenøkkelen. For eksempel kan vi ha en million brukere som plasseres i 255 bøtter, ved at hashfunksjonen påføres en verdi hos hver bruker og regner ut en hashverdi mellom 0 og 255. Flere personer vil plasseres i samme bøtte, men når vi søker etter en bestemt bruker holder det å søke gjennom én bøtte istedenfor en million brukere. **Hashfunksjonen brukes altså for å spre postene utover på en måte som gjør det lettere å finne poster ved søk**.

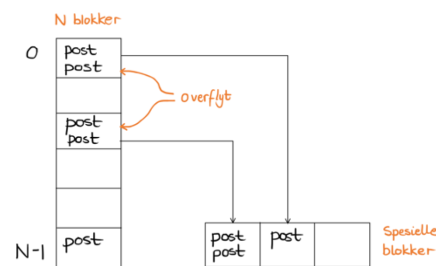
Det finnes mange ulike hashfunksjoner, og et eksempel er $h(K) = K \bmod M$ (der M er antall blokker). **En god hashfunksjon har jevn spredning av postene**, **men den vil være avhengig av at nøklene har et format som gjør at ikke alle poster ender i samme blokk**. Hashfiler blir brukt på to ulike måter avhengig av om antall poster er kjent på forhånd eller ikke.

Statisk hashing

Statisk hashing brukes når vi kjenner antall poster som skal lagres på forhånd, og det er den grunnleggende formen for hashbasert lagring. **En fil vil bestå av *N* blokker som lagrer postene. Ved å bruke hashfunksjonen på søkenøkkelen vil posten plasseres i en bestemt blokk i filen**. Hashfunksjonen vil spre postene over blokkene. For å hente en blokk vil formelen brukes på søkenøkkelen og passende blokk blir funnet. Deretter blir blokken gjennomsøkt.

En god hashfunksjon vil ha en jevn spredning av postene. Hvis hashfilen har skjev last, betyr det at **mange søkenøkler hasher til samme blokk, slik at blokken blir full. Dette kalles overflyt, og må håndteres vha overflyt lagring**:

- **Åpen adressering** – posten blir lagret i første etterfølgende blokk som har ledig rom
- **Separat overløp** – spesielle overflyt blokker brukes for å lagre overflyt poster og de fulle blokkene vil inneholde en peker mot en slik blokk (se figur). Når en post hasher til en full blokk vil den lagres i en overflyt blokk. Disse blokkene kan deles blant flere blokker eller være separat for hver blokk som har behov for overflyt lagring. Når en overflyt blokk blir full vil den få en peker til en ny overflytt blokk. Merk: det blir laget en lenket liste.
- **Multippel hashing** – en ny hashfunksjon blir brukt for å regne ut blokken som skal lagre overflyt posten. Dette kalles distribuert overflyt.



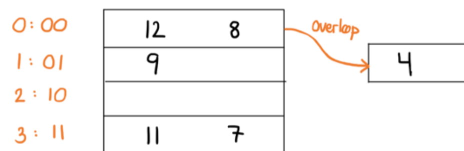

```
CREATE TABLE tm1 (
   s1 CHAR(32) PRIMARY KEY
)
PARTITION BY KEY(s1)
PARTITIONS 10;
```

I MySQL kalles hashing for *partition by key*. Figuren viser et eksempel der s1 er søkenøkkelen og det er 10 partisjoner i filen. Det er altså et fast antall blokker/partisjoner i samme fil, og overflyt lagring må brukes for dynamiske datamengder (ukjent antall poster). **Problemet med statisk hashing er at det ikke er dynamisk**, noe som er problematisk når det er vanskelig å estimere antall poster på forhånd. **Lange overflyt kjeder vil også gi dårlig ytelse**.

Eksempel 1 – statisk hashing

Vi har følgende verdier for søkenøkkelen: 9, 11, 12, 7, 8 og 4 som skal settes inn i en hashfil som har fire blokker som hver har plass til to verdier. Hashfunksjonen er $h(k) = k \bmod N$. Regn ut gjennomsnittlig aksesser for å finne en post.
Vi har at $N = 4$, slik at hashverdiene blir:

$$\begin{aligned} 9 \bmod 4 &= 1 = 01 \\ 11 \bmod 4 &= 3 = 11 \\ 12 \bmod 4 &= 0 = 00 \\ 7 \bmod 4 &= 3 = 11 \\ 8 \bmod 4 &= 0 = 00 \\ 4 \bmod 4 &= 0 = 00 \end{aligned}$$



Legg merke til at vi regner ut binær verdi, siden indekser i hashtabellen som regel er binære tall. Vi bruker de binære hashverdiene til å plassere nøkkelverdiene i blokkene. Vi begynner med 9 som plasseres i 01, 11 som plasseres i 11, osv. Når vi kommer til 4 som skal plasseres i 00, ser vi at denne blokken er full (kun to verdier i hver blokk). Så derfor må 4 plasseres i en overløpsblokk og vi må legge til en peker fra blokk med indeks 00 til denne.

For å finne gjennomsnittlig antall aksesser for å nå en post må vi se på:

$$\frac{\textbf{Totalt antall aksess}}{\textbf{Antall poster (nøkler)}}$$

I vårt tilfelle kan vi finne fem av postene vha én aksess, mens en post krever to aksess. Det er seks nøkler, så derfor vil gjennomsnittlig antall bokser som må aksesseres være $7/6$.

Eksempel 2 – statisk hashing

Gitt tabellen på figuren til høyre, sett inn radene i en statisk hashfil med fire blokker med separat overløp. Det er plass til to poster i hver blokk. Bruk hashing på Nr. Finn gjennomsnittlig aksess
Vi har at $N = 4$, slik at hashverdiene blir:

$$\begin{aligned} 13 \bmod 4 &= 1 = 01 \\ 2 \bmod 4 &= 2 = 10 \\ 1 \bmod 4 &= 3 = 11 \\ 14 \bmod 4 &= 2 = 10 \\ 7 \bmod 4 &= 3 = 11 \\ 5 \bmod 4 &= 1 = 01 \\ 6 \bmod 4 &= 2 = 10 \\ 8 \bmod 4 &= 0 = 00 \\ 11 \bmod 4 &= 3 = 11 \end{aligned}$$

Nr	Navn
13	Anna
2	Bernt
1	Diana
14	Erik
7	Hanne
5	Karl
6	Mina
8	Oscar
11	Synne

Vi bruker de binære hashverdiene til å plassere nøkkelverdiene i blokkene. Når vi kommer til 6 som skal plasseres i 10 og 11 som skal plasseres i 11, ser vi at disse blokken er full og må derfor plasseres i overløpsblokker.

Siden syv av postene kan nås vha aksess til én boks og to poster kan nås vha aksess til to bokser og det er ni nøkler, vil gjennomsnittlig aksess for å finne en post med en gitt søkenøkkelvære være $11/9 = 1{,}22$.

Dynamisk (extendible) hashing

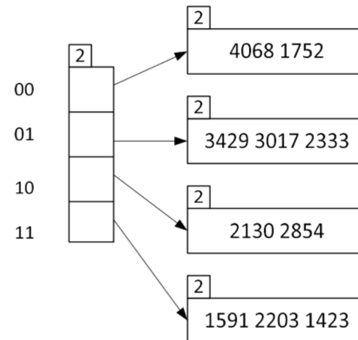
Extendible hashing tillater dynamiske hashfiler, og det brukes når vi ikke kjenner antall poster som skal lagres på forhånd. Ved statisk hashing må antall poster estimeres ved starten og lengden til hashfilen må bestemmes. Når en blokk i en statisk hashfil blir full, er det mulig å doble lengden til filen og sende alle postene gjennom hashfunksjonen på nytt, men dette krever lesing og skriving av alle blokkene i filen. Som regel vil overflyt lagring brukes for å håndtere denne situasjonen, men lange overflytkjeder kan også gi dårlig ytelse.

Extendible hashing bruker en katalog (*directory*) for å peke på den logiske strukturen til hashfilen, og blokkene kan være ulikt fordelt. Dvs. katalogen vil peke på faste plasseringer i hashfilen, og blokkene kan bytte plassering. Når en nøkkel hasher til en blokk som er full, vil kun nøklene i denne blokken rehashe til nye blokker. Dette oppnås vha *Directory doubling* og blokksplitting, som vi ser på neste side.

K	h(K)	binary
4068	4	0100
1752	8	1000
3429	5	0101
2130	2	0010
2854	6	0110
1591	7	0111
2203	11	1011
1423	15	1111
3017	9	1001
2333	13	1101
3923	3	0011
4817	1	0001
4876	12	1100

Eksempel – extendible hashing

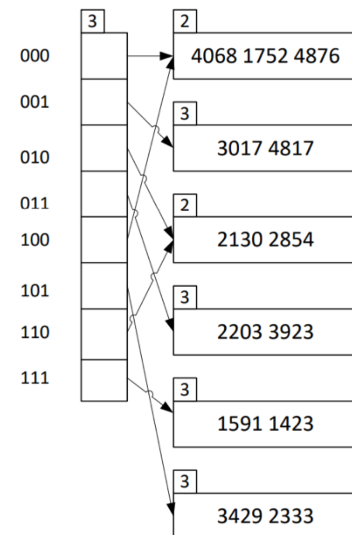
Vi skal illustrere extendible hashing vha et eksempel, der hashfunksjonen er $h(k) = k$ mod 16. Tabellen viser nøklene (K) og hashverdien i desimal- og binærtall. Vi begynner med en katalog som har 4 luker og en hashfil som har 4 blokker som hver har plass til tre nøkler. **Siden vi har 4 luker vil vi bruke de to siste bitene i de binære tallene som indeks i hashingen** (siden 00, 01, 10 og 11 = 4 muligheter). Nøkkelen 4068 har binær hashverdi 0100, slik at de to siste sifrene er 00. Denne nøkkelen hasher derfor til blokk 00. Etter å ha satt inn de første 10 nøklene får vi følgende hashfil:



Tallet som er festet til katalogen (2 på figur) kalles global dybde og gir antall bits som brukes i hashingen av nye nøkler. **Tallet festet til en blokk kalles lokal dybde og gir antall bits som ble brukt i hashingen av nøklene som befinner seg i denne blokken**. Som vi skal se vil ikke disse alltid være like.

Innsetting av nøkkel 3923 vil føre til en *directory doubling,* der katalogen blir doblet i størrelse. Dette skyldes at blokk 11 er full. Hashfilen vil fortsatt inneholde blokkene fra tidligere hashing. **Det er først når de blir overfylt at blokkene splittes opp og nøklene rehasher inn i de to blokkene basert på de tre siste bitene**. De resterende hashingene er:

- Innsetting av nøkkel 3923 vil resultere i en blokksplitt, siden blokk 11 er full. Denne blokken deles inn i to blokker 011 og 111 med lokal dybde 3, og nøklene 1591, 2203 og 1423 rehasher til disse. Nøkkelen 3923 vil hashe til 011, siden den har binærtall 0011.
- Innsetting av nøkkel 4817 vil resultere i en blokksplitt, siden blokk 01 er full. Denne blokken deles inn i to blokker 001 og 101 med lokal dybde 3, og nøklene 3429, 3017 og 2333 rehasher til disse. Nøkkelen 4817 vil hashe til 001, siden den har binærtall 0001.
- Innsetting av nøkkel 4876 vil ikke resultere i en blokksplitt, siden blokk 00 har ledig rom. Denne blokken vil derfor fortsatt ha lokal dybde 2.

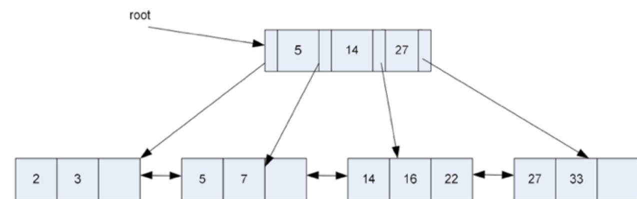


Når en nøkkel hasher til en full blokk og lokal dybde er mindre enn global dybde, vil blokken splittes opp i to blokker og lokal dybde oppdateres. **Når en nøkkel hasher til en full blokk og lokal dybde er lik global dybde, får vi en *directory doubling* (blokken har allerede blitt splittet). Fordelen med extendible hashing er at det er dynamisk og det er kun nøklene i den fulle blokken som rehasher. Ulempen er at søk etter en post vil kreve to aksesser til hashfilen istedenfor én, som er tilfellet ved statisk hashing.**

Merk: B+-trær er ikke fantastisk på noe, men de er gode på det meste

B+-trær

B+-trær er den mest brukte aksesstrukturen i databaser, og det er en generalisering av ISAM strukturen som bruker flere nivåer for å oppnå hurtig henting av poster. **Fordelen med B+-trær er at de er gode for mange ulike situasjoner,** slik som sekvensiell og sortert skanning, direkte aksess basert på en søkenøkkel (likhetssøk), områdesøk, osv. **B+-trær vil organisere indeksnøklene i sortert rekkefølge og treet er alltid balansert**. Derfor er de også gode for innsetting. Ulempen med B+-trær er at nylige innsettinger har en tendens til å spres over blokkene til treet, slik at det blir stor skrivelast til disken.



Figuren viser et B+-tre med høyde to. **Ved bladnivået (nivå 0) ligger de faktiske nøklene som blir indeksert av treet. Ved rotnivået (nivå 1) ligger nøklene som hjelper oss med å navigere til nøklene ved nivået under.** Nøkkel 14 har en peker på venstre side som vil føre til en blokk der nøklene er mindre enn 14, mens pekeren på høyre side vil føre til en blokk der nøklene er 14 eller større. Neste nøkkel på rotnivået er 27, så pekeren mellom 14 og 27 vil føre til en blokk der nøklene er større eller lik 14 og mindre enn 27. **Pekerne vil være blokkID**. **Nøklene ved bladnivået kan være poster som er fullstendige rader fra en tabell eller indekserte nøkler med et felt som peker mot en post i en annen fil eller tre**.

Søk etter post og områdesøk

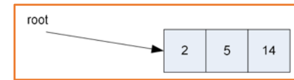
Når vi skal søke etter en bestemt nøkkelverdi, for eksempel nøkkel 3, vil vi starte ved rotnivået. I dette tilfellet er 3 mindre enn den minste nøkkelen 5, derfor vil vi følge pekeren til første blokk ved nivået under, som er bladnivået i dette tilfellet. **Siden nøklene er sorterte blir binærsøk brukt for å søke gjennom en blokk**, og i dette tilfellet vil vi finne nøkkel 3 som den andre nøkkelen i blokken. Vi ønsker nå å søke etter nøkkel 15 og starter i rotblokken, der vi finner pekeren mellom 14 og 27. Det binære søket av bladblokken vil ikke gi noe resultat, siden den ikke finner nøkkel 15. Et slikt søkt kan brukes for å returnere posisjonen der en ny post med nøkkel 15 kan settes inn.

B+-trær er nyttige for områdesøk, for eksempel kan vi søke etter alle nøklene som er større eller lik 14. Det binære søket av rotblokken vil gi pekeren til høyre for nøkkel 14, som fører til blokken med nøkler som er 14 og større. **B+-trær lar oss skanne bladnivået sidelengs til høyre og starter med blokken som inneholder 14**. Denne sidelengs skanningen er en av fordelene med B+-trær og kan ses som piler mellom blokkene på bladnivået. Disse pilene peker begge retningene, noe som betyr at skanningen kan gå både mot venstre (mindre nøkler) og høyre (større nøkler). **Dette er også nyttig for sekvensiell, sortert skanning som starter ved post helt til venstre og henter alle postene til høyre** (eller motsatt).

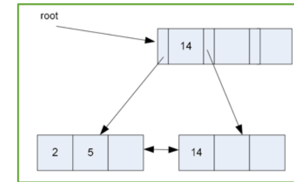
Innsetting

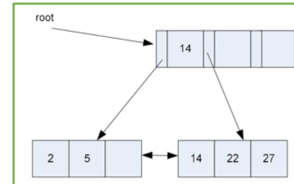
En annen fordel ved B+-trær er at poster kan settes inn hvor som helst i treet uten bemerkelsesverdig tap i effektivitet. Hvis det er ledig rom i blokken, vil posten settes inn. Hvis ikke vil den eksisterende blokken splittes opp i to blokker ved at det legges til en blokk til høyre og eksisterende poster fordeles mellom disse. **Dette kalles blokksplitt, og som regel vil halvparten av eksisterende poster flyttes til den nye blokken til høyre. Deretter blir den nye nøkkelen satt inn**.

Vi ser på et eksempel der følgende nøkler blir satt inn i et tomt B+-tre: 2, 5, 14, 22, 27, 33, 3, 7, 16 og 24. Vi har blokker med rom for tre nøkler og rotblokken kan ha fire pekere. Dette blir gjort på følgende måte

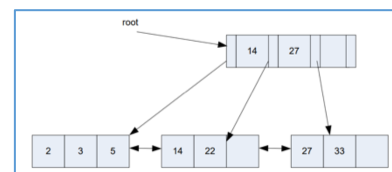


1. Vi setter inn nøklene 2, 5 og 14 og får B+-treet på figuren. Pekeren *root* fører til en bladblokk med ett nivå.

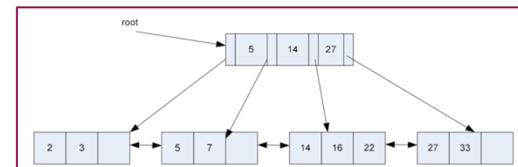




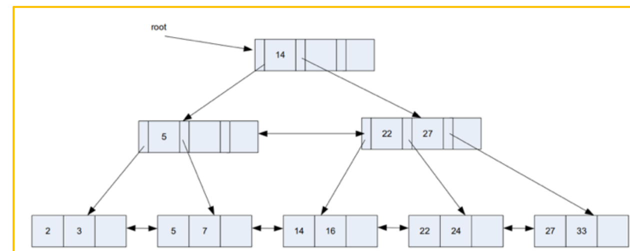
2. Vi setter inn nøkkel 22 og 27. Ved innsetting av nøkkel 22 vil bladblokken være full, siden den inneholder tre nøkler. Derfor vil bladblokken splittes i to og eksisterende poster fordeles mellom disse. Ved en blokksplitt vil halvparten av nøklene flyttes til høyre blokk, før den nye nøkkelen settes inn. Merk: blokksplitt vil alltid utføres før den nye nøkkelen settes inn. **Det blir også laget en ny rotblokk som fylles med splittnøkkelen, som er nøkkelen til venstre i den nye blokken som ble laget ved splitten** (her: nøkkel 14). Siden det er ledig rom i den nye blokken til høyre blir nøkkel 27 satt inn uten blokksplitt. Nå vil *root* peke mot en blokk ved nivå 1.



3. Vi setter inn nøkkel 33 og 3. Når nøkkel 33 settes inn får vi enda en blokksplitt ved bladnivå. Her vil 27 være splittnøkkelen som legges til rotblokken. Nøkkel 3 kan settes inn uten blokksplitt, siden det er ledig rom i venstre bladblokk



4. Vi setter inn nøkkel 7 og 16. Når nøkkel 7 settes inn får vi enda en blokksplitt ved bladnivå. Her vil 5 være splittnøkkelen som legges til rotblokken. Nøkkel 16 kan settes inn uten blokksplitt, siden det er ledig rom i tredje bladblokk
5. Vi setter inn nøkkel 24, og får B+-treet på figuren. Når vi setter inn nøkkel 24 vil vi få enda en blokksplitt ved bladnivået, men i dette tilfellet er rotblokken full. Derfor må også denne splittes. **Når vi splitter en blokk ved nivåer som er over nivå 0, vil splittnøkkelen flyttes til nivået over (dvs. ikke beholdes ved begge nivåene)**. I dette tilfellet betyr det at nøkkel 14 i nivå 1 blir flyttet til nivå 2. Grunnen til dette er at nivå 1 og 2 gir kun retninger, mens i nivå 0 ligger de «ekte» nøkkelindeksene. Derfor kan vi ikke flytte (fjerne) en nøkkel fra nivå 0, siden den er en del av dataen i databasen.



Ved innsetting i en full bladblokk, vil det legges til en blokk til høyre og eksisterende nøkler fordeles mellom de to blokkene. Ved innsetting i full rotblokk vil rotblokken splittes i to og det legges til en ny rotblokk i nivået over. Splittnøkkelen flyttes til nivået over, men nøkler vil ikke fjernes fra nivå 0, siden dette nivået må inneholde alle nøklene.

Noen viktige poeng:

- **Indekspostene blir lagret i bladblokker** (nivå 0), mens blokkene over brukes for å navigere til bladblokkene.
- **Postene/nøklene ved et nivå vil være sorterte**
- **Blokkene må være minst 50% fylt, og ved gjennomsnitt vil $2/3 = 67\%$ være fylt når nøklene blir tilfeldig satt inn**.
- Blokkene har en størrelse som passer disken, for eksempel 5KB, 8KB, 16KB eller 32KB

- Blokkene er diskorienterte og blir som regel lest inn i minnet på diskformat. Det er også mulig å omdanne diskformatet til for eksempel javaobjekter når en blokk leses, og motsatt ved skriving.
- Ved sletting av poster vil nivåer komprimeres og tomme blokker slettes.
- Sammenlignet med LSM tre, har B+-tre bedre leseytelse, mens LSM trær har bedre skriveytelse.
- Høyden til et B+-treet er antall blokker fra og med rotnivået til og med bladnivået, og den er ofte 2, 3 eller 4

B+-trær i praksis

Fanout er antall pekere en blokk ved høyere nivå vil ha (dvs. antall blokker under seg), og den vil vanligvis være rundt 133. Typisk fyllgrad av blokkene er 67%. Hos non-clustered B+-tre med høyde 3 vil typisk kapasitet være $133^3 = 2\ 352\ 636$ poster, mens med høye 4 vil typisk kapasitet være $133^4 = 312\ 900\ 700$ poster. For et clustered B+-tre vil postene ved høyere nivå være mindre (omtrent ¼) enn postene ved bladnivået, siden disse kun inneholder søkenøkkel og peker (blokkID). **Hver blokk ved bladnivået vil derfor ha plass til mindre poster, slik at siste fanout blir mindre** (eks: 20 istedenfor 133). Hos clustered B+-tre med høyde 4 vil typisk kapasitet være $133^3 * 20 = 47\ 052\ 740$ poster, som er mye mindre enn for non-clustered.

I praksis vil de øverste nivåene alltid være i buffer, siden de brukes for alle søkene nedover i treet. Det er som regel kun bladblokkene som kanskje må hentes fra disken.

Poster i B+-trær

Postene som ligger lagret i B+-trær vil være forskjellig avhengig av om de ligger på bladnivået eller høyere nivå. For eksempel kan vi bruke et B+-tre for å lagre tabellen $\text{Student}(\text{pnr}, \text{studentnr}, \text{navn}, \text{adresse})$, der studentnr er brukt som indeks.

- **Bladnivå (nivå = 0)** – hver post ser omtrent slik ut: (‘010195 12345’, 123456, ‘Hans Hansen’, ‘Revekroken 1’). Hver blokk på bladnivå kan holde omtrent 150 poster.
- **Høyere nivå (indeksnivå > 0)** – hver blokk ser omtrent slik ut (‘010195 12345’, blokkID). Hver blokk på indeksnivå kan holde omtrent 600 poster.

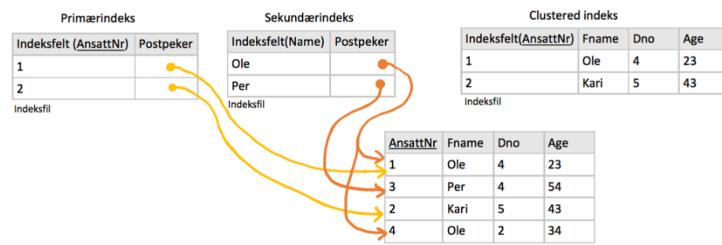
Indeksering

Indeksering brukes for å gjøre spørringer (*queries*) raskere, ved at vi enklere finner poster vi har indeksert (dvs. gitt en indeks). For eksempel kan vi ha tabellen $\text{Student}(\text{pnr}, \text{studentnr}, \text{navn}, \text{adresse}, \text{epost})$, der vi bruker studentnummer som indeks, siden det er det vi skal spørre etter. Når vi bruker $\text{SELECT navn FROM Student WHERE studnr} = 123456$, vil indeksen brukes for å lettere hente Student tuplen som oppfyller denne betingelsen. Hvis vi søker etter noe som ikke er en indeks, for eksempel navn, må vi skanne hele filen. **Indekser blir også brukt for å tvinge gjennom UNIQUE og primærnøkkel-restriksjoner og for å sortere tuplene etter et bestemt attributt som brukes som indeks** (eks: score hos en film).

Indeksfeltet er attributtet i posten som blir indeksert. Det finnes flere typer indekser:

- **Primærindeks** – indeks på primærnøkkelen som er unik og gir en ordning av filen
- **Sekundærindeks** – gir en annen måte å aksessere data for en fil der primæraksess allerede eksisterer (dvs. hos en post som har primærnøkkel, blir et annet felt brukt som

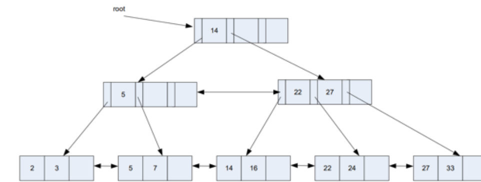
Eksempel: vi har $\text{Employee}(\underline{\text{AnsattNr}}, \text{Fname}, \text{Dno}, \text{Age})$. For primærindeks vil det lages en indeksfil der indeksfeltet er primærnøkkelen AnsattNr. For sekundærindeks vil det lages en indeksfil der indeksfeltet ikke er primærnøkkelen, for eksempel kan den være Fname. Hos disse non-clustered indeksene vil indeksfilen være aksessbaner som består av indeksfeltet og en peker til tilhørende post eller blokk som er lagret i datafilen. For clustered indekser vil postene være lagret sammen med indeksfeltet.



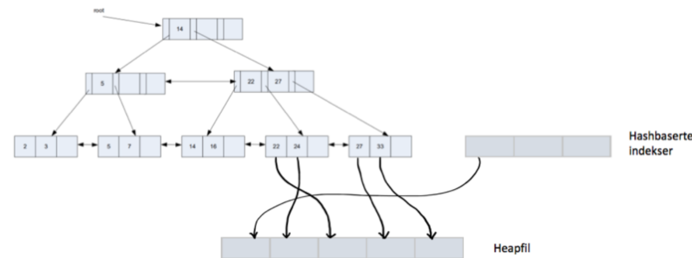
indeksfelt). Sekundærindeksen kan være en kandidatnøkkel (unik verdi) eller et ikke-nøkkelattributt (duplikate verdier). Det brukes for å tvinge gjennom UNIQUE eller få spørringer på andre attributter enn primærnøkkelen til å gå raskere.

- **Clustered indeks** (gruppert) – indeks på en tabell der **postene er fysisk lagret sammen med indeksen** (eks: clustered B+-tre). Indeksen kan være en nøkkel eller et ikke-nøkkelattributt. Ved ikke-nøkkelattributt vil like indekser være gruppert sammen.

Det finnes flere alternativer for lagring og indeksering av en tabell, og hvilke som brukes er avhengig av systemet som brukes (eks: MySQL, Oracle, osv.). Noen alternativer er:



- **Clustered B+-tre** – vanligste lagringsmetode. Består av et B+-tre med primærindekser, der postene er lagret sammen med indeksene i bladblokkene (se figur).
- **B+-tre med heapfil** – postene er lagret i en heap fil og B+-treet har primærindekser (RID) som peker til denne filen. Det kan også være en hashindeks til noen felter hos postene (se figur).



- **Heapfil** – postene blir lagret etter hverandre uten noen annen organisering. Dette brukes når tabellen har ingen definert primærnøkkel, og det er nyttig når det ikke trengs direkte aksess basert på en søkenøkkel eller søk etter spesifikke poster (dvs. når hele tabellen skannes).
- **Clustered hashindeks** – primærnøkkelen brukes som hashindeks, og postene er lagret sammen med indeksene i hashfilen.
- **LSM-trær** – moderne lagrings- og indekseringsmetode som «cacher» de nyeste innsatte eller oppdaterte postene. Dette gir høye skriveytelse, som følge av bedre komprimering av data og lavere antall diskskrivinger. Eldre poster flyttes over i langtidslager, og blir dermed vanskeligere å lese. LSM-trær brukes når man ønsker høy skrivehastighet og kun leser den nyeste dataen. Brukes for Big Data.
- **Kolonnelager** – tradisjonelle SQL databaser bruker radsbasert lagring, der hver rad i tabellen lagres som en post. Datavarehus krever aggregering (eks: sum) av enkelte attributter og bruker derfor heller kolonnebasert lagring av tabeller, der hver kolonne i tabellen lagres som en post. Dette gjør at mindre data må hentes og det blir lettere å komprimere dataen.

Behandling av spørringer (*queries*)

Vi skal nå se hvordan enkle spørringer blir utført ved å bruke indeksering og lagringskonseptene heapfil, Clustered B+-tre, non-clustered B+-tre med heapfil og clustered hashindeks.

Eksempel

Vi har en tabell med ansatte Employee(empno, name, age, depno, salary) og antar at det er 100 000 ansatte i tabellen. Hver blokk har plass til 100 poster. Vi får følgende oppsett:

1. **Heapfil** – siden hver blokk kan inneholde 100 ansatte, vil heapfilen bestå av 1000 blokker. En heapfil er et nyttig startpunkt.

2. **Clustered B+-tre** – vi antar at postene blir tilfeldig satt inn, slik at fyllgraden er 67%. Siden bladblokkene har plass til 100 poster, vil de gjennomsnittlig romme 67 poster. Bladnivået vil derfor inneholder omtrent 1500 blokker. For å finne høyden til treet (ofte 3 eller 4) må vi se på antall pekere ved de ulike nivåene. Bladnivået består av 1500 blokker og siden alle blokkene må ha en peker, må nivå 1 totalt bestå av 1500 poster. Hvis vi antar at disse postene trenger 20% av lagringsplassen til en Employee post, kan hver indeksblokk i nivå 1 inneholde fem ganger så mange poster som bladblokkene. Siden bladblokkene inneholder 67 poster, kan indeksboksene derfor inneholder $67 * 5 = 335$ poster. Derfor vil det være 5 blokker på nivå 1 $(1500/335 = 4.78)$. Hver av disse blokkene må ha en peker på nivå 2, og siden det kun krever 5 poster vil nivå 2 bestå av en blokk. B+-treet har derfor høyde 3.
3. **Non-clustered B+-tre med heapfil** – heapfilen vil bestå av 1000 blokker. Siden vi har et non-clustered B+-tre vil hver post være på formen (empno, RID), der RID er (blokkID, indeks i blokken). Vi antar at disse postene bruker 25% av lagringsplassen til en Employee post. Siden det trengs 1500 blokker for å romme Employee postene, trengs det $25\% * 1500 = 375$ blokker for å romme postene (empno, blokkID). Hver av disse må ha en peker, så derfor må nivå 1 bestå av 375 poster. Over regnet vi ut at hver blokk kan romme gjennomsnittlig 335 slike poster, men splittingen vil som regel ikke skje før blokken er full (dvs. når $335 + 50\% = 503$ poster er satt inn). Dette B+-treet vil derfor ha høyde 2.
4. **Clustered hashfil** – en hashfil vil som regel ha 80% fyllgrad, altså vil 80% av filen ikke være tomme blokker. Siden det trengs 1000 blokker for å få plass til Employee postene, vil derfor totalt antall blokker være $1000/0.8 = 1250$.

Dette er prosentregning:

$$\text{det hele} = \frac{\text{delen}}{\text{prosent}}$$

Her vil 1000 blokker være delen av hashfilen som ikke er tom, og denne delen utgjør 80% av hashfilen. Hele hashfilen vil derfor være $1000/0.8 = 1250$ blokker

Enkle SELECT spørringer

Det er flere typer spørringer som blir utført ulikt for de ulike lagringsmetodene:

- SELECT $*$ FROM table
  1. Vi må skanne alle 1000 blokkene
  2. Vi må skanne bladnivået til B+-treet og indeksnivåene, altså 2 + 1500 blokker
  3. Vi må skanne heapfilen, altså 1000 blokker
  4. Vi må skanne hashfilen, altså 1250 blokker
- SELECT attributter FROM table WHERE key $=$ konstant
  1. I gjennomsnitt må vi skanne halvparten av heapfilen, altså 500 blokker. Antar at nøkkelen er unik
  2. Vi må traversere nedover B+-treet, altså 3 blokker
  3. Vi må traversere nedover B+-treet og aksessere heapfilen, altså 2+1 blokker
  4. I hashfilen vil gjennomsnittlig antall bokser som aksesseres være 1.2, pga. overflyt
- SELECT attributter FROM table WHERE key $>$ konstant
  1. Vi må skanne alle 1000 blokkene
  2. Vi må traversere nedover B+-treet og skanne fremover ved bladnivået. Hvis vi antar at 20% av nøklene matcher, må vi traversere $2 + 0.2 * 1500 = 302$ blokker.
  3. Hvis vi bruker B+-tre indeksen må vi traversere 1 blokk nedover og $0.2 * 375 = 75$ blokker fremover. Videre må vi følge $0.2 * 100\,000 = 20\,000$ pekere til poster i heapfilen. Altså, må $20\,000 + 1 + 75 = 20\,076$ blokker aksesseres. Derfor er det bedre å skanne hele heapfilen som er 1000 blokker.
  4. Vi må skanne hele hashfilen, altså 1250 blokker

```sql
SELECT *
FROM Employee
WHERE dno=4 AND age>50;
```

Sammensatte indekser

Hvis vi har tabellen $\text{Employee}(\text{Ssn}, \text{dno}, \text{age}, \text{street}, \text{Salary}, \text{Skill})$ og bruker spørringen på figuren, vil vi ha flere muligheter for indeksen:

- **Indeks på dno** – finn alle poster med $\text{dno} = 4$ og deretter sjekk om $\text{age} > 50$
- **Indeks på age** – skann indeksene oppover fra 50 og finn alle postene med $\text{dno} = 4$
- **Sammensatt indeks på $(\textbf{age}, \textbf{dno})$ eller $(\textbf{dno}, \textbf{age})$** – bruk den som er mest selektiv først, altså den som gir færrest poster i resultatet.

JOIN spørringer

For at indekser skal være nyttig for JOIN spørringer, må join betingelsen være indeksen.

Vi skal se på **nested loop join**, der en buffer brukes for å slå sammen de to tabellene. Det er kun blokker som befinner seg i bufferen som går igjennom JOIN prosessen, og blokkene blir lest inn i bufferen ved ulik loops. Merk: det er postene som slås sammen dersom de oppfyller join betingelsen. For hver blokk fra en tabell (ytre loop) vil alle blokkene fra den andre tabellen ses gjennom for å undersøke om noen poster skal slås sammen. Hvis bufferen består av N blokker, vil 1 settes av til resultatet (dvs. sammenslåtte poster fra denne loopen), 1 settes av for den ene tabellen og $N - 2$ settes av for den andre tabellen (ytre loop). I hver loop vil de $N - 2$ bufferblokkene fylles med blokker fra den ene tabellen og den ene bufferblokken fylles med alle blokkene fra den andre tabellen en etter en. Poster som slås sammen legges til bufferposten for resultatet. Deretter blir loopen gjentatt helt til alle blokkene er gått igjennom

```sql
SELECT E.empName, E.salary, D.location
FROM Employee AS E JOIN Department AS D ON E.depno = D.dno
WHERE dname = 'Sales';
```

Vi skal illustrere dette vha tabellen $\text{Department}(\underline{\text{dno}}, \text{dname}, \text{manager}, \text{location})$ og spørringen på figuren. Vi antar at det er 500 avdelinger som er lagret i 5 blokker i en heapfil, og 100 000 ansatte som er lagret i 1500 blokker i en heapfil. Bufferen består av 5 blokker, så 3 blokker brukes for å holde blokkene fra den ene tabellen. Vi har to fremgangsmåter:

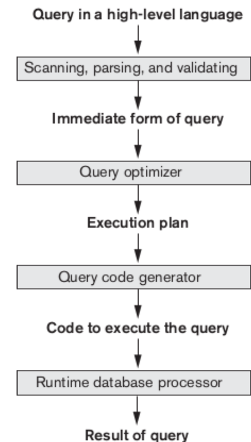
1. De 3 bufferblokkene fylles med Department blokker og de 1000 Employee blokkene blir lest inn i den ene bufferblokken en etter en. Siden det er 5 Department blokker trengs det 2 loops for å få gå igjennom alle disse. For hver loop vil alle Employee blokkene leses en gang. Totalt antall blokklesninger blir derfor $5 + 2 * 1000 = 2005$.
2. De 3 bufferblokkene fylles med Employee blokker og de 5 Department blokkene blir lest inn i den ene bufferblokken en etter en. Siden det er 1000 Employee blokker trengs det (1000/3) loops for å få gå igjennom alle disse. For hver loop vil alle Department blokkene leses en gang. Totalt antall blokklesninger blir derfor $1000 + 1000/3 * 5 = 2670$.

Merk: blokkene som lastes inn i $N - 2$ bufferblokk blir lest én gang, mens blokkene som lastes inn i den ene bufferblokken blir lest en gang per loop. **En generell regel er derfor å ha den minste tabellen i den ytre loopen til nøstingen, siden dette gir færre loops**.

Hvis vi ser nærmere på spørringen kan vi se at den har en betingelse $\text{dname} = \text{'Sales'}$, så det vil være få avdelinger som brukes i sammenslåingen. Derfor er det lurt å bruke Department som ytre join, siden Department postene som oppfyller denne betingelsen sannsynligvis får plass i én blokk. Dermed trenger vi kun én loop og antall blokklesninger blir $1 + 1 * 1000 = 1001$.

Kapittel 18 – Strategier for query prosessering

Dette kapittelet ser på teknikker som brukes internt av en DBMS for å prosessere høy-nivå spørringer (*queries*). Vi vil fokusere på RDBMS (relasjonell DBMS). Figuren viser de ulike stegene i denne prosesseringen. **En spørring uttrykket i høy-nivå språk (eks SQL) må først skannes, analyseres (*parsed*) og valideres**. Skanneren vil identifisere spørresymboler (eks: attributtnavn) i spørreteksten, analysatoren sjekker om spørringen er formulert riktig mht. regler for språket og validatoren vil sjekke at alle attributt- og relasjonsnavnene er gyldige. **Deretter vil det lages en intern representasjon av spørringen, som ofte er et algebratre.** DBMS må deretter lage en utføringsstrategi eller **query plan** for å hente resultatet til spørringen fra databasefilene. **En spørring vil ha flere mulige query planer, og prosessen av å finne den mest passende kalles query optimalisering**. En query plan vil gi algebrauttrykket i tillegg til informasjon om hvilken rekkefølge algebraen skal utføres i, hvordan den skal aksessere indekser, osv.



I dette kapittelet skal vi se hvordan spørringer prosesseres og hvilke algoritmer som brukes for å utføre individuelle operasjoner i en spørring. **Query optimizer** vil lage en god utføringsplan og **kodegeneratoren** vil lage koden som skal utføre denne planen. **Runtime databaseprosessoren** har i oppgave å utføre (*execute*) koden og produsere resultatet til spørringen. Den valgte utføringsplanen vil ikke alltid være optimal, men det vil være den beste tilgjengelige strategien. Dette skyldes at det tar mye tid og krever mye detaljert informasjon for å lage en optimal plan.

Det er to hovedteknikken for query optimalisering:

1. **Heuristiske regler** – operatorene blir ordnet i bestemte rekkefølger
2. **Kostnadsestimering** – estimerer kostnaden ved ulike utføringsplaner og velger den med minst kostnad

18.1 Translasjon av SQL spørringer til relasjonsalgebra

De fleste RDBMS bruker SQL som query språk. En SQL spørring vil først oversettes til et ekvivalent uttrykk i relasjonsalgebra, som deretter blir optimalisert. Uttrykket representeres som et algebratre. **SQL spørringer deles inn i spørreblokker som deretter blir oversatt til algebraoperatorer**. **En spørreblokk vil inneholde et enkelt SELECT-FROM-WHERE uttrykk**, men kan også inkludere GROUP BY, HAVING og aggregat funksjoner.

Nøstede spørringer blir plassert i separate spørreblokker. For eksempel på figuren ser vi en nøstet spørring som vil deles inn i en ytre og en indre blokk. I den ytre blokken er *c* brukt for å representere resultatet fra den indre blokken. Vi får følgende relasjonsalgebrauttrykk for de hhv. indre og ytre blokk:

```
/*Nøstet spørring*/
SELECT Lname, Fname
FROM Employee
WHERE Salary > (SELECT MAX (Salary)
                FROM Employee
                WHERE Dno = 5);

/*Indre blokk*/
SELECT MAX (Salary)
FROM Employee
WHERE Dno = 5;

/*Ytre blokk*/
SELECT MAX (Salary)
FROM Employee
WHERE Salary > c;
```

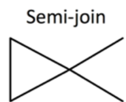
$$\gamma_{\text{Max Salary}}\left(\sigma_{\text{Dno}=5}(\text{Employee})\right)$$
$$\pi_{\text{Lname,Fname}}\left(\sigma_{\text{Salary}>c}(\text{Employee})\right)$$

Query optimizer vil deretter velge en utføringsplan for hver av spørreblokkene. I dette eksempelet må indre blokk evalueres en gang for å produsere et resultat som deretter brukes i ytre blokk. Indre blokk kalles nøstet delspørreblokk.

Flere operatorer – semi-join og anti-join

Noen spørringer vil oversettes til operatorer som ikke er en del av standard relasjonsalgebra vi så på i kapittel 8. To mye bruke operatorer er semi- og anti-join, som vi skal se nærmere på.

Semi-join



Semi-join brukes for å unøste EXISTS IN og ANY delspørringer, og betegnes som $\mathbf{T1.X\ S = T2.Y}$, der T1 er venstre tabell og T2 er høyre tabell i semi-join. Figuren viser notasjonen i relasjonsalgebra. En rad i T1 vil returneres med en gang $T1.X$ finner en match i $T2.Y$ og det vil ikke søkes etter flere matcher. Dette er en kontrast til inner join som finner alle mulige matcher.

```
/*Nøstet spørring med IN connector*/
SELECT COUNT(*)
FROM Department AS D
WHERE Dnumber IN (SELECT E.Dno
                  FROM Employee AS E
                  WHERE E.Salary > 200000);

/*Unøstet spørring med semi-join*/
SELECT COUNT(*)
FROM Department AS D, Employee AS E
WHERE D.Dnumber S = E.Dno AND E.Salary > 200000;
```

Øverst på figuren ser vi et eksempel på en nøstet spørring som slås sammen med IN. Prosessen av å fjerne den nøstede spørringen kalles unøsting, og det fører til spørringen nederst på figuren. Her kan vi se at semi-join ("S =") er brukt for å representere IN og den nøstede spørringen. Feilmelding skyldes at MySQL ikke støtter egen semi-join notasjon.

Anti-join

Anti-join brukes for å unøste NOT EXISTS, NOT IN og ALL delspørringer, og betegnes som $\mathbf{T1.X\ A = T2.Y}$, der T1 er venstre tabell og T2 er høyre tabell i anti-join. Figuren viser notasjonen i relasjonsalgebra. En rad i T1 vil avslås med en gang $T1.X$ finner en match i $T2.Y$. En rad i T1 vil returneres hvis den har ingen matcher med T2.

```
/*Nøstet spørring med NOT IN connector*/
SELECT COUNT(*)
FROM Employee AS E
WHERE E.Dno NOT IN (SELECT D.Dnumber
                    FROM Department AS D
                    WHERE D.zipCode = 30332);

/*Unøstet spørring med anti-join*/
SELECT COUNT(*)
FROM Department AS D, Employee AS E
WHERE E.Dno A = D.Dnumber AND D.zipCode = 30332;
```

Øverst på figuren ser vi et eksempel på en spørring som bruker NOT IN for å hente tupler som ikke matcher med resultatet fra nøstet spørring. Nederst på figuren ser vi resultatet av unøstingen, der anti join ("A =") er brukt for å representere NOT IN og den nøstede spørringen. Feilmelding skyldes at MySQL ikke støtter egen anti-join notasjon

18.2 Algoritmer for ekstern sortering (flettesortering)

Sortering er en av de viktigste algoritmene i query prosessering, for eksempel vil ORDER BY kreve at spørreresultatet er sortert. Sortering er også viktig i algoritmer som brukes for JOIN, UNION, INTERSECTION og PROJECT (duplikateliminasjon). **Ekstern sortering er sorteringsalgoritmer som brukes for å sortere store filer som er lagret på disk og ikke får plass i hovedminnet**, for eksempel databasefiler. Slike algoritmer vil ofte bruke en **sorter-fusjoner strategi**, der små delfiler kalt *runs* blir sortert og deretter slått sammen til større delfiler, som videre fusjonerer. Disse delfilene blir lest inn i bufferrom i hovedminnet der sorteringen og fusjoneringen tar sted, og dette rommet er en del av **DBMS cache** (delen av hovedminnet som kontrolleres av DBMS). Bufferrommet deles inn i flere individuelle buffere som hver rommer én diskblokk. Sorteringsalgoritmen består av en sorterings- og fusjoneringsfase.

Merk: Indeksering av passende attributt kan brukes for å unngå sortering.

Sorteringsfase

I sorteringsfasen vil deler av filen (*runs*) som får plass i bufferrommet leses inn i hovedminnet, sorteres vha en intern sorteringsalgoritme og leses tilbake til disken. Størrelsen til hver *run* og antall initiale *runs* (n_R) bestemmes av antall blokker i filen (b) og tilgjengelig bufferrom (n_B). For eksempel hvis hovedminnet har $n_B = 5$ tilgjengelige buffere

```
set     i ← 1;
        j ← b;          {size of the file in blocks}
        k ← n_B;        {size of buffer in blocks}
        m ← ⌈(j/k)⌉;

{Sorting Phase}
while (i ≤ m)
do {
        read next k blocks of the file into the buffer or if there are less than k blocks
          remaining, then read in the remaining blocks;
        sort the records in the buffer and write as a temporary subfile;
        i ← i + 1;
}

{Merging Phase: merge subfiles until only 1 remains}
set     i ← 1;
        p ← ⌈log_{k−1} m⌉   {p is the number of passes for the merging phase}
        j ← m;
while (i ≤ p)
do {
        n ← 1;
        q ← ⌈(j/(k−1)⌉;   {number of subfiles to write in this pass}
        while (n ≤ q)
        do {
          read next k−1 subfiles or remaining subfiles (from previous pass)
              one block at a time;
          merge and write as new subfile one block at a time;
          n ← n + 1;
        }
        j ← q;
        i ← i + 1;
}
```

og størrelsen til filen er $b = 1024$ diskblokker, så vil sorteringen kreve $n_R = \lceil b/n_B \rceil = 205$ initiale *runs* som hver har størrelse på 5 blokker (bortsett fra siste run som har 4 blokker).

Fusjoneringsfase

I fusjoneringsfasen vil sorterte delfiler slås sammen i løpet av en eller flere fusjoneringspass, som hver består av en eller flere fusjoneringssteg. I et fusjoneringssteg blir et antall delfiler lest inn i bufferen og slått sammen, mens i et fusjoneringspass har alle delfilene ved et bestemt størrelsesnivå blitt slått sammen. Fusjoneringsgraden (d_M) er antall sorterte delfiler som kan slås sammen i hvert fusjoneringssteg. I løpet av fusjoneringssteget vil én bufferblokk holde resultatet og resten ($n_B - 1$) vil holde delfiler som skal slås sammen. Fusjoneringsgraden vil derfor være den minste av ($n_B - 1$) og n_R (ikke nok delfiler til å fylle bufferen). Antall fusjoneringspass er $\log_{\mathrm{dM}}(n_R)$. For $n_B = 5$ bufferblokker, kan fire blokker slås sammen i ett fusjoneringssteg ($d_M = 4$). I første fusjoneringspass vil de 205 initiale, sorterte delfilene slås sammen til 52 større delfiler, som i andre fusjoneringspass slås sammen til 13 større delfiler. I tredje fusjoneringspass vil disse slås sammen til 4 større delfiler, som i fjerde fusjoneringspass slås sammen til én sortert fil. Denne sorteringen krever fire fusjoneringspass.

Ytelsen til sorteringen

Ytelsen til sorter-fusjoner algoritmen kan måles mht. antall diskblokker som leses og skrives før hele sorteringen er ferdig. Følgende er en tilnærming på denne kostnaden:

$$2b + 2b \log_{\mathrm{d_M}}(n_R)$$

Her vil $2b$ være antall blokkaksesser for sorteringen, siden hver blokk blir lest inn i hovedminnet og skrevet tilbake til disken i en av de sorterte delfilene. Antall aksesser for fusjoneringen er $2b \log_{\mathrm{dM}}(n_R)$, siden hver fusjoneringspass involverer lesing og skriving av omtrent alle filblokkene (b) og antall fusjoneringspass er $\log_{\mathrm{dM}}(n_R)$. Worst-case ytelse er ved minimum antall bufferblokker, altså $n_B = 3$, slik at $d_M = 2$.

18.3 Algoritmer for SELECT operatoren

Det er mange algoritmer som kan brukes for å utføre en SELECT operasjon, som er et søk for å finne en post i diskfilen som tilfredsstiller en bestemt betingelse. Noen søkealgoritmer krever at filen har bestemte aksessbaner og seleksjonsbetingelser. Vi ser på følgende seleksjoner:

- OP1: $\sigma_{\mathrm{Ssn=123456789}}(\mathrm{Employee})$
- OP2: $\sigma_{\mathrm{Dnumber>5}}(\mathrm{Department})$
- OP3: $\sigma_{\mathrm{Dno=5}}(\mathrm{Employee})$
- OP4: $\sigma_{\mathrm{Dno=5\ AND\ Salary>30000\ AND\ Sex=F}}(\mathrm{Employee})$
- OP5: $\sigma_{\mathrm{Essn=5123456789\ AND\ Pno=10}}(\mathrm{Employee})$
- OP6: SELECT * FROM Employee WHERE Dno IN (2, 37, 49)

De viktigste metodene i dette faget er S1, S5 og S6

Seleksjon av poster kalles **filskanning** fordi postene> i filen blir skannet for å søke etter og hente poster som tilfredsstiller seleksjonsbetingelsen. Hvis søkealgoritmen bruker en indeks, kalles søket for en **indeksskanning**. Noen søkealgoritmer som implementerer SELECT:

- **S1: Lineært søk (*brute force* algoritme)** – henter hver post i filen og tester om den har attributtverdi(er) som tilfredsstiller seleksjonsbetingelsen. Siden postene er gruppert i diskblokker, vil hver diskblokk leses inn i hovedminnet og deretter søkes gjennom.
- **S2: Binærsøk** – mer effektivt enn lineært søk og brukes når seleksjonsbetingelsen er en ekvivalenssammenligning på et nøkkelattributt som filen er ordnet etter. Eks: OP1 hvis Employee filen er ordnet etter Ssn.
- **S3a: Bruk av primærindeks** – henter maksimalt én post og brukes når seleksjonsbetingelsen er en ekvivalenssammenligning på et nøkkelattributt som er primærindeksen i lagringsstrukturen. Eks: OP1 der Ssn er primærindeks.
- **S3b: Bruk av hashnøkkel –** henter maksimalt én post og brukes når seleksjonsbetingelsen er en ekvivalenssammenligning på et nøkkelattributt som er hashnøkkel i lagringsstrukturen. Eks: OP1 der Ssn er hashnøkkel.
- **S4: Bruk av primærindeks for å hente flere poster** – brukes når seleksjonsbetingelsen innebærer å se om attributtet er $>, \geq, <$ eller $\leq$ et nøkkelattributt som er primærindeks i en ordnet fil. For eksempel ved $\text{Dnumber} > 5$ i S2 vil indeksen brukes for å finne posten som tilfredsstiller $\text{Dnumber} = 5$ og deretter vil alle etterfølgende poster i den ordnete filen hentes. For $<$ vil postene før indeksposten hentes.
- **S5: Bruk av clustered indekser med B+-tre eller hashfil for å hente flere poster -** brukes når seleksjonsbetingelsen er en ekvivalenssammenligning på et ikke-nøkkelattributt som er en clustered indeks. Siden indeksen ikke er en nøkkel, kan det finnes flere poster med samme indeks. Derfor vil indeksen brukes for å finne første post som oppfyller betingelsen og deretter blir etterfølgende poster lest helt til det når en post som ikke oppfyller betingelsen (filen er ordnet etter indeksen, eks B+-tre).
- **S6: Bruk av sekundærindeks med B+-tre (non-clustered)** – brukes for å hente én post når indeksfeltet er en nøkkel (unik verdi) og flere poster når indeksfeltet ikke er en nøkkel. Kan også brukes for sammenligninger som involverer $>, \geq, <, \leq$. For områdesøk (eks: $3000 \leq \text{Salary} \leq 4000$) kan B+-tre brukes siden indeksene er ordnet, så en sekvens av indekser i samme området som forespurt kan brukes for å gi pekere til ønskede poster.

Merk: B+-tre krever at det du søker etter er indeksen!

Husk: sekundærindeks kan være både nøkkel- og ikke-nøkkelattributt

Metode S1 kan brukes på alle filer, men de andre metodene krever bestemte aksessbaner på attributtene som brukes i seleksjonsbetingelsen. Metode S2 krever at filen er ordnet etter søkeattributtet, mens indekssøkene (S3a, S4, S5 og S6) krever at søkeattributtet er indeksert. Metode S4 og S6 kan brukes for områdesøk.

Søkemetoder for konjunktivseleksjon

Hvis seleksjonsbetingelsen er konjunktiv, altså bygd opp av flere enkle betingelser som er koblet sammen med AND, kan DBMS bruke følgende metoder for å implementere operasjonen:

- **S7: Konjunktiv seleksjon vha en individuell indeks** – hvis ett attributt i en av de enkle betingelsene har en aksessbane som kan brukes i metode S2-S6, bruk denne betingelsen til å hente postene og sjekk deretter om disse tilfredsstiller gjenværende betingelser.
- **S8**: **Konjunktiv seleksjon vha en sammensatt indeks –** hvis to eller flere attributter er involvert i ekvivalenssammenligninger i betingelsen og det eksisterer en sammensatt

indeks (eller hashstruktur) på det kombinerte feltet, kan denne indeksen brukes direkte. For eksempel for OP5 vil indeksen være $(\text{Essn}, \text{Pno})$

- **Konjunktiv seleksjon med snitt av postpekere** – hvis flere felt i betingelsene har sekundærindekser (ikke unike) eller andre aksessbaner som inneholder pekere mot poster, kan hver indeks brukes for å hente et sett med postpekere som tilfredsstiller de ulike betingelsene. Snittet (*intersection*) av disse settene vil gi postpekerne som tilfredsstiller den konjunktive seleksjonsbetingelsen, og disse brukes for å hente ønskede poster. Denne metoden antar at sekundærindeksene ikke er unike, fordi hvis en betingelse er på et nøkkelfelt, vil kun én post tilfredsstille betingelsen.

Når seleksjonsbetingelsen er en enkel betingelse (eks: OP1, OP2 og OP3) kan DBMS sjekke om en aksessbane eksisterer på attributtet som er involvert i betingelsen. For eksempel for OP1 holder det å sjekke om det eksisterer en primærindeks $\text{Ssn} = 123456789$. Hvis en aksessbane eksisterer vil metoden som korresponderer til denne banen brukes (eks: S3a for OP1). For konjunktive seleksjonsbetingelser vil query optimalisering brukes for å velge aksessbaner som er mest effektive, dvs. henter postene med lavest kostnad.

Søkemetoder for disjunktiv seleksjon

Disjunktiv seleksjonsbetingelse består av flere enkle betingelser som er koblet sammen med OR, for eksempel:

$$\sigma_{\text{Dno=4 OR Salary>30000 OR Sex=F}}(\text{Employee})$$

Postene som tilfredsstiller den disjunktive betingelsen vil være unionen av postene som tilfredsstiller de individuelle betingelsene. Hvis en av betingelsene ikke har en aksessbane må vi bruke lineær søk. Hvis alle betingelsene har en aksessbane kan metodene S1 til S6 brukes for å hente postene som tilfredsstiller hver betingelse og deretter brukes UNION for å fjerne duplikater. Query optimalisering vil velge aksessbaner som er mest effektive.

Estimering av selektiviteten til en betingelse

For å minimere kostanden til utførelsen av spørringen mht. ressurser og tiden som brukes, vil query optimizer motta input fra DBMS katalogen. Dette er viktig statistisk informasjon, slik som:

- Antall rader/poster (r_R)
- Lengden til hver tuple i relasjonen
- Antall blokker relasjonen okkuperer i lagringen (b_R)
- Antall tupler per blokk, som kalles blokkfaktoren (bfr)
- Antall distinkte verdier for et attributt ($\text{NDV}(A, R)$)
- Max og min verdi for et attributt ($\max(A, R)$ og $\min(A, R)$)

Disse blir oppdatert ofte for å være så nøyaktige som mulig, men ikke hele tiden siden det vil være svært krevende. Når optimalisatoren skal velge mellom betingelsene i en konjunktiv seleksjonsbetingelse, vil den ofte se på selektiviteten til hver betingelse. **Selektiviteten (sl) er forholdet mellom antall poster som tilfredsstiller betingelsen og totalt antall poster i filen**. Null selektivitet betyr at ingen poster tilfredsstiller betingelsen, mens en selektivitet på 1 betyr at alle postene tilfredsstiller betingelsen. **Det er en fraksjon som estimerer hvor stor andel av filen som blir hentet.**

Informasjonen i DBMS katalogen kan brukes for å finne estimater av selektivitetene. For en ekvivalensbetingelse på et nøkkelattributt vil $sl = 1/r_R$, mens for en ekvivalensbetingelse på et ikke-nøkkelattributt med i distinkte verdier vil $sl = 1/i$. Antall poster som tilfredsstiller betingelsen med selektivitet sl vil være $r_R * sl$. **Jo mindre dette estimatet er, desto bedre er betingelsen som brukes for å hente poster (få poster blir hentet).**

18.4 Implementasjon av JOIN operatoren

JOIN operasjonen er en av de mest tidskrevende operasjonene i query prosessering. Vi skal se på EQUIJOIN og NATURAL JOIN, for toveis join der to filer blir slått sammen. Vi har:

- OP6: Employee $\bowtie_{\mathrm{Dno=Dnumber}}$ Department
- OP7: Department $\bowtie_{\mathrm{Mgr_ssn=Ssn}}$ Employee

Følgende er metoder for å implementere join $R \bowtie_{A=B} S$:

- **J1: Nested-loop join** – standard algoritme siden den ikke krever at filene har spesielle aksessbaner. For hver post t i R (ytre loop) vil hver post s fra S hentes (indre loop) og det blir testet om disse to postene tilfredsstiller join betingelsen $t[A] = s[B]$
- **J2: indeksbasert nested-loop join** – brukes hvis en av de to join attributtene er en indeks. Metoden henter alle poster t i R og vil deretter bruke aksessbanen (indeks eller hashnøkkel) for å direkte hente matchende poster s fra S som tilfredsstiller $s[B] = t[A]$. Den sjekker altså om indeksen til S har verdi $t[A]$.
- **J3: sorter-fusjoner join** – mest effektiv og kan brukes når R og S er sortert/ordnet etter verdiene til hhv. A og B. Filene skannes samtidig og postene som har samme verdi for A og B blir slått sammen. Hvis filene ikke er sortert, må de først sorteres vha. for eksempel ekstern sortering. Sekundærindekser kan brukes på join attributtene, men da vil ikke postene være sorterte så dette kan være ineffektivt.
- **J4: partisjon-hash join** – postene i filene R og S blir partisjonert inn i mindre filer ved å bruke en hashfunksjon h på join-attributtene. Partisjoneringsfasen begynner med filen som har minst poster (eks: R), og postene med samme verdi for $h(A)$ vil hashe til samme hashbøtte i hashtabellen. I dette tilfellet antar vi at disse bøttene får plass i hovedminnet. I probefasen vil den samme hashfunksjonen brukes på postene i den andre filen, slik at poster med samme verdi for $h(B)$ hasher til samme hashbøtte. Metoden vil deretter søke (*probe*) gjennom bøtten for å se om posten fra S matcher noen av postene fra R.

Hvordan bufferrom og valg av ytre loop påvirker ytelsen til nested-loop join

Join algoritmene vil aksessere hele diskblokker hos filen, istedenfor individuelle poster. Antall blokker som leses fra filen vil avhenge av antall buffere i minnet, så derfor vil **tilgjengelig bufferrom ha stor effekt på flere av join algoritmene**. Vi ser på metode J1 for OP6, der vi antar at antall tilgjengelige buffere i hovedminnet er $n_B = 7$ blokker. Vi antar at hver buffer er av samme størrelse som en diskblokk. Department filen består av $r_D = 50$ poster som er lagret i $b_D = 10$ diskblokker, mens Employee filen består av $r_E = 6000$ poster som er lagret i $b_E = 2000$ diskblokker. Én bufferblokk må holde resultatet av join og én må brukes for å lese indre-loop filen, så derfor vil $n_B - 2$ bufferblokker være tilgjengelig for å lese ytre-loop filen. Algoritmen vil lese $n_B - 2$ blokker fra ytre-loop fil inn i hovedminnet. Deretter vil den lese alle blokkene fra indre-loop fil en om gangen, og for hver av disse vil den se om den finner matchende poster mellom de to filene som er i bufferen. Poster som matcher blir slått sammen og

lagt til resultatblokken som blir festet til resultatfilen på disken. Dette gjentas helt til alle blokkene fra ytre-loop fil har vært i bufferen.

Denne metoden kan redusere antall blokkaksesser og dermed bedre ytelsen til join, men det er vesentlig hvilken fil som velges for ytre-loop. Hvis b_y er antall blokker i ytre-loop fil og b_i er antall blokker i indre-loop fil, vil totalt antall blokkaksesser være:

$$b_y + b_i * \left(\frac{b_y}{n_B - 2}\right)$$

Alle blokkene fra ytre-loop fil må lastes inn i bufferen og blir dermed lest én gang (derav b_y). $b_y/(n_B - 2)$ er antall ganger blokker fra ytre-loop fil blir lastet inn i bufferen, og for hver av disse må alle blokkene i indre-loop fil leses (derav $b_i * b_y/(n_B - 2)$). For eksempel:

- Vi velger Employee som ytre-loop fil, slik at hver blokk i Employee blir lest én gang og hele Department filen blir lest en gang for hver gang $(n_B - 2)$ blokker av Employee blir lest inn i bufferen. Siden $b_y = 2000, b_i = 10, n_B = 7$ vil totalt antall blokkaksesser være $2000 + 10 * \left(\frac{2000}{7-2}\right) = 6000$.
- Vi velger Department som ytre-loop fil, slik at hver blokk i Department blir lest én gang og hele Employee filen blir lest en gang for hver gang $(n_B - 2)$ blokker av Department blir lest inn i bufferen. Siden $b_y = 10, b_i = 2000, n_B = 7$ vil totalt antall blokkaksesser være $10 + 2000 * \left(\frac{10}{7-2}\right) = 4010$.

Join algoritmen bruker en buffer for å holde resultatet av join. Når denne er full vil innholdet skrives til resultatfilen som er lagret på disken. **Hvis resultatfilen har b_{RES} diskblokker må hver av disse ha blitt skrevet en gang til disken, så derfor må b_{RES} blokkaksesser legges til resultatet for å estimere total kostnad til join operasjonen**. Eksempelet over illustrerer at **det er best å bruke filen med færre blokker som ytre-loop fil i nestet-loop join**.

Hvordan join seleksjonsfaktorer påvirker join ytelse

Join seleksjonsfaktoren til en fil er antall poster i filen som vil slås sammen med poster i en annen fil, og den vil kunne påvirke ytelsen til join (spesielt J2). Denne faktoren vil avhenge av join betingelsen mellom de to filene. For eksempel kan vi se på OP7 der hver Department post (50 stk.) blir slått sammen med én Employee post, og resterende 5950 Employee poster blir ikke slått sammen med noen Department poster.

Anta at sekundære indekser eksisterer for både Ssn i Employee og Mgr_ssn i Department. For å redusere søkeområdet kan indekser ha flere nivåer, slik at en blokk må aksesseres for hvert nivå i tillegg til blokken i datafilen. Vi antar at Ssn og Mgr_ssn er multilevel indekser med hhv. nivå $x_{\text{Ssn}} = 4$ og $x_{\text{Mgr_ssn}} = 2$. Det er to muligheter for å implementere J2:

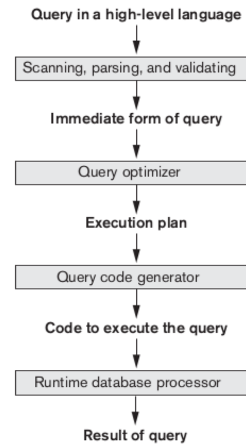
- Henter hver Employee post og bruker deretter Mgr_ssn for å finne matchende Department poster. Antall blokkaksesser er $b_E + \left(r_E\left(x_{\text{Mgr_ssn}} + 1\right)\right) = 2000 + (6000 * 3) = 20\,000$.
- Henter hver Department post og bruker deretter Ssn for å finne matchende Employee poster. Antall blokkaksesser er $b_D + \left(r_D\left(x_{\text{Ssn}} + 1\right)\right) = 10 + (50 * 5) = 260$.

Merk: $x_{\text{Mg_rssn}} + 1$ vil være antall blokker som må aksessere for å finne en Department post vha indeksen, mens r_E er antall Employee poster. $r_E\left(x_{\text{Mgr_ssn}} + 1\right)$ vil derfor være antall aksesser for å finne alle matchende poster mellom Employee og Department. Vi må legge til b_E siden alle blokkene til Employee må aksesseres for å hente postene.

Det andre alternativet er mer effektivt fordi join seleksjonsfaktoren til Department mht. join betingelsen Ssn = Mgr_ssn er 1 (alle poster i Department blir joined), mens join seleksjonsfaktoren til Employee mht. samme join betingelse er $50/6000 = 0.008$ (kun 0.8% av Employee poster blir joined). **For metode J2 bør den minste filen eller filen som har match for alle poster brukes som join-loopen**.

For J3 når begge filene er ordnet etter join-attributtene, vil antall blokkaksesser for join være lik summen av antall blokker i de to filene. I eksempelet over vil antall blokkaksesser være $b_D + b_E = 2010$. Hvis de ikke er sorterte, må de først sorteres, noe som øker kostnaden.

Oppsummering – kapittel 18 (F17)



Figuren viser hvordan en query blir prosessert, fra en SQL spørring i toppen til en utføringsplan som optimaliseres og utføres for å produsere et spørreresultat. Moderne systemer bruker en **kostnadsbasert optimalisator** som vil lage mange utføringsplaner og regner ut kostnaden til disse, for så å velge planen som har minst kostnad. Det finnes ingen standard for query prosessering, fordi det vil avhenge av mange faktorer. Utføringsplanen vil gi et algebrauttrykk i tillegg til informasjon om hvilken rekkefølge algebraen skal utføres, hvordan data skal aksesseres, osv.

Tre hovedteknikker som brukes for å utføre relasjonsalgebraoperasjoner er:

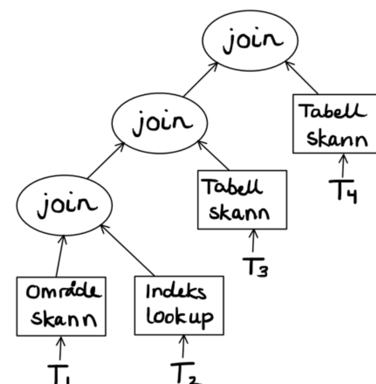
1. **Indeksering** – brukes for seleksjon og join, der WHERE-uttrykket blir brukt for å trekke ut små mengder poster. Datamengden som må hentes ut blir redusert.
2. **Iterasjon** – skanner over hele datamengden og finner det man er på utkikk etter. Det vil ofte være det beste når datamengden er liten.
3. **Partisjonering** – datamengden blir delt opp i mindre mengder som operasjonene blir brukt på. Det kalles også hashing, der input hasher til ulike bøtter.

For å kunne optimalisere og gjøre valg av aksessbaner, må DBMS ha tilgang til **statistikk om dataen**, som oppdateres etterhvert som databasen kjører. Denne statistikken ligger i DBMS katalogen, og den inkluderer blant annet antall rader og blokker for hver tabell, antall nøkler og blokker for hver indeks og trehøyde, lowkey, highkey og antall blokker for hvert B+-tre. For hver indeks kan det også inneholde histogrammer som beskriver fordelingen av indeksverdier. Denne statistikken vil ikke alltid være like nøyaktig, men prøver å gi et anslag.

Aksessbane

Aksessbane er måten DBMS får tak i data på, og optimalisatoren vil velge den billigste aksessbanen. Hvor god en aksessbane er måles i **antall blokker som aksesseres** (og hvor mye CPU som brukes). De vanligste aksessbanene er:

1. **Filskanning** – alle postene leses og de ønskede hentes ut
2. **Indeks** – kan skanne hele indeksen (eks: B+-tre), skanne et område eller søke opp en post med en bestemt indeksverdi.



Figuren viser et optimalisatortre som er resultatet av oversettelsen av en spørring. Her kan vi se at fire tabeller blir sendt inn i tre JOIN operatorer. Vi kan også se de ulike aksessbanene, for eksempel blir det gjort et områdeskann av T_1 og tabellskann av T_3 og T_4. Optimalisatoren velger hvilke aksessbaner som skal brukes og hvilken rekkefølge JOIN operasjonene skal utføres i. Dette er altså en utføringsplan, og hvilken plan som velges vil avgjøre ytelsen til utføringen av spørringen

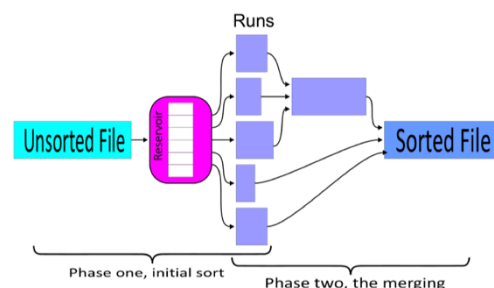
I dette kapittelet er det tre ting vi skal kunne regne på:

1. Flettesortering
2. Kostnaden av SELECT query
3. Kostnaden av JOIN med nested-loop

Flettesortering (merge-sort)

Flettesortering brukes for å sortere store mengder data, og det har to faser:

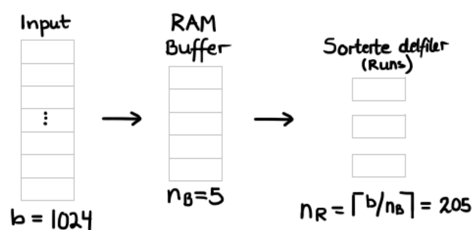
- **Initial sortering (partisjonering)** – en usortert fil sendes inn til et reservoar og det lages sorterte delfiler, som kalles *runs*. Delfilene leses inn i buffere i hovedminnet (RAM) der de blir sortert. Antall delfiler er n_R, antall blokker av data i filen er b og antall tilgjengelige buffere er n_B.
- **Fletting** – de sorterte delfilene (*runs*) blir flettet sammen til større delfiler som igjen flettes sammen. Til slutt vil vi få den sorterte filen. Delfilene blir flettet sammen i et såkalt pass. Det er kun delfilene som er i bufferen som vil bli flettet sammen, så derfor kan det hende det trengs flere pass. Flettepass er antall pass som trengs for å flette sammen hele filen, mens flettegrad (d_M) er antall delfiler som kan flettes sammen i ett pass.



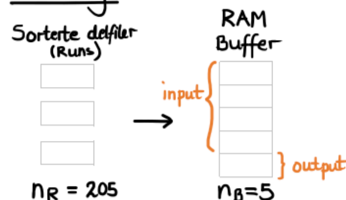
Kostnaden til flettesorteringen gis av total I/O, altså totalt antall blokker som leses eller skrives. Dersom b er antall blokker i filen, d_M er flettegraden og n_R er antall delfiler, vil antall blokkaksesser være gitt av:

$$\textbf{Total I/O} = 2b + 2b \log_{d_M}(n_R)$$

1) Initiell sortering



2) Fletting



Flettegrad:

$d_M = \min(n_R, n_B - 1) = 4$

Antall flettepass:

$FP = \lceil \log_{d_M}(n_R) \rceil$

$= \lceil \log_4(205) \rceil = 4$

Trenger 4 pass for å flette alle delfilene sammen

Total I/O = 1024 · 2 + 1024 · 2 · 4 = 10 240 blokker

(Read + write; FP; hele datamengden)

Kostnaden for SELECT query

På side 143 så vi på ulike måter å implementere en SELECT query, og de viktigste er:

- **S1: Lineær filskanning** – leser alle postene i filen og tester om de tilfredsstiller seleksjonsbetingelsen. Brukes når filen er liten
- **S5: Bruker B+-tre eller hashindeks (clustered indeks)** – regner ut hvor stort B+-treet er og hvor store deler av dette som må aksesseres for å utføre spørringen. Spørringen må matches opp mot det treet indekserer. Krever at seleksjonsbetingelsen er indeksert.
- **S6: Bruk av sekundærindeks med B+-tre** – en heapfil blir ofte brukt for å lagre tabellen og et B+-tre vil inneholde sekundærindeksene og pekere mot heapfilen. Brukes når det er få indeks lookups (dvs. få poster i B+-treet som tilfredsstiller betingelsen) eller spørringen er en indeks-only query der det er kun indeksverdien (eks: Lname) som hentes ut.

Figuren under viser et hvordan vi kan regne ut total I/O (dvs. antall blokkaksesser)

```
/*Henter ansatte med etternavn som begynner med A eller B*/
SELECT *
FROM Employee
WHERE Lname < 'C%';
```

- Antar at 10% av postene tilfredsstiller betingelsen
- 1024 blokker i heapfilen og 200 poster per blokk
  ⇒ totalt 204 800 poster

1) Filskanning

1024 blokkaksesser, siden alle blokkene må leses

2) Clustered B+-tre med søkenøkkel Lname ← seleksjonsbetingelsen er indeksert!

Antar høyde 3

1024 · 1,5 = 1536 blokker på nivå 0

Dette er prosentregning: det hele = delen / prosent

Her vil 1024 være delen av bladnivået som ikke er tom, og denne delen er 67% (=2/3) av det hele.

⇒ det hele = 1024 / (2/3) = 1024 · 3/2

Kostnaden på query:

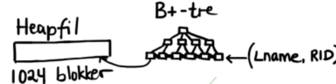
2 (nedover) + 154 (bortover) = 156 blokker

Høyde 3 — 10% matching, så vi starter på første match helt til venstre og leser 10% av 1536 blokker bortover til høyre

3) Heapfil + unclustered B+-tre

Heapfil — 1024 blokker

B+-tre



(Lname, RID)

Merk: Pekerne i det sorterte B+-treet vil peke mot en usortert heapfil, noe som betyr at poster som er i samme blokk i B+-treet ikke nødvendigvis vil være i samme blokk i heapfilen

- Antar at B+-tre er 20% av 2)

  20% · (1024 · 1,5) = 308 blokker på nivå 0

  2 (nedover) + 31 (bortover) = 33 blokker

  Høyde 3 — 10% matching — Dette er blokkene som må aksesseres for å hente Lname som ligger i B+-treet. MEN vi skal hente * og må derfor aksessere heapfilen

- Følg pekere fra hver post som kvalifiserer
- 10% av postene i B+-treet må leses for å finne pekeren til heapfilen:

  10% · 204 800 = 20480 mulige blokkaksesser ← En blokkaksess per peker

- Total I/O:

  Inntil: 33 + 20480 = 20513 blokker

Eksempel – seleksjonsoppgave

Vi har tabellen Student(studno, lastname, firstname, email, startyear). Anta at tabellen er lagret i en heapfil med 1000 blokker der hver blokk får plass til 20 studentposter. Videre er det laget et unclustered B+-tre-indeks på lastname. Vi antar at B+-treet har 500 blokker på løvnivå og har høyde 3. Finn antall blokkaksesser (leses+skrives) ved følgende SQL-setninger:

1. **INSERT INTO Student VALUES (12123, 'Hansen', 'Hans', 'hans@email.org', 2013)**
   Vi må sette inn en ny post i heapfilen, noe som krever at vi leser den siste blokken og skriver inn den nye posten. Dette krever altså 2 blokkaksesser. Vi må også legge til en ny indeks i B+-treet, noe som krever at vi leser tre blokker (høyden) og skriver inn den nye indeksposten. Dette krever altså 4 blokkaksesser. Totalt antall blokkaksesser blir 6 I/O blokker.

Merk: her har vi antatt at innsetting av ny post i B+-treet ikke utløser en blokksplitt, fordi da vil vi få flere skrivinger

2. **SELECT lastname, firstname, email, startyear FROM Student WHERE lastname = Hansen**
   Vi begynner med å lese B+-treet siden det indekserer lastname og det trengs 3 blokkaksesser for å nå blokken der en post som oppfyller betingelsen befinner seg. Siden lastname ikke er et nøkkelattributt, kan det være flere poster som tilfredsstiller betingelsen og hvis det er mange kan det hende vi må lese etterfølgende blokker (får ikke plass i første blokk). I heapfilen vil vi få en read per post i B+-treet som har lastname = Hansen.

3. **SELECT * FROM Student**
   Siden vi skal hente alle Studentpostene i ingen bestemt rekkefølge er det ingen vits å bruke B+-treet, siden det er mer effektivt å skanne hele heapfilen. Dette krever 1000 blokkaksesser.

4. **SELECT DISTINCT lastname FROM Student ORDER BY lastname**
   Dette er et indeks-only query, siden det er kun verdien som er indeksert i B+-treet som skal hentes og resultatet skal være sortert. Derfor vil vi skanne hele B+-treet, noe som krever $2(\text{nedover}) + 500(\text{bortover}) = 502\ I/O$ blokker.

5. **Hvis den samme tabellen hadde fått kravet om at studno skal være PRIMARY KEY, hvordan ville du ha lagret tabellen?**
   For å sikre at et attributt er unikt må vi lage en indeks på studno, for eksempel kan vi lage et clustered B+-tre med studno som indeks.

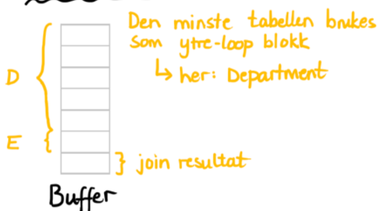
Metoder for utføring av join

På side 144 så vi på ulike måter å implementere en SELECT query, og den viktigste er:

- **J1: Nested-loop join** – for hver blokk i den ene tabellen vil hele den andre tabellen skannes for å se etter matchende poster (dvs. poster som oppfyller join betingelsen). Bufferen fylles med så mange blokker det er plass til fra den ene tabellen, også skannes hele den andre tabellen for å se etter matcher som blir lagt til resultatblokken i bufferen. Dette er standard join algoritme, siden den ikke krever at filene har spesielle aksessbaner.

- Department: 500 poster lagret i 10 diskblokker
- Employee: 60 000 poster lagret i 2000 diskblokker
- Minne/RAM: 7 blokker

J1: Nested loop



For hver lesing i bufferen må 5 Department blokker og alle Employee blokker leses

I/O (lesing) = 2 · (5 + 2000) = 4010

Siden 10 Department blokker skal leses inn i 5 bufferblokker trengs det 2 lesinger inn i bufferen

⇒ Total I/O = 4010 + b_{res}

I/O (skriving) = b_{RES}

Antall diskblokker i resultatfila

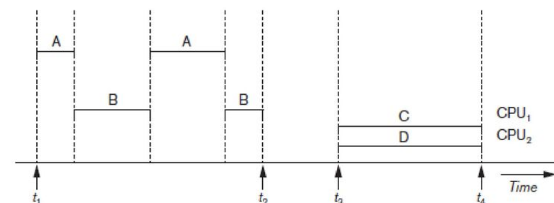
Del 6 – Transaksjonsprosessering, Samtidighetskontroll og gjenoppretting

Kapittel 20 – Introduksjon til transaksjonsprosessering

Transaksjon prosesseringssystem er systemer med store databaser og hundrevis av brukere som samtidig utfører databasetransaksjoner (dvs. lesing/skriving av databasen). For eksempel kan det være flyselskapsreservasjoner, aksjemarkedet, osv. Disse systemene krever høy tilgjengelighet og rask responstid. Dette kapittelet ser på konseptene som trengs i slike system og vil definere transaksjoner som representerer en logisk enhet av databaseprosessering som må fullføres for å sikre korrekthet. En transaksjon vil som regel implementeres av et dataprogram som inkluderer databasekommandoer som henting, innsetting, sletting og oppdatering.

Introduksjon til transaksjonsprosessering

Vi skal nå se på konseptet for samtidig (*concurrent*) utføring av transaksjoner og gjenoppretting fra transaksjonsfeil.



Enkeltbruker vs. flerbruker system

Et databasesystem (DBMS) kan klassifiseres etter antall brukere som kan bruke systemet samtidig, og vi skiller mellom enkelbruker og flerbruker. Enkelbruker DBMS er som regel begrenset til personlige datamaskiner, mens de fleste andre DBMS er flerbruker. For at flere skal kunne bruke databasen samtidig, brukes det to ulike prosesseringsteknikker:

- **Flettet prosessering (*interleaved*)** – basert på **multiprogrammering** som gjør at flere brukere kan aksessere databasen samtidig, siden det **lar operativsystemet utføre flere prosesser samtidig**. En enkel CPU kan utføre én prosess om gangen, men et multiprogram operativsystem kan utføre noen kommandoer fra en prosess, utsette denne prosessen, utføre noen kommandoer fra neste prosess, osv. Samtidig utføring av prosesser er derfor flettet sammen, som vi kan se for prosess A og B på figuren. Fletting lar CPU være opptatt når en prosess krever input eller output operasjoner (eks: lese blokk fra disk). **CPU blir satt til å utføre en annen prosess istedenfor å vente på I/O operasjonen**. Dette hindrer at en lang prosess forsinker andre prosesser
- **Parallell prosessering** – brukes når datasystemet har flere CPUs, for da kan disse brukes for å utføre prosesser samtidig, for eksempel prosess C og D på figuren.

Det meste av teorien for samtidighetskontroll i databaser er utviklet for flettet prosessering.

Transaksjoner, databaseenheter, lese og skriveoperasjoner og DBMS buffere

En transaksjon er et utførende program som utgjør en logisk enhet av database-prosessering, og den kan inkludere en eller flere database aksessoperasjoner (eks: innsetting og sletting). Operasjonene som lager en transaksjon kan være en del av et applikasjonsprogram eller spesifiseres i et spørrespråk som for eksempel SQL. Et applikasjonsprogram kan inneholde flere transaksjoner, og hver transaksjon har grenser som markerer start og slutt. En read-only transaksjon vil kun hente data, mens en read-write transaksjon vil både hente og oppdatere data.

En databasemodell blir brukt for å presentere transaksjonsprosessen. **Databasen blir representert som en samling av navngitte databaseelementer**, og størrelsen til dataelementene kalles granulariteten. **Et dataelement kan være en post, en diskblokk eller individuelle felt (attributter) hos noen poster.** Konseptene i transaksjonsprosessen er uavhengig av granulariteten og vil gjelde generelt for alle dataelement. Navnet til dataelementet vil være unikt, for eksempel for en diskblokk kan det være adressen, mens for en post kan det være RID (Record ID). **Aksessoperasjonene kan være på formen read_item(X) som vil lese databaseelementet med navn X, eller write_item(X) som vil skrive en verdi inn i databaseelementet med navn X**. Grunnleggende enhet for dataoverføring fra disken til hovedminnet er en diskblokk, så **utførelsen av read_item(X) inkluderer følgende steg**:

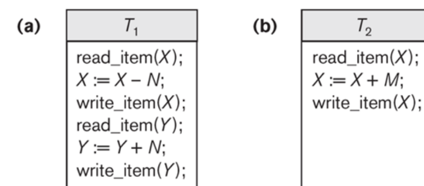
1. Finn adressen til diskblokken som inneholder X
2. Kopier diskblokken inn i bufferen i hovedminnet hvis den ikke allerede ligger i minnet. Størrelsen til bufferen er lik størrelsen til diskblokken
3. Kopier element X fra bufferen til programvariabelen som heter X

Utførelse av write_item(X) inkluderer følgende steg:

1. Finn adressen til diskblokken som inneholder X
2. Kopier diskblokken inn i bufferen i hovedminnet hvis den ikke allerede ligger i minnet.
3. Kopier element X fra programvariabelen X inn i riktig lokasjon i bufferen og lagre den oppdaterte diskblokken fra bufferen tilbake til disken.

Noen ganger vil ikke bufferen umiddelbart lagres på disken, fordi det er flere endringer som skal gjøres. Som regel vil gjenoppretteren i DBMS og operativsystemet bestemme når en endret diskblokk i bufferen skal lagres. DBMS vil kontrollere en **database cache** med et antall databuffere i hovedminnet, der hver buffer rommer en diskblokk fra databasen. Når bufferne er opptatte og en diskblokk må leses inn i minnet, vil **buffer erstatningspolitikk** brukes for å velge hvilken buffer som skal erstattes. For eksempel vil LRU (Least Recently Used) velge bufferen som det er lengst siden ble brukt. Hvis bufferen som erstattes har blitt endret må den først skrives tilbake til disken.

En transaksjon vil inkludere read_item(X) og write_item(X) operasjoner, og figuren viser to enkle transaksjoner. Read-set og write-set er settet av alle elementer som transaksjonen hhv. leser og skriver (eks: $\{X, Y\}$ på figuren).



Samtidighetskontroll og hvorfor det trengs

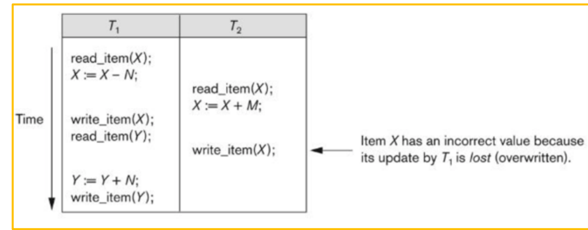
Transaksjoner som innsendt av ulike brukere kan utføres samtidig og dermed aksessere og oppdatere samme databaseelementer. **Hvis denne samtidige utførelsen ikke er kontrollert, kan det føre til flere problemer**. Vi skal se på disse vha en database for flyselskapsreservasjoner, der en post lagres for hver flygning. Hver post inkluderer antall reserverte seter som et databaseelement med unikt navn. Vi har tre transaksjoner:

- T_1: N reservasjoner blir overfør fra en flygning der antall reserverte seter er lagret i databaseelement X til et en annen flygning der antall reserverte seter er lagret i databaseelement Y.
- T_2: M seter blir reservert på første flygning X som det refereres til i transaksjon T_1
- T_3: regner ut totalt antall reservasjoner for alle flygninger

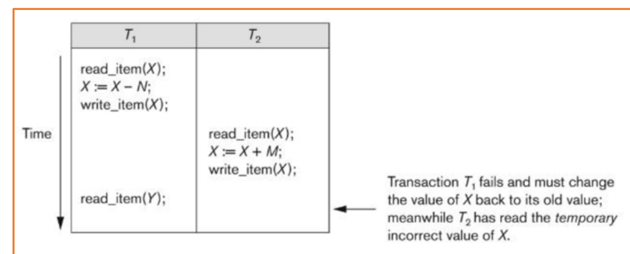
Samtidighetskontroll er viktig for å kontrollere operasjoner som utføres samtidig, slik at vi unngår disse problemene!

Følgende er problem som kan oppstå hvis transaksjonene kan kjøre samtidig:

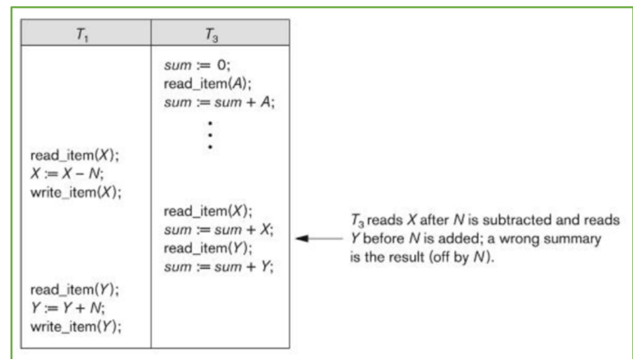
- **Tapt oppdatering** – oppstår når to transaksjoner har operasjoner som flettes sammen på en slik måte at verdien til noen databaseelementer blir ukorrekt. På figuren ser vi at T_1 og T_2 blir sendt inn samtidig og deres operasjoner flettes sammen. Den endelige verdien til X vil være feil, siden T_2 leser verdien til X før T_1 endrer den. **Oppdateringen av X som følge av T_1 bli tapt.** For eksempel hvis det er $X = 80$ reservasjoner ved starten, T_1 vil overføre $N = 5$ reservasjoner til Y og T_2 vil reservere $M = 4$ seter på X, skal antall reserverte seter ved slutten være $X = 79$. Flettingen av operasjoner vil likevel føre til at $X = 84$, fordi oppdateringen i T_1 som fjernet 4 seter blir tapt. read_item(X) i T_2 vil hente den originale verdien til X og ikke den oppdaterte verdien etter T_1, så denne oppdateringen går tapt.



- **Midlertidig oppdatering (*dirty read*)** – oppstår når en transaksjon oppdaterer et databaseelement og deretter feiler, og en annen transaksjon aksesserer det oppdaterte elementet før det endres tilbake til sin originale verdi. På figuren ser vi T_1 som oppdaterer X og deretter feiler før den er ferdig, slik at X vil endres tilbake til sin opprinnelige verdi. Før dette blir gjort vil T_2 lese den midlertidige verdien til X, som kalles *dirty data*, siden den ble laget av en transaksjon som ikke ble fullført.



- **Ukorrekt summering** – oppstår hvis en transaksjon regner ut en aggregerende summeringsfunksjon på et antall databaseelementer, samtidig som en annen transaksjon oppdaterer noen av disse. Aggregatfunksjonen kan bruke noen verdier før de blir oppdatert og andre etter de blir oppdatert. På figuren ser vi T_3 som summerer antall reservasjoner og vil få en feil på N, siden den leser X etter N reservasjoner har blitt trukket fra og leser Y før N reserverasjoner har blitt lagt til.



- **Urepeterbar lesing** – oppstår når en transaksjon leser samme element to ganger og elementet endres av en annen transaksjon mellom de to lesingene. Transaksjonen vil dermed motta ulike verdier for to lesinger av samme verdi, uten å selv ha endret verdien.

Hvorfor gjenoppretting trengs

Når en transaksjon sendes til DBMS er systemet ansvarlig for å sikre at alle operasjonene blir fullført og at endringene blir registrert i databasen, eller at transaksjonen har ingen effekt på databasen eller noen andre transaksjoner. I det første tilfellet vil transaksjonen være **committed**, mens i det andre tilfellet vil den være **abortert**. *Hele* transaksjonen er en logisk enhet for databaseprosessering, så derfor kan ikke noen deler utføres, mens andre ikke! Hvis en transaksjon feiler før alle operasjonene er utført, må operasjonene som ble utført angres slik at de får ingen varende effekt.

Feil blir generelt klassifisert som transaksjon-, system- eller mediafeil. Noen grunner til at transaksjoner feiler i midten av utførelsen er:

1. **Datafeil (systemkræsj)** – en maskinvare, programvare eller nettverksfeil oppstår i datamaskinen ila transaksjonsutførelsen.
2. **Transaksjon eller systemfeil** – en operasjon i transaksjonen kan gjøre at den feiler, for eksempel deling på 0. Det kan også skyldes feil i parameterverdier, logiske programmeringsfeil (bugs) eller at brukeren forstyrrer transaksjonsutførelsen.
3. **Lokale feil eller unntaksforhold som detekteres av transaksjonen** – i løpet av utførelsen oppstår det situasjoner som gjør at transaksjonen må avbrytes, for eksempel at data for transaksjonen ikke blir funnet. Kan også skyldes en unntaksbetingelse, for eksempel kan manglende penger på konto avbryte et uttak av penger.
4. **Samtidighetskontroll** – en transaksjon kan aborteres som en del av samtidighetskontroll
5. **Diskfeil** – noen diskblokker kan miste data pga feil ila lesing eller skriving av blokkene
6. **Fysiske problemer** – inkluderer strømfeil, brann, tyveri, osv.

Disse grupperes i to typer gjenoppretting: **en transaksjon rulles tilbake** (1-4) eller **systemkrasjgjenoppretting** (5-6). 1-4 er mer vanlig og håndteres av gjenoppretteren.

Transaksjon og systemkonsepter

Vi skal nå se på flere konsepter som er relevant for transaksjonsprosessering

Transaksjonstilstander og flere operasjoner

En transaksjon er en atomisk enhet med arbeid som enten skal fullføres fullstendig eller ikke utføres i det hele tatt. Gjenoppretteren til DBMS vil holde styr over når transaksjoner starter, terminerer og committer eller aborterer, gjennom operasjonene:

- **BEGIN_TRANSACTION** – markerer starten på transaksjonsutførelsen
- **READ eller WRITE** – spesifiserer lese eller skriveoperasjoner på databaseelementer
- **END_TRANSACTION** – markerer enden på transaksjonsutførelsen. Det kan sjekkes om endringene introdusert i transaksjonen kan committes eller om transaksjonen må aborteres.
- **COMMIT_TRANSACTION** – signaliserer en suksessfull slutt på transaksjonen, slik at alle endringer utført av transaksjonen blir committed til databasen.
- **ROLLBACK (eller ABORT)** – signaliserer en ikke-suksessfull slutt på transaksjonen, slik at alle endringer utført av transaksjonen blir angret.

Figuren viser et tilstandsdiagram der vi kan se hvordan transaksjonen beveger seg gjennom utførelsestilstandene. Med en gang utførelsen starter vil transaksjonen entre en **aktiv tilstand**, der den kan utføre READ og WRITE operasjoner. Når transaksjonen ender vil den gå over til en **delvis committed tilstand**, der det blir sjekket om transaksjonen kan committes eller ikke. Hvis disse sjekkene er suksessfulle vil transaksjonen nå committ punktet og entrer **committed tilstand**. Når transaksjonen har committet vil den ha blitt utført suksessfullt og alle endringene har blitt permanent registrert i databasen. Hvis sjekkene feiler eller transaksjonen aborteres ila den aktive tilstanden, vil transaksjonen entre **feilet tilstand**. Effekten av alle WRITE operasjoner må angres, transaksjonsinformasjon slettes og transaksjonen entrer **terminert tilstand**, der den forlater systemet. Aborterte transaksjoner kan startes på nytt senere, enten automatisk eller fordi de sendes på nytt av brukeren.

Systemloggen

For å kunne gjenopprette fra feil som påvirker transaksjoner, vil systemet opprettholde en logg som holder styr over alle transaksjonsoperasjoner som påvirker verdien til databaseelementer (eks: WRITE) og annen transaksjonsinformasjon som kan være nødvendig for gjenoppretting. Denne loggen er en sekvensiell og *append-only* fil som lagres i disken, slik at den ikke påvirkes av noen typer feil bortsett fra disk og fysiske feil. Den siste delen av loggfilen vil være lagret i logbuffere i hovedminnet, og når disse blir fulle vil innholdet festes til enden av loggfilen på disken. Følgende er logposter som blir skrevet til loggfilen og handlinger som korresponderer til hver loggpost (T er en unik transaksjonsid som lages for hver transaksjon av systemet):

1. $[\textbf{start_transaction}, \textbf{T}]$ – gir at transaksjon T har startet utførelsen
2. $[\textbf{write_item}, \textbf{T}, \textbf{old_value}, \textbf{new_value}]$ – gir at transaksjon T har endret verdien til databaseelement X fra old_value til new_value
3. $[\textbf{read_item}, \textbf{T}, \textbf{X}]$ – gir at transaksjon T har lest verdien til databaseelement X
4. $[\textbf{commit}, \textbf{T}]$ – gir at transaksjon T har blitt utført suksessfullt og bekrefter at effektene kan bli committed til databasen
5. $[\textbf{abort}, \textbf{T}]$ – gir at transaksjon T har blitt abortert

READ operasjoner vil ikke alltid inkluderes i loggfiler (kan gis hvis filen skal holde styr over alle operasjoner). Vi antar at alle permanente endringer i databasen skjer innenfor transaksjoner, så gjenoppretting fra en transaksjonsfeil vil gå ut på å enten angre (*undo*) eller gjenta (*redo*) operasjonene i loggen (mer i kapittel 22).

Committ punkt i transaksjonen

En transaksjon vil nå committ punktet når alle dens operasjoner er utført suksessfullt og effekten av disse er registrert i loggen. **Forbi dette punktet vil transaksjonen være committet**. Committ posten $[\text{commit}, \text{T}]$ blir skrevet inn i loggen. Når systemet krasjer vil vi søke gjennom loggen etter transaksjoner som har startet, men ikke blitt committet, og i løpet av gjenopprettingen kan disse rulles tilbake for å angre deres effekt på databasen. For transaksjoner som har skrevet committ posten, må WRITE operasjoner være registrert i loggen, slik at deres effekt på databasen kan gjentas (merk: selv om transaksjonen er committet kan det hende at endringene ikke har blitt skrevet til disken enda).

Loggfilen ligger på disken, og de nyeste loggpostene vil ligge i loggbufferen helt til den blir full og innholdet festes på loggfilen. Dette vil føre til mindre skriving til disken, men hvis systemet krasjer kan hovedminnet og dermed loggbufferen bli tapt. **Før transaksjonen når committ punktet vil derfor alle deler av loggen som ligger i bufferen tvangsskrives til disken**.

DBMS-spesifikk erstatningspolitikk for buffere

DBMS cache vil holde diskblokker med informasjon som blir prosessert i databuffere. Hvis alle databufferne er fulle og nye diskblokker må leses inn, vil buffererstatningspolitikk brukes for å velge hvilke buffere som skal erstattes. Noen eksempler på dette er:

- **Domeneseperasjon (DS) metode** – DBMS cache deles inn i separate domener (eks: data, indeks, loggfil, osv.) som hver har et sett med buffere. Hvert domene håndterer en type

diskblokker og blokkerstatning innenfor hvert domene følger LRU (Least Recently Used). Siden antall buffere for hvert domene er forhåndsbestemt, vil denne metoden være statisk og vil ikke kunne tilpasse seg endringer i lasten. Det finnes varianter (eks: gruppering av domenene) som gjør metoden mer dynamisk.

- **Hot set metoden** – brukes for spørringer som må gjentatt skanne et sett med blokker, for eksempel nested-loop join. Diskblokkene som gjentatt skannes (indre loop) blir lastet inn i bufferne og vil ikke erstattes før prosessen er ferdig.
- **DBMIN metoden** – vil på forhånd estimerer antall buffere som trengs for hver fil involvert i operasjonen og tildeler passende antall buffere til hver fil involvert i spørringen

Ønskelige egenskaper ved transaksjoner

De ønskede egenskapene til transaksjoner blir ofte kalt ACID egenskaper, og samtidighetskontroll og gjenopprettingsmetoder i DBMS bør sørge for at disse oppnås:

- **Atomisk egenskap** – en transaksjon er en atomisk prosesseringsenhet, altså må den enten fullføres fullstendig eller ikke i det hele tatt.
- **Konsistensbevaring** – en transaksjon skal bevare konsistensen til databasen, altså hvis den fullføres fullstendig uten forstyrrelser av andre transaksjoner, skal den ta databasen fra en konsistent tilstand til en annen.
- **Isolering** – en transaksjon skal oppføre seg som om den blir utført isolert fra andre transaksjoner, selv om mange transaksjoner utføres samtidig. Utførelsen av en transaksjon skal ikke forstyrres av andre transaksjoner som utføres samtidig.
- **Durabilitet eller permanent** – endringene som påføres databasen av en committet transaksjon skal være permanente. Disse endringene kan ikke tapes pga. en feil

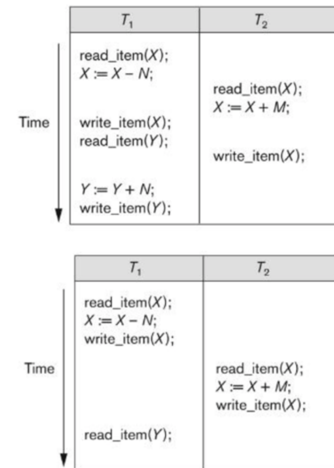
Ansvaret for den atomiske egenskapen ligger på gjenopprettingssystemet i DBMS. For aborterte transaksjoner må systemet angre effekten som transaksjonene har hatt på databasen, mens for committet transaksjoner må effektene skrives til disken. Ansvaret for konsistensbevaring ligger på programmererne av databasen og DBMS som opprettholder integritetsbegrensninger. En konsistent databasetilstand vil tilfredsstille begrensningene som er spesifisert i skjemaet eller generelt gjelder for en database, og et databaseprogram bør skrives slik at dette er tilfellet før og etter en fullført transaksjon uten forstyrrelser. Ansvaret for isoleringen ligger på systemet for samtidighetskontroll i DBMS (kapittel 21), mens ansvaret for durabilitet ligger på gjenopprettingssystemet (kapittel 22).

Karakterisering av planer basert på gjenopprettbarhet

Når transaksjoner blir utført samtidig med fletting av operasjoner, vil rekkefølgen til utføringen av operasjonene kalles planen (eller historien).

Transaksjonsplaner (-historier)

En plan (eller historie) S av n transaksjoner er en ordning av operasjonene i transaksjonene. Operasjoner fra ulike transaksjoner kan være flettet sammen i S. For hver transaksjon må operasjonene være i samme rekkefølge i S som de er i transaksjonen. Planen S har en total orden, som vil si at for to operasjoner må en av de komme før den andre.



Vi er først og fremst opptatt av operasjonene read_item (r), write_item (w), committ(c) og abort (a). I tillegg har vi begin_transaction (b) og end_transaction (e). **Transaksjonsid legges til hver operasjon** (eks: r_1 for lesing i transaksjon 1), for å holde styr over hvilken transaksjon operasjonene hører til. Vi vil også gi **hvilken databaseelement som aksesseres** (eks: $r_1(X)$ for lesing av databaseelement X). Vha denne notasjonen kan vi skrive historiene for transaksjonene vi ser på de to figurene:

$$S_a: r_1(X);\ r_2(X);\ w_1(X);\ r_1(Y);\ w_2(X);\ w_1(Y)$$
$$S_b: r_1(X);\ w_1(X);\ r_2(X);\ w_2(X);\ r_1(Y); a_1$$

Operasjoner i konflikt

To operasjoner i en historie sies å være i konflikt, hvis de tilfredsstiller alle følgende krav:

1. **De hører til ulike transaksjoner**
2. **De aksesserer samme element X**
3. **Minst én av operasjonene er en write_item(X)**

For eksempel for S_a vil det være konflikt hos operasjonene $r_1(X)$ og $w_2(X)$, operasjonene $r_2(X)$ og $w_1(X)$ og operasjonene $w_1(X)$ og $w_2(X)$ fordi disse oppfyller alle tre kravene. Det vil ikke være konflikt hos operasjonene $r_1(X)$ og $r_2(X)$ siden de begge er leseoperasjoner, $w_2(X)$ og $w_1(Y)$ siden de opererer på ulike elementer og $w_1(X)$ og $r_1(X)$ siden de tilhører samme transaksjon.

To operasjoner vil være i konflikt hvis det å endre på rekkefølgen deres vil gi et annet resultat. Dette blir synlig i to ulike typer konflikter:

- **Read-write konflikt** – hvis vi endrer $r_1(X); w_2(X)$ til $w_2(X); r_1(X)$ vil verdien til X som leses av T_1 være endret. I første orden leser vi før verdien endres, mens i andre orden leser vi etter verdien er endret.
- **Write-write konflikt** – hvis vi endrer $w_1(X); w_2(X)$ til $w_2(X); w_1(X)$ vil den siste verdien til X være endret. I første orden vil transaksjon 2 skrive X, mens i andre orden vil transaksjon 1 skrive X.

To lesinger vil ikke være i konflikt, fordi hvis vi endrer på deres rekkefølge vil utfallet fortsatt være det samme.

En historie vil være fullstendig hvis den inneholder alle operasjonene til transaksjonene (inkludert committ eller abort), operasjonene hos en transaksjon har samme rekkefølge i historien og for operasjoner i konflikt må én av de komme før den andre. For operasjoner som ikke er i konflikt er det ikke nødvendig å definere hvilken som skjer først i historien (= delvis orden). For operasjoner i konflikt og operasjoner som hører til samme transaksjon, må derimot rekkefølgen være bestemt (= full orden). Historier vil sjeldent være fullstendige fordi nye transaksjoner blir kontinuerlig lagt til.

Karakterisering av historier basert på gjenopprettbarhet

For noen historier er det lett å gjenopprette fra transaksjonsfeil, mens for andre kan det være mer komplisert eller til og med umulig. Historier kan derfor karakteriseres basert på hvilken gjenoppretting som er mulig.

Merk: det er først når transaksjonene blir committet at verdien som er skrevet inn blir lagret permanent i databasen og dermed blir gyldig.

Vi skal nå se på ulike karakteriseringer av historier, som blir stadig strengere.

Gjenopprettbar historie

En historie er gjenopprettbar hvis hver transaksjon committer etter at transaksjoner de har lest fra har committet. Hvis historien inneholder $w_1(X) \dots r_2(X)$ må vi sjekke at $c_1 \dots c_2$, siden transaksjon 2 leser fra transaksjon 1. **Dette gjør at committet transaksjoner ikke må aborteres, noe som sikrer durabilitet.** Merk: vi sier at en transaksjon T_1 *leser fra* transaksjon T_2 hvis $w_2(X)$ kommer før $r_1(X)$, T_2 er ikke abortert før $r_1(X)$ og ingen andre transaksjoner skriver X etter $w_2(X)$ og før $r_1(X)$ (med mindre de aborteres før $r_1(X)$).

Hvis loggfilen inneholder tilstrekkelig informasjon kan det lages en gjenopprettingsalgoritme for alle gjenopprettbare historier. Det motsatte kalles **ikke-gjenopprettbar** og bør ikke tillates av DBMS. Noen eksempler på gjenopprettbare og ikke-gjenopprettbare historier:

- $S_a'\colon r_1(X);\ r_2(X);\ w_1(X);\ r_1(Y);\ w_2(X); c_2;\ w_1(Y); c_1$
  Gjenopprettbar – den lider av tapt-oppdateringsproblemet (løses med serialiserbarhet), men den er gjenopprettbar siden den oppfyller definisjonen over.
- $S_c\colon r_1(X);\ w_1(X);\ r_2(X);\ r_1(Y); w_2(X); c_2; a_1;$
  Ikke-gjenopprettbar – T_2 leser X fra T_1, men T_2 committer før T_1. Problemet oppstår hvis T_1 aborterer etter c_2 operasjonen, fordi da vil verdien til X som T_2 leser ikke lenger være gyldig. Dermed må T_2 aborteres (rulles tilbake) etter den har blitt committet, slik at historien blir ikke-gjenopprettbar.
- $S_d\colon r_1(X);\ w_1(X);\ r_2(X);\ r_1(Y); w_2(X); w_1(Y); c_1; c_2;$
  Gjenopprettbar – c_2 er flyttet etter c_1, slik at hvis T_1 aborterer vil også T_2 abortere, og dette skjer før de committer: $r_1(X);\ w_1(X);\ r_2(X);\ r_1(Y); w_2(X); w_1(Y); a_1; a_2;$

ACA (Avoid Cascading Abort) historie

En historie er ACA hvis hver transaksjon kun leser verdier som er skrevet av committet transaksjoner. Hvis historien inneholder $w_1(X) \dots r_2(X)$ må vi sjekke at $c_1 \dots r_2(X)$, altså at transaksjon 1 committes før transaksjon 2 leser fra den.

En gjenopprettbar historie tillater *cascading abort*, der ikke-committet transaksjoner må aborteres fordi de leser et element fra en transaksjon som har feilet. Dette kan være tidskrevende, siden det kan føre til at mange transaksjoner vil aborteres. Derfor har ACA historier blitt definert for å garantere at dette ikke skjer. Noen eksempler:

- $S_d\colon r_1(X);\ w_1(X);\ r_2(X);\ r_1(Y); w_2(X); w_1(Y); c_1; c_2;$
  Ikke-ACA – T_2 leser fra T_1 før T_1 har committet. Problemet oppstår hvis T_1 aborterer, fordi da må også T_2 aborteres, siden den har lest et element fra en transaksjon som feilet.
- $S_e\colon r_1(X);\ w_1(X);\ r_1(Y);\ w_1(Y);\ c_1;\ r_2(X);\ w_2(X);\ c_2;$
  ACA - $r_2(X)$ er flyttet etter c_1, slik at transaksjon 2 vil lese fra en committet transaksjon 1. Legg merke til at vi også må flytte $w_2(X)$ fordi rekkefølgen til operasjonene i en bestemt transaksjon må være den samme i historien! Transaksjon 2 blir derfor forsinket, men vi unngår *cascading abort*. Legg også merke til at vi ikke må flytte $r_1(X)$ og $r_1(Y)$ fordi disse leser ikke av en annen transaksjon, men leser verdiene til X og Y som er lagret fra før av.

Merk: ACA unngår **galopperende abort**, der abort av en transaksjon (den som skriver enheten) fører til at en annen transaksjon (den som leser enheten) også må aborteres. Ved ikke-gjenopprettbare historier må committet transaksjoner rulles tilbake, mens ved ikke-ACA må ikke-committet transaksjoner rulles tilbake.

Strikt historie

En historie er strikt når transaksjoner ikke kan lese eller skrive ikke-committet verdier. En transaksjon kan ikke lese eller skrive element X før siste transaksjon som skrev X har committet eller abortert. Hvis historien inneholder $w_1(X) \dots r_2/w_2(X)$ må vi sjekke at $c_1 \dots r_2/w_2(X)$, altså at transaksjon 2 kun leser og skriver committet verdier. Noen eksempler:

- $\underline{S_d: r_1(X);\ w_1(X);\ r_2(X);\ r_1(Y); w_2(X); w_1(Y); c_1; c_2;}$
  Ikke-strikt – T_2 leser og skriver X før T_1 som sist skrev X har blitt committet. Problemet oppstår når T_1 aborterer og verdien til X skal gjenopprettes, siden dette kan krasje med at en annen transaksjon leser og skriver X
- $\underline{S_e: r_1(X);\ w_1(X);\ r_1(Y);\ w_1(Y);\ c_1;\ r_2(X);\ w_2(X);\ c_2;}$
  Strikt – transaksjonene leser og skriver kun committet verdier.

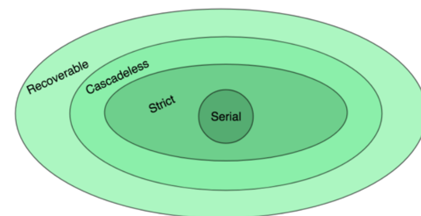
Strikte historier vil gjøre gjenopprettingen enklere. For eksempel for å angre en write_item(X) operasjon hos en abortert transaksjon, må vi kun gjenopprette ***before image*** (old_value) til dataelement X. Ingen andre transaksjoner må aborteres, siden ingen har lest X etter endringen. Dette kalles *undo recovery* av *before image* fra loggen, og vil alltid fungere for strikte historier. For gjenopprettbare eller ACA historier kan det hende den feiler. For eksempel for $w_1(X,5); w_2(X,8); a_1$ der X opprinnelig var 9 (*before image*), vil abortering av T_1 føre til at X settes lik 9, noe som blir feil, siden T_2 forventer at X skal ha verdien 8. Denne historien er ikke strikt, siden T_2 får skrive element X, selv om T_1 som sist skrev X ikke har committet.

Oppsummert

Vi har følgende karakteriseringer av historier:

- **Gjenopprettbar** – for $w_1(X) \dots r_2(X)$ må $c_1 \dots c_2$, altså transaksjoner må committes etter transaksjoner de har lest fra har committet
- **ACA** – for $w_1(X) \dots r_2(X)$ må $c_1 \dots r_2(X)$, altså transaksjoner kan kun lese fra committet transaksjoner
- **Strikt** – for $w_1(X) \dots r_2/w_2(X)$ må $c_1 \dots r_2/w_2(X)$, altså transaksjoner kan ikke lese eller skrive ikke-committet verdier.

Merk: Strikt $\subset$ ACA $\subset$ Gjenopprettbar $\subset$ Alle historier, altså alle strikte historier er ACA, og alle ACA historier er gjenopprettbare. Hvis vi ser at en historie er strikt, vil vi derfor også vite at den er gjenopprettbar og ACA. En historie kan være gjenopprettbar uten å være ACA eller strikt.

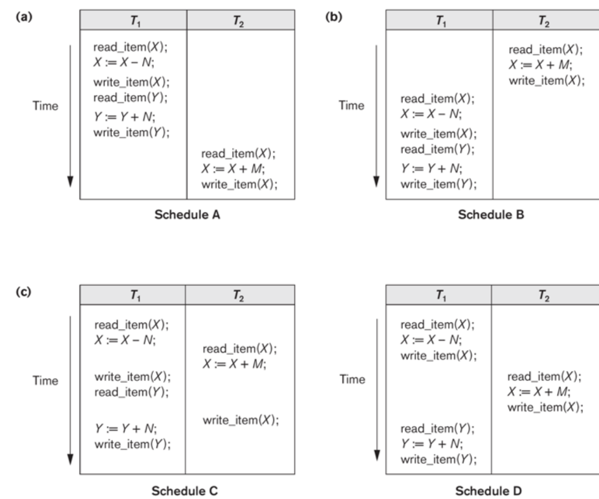


Karakterisering av historier basert på serialiserbarhet

Serialiserbare historier er historier som alltid er korrekte når samtidige transaksjoner utføres. Hvis to transaksjoner T_1 og T_2 sendes til DBMS samtidig og fletting av operasjoner ikke er tillatt, vil vi få to mulige historier:

1. Utfør alle operasjonene i T_1 (i sekvens) etterfulgt av alle operasjonene i T_2 (i sekvens)
2. Utfør alle operasjonene i T_2 (i sekvens) etterfulgt av alle operasjonene i T_1 (i sekvens)

Disse historiene er **serielle** (figur a og b, neste side). Hvis fletting av operasjoner er tillatt, vil vi få mange mulige historier (eks: figur c). **Serialiserbarhet brukes for å identifisere hvilke historier som er riktige når operasjonene er flettet sammen.**



Serielle, ikke-serielle og konflikt-serialiserbare historier

Vi skiller mellom:

- **Seriell historie** – transaksjonene kjører etter hverandre og operasjoner fra ulike transaksjoner blir ikke flettet sammen
- **Serialiserbar historie** – fletting av operasjoner er tillatt og historien vil ha samme effekt på databasen som en seriell historie av samme transaksjoner (dvs. de er resultatekvivalent)

Historiene A og B er serielle fordi operasjonene til hver transaksjon utføres i en sekvens uten flettede operasjoner fra den andre transaksjonen. I en seriell historie blir hele transaksjoner utføres i en rekkefølge, for eksempel T_1, T_2 i historie A. Historiene C og D er ikke-serielle siden operasjonene er flettet sammen.

Serielle historier – ingen fletting av operasjoner

I en seriell historie vil **kun én transaksjon være aktiv om gangen**, og committ (eller abort) av denne vil initiere utførelsen av neste transaksjon. **Hvis transaksjonene er uavhengige kan vi anta at alle serielle historier er korrekte**, siden det antas at alle transaksjoner er riktige hvis de gjennomføres alene (konsistensbevaring). Rekkefølgen av transaksjonene er ikke viktig, så lenge de fullføres isolert fra hverandre. **Problemet med serielle historier er:**

1. **De begrenser samtidig utførelse ved å hindre fletting av operasjoner**. Hvis en transaksjon venter på I/O kan ikke CPU svitsje til en annen transaksjon, noe som fører til bortkastet CPU prosesseringstid.
2. **Hvis en transaksjon er lang, må de andre transaksjonene vente på at denne blir ferdig utført**.

Serielle historier blir derfor ikke brukt i praksis.

Serialiserbare historier – fletting av operasjoner

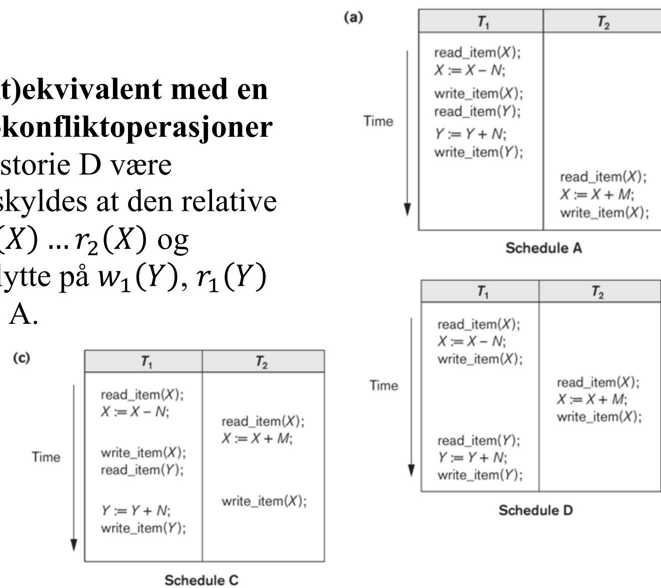
Vi ser på historiene i figuren over, og antar at initiale verdier i databaseelementene er $X = 90$ og $Y = 90$, og at $N = 3$ og $M = 2$. Etter utføringen av transaksjonene T_1 og T_2 vil vi forvente at $X = 89$ og $Y = 93$, og hvis vi utfører historie A eller B vil vi få dette resultatet. Historie C vil gi feil resultat $X = 92$ som følge av tapt-oppdateringsproblemet (s. 152), mens historie D vil gi riktig resultat. **For å bestemme hvilke ikke-serielle historier som alltid gir riktig resultat må vi se på serialiserbarheten til historiene. En historie vil være serialiserbar hvis den er ekvivalent en seriell historie av samme transaksjoner**. Å si at en ikke-seriell historie er serialiserbar er det samme som å si at den er korrekt, siden den er lik den serielle historien som alltid er korrekt.

Det er flere måter å definere at to historier er ekvivalente, for eksempel kan vi se på effekten historiene har på databasen. **To historier er resultatekvivalente hvis de produserer samme endelige tilstand av databasen**. Det er likevel mulig at to ulike historier ved en tilfeldighet kan gi samme resultat. For eksempel kan to ikke-ekvivalente historier gi samme resultat for noen initiale verdier av X og ulik resultat for andre initiale verdier. Ekvivalente historier må utføre like transaksjoner, og resultatekvivalens vil ikke si noe om det er tilfellet. **Resultatekvivalens alene kan derfor ikke brukes for å definere ekvivalente historier**.

Den tryggeste og mest generelle måten å definere ekvivalente historier er ved å fokusere på read_item og write_item operasjonene. For at to historier skal være ekvivalente må operasjonene som brukes på hvert dataelement utføres i samme rekkefølge i de to historiene. En vanlig definisjon av ekvivalente historier er **konfliktekvivalens.**

Konfliktekvivalens av to historier

To historier er konfliktekvivalente hvis den relative rekkefølgen til to operasjoner i konflikt er lik i begge historiene. To operasjoner er i konflikt hvis de hører til ulike transaksjoner, aksesserer samme databaseelement og minst én er write_item. **Hvis to konfliktoperasjoner blir brukt i ulik rekkefølge i de to historiene, kan det bli ulik effekt på databasen eller transaksjonene i historiene. Dermed vil ikke historiene være konfliktekvivalente**. For eksempel hvis historie 1 har $r_1(X); \dots; w_2(X)$ og historie 2 har $w_2(X); \dots; r_1(X)$, kan verdien som leses av r_1 være ulik i de to historiene. De to historiene vil dermed ikke være konfliktekvivalente.



Serialiserbare historier

En historie S_1 vil være serialiserbar hvis den er (konflikt)ekvivalent med en seriell historie S_2. Hvis vi flytter på rekkefølgen til ikke-konfliktoperasjoner i S_1 vil vi kunne finne S_2. I følge denne definisjonen vil historie D være serialiserbar, fordi den er ekvivalent med historie A. Dette skyldes at den relative rekkefølgen til operasjonene i konflikt ($r_1(X) \dots w_2(X)$, $w_1(X) \dots r_2(X)$ og $w_1(X) \dots w_2(X)$) er den samme i begge historiene. Vi kan flytte på $w_1(Y)$, $r_1(Y)$ og $w_1(Y)$ som ikke er i konflikt i historie D for å få historie A.

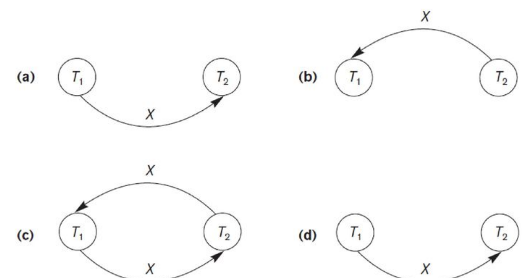
Historie C er ikke serialiserbar, fordi den er ikke ekvivalent til noen serielle historier. Dette skyldes at vi ikke kan flytte på operasjoner uten å endre på rekkefølgen til konfliktoperasjoner.

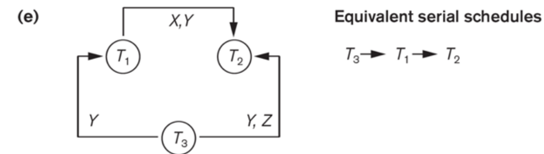
Test av serialiserbarhet

En algoritme brukes for å bestemme om en historie er serialiserbar. **Denne algoritmen bruker read_item og write_item operasjonene for å lage en presedensgraf som er en rettet graf med en node for hver transaksjon i historien**. Kantene vil være mellom to transaksjoner som har konfliktoperasjoner. En kant vil peke fra node T_i til node T_j hvis disse inneholder operasjoner i konflikt og konfliktoperasjonen i T_i kommer først i historien. Følgende er algoritmen:

1. **Lag en node for alle transaksjonene i historien**
2. **Lag en kant $(T_i \rightarrow T_j)$ for hvert tilfelle der:**
    a. $\boldsymbol{w_i(X); \dots; r_j(X)}$ – T_j utfører en read_item(X) etter T_i utfører en write_item(X)
    b. $\boldsymbol{r_i(X); \dots; w_j(X)}$ – T_j utfører en write_item(X) etter T_i utfører en read_item(X)
    c. $\boldsymbol{w_i(X); \dots; w_j(X)}$ – T_j utfører en write_item(X) etter T_i utfører en write_item(X)
3. **Historien er serialiserbar hvis og bare hvis presedensgrafen ikke har noen sykluser**

Hvis presedensgrafen ikke inneholder noen sykluser, vil altså historien være serialiserbar. Den ekvivalente serielle historien vil ha samme retning på kantene, siden retningen gir rekkefølgen på konfliktoperasjonene og den skal være bevart (se figur for historie A og D). På figuren kan vi se at presedensgrafen til historie C har en syklus, så historie C er ikke serialiserbar.





For å finne den ekvivalente serielle historien, start med en node som ikke har noen innkommende kanter og lag kanter til de andre nodene hvis kantene ikke bryter den opprinnelige noderekkefølgen. For eksempel for presedensgraf E, begynner vi i T_3 og lager kantene $(T_3 \rightarrow T_1)$ og $(T_1 \rightarrow T_2)$. Dermed får vi ekvivalent sekvensiell historie $T_3 \rightarrow T_1 \rightarrow T_2$. Legg merke til at vi ikke kan ta den andre veien, for det er ingen måte å nå T_1 fra T_2 uten å bryte den originale noderekkefølgen.

Hvordan serialiserbarhet brukes i samtidighetskontroll

Når en historie er serialiserbar betyr det at den er korrekt, i likhet med den ekvivalente serielle historien. Forskjellen er at den serielle historien representerer ineffektiv prosessering, siden det ikke tillater flettet operasjoner. Dette fører til at CPU ikke brukt når transaksjonen venter på I/O og lange transaksjoner kan forsinke andre transaksjoner. **En serialiserbar historie gir fordelene ved samtidig utføring samtidig som det sikrer korrekthet**.

I praksis er det vanskelig å teste serialiserbarheten til en historie, fordi det er vanskelig å på forhånd bestemme hvordan operasjonene vil flettes sammen (avhenger av flere faktorer, for eksempel systemlast, prioriteter, osv.). **De fleste DBMS har derfor designet protokoller (sett med regler) som vil sikre serialiserbarhet hos historier der alle transaksjonene følger reglene**. Et annet problem er at transaksjoner blir mottatt kontinuerlig, så det er vanskelig å avgjøre når en historie begynner og slutter. For å løse dette problemet kan vi se på **committed projeksjon av historien**, som kun inkluderer operasjoner som hører til committet transaksjoner.

Transaksjonsstøtte i SQL

En SQL transaksjon er atomisk, altså vil den fullføres uten feil eller feile og dermed etterlate databasen uendret. SQL har **ingen eksplisitt Begin_Transaction** påstand, men alle transaksjoner må ha en **eksplisitt endepåstand** (COMMIT eller ROLLBACK). **Alle transaksjoner vil også ha følgende egenskaper som spesifiseres i en SET TRANSACTION påstand:**

- **Aksessmodus** – standard er READ WRITE, men kan spesifiseres som READ ONLY eller WRITE ONLY. Hvis isolasjonsnivået er READ UNCOMITTED vil standard være READ ONLY. Modus READ WRITE tillater seleksjon, oppdatering, innsetting, sletting og *create* kommandoer, mens READ ONLY tillater kun henting.
- **Diagnostikk områdestørrelse** – gir antall betingelser som kan gjelde samtidig for diagnostikkområdet.
- **Isolasjonsnivå** – gis som ISOLATION LEVEL⟨isolation⟩, der ⟨isolation⟩ kan være:
  - READ UNCOMITTED
  - READ COMITTED
  - REPEATABLE READ
  - SERIALIZABLE

  Jo lengre ned på listen desto mer isolasjon og korrekthet, men mindre samtidighet. Standard nivå er SERIALIZABLE og bruk av lavere nivå kan føre til følgende egenskaper:
  - **Dirty read** – en transaksjon T_1 kan lese en oppdatert verdi fra T_2 som enda ikke har blitt committet. Hvis T_2 feiler vil T_1 ha lest en verdi som ikke eksisterer.
  - **Unrepeatable read** – en transaksjon T_1 kan lese en verdi som senere blir endret av en annen transaksjon. Hvis T_1 leser den samme verdien, vil den ikke være lik

- **Fantomer** - en transaksjon T_1 leser et sett med rader fra en tabell, for eksempel basert på en betingelse i WHERE-klausulen. Før T_1 er ferdig vil en annen transaksjon T_2 sette inn en ny rad som tilfredsstiller betingelsen. Denne raden kalles en fantompost, siden den ikke var der når T_1 startet, men er der når den slutter. Hvis ekvivalent seriell rekkefølge er $T_1 \rightarrow T_2$, så skal ikke T_1 se posten, mens hvis den er $T_2 \rightarrow T_1$, så skal T_1 se posten. Systemet må sørge for riktig oppførsel

	Type of Violation		
Isolation Level	Dirty Read	Nonrepeatable Read	Phantom
READ UNCOMMITTED	Yes	Yes	Yes
READ COMMITTED	No	Yes	Yes
REPEATABLE READ	No	No	Yes
SERIALIZABLE	No	No	No

Tabellen viser de mulige bruddene for de ulike isolasjonsnivåene. Som vi kan se vil SERIALIZABLE være strengest og unngår dermed alle tre problemene. Et annet isolasjonsnivå er snapchat isolering, der transaksjonen kun ser dataelement med verdier den leste fra databasetilstanden ved tidspunktet transaksjonen startet. Dette unngår fantomer.

Figuren viser en mulig SQL transaksjon, som vil sette inn en ny rad i Employee tabellen og deretter oppdatere lønnen til alle ansatte ved avdeling 2. Hvis en feil oppstår vil hele transaksjonen aborteres, altså vil lønnen settes tilbake til gammel verdi og Employee raden slettes.

```
EXEC SQL WHENEVER SQLERROR GOTO UNDO;
EXEC SQL SET TRANSACTION
    READ WRITE
    DIAGNOSTIC SIZE 5
    ISOLATION LEVEL SERIALIZABLE;
EXEC SQL INSERT INTO EMPLOYEE (Fname, Lname, Ssn, Dno, Salary)
    VALUES ('Robert', 'Smith', '991004321', 2, 35000);
EXEC SQL UPDATE EMPLOYEE
    SET Salary = Salary * 1.1 WHERE Dno = 2;
EXEC SQL COMMIT;
GOTO THE_END;
UNDO: EXEC SQL ROLLBACK;
THE_END: ... ;
```

Oppsummering – kapittel 20 (F18, F19)

Det er to grunner til at vi har transaksjoner:

1. **Det støtter deling og samtidig aksess av data**. Samtidighetskontroll brukes for å kontrollere operasjoner som blir utført samtidig, slik at konflikt unngås.
2. **Det støtter sikker, pålitelig og atomisk aksess til store mengder data.** Gjenoppretting brukes for å sikre at databasen er konsistens.

I SQL kan man gruppere et vilkårlig spørringer i en transaksjon som deretter sendes til DBMS for å utføres.

Databaseoperasjoner

Databasen består av databaseelementer (-objekt) som kan være en post eller en blokk. All aksess til databasen kan deles inn i $\mathbf{read(X)}$ og $\mathbf{write(X)}$. Vi bruker notasjonen $r_1(X)$ og $w_1(X)$ når transaksjon 1 leser og skriver element X. To spesielle transaksjonsoperasjoner er **committ** (c_1) og **abort** (a_1). Ved committ vil transaksjonen fullføres suksessfullt og endringene lagres i permanent i databasen, mens ved abort vil alle endringene rulles tilbake.

Samtidighetsproblemer – hvorfor trenger vi det?

```
T1:  BEGIN  A=A+100,  B=B-100  END
T2:  BEGIN  A=1.06*A,  B=1.06*B  END

T1:  A=A+100,               B=B-100
T2:            A=1.06*A,              B=1.06*B

T1:  A=A+100,                          B=B-100
T2:            A=1.06*A, B=1.06*B
```

Øverst på figuren ser vi to transaksjoner som gjør endringer på to ulike bankkontoer A og B. T1 flytter 100kr fra B til A, mens T2 legger til renter. **Avhengig av hvilken rekkefølge operasjonene kjører, vil vi få ulike resultat**. Tiden forløper mot høyre for midterste og nederste figur. For eksempel i midten vil den legge til 100kr og regne ut renten til A og deretter fjerne 100kr og regne ut renten til B. Den nederste gir feil resultat, siden renten på 100kr blir lagt til i både A og B.

Dersom systemet ikke har god samtidighetskontroll, kan følgende samtidighetsproblem oppstå:

- **Dirty read** – en transaksjon har lest noe som er skittent (skrevet, men ikke committet). T1 skriver A som deretter blir lest av T2 som committer. T1 vil deretter abortere, så T2 har lest en ugyldig verdi.

T1:	R(A), W(A),		R(B), W(B), Abort
T2:		R(A), W(A), C	

- **Lost update** – skriveoperasjonen til en transaksjon blir borte, fordi den blir skrevet over av en annen transaksjon før den har blitt committet.

T1:	W(A),		W(B), C
T2:		W(A), W(B), C	

- **Unrepeatable read** – samme data leses flere ganger, men resultatet blir ulikt fordi dataen har blitt endret av en annen transaksjon.

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

- **Incorrect summary** – transaksjonen utfører en aggregatfunksjon på en tabell (eks: summering), mens en annen gjør en oppdatering av tabellen. For eksempel på figuren ser vi at summen vil mangle N siden den trekkes fra X før summeringen og legges til Y etter summeringen.

T_1	T_3
	$sum := 0$; read_item(A); $sum := sum + A$; ⋮

read_item(X);
$X := X - N$;
write_item(X);	

	read_item(X);
	$sum := sum + X$;
	read_item(Y);
	$sum := sum + Y$;

read_item(Y);
$Y := Y + N$;
write_item(Y);	

Eksempel – identifikasjon av samtidighetsproblem

Se på følgende historier:

$$H_1: \ r_1(A); \ w_1(A); \ r_1(B); w_2(A); w_1(B); c_1; c_2$$
$$H_2: \ r_1(A); \ w_1(A); \ r_2(A); w_2(A); c_2; \ r_1(B); a_1$$

Merk: for å identifisere hvilke samtidighets-problem en historie lider av, går vi igjennom alle problemene og undersøker disse opp mot historien

Hva er problemet for H_1 og H_2?

a) **Dirty read** - H_2 har dirty read, siden T2 leser A som blir skrevet av T1, og T1 aborterer etter T2 har committet.
b) **Lost update** – Både H_1 *og* H_2 har lost update, fordi i begge historiene vil T1 skrive A som senere blir skrevet av T2 før T1 har committet.
c) **Unrepeatable read** – verken H_1 eller H_2 har unrepeatable read, fordi transaksjonene leser elementene kun én gang hver.
d) **Incorrect summary** – det er ingen aggregatfunksjon

Gjenoppretting – hvorfor trenger vi det?

Det er to typer gjenoppretting (*recovery*) av transaksjoner:

1. **En transaksjon rulles tilbake (aborteres)** – det er en uventet situasjon som gjør at transaksjonen må rulles tilbake. Dette kan skyldes manglende data eller at brukeren eller samtidighetskontrolleren (CC) bestemmer det.
2. **Systemkrasjgjenoppretting** – en feil i databasesystemet, operativsystemet eller datamaskinen krever en restart av systemet. Databasen blir utilgjengelig og vil gjennomføre en redo og undo av transaksjoner avhengig av hvor langt de har kommet i prosesseringen.

Egenskaper ved en transaksjon

En transaksjon er en gruppering av operasjoner mot databasen og har følgende egenskaper:

- **A – Atomiske**: en transaksjon vil enten kjøre fullstendig, eller ikke kjøre i det hele tatt
- **C – Consistency**: operasjonene i transaksjonen vil overholde konsistenskrav (eks: primærnøkler, fremmednøkler, check, osv.)

- **I – isolering**: transaksjoner er isolert fra hverandre, slik at de ikke merker at noen kjører samtidig. Det finnes mange isolasjonsnivåer (mer senere).
- **D – Durabilitet**: endringene gjort av transaksjonen er permanente, altså vil de ikke mistes etter committ. Dette oppnås vha en logg.

En transaksjon er som regel en logisk operasjon som endrer databasen fra en tilstand til en annen.

Transaksjoner i SQL

Committ/abort

En programmerer (for eksempel i Java) kan velge å slutte en transaksjon på to ulike måter:

- **COMMIT** – alt gikk bra og endringene fra transaksjonen lagres permanent i databasen. For å committe en transaksjon brukes kommandoen Connection. commit();
- **ROLLBACK (abort)** – transaksjonen rulles tilbake (aborteres) og ingen endringer fra transaksjonen finnes i databasen. For å rulle tilbake en transaksjon brukes kommandoen Connection. rollback();

I SQL blir Autocommit brukt, noe som betyr at hver SQL-setning er en egen transaksjon. Dette vil være standard i JDBC og kan skrus av vha kommandoen Connection. setAutoCommit(false);

Figuren viser eksempel på to ulike transaksjoner, der vi bruker COMMIT for å lagre endringene i databasen.

```
SET AUTOCOMMIT=0;
UPDATE Account SET b = b - 1000 WHERE id=123123;
UPDATE Account SET b = b + 1000 WHERE id=234234;
COMMIT;
Ekt-eksempel RegMålCtrl
INSERT INTO Reg VALUES (1,123123,31,100);
INSERT INTO Reg VALUES (2,123123,32,120);
….
INSERT INTO Reg VALUES (9,123123,175,245);
UPDATE Loper SET status = 'ok'
WHERE brikkenr=123123;
COMMIT;
```

SQL isolasjonsnivå

Programmereren kan velge hvilket isolasjonsnivå som skal brukes. I SQL blir disse gitt i kommandoen SET TRANSATION ISOLATION LEVEL, og de ulike isolasjonsnivåene er:

- **READ UNCOMITTED**
- **READ COMITTED**
- **REPEATABLE READ**
- **SERIALIZABLE** – standard isolasjonsnivå og den sterkeste isolasjonen som gir mest korrekt utførelse. Ulempen er at den tillater lite samtidighet.

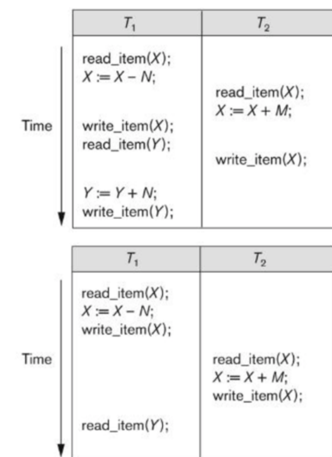
Isolasjonen og korrekthet vil øke nedover i listen, fordi flere låser blir satt slik at transaksjonen blir mindre forstyrret. **Samtidigheten vil reduseres nedover listen**. Ulempen med mer korrekthet er altså at vi får mindre samtidighet (parallellitet). Det er tre egenskaper vi vil unngå:

1. **Dirty read** – en transaksjon leser en verdi fra en annen transaksjon som senere aborterer
2. **Unrepeatable read** – en transaksjon leser samme element flere ganger, men får ulikt resultat
3. **Fantomer** - en transaksjon leser en mengde verdier basert på en søkebetingelse, og imens blir denne mengden endret av en annen transaksjon. Dette er aktuelt ved reskanning (eks: nested loop).

Merk: ved unrepeatable read vil rader som er der fra før ha blitt endret etter første gang de ble lest, mens ved fantomer vil nye rader være lagt til etter første gang tabellene ble skannet

Tabellen viser hvilke egenskaper de ulike isolasjonsnivåene unngår. For eksempel kan vi se at Read Committet unngår Dirty Read, fordi den låser elementer som er skrevet, slik at de ikke kan leses før de er committet. SERIALIZABLE er eneste nivå som med sikkerhet unngår alle problemene.

Isolation Level	Dirty Read	Unrepeatable Read	Phantom Problem
Read Uncommitted	Maybe	Maybe	Maybe
Read Committed	No	Maybe	Maybe
Repeatable Reads	No	No	Maybe
Serializable	No	No	No



Generelt om transaksjoner

Transaksjonshistorie

Historier (*schedule*) er en liste av operasjonene read, **w**rite, **a**bort, **c**ommit **for en mengde transaksjoner**. For eksempel for transaksjonene på figuren, kan vi definere følgende historier:

$$H_a: r_1(X);\ r_2(X);\ w_1(X);\ r_1(Y);\ w_2(X);\ w_1(Y)$$
$$H_b: r_1(X);\ w_1(X);\ r_2(X);\ w_2(X);\ r_1(Y); a_1$$

Vi inkluderer commit og abort når vi skal se på gjenopprettbarheten til historien og ser bort fra disse når vi skal se på samtidighetsegenskaper.

Konflikt i historier – samtidighet

To operasjoner fra en historie er i konflikt hvis alle følgende egenskaper er oppfylt:

1. **De tilhører ulike transaksjoner**
2. **De bruker samme dataelement**
3. **Minst én av operasjonene er en write**

To operasjoner er i konflikt hvis endring av rekkefølgen til operasjonene endrer resultatet på databasen. Dette brukes i konfliktserialiserbarhet, som vi skal se på senere.

Karakterisering av historier basert på gjenopprettbarhet

Vi har følgende karakteriseringer av historier basert på gjenopprettbarhet:

- **Gjenopprettbar** – **hver transaksjon comitter etter at transaksjoner de har lest fra har comittet**. For $w_1(X) \dots r_2(X)$ må $c_1 \dots c_2$.
- **ACA** – **hver transaksjon kan kun lese verdier skrevet av comittet transaksjoner**. For $w_1(X) \dots r_2(X)$ må $c_1 \dots r_2(X)$.
- **Strikt** – **hver transaksjon kan kun lese og skrive comittet verdier**. For $w_1(X) \dots r_2/w_2(X)$ må $c_1 \dots r_2/w_2(X)$.

Vi har at **Strikt** ⊂ **ACA** ⊂ **Gjenopprettbar** ⊂ **Alle historier**, altså alle historier som er strikt vil også være ACA og gjenopprettbar (ikke nødvendigvis motsatt).

Karakterisering av historier basert på samtidighet

Vi har følgende karakteriseringer av historier basert på samtidighet:

- **Serielle historier** - historier som **ikke fletter operasjoner** fra ulike transaksjoner, men kjører heller transaksjonene etter hverandre
- **Serialiserbare historier** – historier som **fletter operasjoner** og har samme effekt på databasen som en seriell historie med samme transaksjoner (de er resultatekvivalent)

Serielle historier tillater ikke samtidighet. Dermed blir det ikke mulig med parallelle tråder (flere transaksjoner kjører samtidig) og hvis en transaksjon krever diskaksess må andre transaksjoner vente. **Derfor blir kun serialiserbare historier brukt i praksis, siden det gir bedre utnyttelse av maskinressursene** (eks: CPU prosessering).

Konfliktserialiserbarhet

For at en historie skal være serialiserbar, må den være ekvivalent med en seriell historie av samme transaksjoner. For å avgjøre om en historie er ekvivalent med en seriell historie og dermed serialiserbar, kan vi se på konfliktene mellom transaksjonene. Mulige konflikter er:

- $r_1(A)$ og $w_2(A)$
- $w_1(A)$ og $r_2(A)$
- $w_1(A)$ og $w_2(A)$

To historier er konfliktekvivalente hvis de har samme rekkefølge på operasjoner med konflikt. En historie er konfliktserialiserbar hvis den er konfliktekvivalent med en seriell historie. Konfliktserialiserbarhet impliserer serialiserbarhet, men ikke nødvendigvis motsatt.

Presedensgraf

For å finne ut om en historie er konfliktserialiserbar, kan vi bruke en presedensgraf. Dette er en rettet graf med noder som representerer transaksjon i historien. **Grafen vil inneholde kanten $T_1 \rightarrow T_2$, hvis det finnes en operasjon i T_1 som er i konflikt med og skjer før en operasjon i T_2. Historien er konfliktserialiserbar hvis presedensgrafen ikke har sykluser.**

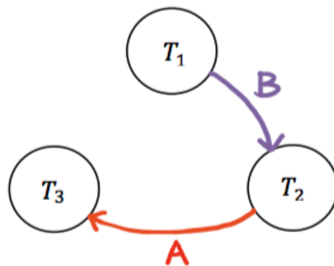
Oppskrift for å lage presedensgraf

1. Lag en node for alle transaksjonene i historien
2. Gå igjennom enhetene en etter en og lag kanter for konflikter
3. Historien er serialiserbar hvis presedensgrafen ikke har noen sykluser

Eksempel 1

$H_1: r_2(A); r_1(B); w_2(A); r_3(A); w_1(B); w_3(A); r_2(B); w_2(B);$

Konfliktene er markert med piler i uttrykket over.
Presedensgrafen blir:

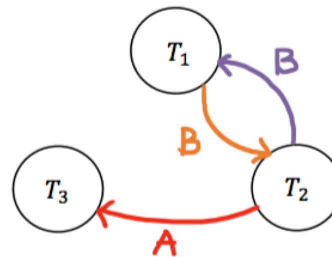


Denne historien er konfliktserialiserbar, siden det ikke er noen sykluser. Den vil være konfliktekvivalent med den serielle historien

Eksempel 2

$H_2: r_2(A); r_1(B); w_2(A); r_2(B); r_3(A); w_1(B); w_3(A); w_2(B);$

Konfliktene er markert med piler i uttrykket over.
Presedensgrafen blir:

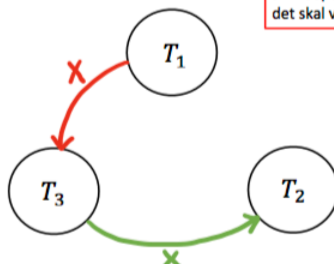


Denne historien er ikke konfliktserialiserbar, siden det er en syklus i presedensgrafen.

Eksempel 3

$H_3: r_1(X); r_2(Y); w_3(X); r_2(X); r_1(Y)$

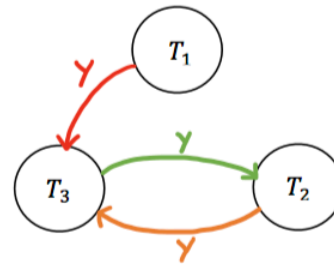
OBS! Det MÅ være minst én skriveoperasjon for at det skal være konflikt!



Denne historien er konfliktserialiserbar, siden det ingen syklus

Eksempel 4

$H_4: r_1(X); r_1(Y); w_1(X); r_2(Y); w_3(Y); w_1(X); r_2(Y);$



Denne historien er ikke konfliktserialiserbar, siden det en syklus

Kapittel 21 – Samtidighetskontroll (CC)

Dette kapittelet handler om teknikkene for samtidighetskontroll som brukes for å isolere transaksjoner som utføres samtidig. De fleste av disse teknikkene vil gi serialiserbare historier ved å bruke samtidighetskontroll protokoller. Et viktig sett med protokoller **er to-fase låseprotokollen som låser dataelementer for å hindre at flere transaksjoner aksesserer samme element samtidig.** Et annet sett med protokoller bruker **tidsstempler** som unikt identifiserer hver transaksjoner og lages i samme rekkefølge som starttidene til transaksjonene. M**ultiversjon samtidighetskontroll bruker flere versjoner av er dataelement**.

To-fase låseprotokoll (2PL)

To-fase låseprotokoll er et sett med regler som kontrollerer samtidig utføring av transaksjoner ved å låse databaseelementer. En **lås** er en variabel assosiert med dataelementet, og den beskriver statusen til elementet mht. operasjoner som kan brukes på den. Det vil generelt være en lås per dataelement i databasen. Låsene brukes for å synkronisere aksessen til transaksjoner som utføres samtidig.

Typer låser og låsetabeller

Det finnes flere typer låser i samtidighetskontroll, blant annet:

- **Binære låser** – enkle, men for begrenset for samtidighetskontroll av databaser
- **Delte/eksklusive låser** – gir mer generelle låseegenskaper og kalles også lese/skrive låser

Binære låser

En binær lås kan ha to tilstander eller verdier: låst eller ulåst (1 eller 0). **Hvert databaseelement $\boldsymbol{X}$ vil ha en distinkt lås, og hvis verdien til denne låsen er 1 kan ikke $\boldsymbol{X}$ aksesseres av en databaseoperasjon**. Hvis verdien til låsen er 0 kan X aksesseres av en operasjon og låsen endres til 1. Nåværende verdi til låsen X kalles $\mathbf{lock(X)}$. For binære låser blir to operasjoner brukt:

1. $\mathbf{lock_item(X)}$ – brukes når en transaksjon etterspør aksess til element X. Hvis $\mathrm{LOCK}(X) = 1$ tvinges transaksjonen til å vente, mens hvis $\mathrm{LOCK}(X) = 0$ vil den settes til 1 (elementet låses) og transaksjonen får aksessere element X.
2. $\mathbf{ulock_item}(\boldsymbol{X})$ – brukes når transaksjonen er ferdig med element X. Dette vil sette $\mathrm{LOCK}(X) = 0$ (elementet låses opp), slik at X kan aksesseres av andre transaksjoner

En binær lås har altså gjensidig eksklusjon på dataelementet. Figuren viser disse operasjonene som må implementeres som individuelle enheter. Når en av operasjonene har startet vil altså ingen fletting være tillatt før operasjonen terminerer eller transaksjonen plasseres i kø. Kommandoen *wait* blir vanligvis implementert ved å plassere transaksjonen i en **ventekø for enhet $\boldsymbol{X}$**, helt til X blir låst opp og transaksjonen kan få aksess til den. Andre transaksjoner blir plassert i samme kø.

```
lock_item(X):
B:  if LOCK(X) = 0              (*item is unlocked*)
        then LOCK(X) ←1         (*lock the item*)
    else
        begin
        wait (until LOCK(X) = 0
            and the lock manager wakes up the transaction);
        go to B
        end;
unlock_item(X):
    LOCK(X) ← 0;                (* unlock the item *)
    if any transactions are waiting
        then wakeup one of the waiting transactions;
```

For å implementere en binærlås bruker vi binærvariabelen LOCK. Hver lås som hører til en enhet, representeres som en post med tre felt: $\langle\mathbf{Data_item_name}, \mathbf{LOCK}, \mathbf{Locking_transaction}\rangle$, altså enhetsnavnet, låsvariabelen og transaksjonen som har låst enheten. Hver lås må også ha en kø for transaksjoner som venter på å aksessere enheten. **Det er kun låste enheter som vil ha**

slike poster lagret i låstabellen til systemet, og de kan organiseres i for eksempel en hashfil på enhetsnavnet. Enheter som ikke er i låstabellen blir sett på som ulåst. DBMS har et delsystem for låskontroll som vil holde styr over og kontrollere aksessen til låser. For **binærlås skjemaet** må transaksjonene følge følgende regler:

1. $\text{lock_item}(X)$ må utføres før første $\text{read_item}(X)$ eller $\text{write_item}(X)$
2. $\text{unlock_item}(X)$ må utføres etter alle $\text{read_item}(X)$ og $\text{write_item}(X)$ er ferdige
3. $\text{lock_item}(X)$ blir ikke utført hvis transaksjonen allerede har låst X
4. $\text{unlock_item}(X)$ blir ikke utført hvis transaksjonen ikke har en lås på enhet X

Disse reglene blir håndhevet av låskontrolleren i DBMS.

Systemet vil altså ha en egen låstabell som inneholder låste enheter. Når en transaksjon skal utføres må den sjekke om enheten er låst ved å se om den er registrer i låstabellen. Hvis det er tilfellet vil den legges til i en kø, hvis ikke vil enheten låses og transaksjonen utføres. **Det er kun én transaksjon som kan holde låsen på et bestemt element om gangen, så to transaksjoner kan ikke aksessere samme element samtidig.**

Delte/eksklusive (skrive/lese) låser

Det binære låssystemet er for begrenset, siden det lar kun én transaksjon om gangen holde låsen på et element. **Flere transaksjoner bør få aksess til samme element hvis de kun leser dette elementet, siden leseoperasjoner på samme element av ulike transaksjoner ikke vil kollidere**. **Hvis en transaksjon skriver et element, bør den ha eksklusiv aksess til elementet.** Dette kalles **delt/eksklusiv** eller **skrive/lese låser**, og det er tre låseoperasjoner:

1. $\textbf{read_lock}(\mathbf{X})$ – brukes når transaksjonen skal lese X og lar andre transaksjoner lese (men ikke skrive) X samtidig. Den vil sette $\textbf{LOCK}(\boldsymbol{X}) = \textbf{read-locked}$ og kalles *shared-lock* siden låsen deler elementet med andre lesende transaksjoner,
2. $\textbf{write_lock}(\boldsymbol{X})$ – brukes når transaksjonen skal skrive X og låser elementet slik at ingen andre transaksjoner får aksess (verken lese eller skrive). Den vil sette $\textbf{LOCK}(\boldsymbol{X}) = \textbf{write-locked}$ og kalles *exclusive-lock* siden kun låsende transaksjon får aksess til elementet.
3. $\textbf{unlock}(\boldsymbol{X})$ - brukes når transaksjonen er ferdig med element X. Den vil sette $\text{LOCK}(X) = \text{unlocked}$, slik at X kan aksesseres av andre transaksjoner

For å implementere disse operasjonene kan låstabellen inkludere antall transaksjoner som holder en delt leselås på en enhet og en liste av transaksjonsid til transaksjonene som holder den delte låsen. Hver post i låstabellen vil ha fire felt: $\langle \textbf{Data_item_name}, \textbf{LOCK}, \textbf{No_of_reads}, \textbf{Locking_transation}(\mathbf{s}) \rangle$. Det er kun låste enheter som vil ha poster i låstabellen, så derfor vil verdien til LOCK være read-locked eller write-locked. Hvis $\text{LOCK}(X) = \text{write-locked}$, vil $\text{Locking_transation(s)}$ være én transaksjon som holder den eksklusive skrivelåsen på X. Hvis $\text{LOCK}(X) = \text{read-locked}$, vil $\text{Locking_transation(s)}$ være en liste av transaksjoner som holder den delte leselåsen på X.

Figuren viser de tre operasjonene som må implementeres som individuelle enheter. Når en av operasjonene har startet vil altså ingen fletting være tillatt før operasjonen terminerer eller transaksjonen plasseres i kø.

```
read_lock (X):
  B: if LOCK (X)="unlocked"
     then begin LOCK (X)← "read-locked";
                no_of_reads(X)← 1
          end
     else if LOCK(X)="read-locked"
          then no_of_reads(X)← no_of _reads(X) + 1
          else begin wait (until LOCK (X)="unlocked" and
                    the lock manager wakes up the transaction);
                  go to B
                end;

write_lock (X):
  B: if LOCK (X)="unlocked"
     then LOCK (X)← "write-locked"
     else begin
          wait (until LOCK(X)="unlocked" and
            the lock manager wakes up the transaction);
          go to B
          end;

unlock_item (X):
  if LOCK (X)="write-locked"
     then begin LOCK (X)← "unlocked;"
          wakeup one of the waiting transactions, if any
          end
     else if LOCK(X)="read-locked"
          then begin
                no_of_reads(X)← no_of_reads(X) – 1;
                if no_of_reads(X)=0
                  then begin LOCK (X)="unlocked";
                          wakeup one of the waiting transactions, if any
                          end
                  end;
```

For **delt/eksklusiv låsskjemaet** må transaksjonene følge følgende regler:

1. read_lock(X) eller write_lock(X) må utføres før første read_item(X)
2. write_lock(X) må utføres før første write_item(X)
3. unlock(X) måt utføres etter alle read_item(X) og write_item(X) er ferdige
4. read_lock(X) eller write_lock(X) blir ikke utført hvis transaksjonen allerede har en lese- eller skrivelås på X
5. unlock_item(X) blir ikke utført hvis transaksjonen ikke har en lås på enhet X

Konvertering av låser (oppgradering, nedgradering)

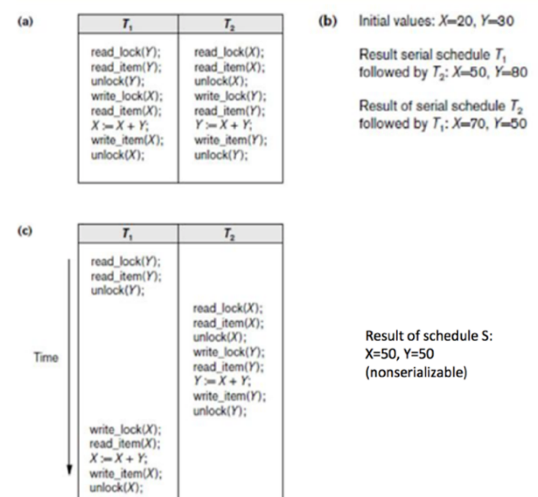
Det er ønskelig å relaksere regel 4 for å tillate **konvertering av låser, der en lås hos en transaksjon kan omdannes fra en tilstand til en annen**. Det er to muligheter:

- **Oppgradering av låsen** – en read_lock(X) kan oppgraderes til en write_lock(X) dersom transaksjonen er den eneste som holder en leselås på X i det øyeblikket den utfører write_lock(X) operasjonen. Hvis ikke må transaksjonen vente.
- **Nedgradering av låsen** – en write_lock(X) kan nedgraderes til en read_lock(X) ved at transaksjonen utfører read_lock(X) operasjonen. Merk: skrivelåsen er eksklusiv.

Når dette er tillatt må Locking_transation(s) for hver lås i låstabellen inneholde transaksjonsid, slik at det er lagret informasjon om hvilke transaksjoner som holder lås på ulike elementer.

Garantert serialiserbarhet vha to-fase låsing (2PL)

Selv om historiene følger reglene for binær- eller delt/eksklusiv låsskjema, vil det ikke være garantert at de er serialiserbare. For å garantere serialiserbarhet trengs det enda en protokoll som gir regler for posisjonering av låsing og opplåsning av operasjoner. Den mest kjente er to-fase låseprotokollen (2PL). **En transaksjon sies å følge 2PL protokollen hvis alle låsoperasjonene (read_lock og write_lock) kommer før den første unlock operasjonen i transaksjonen**. En slik transaksjon kan deles inn i to faser: (1) **utvidende (første) fase** der nye låser blir laget og ingen låser åpnes og (2) **krympende (andre) fase** der eksisterende låser åpnes og ingen nye låser blir laget. Hvis konvertering av låser er tillatt kan låser oppgraderes i utvidende fase og nedgraderes i krympende fase.



Transaksjon T_1 og T_2 på figur A følger ikke 2PL protokollen fordi write_lock(X) kommer etter unlock(Y) i T_1 og write_lock(Y) kommer etter unlock(X) i T_2. På figur B ser vi at det er to mulige serielle historier avhengig av rekkefølgen på transaksjonene (begge gir «korrekt» resultat). Figur C viser en historie som ikke er resultatekvivalent med en av de serielle historiene, og dermed ikke serialiserbar. **Selv om transaksjonene følger reglene for låsskjemaet, kan operasjonene ordnes på en måte som gjør at historien ikke blir serialiserbar.** Dette skyldes at Y og X blir låst opp for tidlig i hhv. T_1 og T_2. Dvs. transaksjonene har ikke 2PL!

T_1'

read_lock(Y);
read_item(Y);
write_lock(X);
unlock(Y)
read_item(X);
$X := X + Y$;
write_item(X);
unlock(X);

T_2'

read_lock(X);
read_item(X);
write_lock(Y);
unlock(X)
read_item(Y);
$Y := X + Y$;
write_item(Y);
unlock(Y);

Figuren til venstre viser hvordan vi kan **omforme transaksjonene, slik at de følger 2PL protokollen**. Her kan vi se at alle låsoperasjonene kommer før den første unlock operasjonen. Merk skillet mellom ... _lock og ... _item. Hvis transaksjonene oppfyller reglene for låsskjemaet vil disse transaksjonene garantere at historien er serialiserbar. Historien på

figur C er ikke mulig siden T_1' vil sende write_lock(X) før den låser opp Y, slik at når T_2' sender read_lock(X) tvinges den til å vente helt til T_1' slipper skrivelåsen på X. Dette kan likevel føre til *vranglås* (mer senere).

Hvis alle transaksjoner i en historie følger to-fase låseprotokollen, er det garantert at historien er serialiserbar. Dvs. historier som består av transaksjoner som følger 2PL protokollen og reglene for låsskjemaet vil være serialiserbare. Dermed slipper systemet å teste for serialiserbarhet. **2PL protokollen kan begrense mengden samtidighet i en historie**, fordi det kan hende transaksjonen må vente med å slippe en elementlås eller låse et element før den trenger det, for å oppfylle definisjonen til 2PL (alle låsene må lages før første unlock). Dermed kan det hende at andre transaksjoner må vente på å få tilgang til bestemte elementer, selv om disse elementene egentlig ikke brukes.

Varianter av to-fase låsing

Det finnes flere ulike varianter av to-fase låsingen (2PL):

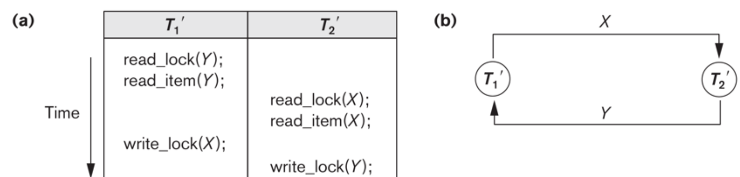
- **Grunnleggende 2PL** – alle låseoperasjonene må komme før den første unlock
- **Konservativ 2PL** – transaksjonen må låse alle elementene den aksesserer før den starter utførelsen, vha forhåndserklærte *read*- og *write*-sett som inneholder alle elementene som skal hhv. leses og skrives. Hvis noen av elementene ikke kan låses, vil ikke transaksjonen låse noen av elementene, men heller vente til alle blir tilgjengelig. Dette er en vranglås-fri protokoll (mer senere), men den er vanskelig å bruke i praksis.
- **Strikt 2PL** – garanterer strikte historier, siden transaksjoner ikke slipper skrivelåser før de har committet eller abortert. Ingen andre transaksjoner kan lese eller skrive et element som er skrevet av T, før T har committet. Strikt 2PL er ikke vranglås-fri.
- **Streng 2PL (*rigorous*)** – garanterer strikte historier, siden transaksjoner ikke slipper noen låser før de har committet eller abortert (dvs. lese- og skrivelåser). Dette er en strengere versjon av strikt 2PL, men den er enklere å implementere.

Som regel vil delsystemet for samtidighetskontroll være ansvarlig for å generere forespørsler for skrive- og leselåser. Når en transaksjon sender en read_item eller write_item, vil systemet kalle hhv. read_lock og write_lock for transaksjonen. Hvis elementet er ledig vil operasjonen utføres, mens hvis elementet allerede er låst vil systemet plassere transaksjonen i kø. Systemet vil oppdatere låstabellen underveis.

Låsing har et stort overhead, siden en låsforespørsel må sendes før hver lese- og skriveoperasjon. Bruken av låser kan også føre til to problemer: **vranglås** og **sult**.

Vranglås (*deadlock*)

Vranglås skjer når alle transaksjoner i et sett av to eller flere venter på et element som er låst av en annen transaksjon i settet. Hver transaksjon er i kø og venter på at en av de andre transaksjonen skal slippe låsen på et element, men **siden den andre transaksjonen også er i kø vil den aldri slippe låsen**. Figuren viser et eksempel. De to transaksjonene T_1' og T_2' er i vranglås siden T_1' venter i kø på X som er låst av T_2', mens T_2' venter i kø på Y som er låst av T_1'.



Vranglås forebyggingsprotokoller

En vranglås forebyggingsprotokoll kan brukes for å unngå vranglås. En protokoll vil sørge for at alle transaksjonene må låses på forhånd og hvis noen ikke kan nås, vil ingen av elementene låses (brukes av konservativ 2PL). En annen protokoll vil ordne alle elementene i databasen og sørger for at transaksjonen låser elementer i den rekkefølgen. Disse begrenser samtidighet.

Andre vranglås forebyggingsprotokoller ser på hva som skal gjøres med en transaksjon som er involvert i en mulig vranglås. Noen av disse bruker **tidsstempel** $TS(T')$, som er en unik identifikator av hver transaksjon. Tidsstemplene er basert på når transaksjonene startet, så hvis T_1 startet før T_2 vil $TS(T_1) < TS(T_2)$. Den eldste transaksjonen har minst tidsstempel. Hvis T_i prøver å låse element X som er låst av T_j, kan følgende skjema brukes for å hindre vranglås:

1. **Wait-die** – hvis $TS(T_i) < TS(T_j)$, vil T_i være eldst og den får vente, mens hvis T_i er yngst vil den abortere (dø) og restarte senere med samme tidsstempel
2. **Wound-wait** – hvis $TS(T_i) < TS(T_j)$, vil T_i være eldst og T_j vil abortere (T_i skader T_j) og restarte senere med samme tidsstempel, mens hvis T_i er yngst vil den vente

I Wait-die får en eldre transaksjon vente på en yngre transaksjon, mens en yngre transaksjon vil aborteres og restartes. Wound-wait tilnærmingen gjør det motsatte. En yngre transaksjon får vente på en eldre, mens en eldre transaksjon vil abortere den yngre transaksjonen som har låst elementet som aksesseres. Begge vil abortere den yngste transaksjonen som kanskje er involvert i en vranglås, siden dette kan føre til mindre bortkastet prosessering. **Disse er vranglås-frie, siden kun en av transaksjonene vil vente på den andre (en må være yngst/eldst) og vranglås krever gjensidig venting. Ulempen er at noen transaksjoner kan aborteres selv om de ikke forårsaker vranglås.**

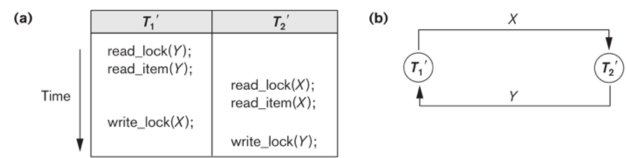
To andre vranglås forebyggingsprotokoller er:

1. **No Waiting (NW) algoritme** - hvis transaksjonen ikke får laget en lås vil den umiddelbart aborteres og restarte etter en bestemt tidsforsinkelse. Det blir ikke sjekket om en vranglås kan oppstå, men siden ingen transaksjoner venter vil ingen vranglås dannes. Transaksjoner kan aborteres selv om de ikke forårsaker vranglås.
2. **Cautious Waiting (CW) algoritme** – hvis transaksjonen forsøker å låse et element som er låst av en annen transaksjon, vil den vente dersom den andre transaksjonen ikke er blokkert (dvs. ikke venter på et annet låst element). Hvis ikke vil den aborteres. Dette vil redusere antall unødvendige aborter.

Deteksjon av vranglås

Vranglås deteksjon er en alternativ måte å håndtere vranglåser som blir brukt når transaksjoner sjeldent aksesserer samme element samtidig (lite interferens). Dette vil være tilfellet når transaksjonene er korte og låser få elementer. For lange transaksjoner som låser mange elementer vil forebyggingsprotokoll være bedre.

Systemet kan detektere vranglåser ved å lage og opprettholde en **wait-for graf**, der det lages en node for alle aktive transaksjoner. Når T_i venter på å låse et element X som er låst av T_j, vil rettet kant $(T_i \rightarrow T_j)$ legges til grafen. Når T_j slipper låsen, blir kanten fjernet. **Historien vil ha en vranglås hvis og bare hvis wait-for grafen har en syklus**. En utfordring ved denne metoden er



å bestemme når systemet skal sjekke etter en vranglås. En mulighet er å sjekke hver gang en kant blir lagt til grafen, men dette kan føre til mye overhead. I stedet kan man bruke antall transaksjoner som venter eller ventetiden for flere transaksjoner som kriterier. Figur b viser wait-for grafen for historien på figur a. **Hvis systemet har en vranglås, må noen av transaksjonene som forårsaker vranglåsen aborteres.** Dette valget kalles **offerseleksjon** og transaksjonene som aborteres vil som regel være de yngste.

Timeout

Vranglåser kan også håndteres vha timeouts, der transaksjoner aborteres hvis de har ventet lengre enn systemets definerte timeout periode. Hver transaksjon får altså en timeout som sammenlignes med systemets timeout. Denne metoden er enkel og har lite overhead, men det er vanskelig å sette timeouten riktig.

Sult (*starvation*)

Et annet problem som kan oppstå ved låsing er sult, der en transaksjon ikke vil fortsette på ubestemt tid, mens andre transaksjoner fortsetter normalt. Dette kan skje hvis venteskjemaet for låste elementer er urettferdig og gir prioritet til noen transaksjoner over andre. For å unngå sult kan vi bruke et rettferdig venteskjema, for eksempel **first-come-first-served kø**, der transaksjoner får låse elementet i den rekkefølgen de ankom køen. En annen mulighet er å **øke prioriteten til transaksjonen ettersom den venter**, slik at den til slutt vil få høyest prioritet og dermed fortsette. Sult kan også oppstå hvis samme transaksjon blir valgt flere ganger ved offerseleksjon, noe som kan unngås ved å gi høyere prioritet til transaksjoner som har blitt valgt flere ganger. Wait-die og Wound-die metodene unngår sult, siden de restarter transaksjoner med samme tidsstempel.

Husk: samtidighetskontroll brukes for å sikre serialiserbarhet

Multiversjon samtidighetskontroll (CC)

Multiversjon samtidighetskontroll (CC) vil ta vare på kopier av gamle verdier til et dataelement når det oppdateres (skrives), altså blir flere versjoner av et element holdt i systemet. Dette er en metode som brukes mye i dagens SQL-databaser. Når en transaksjon vil lese et element, vil passende versjon velges for å opprettholde serialiserbarheten til historien. Hvis en leseoperasjon er i konflikt, kan dette løses ved at den får lese en gammel versjon av elementet og dermed vil historien fortsatt kunne være serialiserbar. Når en transaksjon skriver et element vil den skrive en ny versjon og gamle versjoner blir tatt vare på. **Ulempen med multiversjon CC er at det krever mer lagring**, men i noen tilfeller hender det at gamle versjoner må tas vare på uansett for gjenoppretting.

Multiversjon teknikk basert på tidsstempel

I denne metoden blir flere versjoner $X_1, X_2, \ldots, X_k$ av et element X tatt vare på, og for hver versjon X_i vil verdien og følgende to tidsstempel lagres:

- $\mathbf{read_TS(X_i)}$ – det største tidsstempelet til transaksjonene som har lest X_i
- $\mathbf{write_TS(X_i)}$ – tidsstempelet til transaksjonen som skrev X_i

Når en transaksjon T utfører $\mathrm{write_item(X)}$, vil det lages en ny versjon X_{k+1} og både $\mathrm{read_TS(X_{k+1})}$ og $\mathrm{write_TS(X_{k+1})}$ settes lik $TS(T)$. Når T får utføre $\mathrm{read_item(X)}$, vil $\mathrm{read_TS(X_i)}$ settes lik den største av nåværende $\mathrm{read_TS(X_i)}$ og $TS(T)$.

Husk: for $TS(T_1) < TS(T_2)$ vil T_1 være eldst og starter før T_2. Den eldste transaksjonen har minst tidsstempel

For å sikre serialiserbarhet, vil følgende regler brukes:

1. **Hvis T ønsker å utføre en $\text{write_item}(X)$ vil den rulles tilbake dersom $\text{write_TS}(X_i) \leq TS(T)$ og $\text{read_TS}(X_i) > TS(T)$.** Hvis ikke vil det lages en ny versjon X_j der $\text{read_TS}(X_j) = \text{write_TS}(X_j) = \text{TS}(T)$.
2. **Hvis T ønsker å utføre en $\text{read_item}(X)$, finn versjonen X_i som har høyest verdi for $\text{write_item}(X_i) \leq TS(T)$ og returner denne.** Verdien til $\text{read_item}(X_i)$ settes lik den største av nåværende $\text{read_TS}(X_i)$ og $TS(T)$.

For regel 1 ser vi tilfellet der T ønsker å skrive et element, men en annen transaksjon skriver X før T starter og leser X etter T starter (= T prøver å overskrive elementet = tapt oppdatering). Da får ikke T skrive elementet, fordi da kan det hende historien ikke blir serialiserbar. For regel 2 vil T lese versjonen av X som sist ble skrevet samtidig eller etter T startet, altså den nyeste versjonen i det T begynte utførelsen. Dette vil sikre at transaksjonene alltid leser riktig verdi.

Oppsummering – kapittel 21 (F19)

Dette kapittelet ser hvordan serialiserbarhet kan oppnås ved å følge protokoller.

Serialiserbarhet ved låsing (to-fase låseprotokoll)

Låsing av dataelementer (poster eller blokker) kan brukes for å garantere konfliktserialiserbarhet. To typer vanlige låser er:

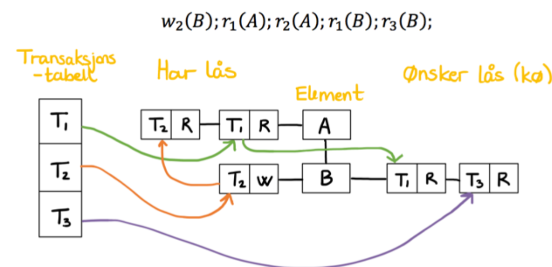
- **$\text{Read_lock}(X)$** – delt lås. Hindrer at andre transaksjoner skriver X, men lar de lese
- **$\text{Write_lock}(X)$** – eksklusiv lås. Hindrer at andre transaksjoner leser eller skriver X

$\text{Read_lock}(X)$ kan oppdateres til $\text{Write_lock}(X)$ og $\text{Write_lock}(X)$ kan nedgraderes til $\text{Read_lock}(X)$. Oppgraderingen er mulig hvis transaksjonen er den eneste som har leselås på X. Siden skrivelåsen er eksklusiv vil nedgradering alltid være mulig.

Implementasjon av låser

Låser blir satt mens transaksjonen kjører og når transaksjonen er ferdig vil låsene fjernes. Derfor er det tilstrekkelig å lagre låsetabellen i minnet (og ikke på disken). Det finnes flere låstyper for eksempel postlåser, blokklåser, tabellåser, verdiområdelåser (unngår fantomer) og predikatlåser (unngår fantomer).

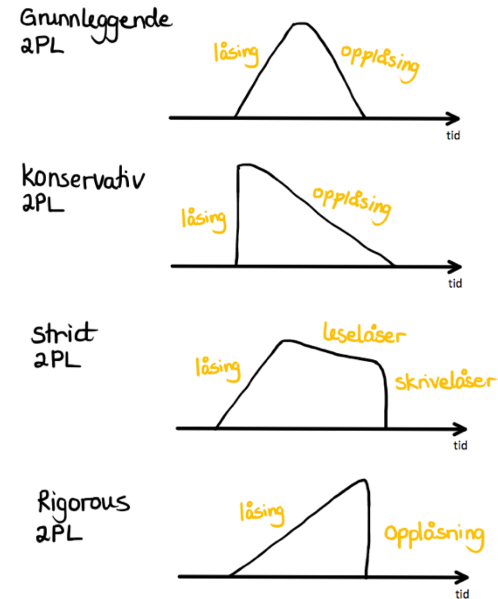
Figuren viser hvordan et låssystem kan se ut for en gitt historie. Transaksjonstabellen vil lenke sammen låsene til en transaksjon, slik at låsene kan slippes når transaksjonen er ferdig.



2PL (tofaselåsing)

Tofaselåsing brukes for sikre at historien er konfliktserialiserbar. **En transaksjon har tofaselåsing hvis alle låseoperasjoner skjer før alle opplåsningsoperasjoner.** På figuren kan vi se to transaksjoner som har tofaselåsing. Legg merke til at T1 må vente, siden den forsøker å skrive på element X som er låst av T2. **Merk: hvis en transaksjon blir blokkert vil alle operasjoner i denne transaksjonen settes på vent, mens de neste operasjonene hos andre transaksjoner i historien blir utført i sekvens.**

T1	T2
	Write_lock(X)
Write_lock(X)	Read(X)
wait	X = X + 1000
wait	Write(X)
wait	Commit / Unlock(X)
Read(X)	
X = X – 100	
Write(X)	
Commit / Unlock(X)	



Tofaselåsing impliserer serialiserbarhet, altså hvis en historie består av transaksjoner som har 2PL vil historien være serialiserbar. Det finnes flere typer 2PL:

- **Grunnleggende 2PL** – symmetrisk feil (alle låser før opplåsning)
- **Konservativ 2PL** – låser alt man trenger aller først
- **Strikt 2PL** – opplåsning av skrivelåser etter commit/abort
- **Rigorous (streng) 2PL** – opplåsning etter commit/abort

Konservativ 2PL krever at programmet må analyseres for å identifisere elementene som kan aksesseres og derfor må låses. Det kan hende at elementer som ikke brukes blir låst, for eksempel ved if-else-setninger. Fordelen ved konservativ 2PL er at den hindrer vranglåser.

T1	T2
Read_lock(X)	Read_lock(Y)
Write_lock(Y)	Write_lock(X)

Vranglås

Vranglås er når to eller flere transaksjoner venter gjensidig på hverandre låser. Figuren viser et eksempel der T1 vil vente på T2 og T2 vil vente på T1. Dette kan løses på forskjellige metoder:

1. **Unngåelse** – låsene settes på en måte som gjør at man unngår vranglåser, for eksempel kan alle transaksjoner sette låsene i samme rekkefølge. Dette er vanskelig å få til i databaser, så de to andre metodene er mer vanlig.
2. **Oppdagelse** – de fleste databaser bruker denne metoden som er basert på en **wait-for graf**. I denne grafen vil hver transaksjon være en node, og den vil inneholde kant $(T_i \rightarrow T_j)$ hvis T_i venter på en lås holdt av T_j. **Vi har vranglås hvis grafen har syklus.** Systemet vil da forsøke å abortere en transaksjon og se om syklusen forsvinner. Det kan være vanskelig å bruke ved distributerte systemer (vanskelig å oppdage sykluser).
3. **Timeout** – dette er den enkleste løsningen som er basert på at hver transaksjon får en timeout. **Hvis timeouten utgår vil transaksjonen aborteres**. Det kan være vanskelig å sette timeouten riktig.

Eksempel – rigorous 2PL

Gitt historien nedenfor, finn rekkefølgen for utføringen av historien gitt at den setter låser og bruker rigorous 2PL metode.

$$H_1: r_1(A); w_2(A); w_2(B); w_3(B); w_1(B); c_1; c_2; c_3$$

T_1	T_2	T_3
$rl_1(A)$		
$r_1(A)$		
	$trylock\ A$	
		$wl_3(B)$
		$w_3(B)$
$trylock\ B$		
		$c_3; unlock(B);$
$wl_1(B)$		
$w_1(B)$		
$c_1; unlock(A,B);$		
	$wl_2(A)$	
	$w_2(A)$	
	$wl_2(B)$	
	$w_2(B)$	
	$c_2; unlock(A,B);$	

tid ↓

T_2 forsøker å skrivelåse A, men den er allerede leselåst av T_1, derav $trylock\ A$ (kan droppes)

Når B låses opp vil $w_1(B)$ være den første operasjonen som utføres. $w_2(B)$ står lengre frem i køen, men den venter på at $w_2(A)$ skal gjennomføres!

Her kan vi se at det er rigorous 2PL, siden alle låsene blir låst opp etter transaksjonen har committet

rl: leselås
wl: skrivelås

Gitt historien nedenfor, finn rekkefølgen for utføringen av historien gitt at den setter låser og bruker rigorous 2PL metode.

$H_2: r_1(A); w_2(B); w_2(A); w_3(B); w_1(B); c_1; c_2; c_3$

T_1	T_2	T_3
$rl_1(A)$		
$r_1(A)$		
	$wl_2(B)$	
	$w_2(B)$	
	$trylock\ A$	
		$trylock\ B$
$trylock\ B$		

tid

Denne historien vil gi en vranglås! Systemet vil velge å abortere en av transaksjonene, for eksempel T_2. Rekkefølgen til resten av utførelsen vil dermed avhenge av låsesystemet.

Gitt historien nedenfor, finn rekkefølgen for utføringen av historien gitt at den setter låser og bruker rigorous 2PL metode.

$H_3: r_1(A); r_2(B); w_1(B); w_1(C); r_2(A); c_1; c_2;$

T_1	T_2
$rl_1(A)$	
$r_1(A)$	
	$rl_2(B)$
	$r_2(B)$
$trylock\ B$	
	$rl_2(A)$
	$r_2(A)$
	$c_2; unlock(A, B);$
$wl_1(B)$	
$w_1(B)$	
$wl_1(C)$	
$w_1(C)$	
$c_1; unlock(A, B);$	

Leselåser kan deles!

$w_1(C)$ må vente på at $w_1(B)$ blir utført!

tid

Multiversjon samtidighetskontroll (CC)

Et alternativ til tofaselåsing som kan brukes for å garantere serialiserbarhet er multiversjon samtidighetskontroll, som brukes mye i dagens SQL-databaser. Konseptet til denne metoden er å lagre flere versjoner av elementet, slik at leseoperasjoner i konflikt kan lese en gammel elementversjon.

Multiversjon CC baseres på **tidsstempelordning**, der hver transaksjon har et tidsstempel $TS(T)$. Hvert element kan finnes i flere versjoner $X_1, X_2, \ldots, X_k$, og for hver versjon vil følgende lages:

- $\mathbf{read_TS(X_i)}$ – det største tidsstempelet til transaksjonene som har lest X_i
- $\mathbf{write_TS(X_i)}$ – tidsstempelet til transaksjonen som skrev X_i

Når en transaksjon skriver X_i, vil den sette begge tidsstemplene til $TS(T)$, mens når den leser X_i vil $\mathrm{read_TS(X_i)}$ settes lik den største av nåværende $\mathrm{read_TS(X_i)}$ og $TS(T)$.

Det er to regler som må oppfylles for å sikre serialiserbarhet:

1. **Hvis T ønsker å utføre en $\mathbf{write_item(X)}$**, vil den rulles tilbake dersom $\mathrm{write_TS(X_i)} \leq TS(T)$ og $\mathrm{read_TS(X_i)} > TS(T)$. Hvis ikke vil det lages en ny versjon X_j der $\mathrm{read_TS(X_j)} = \mathrm{write_TS(X_j)} = \mathrm{TS}(T)$.
2. **Hvis T ønsker å utføre en $\mathbf{read_item(X)}$,** finn versjonen X_i som har høyest verdi for $\mathrm{write_item}(X_i) \leq TS(T)$ og returner denne. Verdien til $\mathrm{read_item(X_i)}$ settes lik den største av nåværende $\mathrm{read_TS(X_i)}$ og $TS(T)$.

Lesing vil alltid lykkes, mens det kan hende skriving fører til at transaksjonen må aborteres. Ulempen med multiversjon samtidighetskontroll er at det krever administrasjon og lagring av mange versjoner, noe som krever mer plass i minnet.

I praksis er det to måter å implementere multiversjon samtidighetskontroll:

1. **Lagrer flere versjoner av poster i databasen og kjøre GC (søppeltømming) når de gamle versjonene ikke trengs lengre**. Brukes av Microsoft SQL server
2. **Lagrer kun siste versjon av posten, men kan konstruere den forrige versjonen vha undo.** Brukes av Oracle og MySQL

Noen systemer bruker **Multiversjon 2PL**, som er en kombinasjon av låsing og multiversjon-CC. I disse systemene vil Read/write-transaksjoner bruke låser og 2PL, mens Read-transaksjoner bruker multiversjon-CC.

Kapittel 22 – Gjenoppretting

Dette kapittelet ser på noen teknikker som kan brukes for gjenoppretting av databasen ved systemfeil. I kapittel 20 så vi på ulike typer feil og introduserte noen konsepter som brukes i gjenoppretting, for eksempel systemlogg og committ punkt. Vi skal nå se på flere konsepter som er relevante for gjenopprettingsprotokoller og gi et overblikk over de ulike gjenopprettings-algoritmene. Gjenoppretting er ofte knyttet sammen med mekanismer for samtidighetskontroll.

Konsepter ved gjenoppretting

Gjenoppretting fra transaksjonsfeil vil som regel bety at databasen blir gjenopprettet til den mest nylige, konsistente tilstanden før feilen skjedde. Systemet må derfor ta vare på informasjon om endringer som ble gjort av ulike transaksjoner, og denne informasjonen vil som regel lagres i **systemloggen**. To situasjoner som krever gjenoppretting:

- **Ved store skader på databasen som følger av katastrofale feil** (dvs. systemkrasj), vil gjenopprettingsmetoden hente en tidligere sikkerhetskopi av databasen fra arkivet. Metoden vil deretter rekonstruere en mer nylig tilstand ved å gjenta operasjoner til committet transaksjoner fra den sikkerhetskopierte loggen, opp til da feilen skjedde.
- **Ved ikke-katastrofal feil av type 1-4** (s. 153) vil gjenopprettingsstrategien være å identifisere noen endringer som kan forårsake inkonsistens i databasen. Hvis en ikke-committet transaksjon har oppdatert databaseelementer, må disse endringene fjernes. Hvis skriveoperasjonene til en committet transaksjon ikke har blitt skrevet til disken, må disse operasjonene gjøres på nytt. Dette blir gjort vha systemloggen som er lagret på disken.

Ved ikke-katastrofale transaksjonsfeil kan vi skille mellom to hovedteknikker for gjenoppretting:

1. **Utsatt oppdatering** – databasen på disken blir ikke oppdatert før transaksjoner har committet, så før dette vil oppdateringene være registrert i det lokale arbeidsområdet til transaksjoner eller i bufferne i hovedminnet. Oppdateringen av databasen blir altså utsatt til etter committ. Før committ vil oppdateringene registreres i loggfilen på disken og etter committ vil de skrives til databasen fra bufferne. Hvis transaksjonen feiler før den når committ, vil den ikke ha endret databasen på disken, så UNDO er ikke nødvendig. Det kan være nødvendig med REDO av effektene til operasjonene hos en committet transaksjon fra loggen, fordi det kan hende effektene ikke har nådd databasen på disken. Utsatt oppdatering kalles derfor **NO-UNDO/REDO algoritme**
2. **Umiddelbar oppdatering** - databasen på disken kan oppdateres av operasjoner hos en transaksjon før transaksjonen committer. Disse operasjonene må registreres i loggen på disken ved tvangsskriving (*force writing*) før de kan påføres databasen på disken, slik at gjenoppretting fortsatt er mulig. Hvis transaksjonen feiler før den committer, må transaksjonen rulles tilbake, altså endringene på databasen må angres. Umiddelbar oppdatering kan kreve både UNDO og REDO og kalles derfor **UNDO/REDO algoritme**. Denne metoden blir mest brukt i praksis. En variant av denne er UNDO/NO-REDO der alle oppdateringene må lagres i databasen før transaksjonen committer.

Gjenopprettingsprosessen (inkludert REDO og UNDO operasjoner) må være **idempotent**, som betyr at når operasjonen blir utført flere ganger er dette ekvivalent med at den utføres én gang.

Merk: REDO er nødvendig når transaksjonen har committet, men endringene har ikke blitt skrevet til databasen på disken. UNDO er nødvendig når endringene har blitt skrevet til databasen på disken og transaksjonen feiler.

Caching av diskblokker

Gjenopprettingen er koblet sammen med buffring av database disksider til hovedminnet. Disksidene vil inneholde dataelementene som skal oppdateres og de blir hentet inn i bufferne, oppdatert i minnet og deretter skrevet tilbake til disken. Denne prosessen blir håndtert av DBMS, som vil kontrollere samlingen av buffere som kalles **DBMS cache** og bruker en **cache katalog** for å holde styr over databaseelementene som er i bufferne. Når en operasjon skal bruke et element vil DBMS sjekke cache katalogen for å se om disksiden som inneholder elementet er i bufferen. Hvis ikke må elementet lokaliseres på disken og passende disksider må kopieres inn i cache. Hvis cache er full, må noen buffere erstattes.

Cache katalogen vil også inneholde mer informasjon som brukes for å kontrollere bufferne. For eksempel vil den inneholde en **dirty bit** som gir om bufferen har blitt endret og i så fall må skrives tilbake til disken før den kan erstattes. Transaksjonsid til transaksjonen(e) som endret bufferen blir inkludert. Katalogen kan også inneholde en **pin-unpin bit** som brukes for å holde tilbake bufferen dersom den ikke kan skrives tilbake til disken enda.

Når bufferen skal skrives tilbake til disken er det to strategier:
1. **In-place oppdatering** – bufferen skrives til samme disklokasjon og vil overskrive den gamle verdien til elementer som har blitt endret.
2. **Shadowing** – en oppdatert buffer skrives ved en annen disklokasjon, slik at det blir tatt vare på flere versjoner av dataelementene. Denne blir sjeldent brukt.

Den gamle verdien til dataelementet før oppdateringen kalles BFIM (*before image*), mens den nye verdien kalles AFIM (*after image*).

Write-Ahead logging, Steal/No-steal og Force/No-Force

Ved in-place oppdatering vil det være nødvendig å bruke en logg for gjenopprettingen. **Ved write-ahead logging (WAL) vil gjenopprettingsmekanismen sørge for at BFIM hos elementet blir lagret i en loggpost som sendes til disken før BFIM blir overskrevet av AFIM i databasen på disken**. Ved WAL vil altså den gamle verdien lagres i loggen før den nye verdien lagres i databasen. Dette er nødvendig for at vi skal kunne angre (UNDO) operasjonen ved en eventuell gjenoppretting. Før vi kan beskrive WAL protokollen må vi se på to ulike typer loggposter som inkluderes for skriveoperasjoner:

- **REDO-type loggposter** – gir den nye verdien (AFIM) til elementet skrevet av operasjonen, siden dette trengs for å gjenta (*redo*) effekten til operasjonen fra loggen (elementverdien i databasen settes lik AFIM)
- **UNDO-type loggposter** – gir den gamle verdien (BFIM) til elementet, siden dette trengs for å angre (*undo*) effekten til operasjonen fra loggen (elementverdien i databasen settes tilbake til AFIM)

I UNDO/REDO algoritmen vil loggposten inneholde både BFIM og AFIM.

Loggen som ligger lagret på disken vil være en sekvensiell (append-only) diskfil, og de siste blokkene i loggfilen vil som regel være lagret i loggbuffere i DBMS cache. Når en datablokk blir oppdatert vil den assosierte loggposten skrives til den siste loggbufferen. **Med WAL tilnærmingen må loggbufferne som inneholder de assosierte loggpostene for en bestemt datablokkoppdatering skrives til disken før datablokken kan skrives til disken fra bufferen**.

Begrepene steal/no-steal og force/no-force brukes for å spesifisere reglene som styrer når en blokk fra database cache kan skrives til disken:

- **Steal/no-steal** – når en transaksjon ønsker å lese et dataelement fra disken, men hovedminnet er fylt med elementer som andre transaksjoner arbeider med, må buffere i DBMS cache erstattes. En oppdatert bufferblokk må skrives til disken, og hvis transaksjonen som har endret bufferblokken ikke er committet sier vi at blokken er skitten. **Steal tilnærming** lar skitne blokker skrives til disken, mens **no-steal tilnærming** tillater ikke dette (dvs. den krever at transaksjonen som har oppdatert bufferblokken er committet før blokken kan skrives til disken). Vi sier at **steal lar en transaksjon stjele plassen til en skitten blokk i bufferen.** No-steal regelen gjør at UNDO ikke trengs, siden endringene til en transaksjon ikke blir lagret på disken før den har committet.
- **Force/No-force** – når en transaksjon committer vil **force tilnærmingen** kreve at alle blokker som er oppdatert av denne transaksjonen skrives til disken. Den alternative tilnærmingen er **no-force** (kun loggen skrives til disk). Force regelen gjør at REDO ikke trengs, siden alle endringene til en committet transaksjon er lagret på disken.

Utsatt oppdatering bruker no-steal, mens vanlige databasesystem bruker steal/no-force (UNDO/REDO) strategi. Fordelen med steal er at den unngår behovet for en stor buffer som kan romme alle skitne blokker. Fordelen med no-force er at en oppdatert blokk fortsatt kan være i bufferen når den trengs av en annen transaksjon. Dermed blir det mindre I/O kostnad, siden blokken ikke må skrives til og leses fra disken for hver gang den skal oppdateres.

For at gjenoppretting skal være mulig ved in-place oppdatering må bestemte poster lagres i loggen på disken før endringene gjøres på databasen. For eksempel kan vi se at WAL protokollen har følgende to regler:

Merk: regel 1 sørger for at operasjoner kan angres hvis transaksjonen feiler underveis, mens regel 2 sørger for at operasjoner kan angres/gjentas hvis systemet feilet etter transaksjonen har committet

1. **AFIM til et element kan ikke overskrive BFIM i databasen på disken, før UNDO-type loggposter (dvs. BFIM) har blitt *force*-skrevet til disken**. Dvs. loggposten som hører til en oppdatering av datablokken må skrives til disken før datablokken skrives til disken. Brukes for undoformål (gammel verdi trengs ved angring).
2. **Committ operasjonen til transaksjonen kan ikke fullføres før alle REDO-type og UNDO-type loggposter har blitt *force*-skrevet til disken**. Dvs. loggen må skrives til disken før transaksjonen committer. Brukes for redoformål (sikrer at alle committet transaksjoner kan gjenopprettes ved å gjenta operasjoner fra loggen).

Sjekkpunkt i systemloggen

For å gjøre gjenopprettingen mer effektiv vil DBMS opprettholde lister for aktive transaksjoner og transaksjoner som har committet eller abortert. DBMS vil periodisk lage en **[checkpoint] post** som består av et sjekkpunkt og en liste over alle aktive transaksjoner, og denne **lages hver gang alle oppdaterte DBMS buffere skrives til disken**. **Skriveoperasjonene til alle transaksjoner som har [commit, T] i loggen før [checkpoint] trenger ikke å gjentas ved systemkrasj, siden disse oppdateringene ble registrert i databasen ved sjekkpunkting.** Sjekkpunktposten inneholder en liste over aktive transaksjoner, slik at de lett kan identifiseres ila gjenopprettingen. DBMS må bestemme hvor ofte den skal lage et sjekkpunkt, noe som kan måles i tid (eks: per m minutt) eller antall committet transaksjoner siden sist sjekkpunkt. Følgende er prosessen ved sjekkpunkting:

1. Midlertidig utsett utføring av transaksjoner
2. *Force-write* alle oppdaterte buffere til disken

Merk: ARIES gjenoppretting som vi skal se på senere, vil ikke gjennomføre steg 2, altså skrive skitne blokker til disken

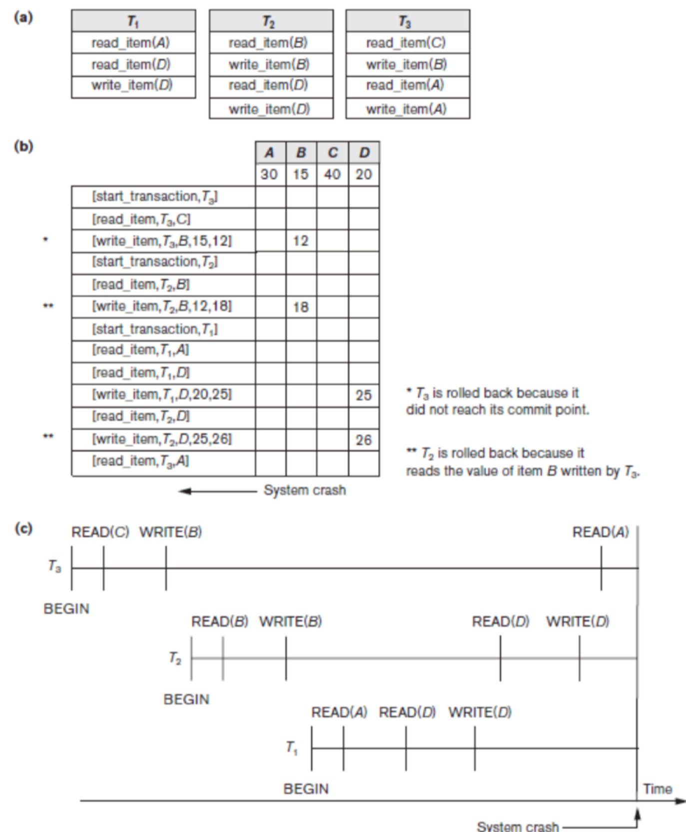
3. Skriv en [checkpoint] post til loggen og *force-write* loggen til disken
4. Gjenta utføring av transaksjoner

En [checkpoint] post kan også inneholde en liste over aktive transaksjonsid og lokasjonen til første og mest nylige poster i loggen for hver aktive transaksjon. Dette vil gjøre det lettere å rulle tilbake transaksjoner ved gjenoppretting.

Steg 2 kan føre til at transaksjonsprosesseringen blir forsenket. Ved fuzzy sjekkpunkt kan prosesseringen gjentas etter at en [begin_checkpoint] post har blitt skrevet til loggen og dermed slipper systemet å vente på steg 2. Når steg 2 er ferdig vil en [end_checkpoint] post skrives til loggen med relevant informasjon. Det er først da det nye sjekkpunktet erstatter det gamle.

Galopperende abort

Hvis transaksjonen feiler før den committer, kan det være nødvendig å rulle tilbake transaksjonen. Hvis noen dataelementer har blitt endret av transaksjonen og skrevet til databasen på disken, må disse elementene gjenopprettes til sine tidligere verdier (BFIM) vha UNDO-type loggposter. Dersom de oppdaterte dataelementene er lest av andre transaksjoner, må også disse rulles tilbake. Det samme gjelder andre transaksjoner som har lest fra disse, osv. Dette kalles **galopperende abort** (*cascading rollback*) og kan oppstå når gjenopprettingsprotokollen ikke er strikt eller ACA (s. 157). Dette kan være kompleks og tidskrevende, så de fleste gjenopprettingsmekanismene designes slik at det aldri blir nødvendig.



Figuren viser et eksempel på galopperende abort. Her vil transaksjon T_3 feile før den blir committet og må derfor rulles tilbake. Siden T_2 leser verdien til B etter den er skrevet av T_3, må også T_2 rulles tilbake. **Det er kun write_item operasjoner som må angres når en transaksjon rulles tilbake. read_item operasjonene inkluderes i loggen kun for å bestemme om galopperende abort av flere transaksjoner er nødvendig**. T_1 trenger ikke å rulles tilbake siden den ikke leser noen elementer skrevet av T_2 eller T_3 (merk den leser D før den blir skrevet av T_2).

Merk: dette vil aldri skje for strikt eller ACA historier! I dette tilfellet vil ikke read_item være nødvendig å inkludere i loggen.

Transaksjonshandlinger som ikke påvirker databasen

En transaksjon vil ofte ha handlinger som ikke påvirker databasen, for eksempel generering og printing av meldinger eller rapporter med informasjon hentet fra databasen. Hvis transaksjonen feiler før den har committet, kan det hende vi ønsker at brukeren ikke mottar disse, siden brukeren kan handle basert på dem. Slike handlinger bør derfor ikke skje før etter transaksjonen har committet, slik at de kan aborteres hvis transaksjonen feiler.

No-Undo/Redo gjenoppretting ved utsatt oppdatering

Ved utsatt oppdatering vil oppdateringen av databasen på disken utsettes til transaksjonen fullfører utføringen og committer. Den bruker altså en **no-steal tilnærming**, der skitne blokker ikke kan skrives til disken. Når transaksjonen utføres vil oppdateringene kun registreres i loggen og i cache bufferne. Ved committ punktet vil loggen tvinges (*force*) til disken, slik at oppdateringene registreres i databasen. **Hvis transaksjonen feiler før den committer vil det ikke være behov for noen UNDO-operasjoner siden ingen endringer er utført på databasen**. REDO-operasjoner vil derimot være nødvendig hvis systemet feiler etter en transaksjon har committet, men før endringene er registrert i databasen på disken. Loggen vil derfor kun inneholde REDO-type loggposter som inkluderer den nye verdien (AFIM) til elementet. Dette vil gjøre gjenopprettingen enklere, men **metoden kan ikke brukes med mindre transaksjonene er korte (raskt committer) eller oppdaterer få elementer.** For andre transaksjoner kan det hende det blir tomt med bufferrom, siden alle endringer må være lagret i bufferen til transaksjonene committer. Altså, mange buffere vil være *pinned* og kan ikke erstattes.

En protokoll for utsatt oppdatering kan være:

1. En transaksjon kan ikke endre databasen på disken før den committer. Alle buffere som har blitt endret av transaksjonen kan derfor ikke skrives til disken før transaksjonen committer (= no-steal)
2. En transaksjon vil ikke nå committ punktet før alle REDO-type loggposter er registrert i loggen og loggbufferen er tvunget til disken (tilsvarer WAL protokollen)

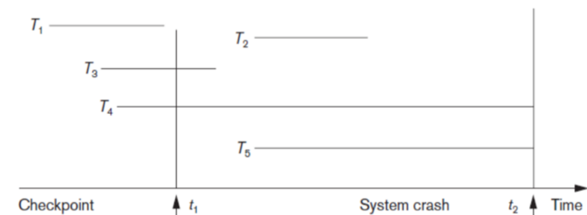
Samtidighetskontroll og gjenoppretting – utsatt oppdatering

Samtidighetskontrollen og gjenopprettingen er ofte flettet sammen. Vi ser på et system med samtidighetskontroll som bruker strikt 2PL, slik at skrivelåsene holdes til transaksjonene committer. Dette gjør at historiene er strikte og serialiserbare. Vi antar at [checkpoint] poster er inkludert i loggen, og får følgende No-Undo/Redo gjenopprettingsalgoritme:

Merk: Vranglås kan oppstå i strikt 2PL, så derfor kan abort av transaksjon være nødvendig

Algoritme RDU-M (No-Undo/Redo med sjekkpunkting)

Systemet opprettholder to lister med transaksjoner, der en er transaksjoner som har committet etter forrige sjekkpunkting og en er aktive transaksjoner. Transaksjoner som har committet før sjekkpunktet vil ikke gjentas. Skriveoperasjonene til transaksjoner som har committet etter sjekkpunktet må gjentas (*redo*) fra loggen i rekkefølgen de ble skrevet inn. Ved REDO vil elementene settes lik AFIM som gis i loggposten. Aktive transaksjoner blir kansellert og må utføres på nytt etter gjenopprettingen.

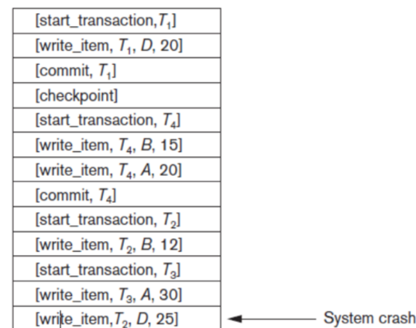


Figuren viser en tidslinje for en historie. Ved sjekkpunktet vil transaksjon T_1 være den eneste som har committet. Ved systemkrasjet har T_2 og T_3 committet, mens T_4 og T_5 har ikke det. I følge RDU-M metoden trenger ikke T_1 å gjentas, mens skriveoperasjonene til T_2 og T_3 må gjentas siden de ble committet etter sjekkpunktet. Husk at loggen blir tvunget til disken før transaksjonene når committ punktet. Transaksjonene T_4 og T_5 blir ignorert, siden no-steal tilnærmingen gjør at deres endringer ikke er registrert på disken (ingen UNDO). I tilfeller der et element har blitt oppdatert flere ganger, kan vi gjøre algoritmen mer effektiv ved å kun gjenta den siste oppdateringen (dvs. den siste skriveoperasjonen).

T_1	T_2	T_3	T_4
read_item(*A*)	read_item(*B*)	read_item(*A*)	read_item(*B*)
read_item(*D*)	write_item(*B*)	write_item(*A*)	write_item(*B*)
write_item(*D*)	read_item(*D*)	read_item(*C*)	read_item(*A*)
	write_item(*D*)	write_item(*C*)	write_item(*A*)

En ulempe med denne metoden er at samtidig utføring reduseres, siden skrivelåser holdes til transaksjoner committes. I tillegg krever det mye bufferrom å holde alle oppdaterte blokker helt til transaksjonen committer. **Fordelen er at transaksjoner aldri må rulles tilbake**, fordi den vil ikke endre databasen før den committer (ingen UNDO av operasjoner) og den vil ikke lese verdien skrevet av ikke-committet transaksjoner siden skriveoperasjoner er låst (ingen galopperende abort).

Figuren viser et eksempel for gjenoppretting som bruker denne metoden.


[start_transaction, T_1]
[write_item, T_1, *D*, 20]
[commit, T_1]
[checkpoint]
[start_transaction, T_4]
[write_item, T_4, *B*, 15]
[write_item, T_4, *A*, 20]
[commit, T_4]
[start_transaction, T_2]
[write_item, T_2, *B*, 12]
[start_transaction, T_3]
[write_item, T_3, *A*, 30]
[write_item, T_2, *D*, 25] ← System crash

T_2 and T_3 are ignored because they did not reach their commit points.

T_4 is redone because its commit point is after the last system checkpoint.

Gjenoppretting ved umiddelbar oppdatering

Ved umiddelbar oppdatering *kan* oppdateringen utført av transaksjonen umiddelbart lagres i databasen på disken, selv om transaksjonen ikke har committet enda. Legg merke til at det ikke er noe krav at alle oppdateringer skrives umiddelbart til disken, men det vil være tilfellet for *noen*. Metoden bruker en steal-tilnærming. Effekten til skriveoperasjoner på databasen må kunne angres ved transaksjonsfeil, altså transaksjonen må kunne rulles tilbake. Dette blir gjort vha UNDO-type loggposter som inneholder BFIM til elementet som har blitt oppdatert til AFIM.

Vi skiller mellom **to typer umiddelbar oppdateringsalgoritmer**:

1. **Undo/No-Redo gjenoppretting** – *alle* oppdateringene til en transaksjon blir registrert i databasen på disken før transaksjonen committer, slik at det ikke er behov for REDO-operasjoner hos committet transaksjoner. Metoden bruker **steal/force tilnærmingen for** å bestemme når oppdaterte buffere skal skrives tilbake til disken.
2. **Undo/Redo gjenoppretting** – *noen* oppdateringer blir registrert i databasen på disken før transaksjonen committer, mens andre blir registrert etter transaksjonen committer, slik at det blir behov for REDO-operasjoner hos committet transaksjoner. Metoden bruker **steal/no-force tilnærmingen for** å bestemme når oppdaterte buffere skal skrives tilbake til disken. Dette er den mest komplekse, men mest brukte metoden i praksis.

Samtidighetskontroll og gjenoppretting – umiddelbar oppdatering

Ved samtidige utføringer vil gjenopprettingen avhenge av samtidighetsprotokollen. Vi ser på et system med samtidighetskontroll som bruker strikt 2PL, slik at historiene er strikte (ingen lesing/skriving av element skrevet av ikke-committet transaksjon) og serialiserbare. Vi antar at [checkpoint] poster er inkludert i loggen, og får følgende Undo/Redo gjenopprettingsalgoritme:

Merk: Vranglås kan oppstå i strikt 2PL, så derfor kan abort og UNDO av transaksjon være nødvendig

Algoritme RIU-M (Undo/Redo med sjekkpunkting)

Systemet opprettholder to lister med transaksjoner, der en er transaksjoner som har committet etter forrige sjekkpunkting og en er aktive transaksjoner. To operasjoner:

- **Angre alle write_item operasjoner til aktive transaksjoner** vha UNDO prosedyren (elementet settes lik BFIM fra loggposten). Operasjonene angres i motsatt rekkefølge de ble skrevet inn i loggen. (dvs. fra slutt til start av loggen).
- **Gjenta alle write_item operasjoner til committet transaksjoner** vha REDO prosedyren (elementet settes lik AFIM fra loggposten). Operasjonene gjentas i samme rekkefølge de ble skrevet inn i loggen (dvs. fra start til slutt av loggen).

Som vi så på side 180 kan REDO-prosedyren bli mer effektiv dersom den starter ved enden av loggen og gjentar kun de siste oppdateringene av elementene. Når en oppdatering av et element er gjentatt vil elementet legges til en liste og blir dermed ikke gjentatt igjen (gjenoppretteren sjekker listen før senere REDO-operasjoner). UNDO-prosedyren kan bli mer effektiv dersom den starter ved begynnelsen av loggen og angrer kun den første oppdateringen av elementene. Et element som angres blir lagt til en liste og blir dermed ikke angret igjen.

Shadow paging

Shadow paging er en gjenopprettingsmetode som ikke bruker loggen ved enkeltbruker-miljø. Ved flerbruker-miljø kan det hende loggen trengs for samtidighetskontroll. Metoden ser på databasen som et fast antall diskblokker og en katalog med samme antall poster, der post i peker til databaseblokk i. Hvis det er plass vil katalogen holdes i hovedminnet og alle skrive- og lesereferanser til databasen går via katalogen. **Når en transaksjon begynner utføringen vil nåværende katalog kopieres inn i en skyggekatalog som er lagret på disken. Nåværende katalog blir deretter brukt av transaksjonen, mens skyggekatalogen blir ikke endret.**

Når write_item operasjonen utføres vil det lages en ny kopi av den oppdaterte blokken som blir skrevet inn i en ledig diskblokk (ingen peker i katalogen), slik at den gamle blokken ikke overskrives. Posten i nåværende katalog blir oppdatert til å peke mot den nye diskblokken, mens posten i skyggekatalogen vil fortsatt peke mot den gamle diskblokken (se figur). **Databasen på disken vil altså inneholde to versjoner av blokker som oppdateres av transaksjonen, og de to katalogene vil peke mot hver sin versjon**. Ved transaksjonsfeil vil gjenopprettingen gå ut på å slette de nye databaseblokkene og nåværende katalog. Tilstanden til databasen før transaksjonen er gitt av skyggekatalogen. Hvis transaksjonen committer vil skyggekatalogen slettes (pekere til gamle blokker fjernes, slik at de blir «ledige blokker»). Siden gjenopprettingen ikke involverer å angre eller gjenta operasjoner, kan shadow paging kategoriseres som **No-Undo/No-Redo gjenoppretting**.

Når transaksjoner kan utføres samtidig, må logger og sjekkpunkting inkluderes i shadow paging metoden. En ulempe med shadow paging er at de oppdaterte databaseblokkene endrer lokasjon på disken, fordi dette gjør det vanskelig å holde relaterte blokker nær hverandre. I tillegg vil skriving av katalogen til disken ha stor overhead hvis katalogen er stor. Håndteringen av søppelsamlingen når en transaksjon committer kan være komplisert. De gamle blokkene som skyggekatalogen refererer til må slippes fri og legges til en liste over ledige blokker.

ARIES gjenopprettingsalgoritmen VIKTIG

ARIES algoritme er en gjenopprettingsalgoritme som brukes i databasesystem. Den bruker en steal/no-force tilnærming for skriving og er basert på tre konsepter:

1. **Write-ahead logging** – BFIM hos elementet blir lagret i en loggpost som sendes til disken før BFIM blir overskrevet av AFIM i databasen på disken
2. **Gjenta historikk ved redo** – ARIES vil spore alle endringene på databasen før krasjet for å rekonstruere databasetilstanden ved krasjtidspunktet. Aktive transaksjoner ved krasjtidspunktet (dvs. ikke-committet) vil rulles tilbake
3. **Logge endringer ila. undo** – når en transaksjon skal rulles tilbake vil LastLSN brukes for å finne loggposten som sist ble utført av transaksjonen. Deretter vil den følge kjeden av loggposter via forrige LSN verdien til postene. **For hver loggpost vil ARIES lage en CRL (kompenserende loggpost) som gjør det motsatte av loggposten**. Ved å kjøre REDO på CRL postene vil transaksjonen rulles tilbake. Dette gjør at ARIES ikke må gjenta fullførte UNDO-operasjoner hvis gjenopprettingen må starte på nytt som følge av feil ila gjenopprettingen.

ARIES gjenopprettingen består av tre steg:

1. **Analysefasen** – identifiserer aktive transaksjoner og de skitne bufferblokker (dvs. blokker oppdatert av ikke-committet transaksjoner). Det blir også bestemt passende punkt i loggen der REDO-operasjonen bør starte.
2. **REDO fasen – kun nødvendige REDO-operasjoner blir utført**. Tidligere har REDO-operasjoner kun blitt brukt på committet transaksjoner, men i ARIES vil REDO-operasjonene begynne ved punktet bestemt i analysesteget og brukes til de når enden av loggen. ARIES bruker informasjon den har lagret for å bestemme om en operasjon fra loggen har blitt påført databasen og trenger derfor ikke å gjentas. Derfor er det kun nødvendige REDO-operasjoner som blir utført.
3. **UNDO fasen** – loggen skannes fra slutten og operasjoner som hører transaksjoner som var aktive ved krasjtidspunktet blir angret (*undo*). For å oppnå dette bruker ARIES sjekkpunkting og informasjon fra loggen, transaksjonstabellen og Dirty Page tabellen. Disse tabellene blir skrevet til loggen ved sjekkpunkting.

Loggen i ARIES

I ARIES vil alle loggposter ha et loggsekvensnummer (LSN) som øker og gir adressen til loggposten på disken. Hver LSN korresponderer til en spesifikk endring ved en transaksjon og hver datablokk lagrer pageLSN som peker mot loggposten som sist endret blokken. En loggpost blir skrevet for følgende handlinger: blokken oppdateres (skriving), transaksjonen committes, transaksjonen aborteres, oppdateringen angres (*undo*) og transaksjonen ender. Ved UNDO av en operasjon vil en loggpost skrives for å hindre at dette må gjentas. Loggposter vil inneholde transaksjons ID, forrige LSN for transaksjonen og type loggpost. **Det er viktig å inkludere forrige LSN fordi det kobler loggpostene sammen i reversert rekkefølge for hver transaksjon (det dannes en oppdateringskjede)**. Ved oppdatering (skriving) vil loggposten også inneholde blokkID til blokken som inneholder elementet, lengden til oppdatert element, offset fra begynnelsen av blokken (dvs. lokasjon), AFIM og BFIM.

Merk: *page* = blokk

Transaksjonstabellen og Dirty Page tabellen

I tillegg til loggen vil ARIES bruke transaksjonstabellen og Dirty Page tabellen for å oppnå effektiv gjenoppretting:

- **Transaksjonstabellen** – inneholder ett element per aktive transaksjon. Hvert element gir transaksjonsID, tilstand (dvs. aktiv, committet, abortert) og **LastLSN som peker til nyeste loggpost i transaksjonen**.
- **Dirty Page tabellen** – inneholder ett element per skitten blokk i bufferen. Hvert element gir blokkID og **RecLSN som peker til eldste loggpost som gjorde blokken skitten**.

Merk: blokken kan ha blitt oppdatert av tidligere endringer, men disse er skrevet til databasen på disken, og har derfor ikke gjort blokken skitten.

Sjekkpunkting

I ARIES vil sjekkpunkting gå ut på å:

1. Skrive en **begin_checkpoint** post til loggen
2. Skrive en **end_checkpoint** post til loggen, og legg til transaksjonstabellen og Dirty page tabellen i denne posten.
3. Skrive **LSN for begin_checkpoint** posten til en spesiell fil

Ved gjenoppretting vil den spesielle filen aksesseres for å lokalisere informasjon om siste sjekkpunkt. Fuzzy sjekkpunkting blir brukt for at DBMS skal kunne fortsette utføringen av

transaksjoner i løpet av sjekkpunktingen. **Siden transaksjonstabellen og Dirty Page tabellen gir informasjonen som trengs for gjenopprettingen, trenger ikke innholdet i DBMS cache å bli *flushed* til disken i løpet av sjekkpunktingen.** Ved krasj i løpet av sjekkpunktingen vil den spesielle filen referere til forrige sjekkpunkt, som dermed brukes for gjenoppretting.

ARIES gjenoppretting

Etter et krasj vil ARIES gjenopprettingsmanager ta over, og informasjon fra forrige sjekkpunkt blir aksessert gjennom den spesielle filen. Vi ser nærmere på de tre stegene.

Analysefasen

Analysen begynner ved begin_checkpoint posten og fortsetter til enden av loggen. Transaksjons-tabellen og Dirty Page tabellen aksesseres fra end_checkpoint posten. Analysen vil deretter gå igjennom etterfølgende loggposter som representerer utførte endringer etter forrige sjekkpunkt. **Tabellene som gis i end_checkpoint posten representerer aktive transaksjoner og skitne blokker ved tidspunktet sjekkpunktet ble laget, og analysen vil sørge for at tabellene endres basert på det som er gjort etter sjekkpunktet ble laget frem til krasjtidspunktet.** For eksempel hvis analysen finner en endeloggpost for en transaksjon, betyr det at transaksjonen har committet eller abortert etter sjekkpunktingen og transaksjonen blir derfor fjernet fra transaksjonstabellen. Hvis analysen finner en annen type loggpost for en transaksjon som ikke er tilstede i transaksjonstabellen, vil denne legges til tabellen og LastLSN blir oppdatert.

Når analysesteget er ferdig vil tabellene inneholde all informasjon som er nødvendig for at UNDO og REDO skal kunne gjenopprette tilstanden databasen hadde ved krasjtidspunktet.

REDO fasen

For å redusere mengden unødvendig arbeid vil ARIES starte REDO-operasjonen ved et punkt i loggen der den vet at tidligere endringer på skitne blokker har blitt påført databasen (unngår å gjenta operasjoner som allerede er gjentatt). **Dette punktet vil være M som er den minste RecLSN blant alle de skitne blokkene i Dirty Page tabellen**. Husk at RecLSN peker til den eldste loggposten som gjorde blokken skitten. Alle endringer som har $LSN < M$ må derfor allerede ha blitt påført databasen på disken eller blitt skrevet over, fordi hvis ikke måtte den tilhørende blokken vært i Dirty Page tabellen og $M = LSN$ (motsigelse).

REDO-operasjonen vil altså starte ved loggposten der $LSN = M$ og skanner fremover mot enden av loggen. **For hver endring registrert i loggen, vil REDO algoritmen verifisere om endringen har blitt gjentatt eller ikke**. Hvis endringen på en blokk allerede er skrevet til disken, vil Dirty Page tabellen enten ikke inneholde blokken eller inneholde blokken med $RecLSN > LSN(\text{endringen})$ (dvs. endringen som gjort blokken skitten skjedde etter den gitte endringen). I så fall vil ikke endringen gjentas av REDO-operasjonen. Hvis disse betingelsene ikke er oppfylt vil blokken leses fra disken. Hvis $\text{pageLSN} \geq \text{LSN}(\text{endring})$, betyr det at endringen har blitt påført databasen ved en tidligere anledning, siden pageLSN peker mot loggposten som sist endret blokken. I dette tilfellet vil ikke blokken skrives tilbake til disken. Hvis pageLSN $< LSN(\text{endring})$, vil endringen påføres blokken som deretter skrives til disken.

Merk: her er det snakk om pageLSN til diskblokken, altså loggposten som sist endret blokken før blokken ble skrevet til disken

UNDO fasen

Når REDO fasen er ferdig, vil databasen være i samme tilstand som ved krasjtidspunktet. I løpet av analysefasen har settet av aktive transaksjoner, kalt undo_set, blitt identifisert. **UNDO-fasen vil skanne loggen bakover fra enden og vil angre de passende handlingene, helt til den har gått igjennom alle transaksjonene i undo_set**. En loggpost blir laget for hver UNDO-operasjon. Når dette er gjort vil gjenopprettingen være fullført og normal prosessering kan begynne igjen

Eksempel – ARIES gjenoppretting

Figuren viser et eksempel på ARIES gjenoppretting når det er tre transaksjoner T_1 (oppdaterer blokk C), T_2 (oppdaterer blokk B og C) og T_3 (oppdaterer blokk A). Figur a viser deler av loggen, mens figur b viser innholdet i transaksjonstabellen og Dirty Page tabellen. Et krasj skjer og ARIES gjenopprettingsmanager tar over. Siden sjekkpunkting har skjedd vil adressen til begin_checkpoint hentes, og den er lokasjon 4.

(a)

Lsn	Last_lsn	Tran_id	Type	Page_id	Other_information
1	0	T_1	update	C	...
2	0	T_2	update	B	...
3	1	T_1	commit		...
4	begin checkpoint				
5	end checkpoint				
6	0	T_3	update	A	...
7	2	T_2	update	C	...
8	7	T_2	commit		...

(b)

TRANSACTION TABLE

Transaction_id	LastLsn	Status
T_1	3	commit
T_2	2	in progress

DIRTY PAGE TABLE

Page_id	RecLsn
C	1
B	2

(c)

TRANSACTION TABLE

Transaction_id	LastLsn	Status
T_1	3	commit
T_2	8	commit
T_3	6	in progress

DIRTY PAGE TABLE

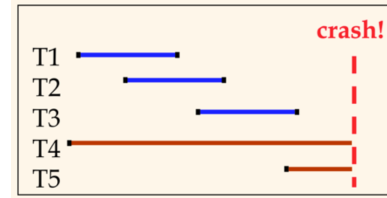
Page_id	RecLsn
C	1
B	2
A	6

De tre stegene blir:

1. **Analysefasen** – analysen begynner ved lokasjon 4 og fortsetter til den når enden av loggen. end_checkpoint inneholder tabellene som aksesseres.
   Når analysen møter loggpost 6 vil transaksjon T_3 legges til transaksjonstabellen og et nytt element for blokk A blir lagt til Dirty Page tabellen. Når analysen møter loggpost 8 vil LastLSN til T_2 i transaksjonstabellen endres til 8 (merk: Dirty Page tabellen blir ikke endret, siden den gir RecLSN som peker til eldste loggpost som gjorde blokken skitten). Figur c viser resultatet etter analysen
2. **REDO fasen** – den minste RecLSN i Dirty Page tabellen er 1, så loggpost ved lokasjon 1 vil være den første posten i loggen som gjorde en blokk skitten. REDO-operasjonen vil derfor starte ved loggpost 1. LSN verdiene $\{1, 2, 6, 7\}$ korresponderer til oppdateringer av hhv. blokk C, B, A og C, og siden de er større enn RecLSN til blokkene betyr det at disse endringene har skjedd etter blokken ble skitten. Blokkene blir derfor lest inn på disken og hvis pageLSN til diskblokken er mindre enn disse LSN verdiene, vil endringene påføres databasen på nytt.
3. **UNDO fasen** – siden T_3 er den eneste transaksjonen i transaksjonstabellen som har status *in progress* er det kun denne som blir rullet tilbake. UNDO-fasen begynner ved loggpost 6 (gitt av LastLSN) og fortsetter bakover langs oppdateringskjeden (s. 183) til T_3. I dette tilfellet er det kun loggpost 6 som blir angret (dvs. lages CRL post som gjør det motsatte).

Oppsummering – kapittel 22 (F20)

Databasesystem støtter sikker, atomisk aksess til store mengder data, altså må transaksjonene være atomiske (kjøre helt eller ikke) og de må ha durabilitet (endringene er permanente). Gjenoppretting brukes for å sikre dette.



Ved gjenoppretting blir transaksjonene delt inn i to grupper:

1. Vinnere = transaksjoner som har committet før krasjet (T1, T2, T3). Disse kan sendes til REDO-operasjoner.
2. Tapere = transaksjoner som ikke har committet før krasjet (T4 og T5). Disse må rulles tilbake med UNDO-operasjoner

Force/Steal-klassifisering

Merk: en blokk er skitten hvis den inneholder oppdateringer som ikke er skrevet til disken enda

Alle gjenopprettingsalgoritmer kan klassifiseres vha Force/No-force og Steal/No-steal tilnærminger. Utgangspunktet for denne klassifiseringen er hvor fleksibel buffermanageren til logging og gjenoppretting. For å klassifisere en gjenopprettingsalgoritme må vi se på når skitne blokker *kan* og når de *må* skrives til disken. Det er to begreper vi bruker:

- **Force** – skitne blokker som er endret av en transaksjon må tvinges til disken når transaksjonen committer. Blir sjeldent brukt fordi data er spredd over store deler av disken, så dette krever mye skriving til disken.
- **Steal** – en transaksjon kan stjele plassen til den skitne blokken i bufferen. Ved No-steal må en aktiv transaksjon holde alle skitne blokker i bufferen helt til den committer.

Figuren viser de fire klassene gjenopprettingsalgoritmer kan plasseres i. Den beste tilnærmingen er Steal/No-force som gir Undo/Redo logging og brukes av ARIES gjenoppretting som brukes i nesten alle moderne SQL system.

	No steal	Steal
Force	Shadowing (ikke logging)	Undo-logging No-redo
No-force	Redo-logging No-undo	Undo/redo-logging Aries

Write-ahead logging (WAL)

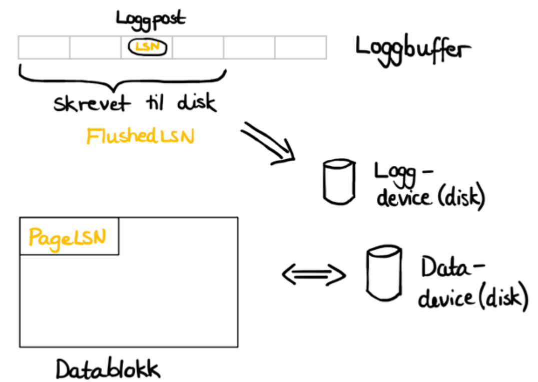
WAL er grunnlaget for undo/redo-logging, og det går ut på at alle endringer (insert/delete/update) blir representert som loggposter i loggen. Lesinger blir ikke logget. WAL protokollen består av to regler:

1. En loggpost som endret en datablokk må skrives til disken før datablokken skrives til disken. Dette er for undoformål
2. Loggen må skrives til disken før transaksjonen committer. Dette er for redoformål

WAL-konsepter i ARIES

ARIES bruker Write-ahead logging og følgende konsepter:

- **LSN** = loggsekvensnummer som er ID for loggposter i loggen. Dette er stigende nummer
- **PageLSN** = tilhører blokken og vil være LSN til loggposten som sist endret blokken. Brukes for å sjekke om en endring har blitt skrevet til disken ved gjenoppretting, ved å sammenligne endringens LSN med pageLSN.
- **FlushedLSN** = LSN til loggposten som sist ble skrevet til disken. Brukes for å sikre at loggen blir skrevet til disk før datablokken (WAL). Datablokken kan skrives til disken når pageLSN $<$ FlushedLSN, fordi da har loggen blitt skrevet til disken.



Loggbuffer er et området i minnet der loggpostene ligger etter hverandre. Hver loggpost har en LSN og FlushedLSN er lik LSN til den siste loggposten som ble skrevet til disken. Legg merke til at loggposter kan kun skrives til disken og ikke leses fra disken. Datablokken ligger i bufferen og har en pageLSN. Datablokker kan både skrives til og leses fra disken

Loggpost i ARIES

Figuren viser feltene i en loggpost i ARIES:

- **PrevLSN** = peker til forrige loggpost i samme transaksjon og brukes når transaksjonen rulles tilbake vha UNDO-operasjoner
- **OpType** = gir om loggposten representerer en update, insert eller delete
- **PageID** = gir hvilken blokk som ble endret (BlokkID)
- **Offset** = gir hvor i blokken det ble endret
- **BeforeImage** = verdi før endring
- **AfterImage** = verdi etter endring

LSN	TransID	PrevLSN	OpType	PageId	Offset	BeforeImage	AfterImage

Datastrukturer for gjenoppretting i ARIES

To datastrukturer som brukes i ARIES gjenoppretting er:

1. **Transaksjonstabell** – inneholder ett element per aktiv transaksjon. Hvert element gir transaksjonsID, tilstand (aktiv, committet, abortert) og LastLSN som peker til nyeste loggpost i transaksjonen (brukes i UNDO)
2. **Dirty Page tabell (DPT)** – ett element per skitten blokk i bufferen. Hvert element gir PageID og RecLSN som peker til eldste loggpost som gjorde blokken skitten (brukes i REDO)

Sjekkpunkting

DBMS vil periodisk lage et sjekkpunkt i loggen som skal minimalisere tiden det tar å gjenopprette databasen, siden den slipper å skanne gjennom hele loggen ved gjenoppretting. Ved sjekkpunkting vil det lages to loggposter:

1. Begin_checkpoint post
2. End_checkpoint post som inneholder transaksjonstabell og DPT, slik de var når sjekkpunktingen startet

LSN til sjekkpunktloggposten blir lagret på et sikkert sted (spesiell fil), og dette kalles et logganker. I noen systemer vil skitne blokker skrives til disk ved sjekkpunkting, men det er ikke tilfellet ved ARIES.

Abortering av transaksjoner

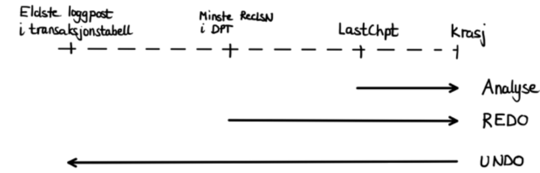
Når transaksjonen skal aborteres vil systemet finne LastLSN til transaksjonen fra transaksjonstabellen. Dermed vil den finne loggposten som sist ble utført av transaksjonen og den vil videre følge kjeden av loggposter via PrevLSN verdien i postene. For hver loggpost vil systemet lage en CRL (kompenserende loggpost) som vil gjøre det motsatte av loggposten (non-CLR). Istedenfor å fjerne effekten av en loggpost, vil systemet lage en loggpost som gjør det motsatte. Ved å gjøre REDO av alle CRL poster vil transaksjonen rulles tilbake og kan dermed fjernes fra transaksjonstabellen. Fordelen med CRL er at de gir grunnlag for låser på radnivå, som er mer presise enn låser på blokker.

ARIES gjenoppretting etter krasj

Målet med gjenopprettingen er å:

- Sørge for at vinnertransaksjoner (committet før krasj) er permanente
- Sørge for at tapertransaksjoner (ikke-committet før krasj) blir borte

ARIES gjenoppretting har tre faser:



1. **Analyse** – finner vinnere og tapere ved å rekonstruere DPT og transaksjonstabellen. Analysen bruker loggankret for å finne LastChpt (siste sjekkpunkt) og henter DPT og transaksjonstabellen. Deretter går analysen forover i loggen for å se på endringene som har blitt gjort etter sjekkpunktet og tabellene oppdateres.
2. **REDO** – REDO av nødvendige loggposter. REDO vil finne den minste RecLSN i DPT, fordi det er den eldste loggposten som kanskje ikke har blitt skrevet til disken før krasjet. Metoden vil deretter gå forover i loggen og utfører en redotest på LSN til hver loggpost
3. **UNDO** – UNDO av alle tapertransaksjonene. For en transaksjon som skal aborteres vil UNDO begynne i LastLSN og deretter følge kjeden av loggposter via PrevLSN verdien til postene. For hver loggpost vil det lages en CRL post som gjør det motsatte. Merk at dette kan gå langt tilbake i posten avhengig av hva som er transaksjonens eldste loggpost.

REDO av loggpost i ARIES (redotest)

Loggposten trenger ikke å gjøre en REDO hvis:

1. **Den tilhørende blokken er ikke i Dirty Page tabellen (DPT)**. Siden blokken ikke er i DPT er den skrevet til disken.
2. **Blokken er i DPT, men RecLSN for blokken er større enn loggpostens LSN**. Loggposten er eldre enn den eldste loggposten som gjorde blokken skitten, og må derfor være skrevet til disken.
3. **Blokkens pageLSN er større eller lik loggpostens LSN. Her må blokken leses inn fra disken**

Hvis ingen av disse betingelsene oppfylles må redo av loggposten gjennomføres:

1. AFIM blir skrevet inn i blokken
2. Blokkens PageLSN blir oppdatert til loggpostens LSN, siden det er den siste loggposten som endret blokken

Oppgave – ARIES gjenoppretting

Anta at følgende logg blir funnet etter krasj:

LSN	PrevLSN	TransID	Operation	PageID
237			End_ckpt	
238	0	T1	Update	A
239	0	T2	Update	B
240	239	T2	Commit	
241	238	T1	Update	A
242	0	T3	Update	B

A) Anta at Dirty Page Tabell (DPT) og transaksjonstabell (TT) i sjekkpunktloggposten med LSN 237 er tomme. Hvordan ser DPT og TT ut etter analyse i Recovery er ferdig? DPT har feltene PageID og RecLSN, mens TT har feltene TransID, LastLSN og status

Analysen henter frem de tomme tabellene fra loggpost 237 og vil deretter gå igjennom alle loggpostene helt til den når enden av loggen. Hvis den møter en transaksjon som ikke er i TT vil denne legges til TT. Hvis den møter en transaksjon som er i TT blir elementet i TT oppdatert. Vi får:

TransID	LastLSN	Status
T1	241	Active
T2	240	Commit
T3	242	Active

Når vi møter en ny loggpost for samme transaksjon blir LastLSN og eventuelt statusen oppdatert. Vi bruker Status Active for Update og Committ

PageID	RecLSN
A	238
B	239

I DPT vil ikke RecLSN oppdateres (det er den første verdien)

B) Hva skjer under UNDO fasen av recovery? Hvilke transaksjoner rulles tilbake og hvilke nye loggposter lages?

Ved UNDO fasen må alle aktive transaksjoner i TT rulles tilbake. For hver aktive transaksjon vil metoden bruke LastLSN for å finne loggposten som sist ble utført av transaksjonen. Deretter vil den følge kjeden av loggposter via PrevLSN og for hver loggpost frem til den eldste i kjeden vil den lage en CRL post som gjør det motsatte. Dermed blir transaksjonene rullet tilbake. Vi får følgende nye CRL loggposter:

LSN	PrevLSN	TransactionID	Operation	PageID
243	242	T3	CLR	B
244	243	T3	Abort	
245	241	T1	CLR	A
246	245	T1	CLR	A
247	246	T1	Abort	

Første CRL post hos en aktiv transaksjon vil ha PrevLSN = LastLSN fra transaksjonstabellen. CRL posten til neste loggpost i kjeden vil ha PrevLSN = LSN til forrige CRL. Når enden av kjeden er nådd, vil det lages en CRL abort post, der PrevLSN = LSN til forrige CRL. LSN vil være monotont stigende

Andre gjenopprettingsteknikker

Noen andre gjenopprettingsteknikker er:

- **UNDO/NO-REDO**: som ARIES, men kun UNDO-logging
- **NO-UNDO/REDO**: som ARIES, men kun REDO-logging
- **Shadowing** – bruker ikke logging, men lager kopier av datablokker ved oppdatering. Transaksjoner blir committet ved å kopiere inn pekere til nye datablokker. Dette krever en katalog som inneholder pekere til datablokker

Vi skiller mellom in-place oppdatering, der dataen blir oppdatert der den ligger, og shadowing, der dataen lagres i nye datablokker.