Arbeidsnotater fra Høgskolen i Buskerud nr. 71

ARBEIDSNOTAT

ARBEIDSNOTAT

Praktisk database-administrasjon med persistens av objektorienterte systemer

Kompendium

Knut W. Hansson

HØGSKOLEN i Buskerud

Arbeidsnotater fra Høgskolen i Buskerud

Nr. 71

Praktisk database-administrasjon med persistens av objektorienterte systemer

Kompendium

Av

Knut W. Hansson

Hønefoss 2012

Tekster fra HiBus skriftserier kan skrives ut og videreformidles til andre interesserte uten avgift.

En forutsetning er at navn på utgiver og forfatter(e) angis- og angis korrekt. Det må ikke foretas endringer i verket. Verket kan ikke brukes til kommersielle formål.

ISSN 1893-2398 (online)

HØGSKOLEN
i Buskerud

Praktisk database-administrasjon

med persistens av objektorienterte systemer

Kompendium

Knut W. Hansson
Førstelektor IT

6 November 2012

HiBus publikasjoner kan kopieres fritt og videreformidles til andre interesserte uten avgift.

En forutsetning er at navn på utgiver og forfatter angis – og angis korrekt. Det må ikke foretas endringer i verket.

Emneord:
databaseadministrasjon
objektorientert
lagring
database
eksempler
Oracle
C#, C sharp

English keywords:
database administration
object oriented
storing
database
examples
Oracle
C#, C sharp

Innhold

SAMMENDRAG/SYNOPSIS

Sammendrag

Ved høyskolen i Buskerud, bachelorstudiene i IT, undervises kurset "Praktisk databaseadministrasjon". Kurset består av to hoveddeler: En del fokuserer på teori om store databaser i praksis og en annen del er praktisk med bruk av Oracle (PL/SQL og objekter) og persistens av objektorienterte systemer med Visual C# og NoSQL teknikker.

Den teoretiske delen foreleses og er gjenstand for vanlig, skriftlig, individuell eksamen.

Den praktiske delen undervises med sterk vekt på det praktiske og med øvinger og obligatoriske gruppeoppgaver.

Dette kompendiet inneholder alle forelesningene mine i kurset, samt noe tilleggsstoff for spesielt interesserte.

Det er laget en egen elektronisk lærerressurs med oppgavene og løsningsforslag for den praktiske delen.

Synopsis in English

At Buskerud College, the course "Practical data base administration" is part of the bachelor IT education. The course consists of two main parts: One part is concentrated on theory of large databases in practice and the other part is practical using Oracle (PL/SQL and objects) and persistence of object-oriented systems with Visual C# and NoSQL techniques.

The theoretical part is taught in lectures with a written, individual exam at the end.

The practical part is taught with a strong emphasis on the practical aspects and with exercises and compulsory group assignments.

This compendium contains all my lectures in the course, and some additional subjects for interested students.

An electronic teacher's resource has been made with the assignments and suggested solution for the practical part of the course.

DEL 1 TEORI: FORELESNINGER I TILKNYTNING TIL LÆREBOKEN

Som grunnlag for forelesningene brukes læreboken R. Elmasri & S. B. Navathe: "Fundamentals of Database Systems", 6th ed., Addison-Wesley, 2011, ISBN: 9780136086208. Tidligere utgave kan ikke brukes, da det er strukturelle forskjeller.

Nedenfor henvises til læreboken som "E&N".

E&N 16: Lagring, filstrukturer og hashing

Typiske databaseoppgaver

I relasjonsteorien snakker vi om tupler som samles i relasjoner (repetisjon senere), som logisk sett lagres i tabeller med rader. På en harddisk vil relasjonene/tabellene bli lagret som *poster* i *filer*.

Typiske databaseoppgaver er

• lagre ny post (insert) • endre eksisterende post (update) • slette post (delete) • finne poster (select)	Alle disse er tidskritiske, og alle krever at poster finnes for å sikre integritet. Rask gjenfinning er avgjørende for alt, enten problemstillingen er ”finnes denne” (ref. "Autopass" når en bil fyker forbi) eller ”hent denne”.
• håndtere transaksjoner, dvs. grupper av oppgaver som utgjør et logisk hele (begin transaction/commit/rollback)	Tidskritisk. Krever overhead da ”gamle” data må huskes for å kunne endre dem tilbake ved rollback.
• administrasjon av metadata (create/drop/alter table/index/user/database) • administrasjon av databasen (startup/alter/shutdown backup/recover)	Oppgaver for databaseadministrator. Disse er ikke tidskritiske og noen av dem er også ganske tidkrevende, f.eks. backup/recover.

Det kan synes som harddisker er raske, men i forhold til serverens kapasitet ellers, er harddisker svært trege. Hvordan data kan gjenfinnes raskt, er derfor helt avgjørende for svartider og kapasitet. Det er også slik at mange av databaseoppgavene krever betydelig overhead av hensyn til sikkerhet f.eks. adgangskontroll, rettighetskontroll, integritetskontroll (entitetsintegritet, referanseintegritet, domeneintegritet), håndtering av køer, logging osv. Praktiske databaser får påtrykk fra mange klienter som vil ha noe gjort, men flesteparten av dem er vanligvis spørringer. Uansett er gjenfinningstiden kritisk, og derfor legges det betydelig vekt på å organisere dataene slik at gjenfinning kan gjøres kjapt. Dette kan gjøres på flere måter med forskjellige konsekvenser, og det er noe av det som dette emnet dreier seg om.

Litt repetisjon av relasjonsteorien

Relasjoner

En relasjon (også kalt relasjonstabell) defineres av C. J Date som en **heading** og en **body**:

1. Heading er en mengde med n attributter (A_i, T_i) der A_i er attributtnavnet og T_i er et typenavn (domene)
2. Body er en mengde med m n-tupler (A_i, v_i) der A_i er attributtnavnet og v_i er den atomære verdien og kan være *NULL*. m (antallet i mengden) er kardinaliteten og n er graden.

Relasjonen er navngitt.

Tabeller

Vi tenker litt enklere på dette som en **navngitt tabell** med **overskrift** og **rader**.

tblPerson

nr - PK (heltall >10)	navn (streng 30 tegn)	tlf (heltall 8 siffer)	postnr - FK (heltall 4 siffer)
56	Knut	12345678	1532
32	Kari	87654321	0280
99	Ole	44332211	0280
15	Nina	99998888	NULL

tblPost

postnr - PK	sted
0280	Oslo
4450	Kompe
1532	Latia

Overskriften er en sekvens av tekster og tolkes som navnet på kolonnen og dermed også navnet på verdien i radene under. Hver rad utgjør en sekvens med atomære verdier i samme rekkefølge som overskriften. Det er en *mengde* med slike rader – altså er ingen av dem like og de er uten rekkefølge. Vanligvis angir vi jo bare navnet på kolonnen og ikke typen.

Siden radene i tabellen er uordnet, er det svært tungt å finne igjen én, bestemt rad. Gjennomsnittlig må man gå igjennom halvparten av radene for en finne en bestemt rad, og man må gå igjennom alle for å være sikker på at den ikke finnes. Vi kan f.eks. ikke være sikker på at personen med *nr* = 70 ikke finnes uten å sjekke samtlige.

Integritet

Databasen krever at vi angir at én - eller flere kolonner tilsammen er **primærnøkkel** (primary key). Det er det samme som å angi at alle verdiene i kolonnen – eller kolonnene til sammen – skal være forskjellige (unike) og ingen av verdien skal være *NULL*. Hvis vi ikke finne på noe annet, kan vi angi at *alle* kolonnene i tabellen inngår i primærnøkkelen, men det er ikke særlig praktisk og har betydelige ulemper ved gjenfinning (dette kommer jeg tilbake til). Dette sikrer **entitetsintegriteten**, dvs at ingen rader er like – de skal jo utgjøre en mengde. Ovenfor er *tblPerson.nr* og *tblPost.postnr* angitt som primærnøkler (PK).

Det er gjerne sammenhenger mellom rader. De knyttes sammen ved at den ene raden har en verdi, som gjenfinnes i én – og bare én – annen rad i en annen eller samme tabell. Den kolonnen som inneholder referansen, deklareres som **fremmednøkkel** (foreign key, FK). Den skal vise til én og bare én annen rad, og derfor refererer den alltid til en primærnøkkel. Grunnen er at hvis den henviste til flere, ville informasjonen være tvetydig (ambiguous). Ovenfor er *tblPerson.postnr* fremmednøkkel som henviser til *tblPost.postnr*. Verdien for en fremmednøkkel kan godt være *NULL* – den henviser da ikke til noen annen rad. Den behøver ikke være unik – mange rader i *tblPerson* kan henvise til samme rad i *tblPost*. Det er et absolutt krav at fremmednøkkelverdien (når den ikke er *NULL*) gjenfinnes i den kolonnen den henviser til. Dette sikrer **referanseintegriteten**. Primærnøkkelen som det blir henvis til, behøver ikke bli henvist til av noen. Ovenfor ser vi at fremmednøklene for person nr 56 og 32 henviser til hver sin rad i *tblPost*, men ingen henviser til *postnr* 4450.

Det angis hvilke verdier som er lovlige for hver kolonne. Dette kalles **domene**. Det enkleste er å angi en datatype som databasen kjenner, som f.eks. streng, Boolsk verdi, byte, positivt heltall, heltall, reelt tall, BLOB og CLOB. Videre kan man angi **beskrankninger** (også kalt skranker eng. constraint) dvs. krav til verdien som ytterligere innskrenker hvilke verdier som er lovlige. Enkle eksempler kan være heltall med fire siffer eller streng med 32 tegn o.l. En svært vanlig beskrankning er at verdien skal være unik. Fremmednøkler har en slik beskrankning ved at verdien skal finnes igjen i primærnøkkelen som den henviser til. Andre beskrankninger kan være at verdien i én kolonne skal være større enn verdien i en annen kolonne på samme rad, at første tegn skal være ”A”, at strengen skal inneholde ”@” osv.

I tillegg angir man om *NULL* er lovlig (standard) eller ikke – husk da at *NULL* ikke er en verdi men bare *mangel* på verdi. Et domene som beskrankes med unik og ikke *NULL*, utgjør en **alternativ**

nøkkel – den har samme beskrankninger som primærnøkkelen men er ikke valgt som det. Hvis alle verdier er lovlige i henhold til domenet, er **domeneintegriteten** overholdt.

Filer
P.g.a av størrelsen og problemer med tap av data hvis maskinen går ned, kan data ikke lagres permanent i RAM. Det er mulig å lagre sikkerhetskopier på tape, CD eller DVD, men de er vanskelige å oppdatere og altfor trege. Dessuten er de offline og må monteres ved behov. I praksis brukes derfor naturligvis harddisker som står online.

Konkret lagres dataene i databasen hverken i relasjoner (en abstrakt fremstillingsmåte) eller tabeller (en logisk fremstillingsmåte), men *fysisk* i filer. Da skilles tabelloverskriftene fra radene. Overskriftene utgjør en del av **metadata** og lagres i egne filer (som også tilsvarer tabeller) som kalles **skjema** (schema av gresk = form, plan). Dataene lagres rad for rad i **poster** som består av verdier i **felt**, vanligvis med postene fra én tabell i hver fil.

Selvom tabellens rader utgjør en mengde, uten rekkefølge, vil det i en fil oppstå en **rekkefølge** mellom postene og posten får en **fysisk referanse** (sted på harddisken der den er lagret). Man bestemmer rekkefølgen bevisst, utfra behov. Vanligst er å tenke mest på hvor raskt dataene kan finnes igjen, men det vil ha konsekvenser for tiden det tar å endre dem (insert, update og delete). Systemer med svært mye endringer i data, vil derfor organisere filene annerledes. Det kan også være aktuelt å ha forskjellig organisering i samme database – f.eks. oppdateres en fil med postnr/poststed sjelden, mens fil med bestillinger oppdateres mye.

16.2. Sekundærlager
Det primære her å merke seg at harddisker må formatteres. Da bestemmes størrelsen på den minste, adresserbare enheten – en **blokk** (page). Alle blokker er like store og de er faste – de kan ikke endre fysisk plassering og størrelse uten reformatering, og da tapes alle data som er lagret der.

Harddisken har egen programvare – ofte en SCSI – som får oppgitt et blokknummer, posisjonerer lese-/skrivehodet og leser/skriver data fra/til denne blokken. Det tar typisk millisekunder (f.eks. 3 ms for en god server) å posisjonere lese-/skrivehodet over en gitt blokk, men mye mindre (f.eks. 0,4 ms) å lese/skrive. Hvis man skal lese flere blokker, vil man derfor spare svært mye hvis man kan lese blokker som fysisk ligger etter hverandre på harddisken – såkalt **kontinuerlige** (contiguous) **filer**. Man vil da også spare mye overhead.

Hvis man f.eks. skal ha tak i den femte blokken i en fil, og den er kontinuerlig, så kan man regne ut blokkens adresse utfra hvilken blokk filen starter i (hvis første blokk har adresse 15, vil den femte blokken ha adresse 19). Det vil kunne ta 3+(0,4*5)=5 ms.

Hvis filen ikke er kontinuerlig, vil det på slutten av hver blokk ligge en adresse som angir hvor neste blokk ligger. For å finne femte blokk, må man da første lese første blokk, finne adressen til andre blokk som inneholder adressen til tredje blokk osv. Hver gang må lese/skrivehodet reposisjoneres og innholdet behandles. Det vil kunne ta (3+0,4)*5=17 ms. Med en fil på f.eks. 1 000 blokker (á 0,5 Mb = 500 Mb fil), blir forskjellen stor (403 ms=0,4 sek mot 3400 ms=3,4 sek).

Intet er gratis her i verden(?). Man må gi for å få. TANSTAFL (”There ain’t no such thing as a free lunch”, Robert Heinlein, Milton Friedman m.fl.). Man "tar igen på gungorna vad man förlorar på karusellen” (Tivoliprinsippet).

Slik er det også med kontinuerlige filer. *Fordelen* er altså at man får rask lesing. *Ulempen* er at de er vanskeligere å plassere på en harddisk. Når de øker i størrelse må ofte hele filen flyttes til et annet sted

der det er plass til en ekstra blokk. Dette er meget tidkrevende. Som et kompromiss kan man enten sette av ekstra blokker med en gang (sløsing med plassen) eller godta at noe av filen er kontinuerlig mens ekstra blokker legges her og der – da må det holdes styr på hvor mange av blokken som er i den kontinuerlige delen (gir overhead). I det siste tilfelle kan filen gjøres kontinuerlig igjen fra tid til annen, når det blir for galt.

En annen og vanlig mellomting er å gruppere et antall blokker kontinuerlig med en peker fra gruppe til gruppe. Slike grupper kalles klustere (clusters) eller filsegmenter.

Magnetisk tape brukes fortsatt, men bare til sikkerhetskopier da de er håpløst tungvinte ved gjenfinning/endring av enkeltposter og altfor trege. De egner seg bare for batch-jobber.

16.3 Bufring

Dataene fra harddisken leses til et buffer (cache) i RAM. Vi kan da øke hastigheten betydelig ved å ha flere buffere. Da kan innholdet i et buffer prosesseres mens neste hentes osv. Bufferet kan gjerne inneholde plass til flere blokker, men må kunne ta imot minst én hel blokk.

16.4 Lagring av poster

Her gjelder i høy grad TANSTAFL og man "må velge mellom Belsebub og djevelen".

Postens størrelse

Det er flere sider ved dette:

- Fast størrelse på hvert felt, gjør det raskere å finne et felt inne i en post – man kan regne ut hvor feltet begynner og slipper skilletegn mellom feltene. På den annen side vil ”Are” da ta like stor plass som ”Anne-Sofie Justine Kristine”, så man sløser med plassen.
  Med variabel lengde på feltene, behøver man ikke bruke flere bytes på harddisken enn nødvendig og man behøver ikke ha noen egentlig maksimalgrense. Det er særlig tekstfelt dette er aktuelt for, da andre datatyper gjerne har fast lengde uansett (heltallet 0 tar like stor plass som heltallet 987654321).
  Skilletegnene må være tegn som ikke under noen omstendighet kan brukes til annet. Det er vanskelig å være sikker på, for i strenger kan jo hva som helst legges inn. Derfor vil gjerne strenger omsluttes av anførselstegn som også tar plass (hvis man vil ha anførselstegn inne i en streng, bruker man da en eller annen sekvens av tegn – jfr. VB som bruker dobbelt anførselstegn inne i en streng). En fordel med skilletegn er at man kan ha varierende antall verdier for et felt, f.eks. kan man lagre fra ett til fem telefonnummer for en kunde inne i posten. Verdien er da ikke lenger atomær, og følgelig unormalisert, men det kan være praktisk og gir raskere gjenfinning enn om telefonnumre må lagres i egen fil (tabell).
  Det er fullt mulig å ha noen felt med variabel lengde (f.eks. strengene) og noen med fast lengde (f.eks. alle tall).
- Fast størrelse på postene. Dette vil kreve at alle feltene har en begrenset maksimallengde (men størrelsen behøver ikke være fast). Det blir da raskere å finne en post hvis man vet hvilken plass i filen posten har – man ganger lengden med plassen og vet da hvor i filen posten begynner. Man slipper også skilletegn mellom postene. På den annen side vil også dette sløse med plassen.

Avveining av alt dette kan selvsagt variere, men da de fleste databaser jobber mest med gjenfinning, vil man gjerne ha *fast postlengde*. Da kan man like godt ha *fast feltlengde* også – man vinner ingen plass på harddisken ved å ”pakke” posten og man slipper skilletegn mellom feltene. Når man har fast felt- og postlengde, kan man ikke trikse med ikke-atomære verdier – de må legges i egne filer hvilken gir økt tilgangs- og behandlingstid.

BLOBS og CLOBS vil uansett gjerne lagres i separate filer (gjerne én for hver CLOB/BLOB), da maksimal størrelse er ukjent og de varierer svært meget i størrelse.

Antall poster pr blokk
Enten postene har fast (som er vanlig) eller varierende størrelse, må man tenke igjennom hvordan postene plasseres i blokkene.

Hvis man bestemmer seg for et fast antall poster i hver blokk, må postene ha en maksimal lengde, men de behøver ikke å være like lange. Dette sløser med plassen, men man kan til gjengjeld regne ut hvilken blokk en post ligger, hvis man vet hvilken plass i filen posten har. Poster som evt. deles over to blokker (begynnelsen av posten i én blokk, resten i den neste) sies å være "spanned".

Siden gjenfinning er kritisk (og plass ikke er det) er det vanlig å ha et fast antall poster pr blokk. Man vil velge å *unngå "spanning"* for da kreves det to aksesser for å lese én post.

Eksempel

filPerson (navn har variabel lengde, variabel postlengde og spanning i diskontinuerlig fil)

Blokk 692							Blokk 803					
5	Knut\\	1234567	153$	32	Kari\\	…	8765432	0280$	99	Ole\\	4433221	0280

filPerson (alt er fast og filen kontinuerlig)

Blokk 692					Blokk 693				
5	Knut	1234567	153		32	Kari	8765432	0280	

16.5 Filoperasjoner
Når databasesystemet skal gjennomføre en kommando gitt i et overordnet språk, f.eks. SQL, må den gjøre det om til filoperasjoner for den filen som inneholder data fra den tabellen vi har henvist til.

Det benyttes en "cursor" som angir hvilken post i filbufferet som til enhver tid er den aktuelle (current).

Noen filoperasjoner opererer med én post, andre med flere. F.eks. vil *delete* slette aktuell post fra bufferet og senere oppdatere filen på harddisk, mens *findall* vil finne alle poster som tilfredsstiller gitte kriterier.

Les selv om filoperasjoner i kapittel 16.5.

16.6 – 16.8 Organisering av postene i filen
Siden postene på disk har en rekkefølge (i motsetning til postene i en relasjon/tabell), så reises spørsmålet om hvordan postene skal legges ut i filen. Man har gjerne én primær organisering av postene i filen (primary file organisation) men legger opp andre måter å finne poster på i tillegg – gjerne indekser. Indekser kommer jeg tilbake til, men vi kan straks legge merke til at indekser vil gjøre oppdateringer tyngre, men tilgangen raskere – primærnøkler indekseres uansett alltid fordi det så ofte er behov for å sjekke entitets- og referanseintegritet ("finnes denne fra før" eller "finnes fremmednøkkelens verdi nå blant primærnøklene som det henvises til").

Postene kan organiseres innen filen på flere måter:

- 16.6: Uordnet = Heap files
- 16.7: Sortert = Sorted files
- 16.8: Hashed
  - 16.8.1: Internal
  - 16.8.2: External – statisk og dynamisk der dynamisk kan være utvidbar eller lineær.

16.6 Uordnet = heap

Filen ordnes ikke på noen spesiell måte - nye poster legges til bakerst.

Fordel:

- Meget rask innsetting av nye poster

Ulempe:

- Meget tungt å finne en post

Hva gjør vi når en post slettes? Vi kan

1. Pakke filen, dvs fysisk fjerne den slettede posten så filen blir mindre. Dette er håpløst tungt. Enten må man kopiere (nesten) hele filen, eller man må flytte alle poster bak den slettede ett hakk lenger frem.
2. Etterlate plassen tom (nullet ut) og evt. bruke plassen om igjen senere – hvis det er plass til den nye posten der (og det vil det være jo ved fast poststørrelse).
3. La posten ligge, men merke den som slettet med et eget tegn. Da kan posten senere ”reddes”. Evt. kan man bruke plassen senere til en ny post (hvis det er plass).

Med teknikk 2 og 3 må filen fra tid til annen "pakkes" for ikke å vokse over alle grenser.

16.7 Sortert fil

Vi kan sortere postene etter ett/flere *sorteringsnøkkelfelt*. Det behøver ikke være et unikt (som primær eller alternativ nøkkel) da det ikke gjør noe om flere med samme sorteringsnøkkel ligger etter hverandre, evt. kan man sekundært sortere på et annet felt. Det er slik telefonkatalogen ble sortert.

Fordel:

- Det er enkelt og raskt å hente postene i den sorterte rekkefølgen.
- Raskt å finne frem en post etter nøkkelverdien, særlig hvis postene har fast lengde uten spanning.

Ulempe:

- Sortering gir ingen fordel ved søk etter annet enn nøkkelfeltet (jfr. telefonkatalogen: Hvem har telefonnummer 33224433?)
- Innsetting må skje på ”rett plass” og det er meget tungt. Vi må finne rett plass (relativt raskt med binærsøk/halveringsmetoden) og så må det lages plass. Det siste innebærer gjennomsnittlig flytting av halvparten av postene.

Hva med sletting? Det blir på samme måten som for usortert fil.

Det er en mulighet å sette av åpne poster for senere innsetting, men det hjelper bare en stund. En mulighet er også en egen overflow-fil, men søkingen blir da tyngre (da legges nye i denne overflow-filen og alt reorganiseres jevnlig).

Oppdatering av en post går greit, hvis ikke den endrede posten er større enn den gamle.

16.8.1 Hashing internt, dvs i RAM

Når data er i RAM – enten fordi de inngår i et program eller fordi vi har lest inn hele filen – kan de struktureres med *hashing* (kalles av noen for *nøkkeltransformasjon*). Det foregår slik at man bestemmer et eller en kombinasjon av felt å hashe på, kalt *hashfelt*. Hashfeltet brukes som variabel i en funksjon – *hashingalgoritmen* – som gir et heltall som resultat. Evt. brukes ANSII-kodene for tegn. F.eks. vil algoritmen x MOD 13 gi rest i intervallet [0..12]. Hashverdien angir da plasseringen (indeksen) i en array. Det er vanlig å benytte et primtall i divisjonen, for det gir bedre "spredning" av hashverdiene.

Fordel:

- Rask innsetting og gjenfinning

Ulempe:

- Hashingen gir overhead

Et problem oppstår ved kollisjoner – når flere poster får samme hashverdi. Det finnes flere, mulige løsninger:

1. Open adressing: Prøv neste til du finner en ledig plass. Denne er enkel, men for å være sikker på at en gitt post ikke finnes, må du søke fra "rett plass" til første, ledige plass.
2. Chaining: Legg til en peker i hver post med standardverdi tilsvarende *NULL*. Hvis plassen som den nye posten skal inn i er opptatt, legges den nye i første, ledige plass i et overflow-område og den som lå i veien settes til å peke dit. Når neste får samme problem, legges peker fra den første til den andre osv.
3. Multiple hashing: Hvis plassen er opptatt, så prøv en annen algoritme, så en tredje osv. og bruk open adressing i nødsfall til slutt.

Det bør være litt ledig plass i arrayen – 70% til 90% er erfaringsmessig OK og man vil jo ikke ha altfor mange ledige. Altså setter man f.eks. av $\frac{n}{0,7}$ plasser for *n* poster = 42% for mange, mens $\frac{n}{0,9}$ gir 11% for mange. Antall plasser bør være et primtall, siden det gir bedre spredning av hashverdiene og følgelig mindre sjanse for kollisjon (hashingalgoritmen bruker dette primtallet).

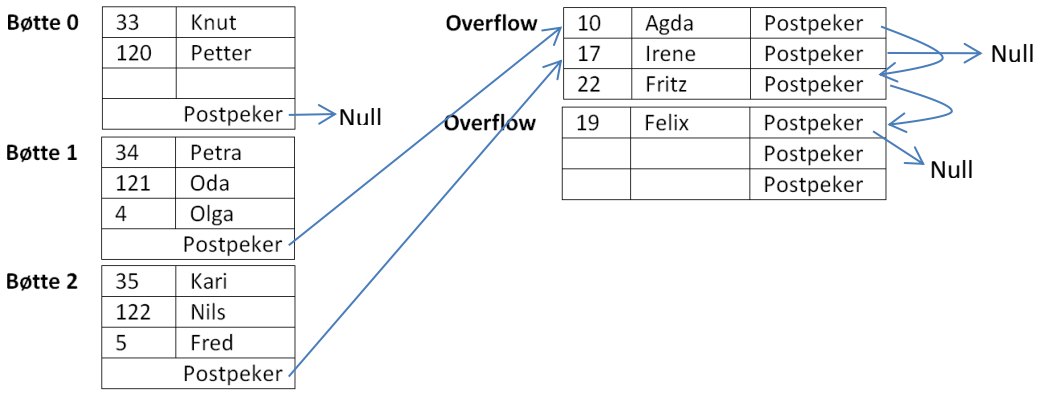
16.8.2 External static hashing

For å tilpasse seg bedre til harddisker, gir hashingtallet her en "bøtte" (bucket) med plass til flere poster. Bøttestørrrelsen er enten en blokk eller et helt antall blokker i en kontinuerlig fil eller kluster.

En spesiell tabell kalt i filheaderen (som står først i hver fil og inneholder opplysninger om filen) tilordner fysisk adresse til hvert bøttenummer. Postene kan gjerne sorteres innen hver bøtte. Siden flere poster får plass i hver bøtte, gir det færre kollisjoner.

Hvis det først blir kollisjon, må posten lagres i en overflow-bøtte og pekere opprettholdes. Overflow-bøtter er lik vanlige bøtter, men har i tillegg en peker i hver post til neste post som skulle hvert i samme bøtte.

Figur 16.10 fra læreboken:
Her brukes siste siffer i primærnøkkelen som bøttenummer. I overflow-bøtten er det en tom post mellom 981 og 182. Den skyldes nok at en post er slettet etter at 182 ble lagt inn.



I denne figuren er det tre bøtter og hashingalgoritmen er *nøkkel mod 3* (dvs. rest etter heltallsdivisjon med 3). Første bøtte har ledig plass, mens de to andre er fulle. De resterende postene som skulle ligget

der er følgelig havnet i overflow-bøttene. Der er det en referanse fra post til post som skulle vært i samme bøtte.

Dette er *statisk hashing* fordi antall vanlige bøtter er fast. Når mange bøtter er fulle, blir lagring og gjenhenting treg og filen må omorganiseres med flere bøtter og ny hashingalgoritme.

Sletting krever egne algoritmer – vi må vedlikeholde evt. pekere og flytte poster fra overflow-bøtten til vanlig bøtte hvis det blir plass.

Merk: External hashing er håpløst for poster med varierende lengde.

16.8.3 External, dynamic hashing

A. Utvidbar (extendible) med to-trinns bøtter

Dette er gjengitt i lærebokens figur 16.11 som du bør studere og forstå.

Man lager en slags "katalog" med plass til 2^d bøttenumre, f.eks. $2^3 = 8$ bøttenumre. De to første binære sifrene i hashverdien, altså intervallet [000..111] = [0..7] angir hvilken plass man skal se etter bøttenummeret.

Flere katalogposter kan vise til samme bøtte, f.eks. kan 010 og 011 vise til samme bøtte. Siden bare de to første sifrene, nemlig 01, brukes, sies da denne bøtten å ha dybde 2 og dette tallet lagres sammen med bøtten. Hvis en slik bøtte med dybde mindre enn antall sifre blir full, må den deles i to, og hver av dem får dybde 3. I eksemplet vil vi få en bøtte som det henvises til fra katalogpost 010 og en bøtte som det henvises til fra katalogpost 011. Hvis en slik bøtte ikke kan deles (den brukes bare av en katalogpost, altså har den dybde 3 i eksemplet) må katalogen dobles til 2^{d+1} bøttenumre. Da kan "problembøtten" (bare denne) deles og alt er i gjenge igjen.

Med statisk hashing kreves bare én diskaksess, mens med utvidbar, dynamisk hashing kreves det to (for å hente katalogen, og for å hente bøtten). Til gjengjeld har dynamisk, utvidbar hashing ingen overflow.

B. Lineær, dynamisk hashing

Man starter med M bøtter og bruker hashingalgoritme h_i, f.eks. K MOD M. Vi merker oss tallet n=0. Når det blir en kollisjon – uansett i hvilken bøtte – deler vi bøtte 0 i to, altså legger til en bøtte nr M, og deler bøtte 0 i to deler med en ny hashingalgoritme h_{i+1}=K MOD 2M som er slik at noen poster (helst omtrent halvparten) havner i bøtte 0 og noen i bøtte M. Vider øker vi n til 1, som angir "neste bøtte som skal splittes".

Når vi søker etter en post, bruker vi først h_i. Hvis hashverdien er mindre enn n, bruker vi isteden h_{i+1}.

Når n=M så er alle bøttene splittet og da brukes h_{i+1} på alle bøttene. Da erstatter vi h_i med h_{i+1} som hovedalgoritme og setter igjen n=0. Overflow fører til ny splitting av bøtte 0 og vi bruker h_{i+2} = K MOD 4M til splittede bøtter.

Slik kan vi fortsette og generelt gjelder at h_{i+j}=K MOD 2^j*M.

Eksempel:

Før innsetting av K = 10:
M = 3, n = 0
h_1 = K MOD 3

Bøtte 0	15
	18

Bøtte 1	13
	4
	25

Bøtte 2	8

Vi skal sette inn K = 10, som skal inn i den fulle bøtten 1. Da må bøtte 0 (fordi n = 0) splittes og h_2 = K MOD 6 brukes på den.

Etter innsetting av K = 10:
M = 4, n = 1
h_2 = K MOD 6

Bøtte 0	18

Bøtte 1	13
	4
	25

Bøtte 2	8

Bøtte 3	15
	10

Bøtte 0 er fordelt med ny algoritme

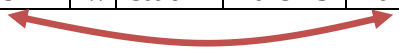
16.9.1 Andre filorganiseringer

A. Logiske referanser
Slike logiske referanser er typisk ”fremmednøkler” (de heter jo ikke det på disken). De fører til at vi må lese flere steder på disken – i forskjellige filer. Det er tungt.

Hvis noen poster ofte hentes sammen, f.eks. *student* og *kurs*, kan de fysisk lagres nær hverandre. Det kalles clustering (av poster). Vi kan til og med lagre to/flere posttyper i samme fil – men se opp for redundans!

Eksempel:
Dårlig, p.g.a redundans (stud⇔kurs er mange-til-mange)

stud1	kurs 15	kurs 12	\\	stud 2	kurs 19	kurs 12	\\	...



OK, ingen redundans (studium⇔stud er en-til-mange):

studium 1	stud 1	stud 3	...	\\	studium 2	stud 5	stud 8	...

Man da blir det tungt å finne én bestemt student. Det kan imidlertid løses med indeksering, som jeg kommer tilbake til.

B. B-trær
B-trær kommer jeg tilbake til under indekser. De kan brukes for små datamengder.

C. Alt annet som er praktisk.
Her er det fritt frem!

16.10 RAID (dette kan studentene sikkert lese selv, her og/eller i boken)

RAID = Redundant Arrays of Independent (eller Inexpensive) Disks, og er mye brukt.

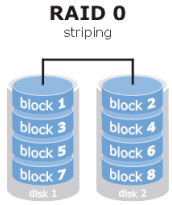
Dataene fordeles på flere filservere (dvs. en datamaskin med O/S og disk). Da kan flere servere jobbe parallelt hvilket gir

- raskere tilgang
- bedre fordeling (load) både av data og prosessering
- bedre stabilitet (reliability) gjennom bevisst redundans

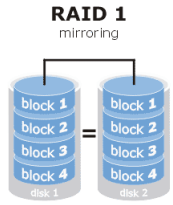
Flere disker gir lavere MTTF (Mean Time to Failure også kalt MTBF). Hvis hver disk har f.eks. MTTF = 200 000 timer = 23 år, så har 100 disker MTTF = 200 000/100 = 2000 timer = 83 dager. Vi må altså forvente hyppigere feil med flere disker, men til gjengjeld er konsekvensene mindre p.g.a redundans. Hvis MTTR (Mean Time To Repair) er forholdsvis liten, er det liten sannsynlighet for at to servere krasjer samtidig, og da kan redundansen redde situasjonen.

Deling av data kan skje ved at
- hver fil deles på flere disker
- hver post deles på flere disker
- hver byte deles på flere disker

RAID deles i mange *nivåer*[1]:

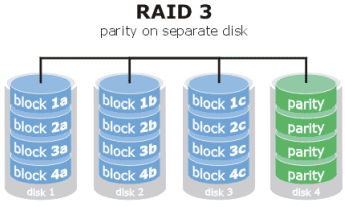


Level 0: Data stripping, der filen fordeles over alle diskene – ingen redundans men raskere.



Level 1: Full mirror – alle forespørsler rutes til den disken som har minst trafikk.

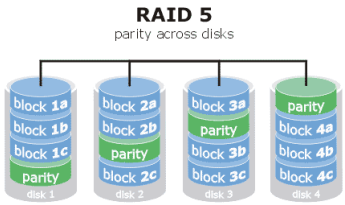
Level 2 (uvanlig): Redundans med hamming codes (slå opp i Wikipedia). Gir færre ekstra disker, f.eks. tre mot fire.



Level 3: Bit-interleaved der bytene splittes på flere disker og med én ekstra disk til redundans med paritetsbit som gjør det mulig å finne hvilken disk som feiler.

Level 4 (uvanlig): Block-interleaved, dvs. hver blokk distribueres til flere disker og én ekstra paritetsdisk redundant.

[1] Figurer fra http://www.prepressure.com/library/technology/raid



Level 5: Som 4, men pariteten spres over alle diskene.

Level 6 (uvanlig): Dobler redundansen og ”tåler” to, samtidige diskfeil. Redundansen spres.

Ytterligere nivåer finnes, blant annet som kombinasjoner av de ovenstående.

Slå også opp i Wikipedia (den engelske er mest grundig).

E&N 17 Filindekser

(Statarisk pensum: 17.1 – 17.4 samt17.6.1 og 17.6.2. Resten av kapittel 17 er ikke pensum.)
Enten man sorterer eller hasher filen, må man vite verdien av sorterings-/hashnøkkelen for å finne igjen posten (uten full gjennomgang). Alle andre søk krever sekvensiell søk, hvilket er ekstremt tung (treg HD, store mengder poster).

Problemet kan løses ved å opprette *indekser*. I prinsippet kan alle felt og kombinasjoner av felt indekseres, og man kan ha mange indekser til hver fil. I *prinsippet* kan man til og med ha flere indekser på ett felt – med hver sin sortering, men noen databaser tillater ikke det. Det er også mulig å indeksere indeksene (ISAM = Indexed Sequentiel Access Method). For raskere å finne frem i en indeks – de kan også bli temmelig store – kan den struktureres som et tre (evt. B-tre eller B^+-tre) eller hashes.

I appendiks F finner du en repetisjon av algoritmen og forklaringen av binærsøk (også kalt "halveringsmetoden") som du kan se på.

17.1 Single level, ordered indexes

Et indekseringsfelt (vanligvis enkle felt, men flere kan kombineres) velges. Verdien kopieres til en annen fil, sammen med pekere til alle poster som har denne verdien – primærnøkler har selvsagt bare én peker. Pekeren refererer ofte til en fysisk blokk. Indeksfilen sorteres, dermed kan den søkes binært.

a. *Primary index* viser til det feltet som er brukt til å sortere en (fysisk) ordnet fil, der ordningsfeltet er unik nøkkel, dvs. en primær eller alternativ nøkkel.
b. *Clustering index* passer der flere poster har samme verdi, jfr. "Hansen" i telefonkatalogen. Filen er altså sortert på clusteringverdien.

Filen kan bare ha én av disse to.

c. *Secondary index* er en indeks på felter som *ikke* er brukt til ordning av filen.

Filen kan ha mange slike indekser.

1) Mer om primærindekser
Siden primærindeksen indekserer unike feltverdier, kan indeksposten ha fast lengde (gitt at feltet/feltene som vanlig har fast og/eller maksimal lengde – dette er diskutert tidligere):

indeksverdi	peker

Indeksverdien er verdien til feltet i første post i en blokk, kalt *anchor record* eller *block anchor*. Pekeren er til en blokk. Notasjon: <K(i), P(i)>.

Se eksemplet i lærebokens figur 14.1.

En *dense index* har post for hver post i hovedfilen, mens en *sparse index* bare har med noen – som primærindeksen forklart ovenfor.

Indeksen blir naturligvis meget mindre enn hovedfilen (den har både færre og kortere poster) og er *mye* raskere å søke i. Vi finner riktig blokk, og så kan vi søke sekvensielt i den – den ligger jo da i RAM.

Et problem er at endring av poster som berører indeksen, blir krevende.

2) Mer om clustering index
Clustering index har en verdi K(i) for hver unike verdi som forekommer i hovedfilen. Videre en peker P(i) som peker til den første blokken som inneholder en post med denne verdien. Husk at denne

indeksen gjelder felt som filen er ordnet etter som sorteringsnøkkelfelt. Postene med samme verdi i det indekserte feltet/feltene, ligger da etter hverandre i filen.

Se lærebokens eksempel i figur 14.2.

Oppdatering er fortsatt tungt. En mulighet er å samle like verdier i blokker (en/flere blokker bare for "Hansen" osv.).

Clustering index er også *sparse* – også kalt *non-dense* – fordi hver indekspost peker til flere poster – det er flere poster i hovedfilen enn i indeksfilen. Siden dette er en sekundær indeks (ikke primær) så er duplikater tillatt.

3) Mer om secondary index
Hvis (a) indekseringsfeltet har unike verdier (det er en alternativ nøkkel), lages én indekspost for hver unike verdi, med verdien og en peker enten (i) til posten med primærnøkkelverdien eller (ii) til blokken i hovedfilen. Denne indekstypen er altså dense. Indeksfilen blir større enn clustering index, men her er alternativet full filsøk.

Hvis (b) indekseringsfeltet har dublettverdier, kan man

i. lage en indekspost for hver post (flere indeksposter får samme indeksverdi)
ii. gi posten variabel lengde med plass til *flere* pekere
iii. la indeksposten peke til en blokk med blokkpekere som igjen peker inn i hovedfilen. Dette er vanlig. I verste fall kan det være behov for å bruke klustere eller lenket liste for den ekstra indeksfilen.

Sekundære indekser gir en *logisk ordning* av filen med et "sortert ikke-nøkkelfelt".

17.2 Multilevel index

Når vi søker binært i en sortert fil, f.eks. en indeksfil, deler vi søkeområdet i to (nesten) like deler for hver iterasjon. Maksimalt antall iterasjoner er altså *omtrent* $\log_2 P$ der P er antall poster, f.eks.:

P	**$\log_2 P$**
16	4
1024	10
2097152	12
...	...

Med multilevel indeksering deler vi ikke i 2 men i flere deler, f.eks. 4, kalt "fo" i boken (fo står for "fan out").

Se lærebokens eksempel i figur 14.6.

I dette eksempelet brukes blokker, både i indeksen og i hovedfilen. Annet nivå indeks peker da til en blokk (via ankerposten) i første nivå indeks, som igjen peker til en blokk i hovedfilen. Dette kan bygges videre ut med ytterligere nivåer.

ISAM – en suksesshistorie
ISAM ble utviklet av IBM. Det var da vanlig med nettverksdatabaser, der hver post inneholdt en fysisk peker til neste post (en lenket liste) og til relaterte poster.

IBM begynte – som pionerer – med harddisker ca. 1950. De ble kalt IBM 350. Før var det hullkort og tape som gjaldt og taper varte lenge p.g.a kapasitetsproblemer med HD – IBM 350 lagret 4,4 MB.

I 1964 kom HD 2311 for stormaskinen ”System/360” som kom samme år. Det var én av de helt store IBM-suksesser. Den var utbyggbar og var designet av Amdahl som senere startet for seg selv.

Harddiskene besto av skiver som var inndelt i *spor* (tracks). Lesehodet leste alle sporene samtidig (ett på hver skive), f.eks. spor 15 på alle skivene. De ble kalt en sylinder. Det var 200 spor på seks skiver.

ISAM var en tonivå indeks. Først var det en *sylinderindeks* med ankerverdier og pekere til en sylinder. Deretter *track index* med ankerverdier for hvert spor i sylinderen. Sporet kunne så hentes og søkes sekvensielt.

Nye poster ble lagt i en overflow-fil, som periodisk ble flettet med hovedfilen. Da ble indeksen oppdatert.

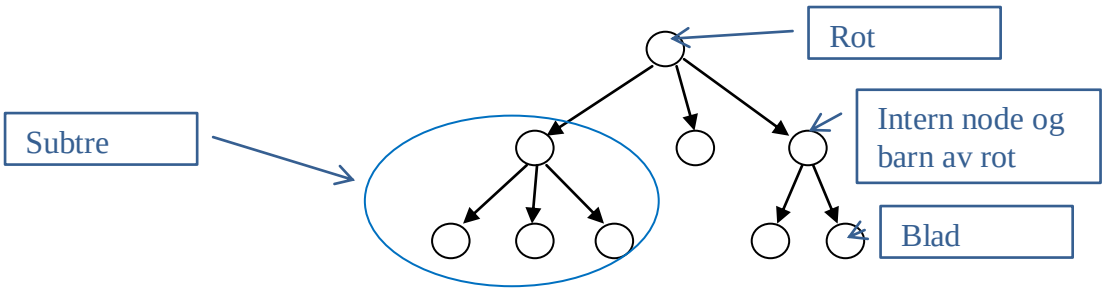
Etterhvert overtok IBMs VSAM og senere DB2 som ligger oppå ISAM/VSAM.

ISAM var ”flate filer”, noe som gir problemer med samtidig bruk av flere. De egnet seg best til batch-prosessering, ellers måtte egne programmer håndtere flerbrukerproblematikk, konflikter, sikkerhet osv.

MySQL brukte også ISAM, senere MyISAM, men nå er det faset ut. Det var hver enkelt tabell som kunne deklareres som ISAM/MyISAM (*Alter table mintbl type = MYISAM;*).

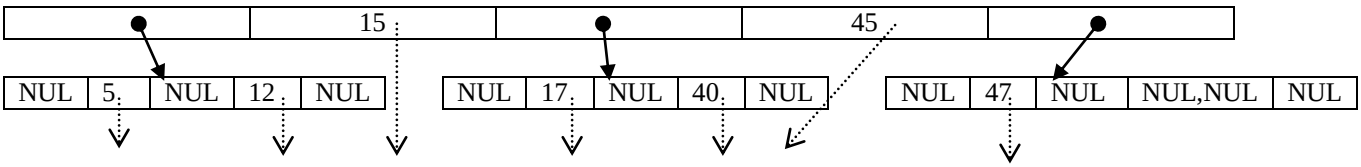
17.3 B-trær

Trær har en kjent, hierarkisk struktur:



Alle nodene unntatt roten, har en og bare én overordnet som de er *barn* av.

Et tre av *orden 3*, har maks tre pekere til barn og maks 2 indeksverdier:



Strukturen i rot og indre noder, er slik at først ligger det en peker til et barn, deretter en verdi sammen med peker til hovedfilen, en ny peker til et annet barn, en ny verdi sammen med peker til hovedfilen, og til slutt en peker til et tredje barn. I bladene er strukturen den samme, men alle indre pekere til barn er NUL. Alle nodene har minst *P DIV 2* indre pekere der *P* er treets orden – her 1 (roten har minst 2 hvis den ikke er eneste node). Pekerne til hovedfilen er enten til en blokk eller til en post (dvs. blokk + postnummer innen blokken). I et ”ekte” B-tre er også alle bladene på samme nivå.

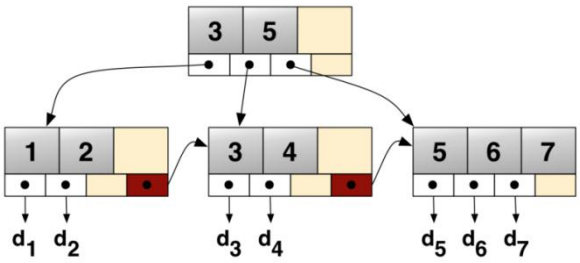
Når (a) en node blir full ved innsetting, eller (b) en blir under halvfull ved sletting, må treet omstruktureres. *Se en artig video av dette på http://www.youtube.com/watch?v=coRJrcIYbF4. Den illustrerer godt hvor mye omstrukturering som kan være nødvendig.*

B-trær brukes vanligvis for indekser, men også hele poster kan legges i nodene. Det vil bare virke for små filer.

17.3.2 B+-trær

Ved flernivå indekser er dette det mest vanlige. I et B^+-tre, finnes indeksverdier og peker til hovedfilen *kun* i bladene, som altså da er forskjellige fra de øvrige nodene (roten og de indre).

Et eksempel[2]:



Merk pekeren fra et blad til neste – det gjør det raskere å gå igjennom alle verdiene – alternativt måtte man opp til rot og starte igjen for å finne neste blad.

17.4 Indekser på flere felt

Hvis vi søker en post etter to felt samtidig, f.eks. navn = ”Hansson” OG alder = 48, bør vi søke slik:

a. Hvis ett av feltene er indeksert: Start med det. Det finner raskt alle aktuelle poster og vi kan glemme resten (p.g.a OG i søkekriteriet). Sjekk deretter disse postene mot det andre kriteriet, en for en.
b. Hvis begge feltene er indeksert i hver sin indeks: Søk hvilken som helst først, sjekk deretter disse postene mot det andre kriteriet. (Hvis det er kjent at det ene kriteriet forekommer sjeldnere enn det andre, bør det søkes med indeksen for det først. I Oracle kan vi utnytte slik kunnskap ved å gi et ”hint” i select-setningen.)
c. Hvis ingen av dem er indeksert: Dette blir tungt – hent kaffe☺!

Hvis dette søket forekommer ofte, bør de to feltene indekseres sammen – som kombinert nøkkel. (Jeg har vært borti databaser som kunne settes opp til å analysere slikt selv, og selv legge til indekser for vanlige søk.)

17.6 Some general issues

Det er mer å si om indeksering enn det som er gjort ovenfor. Her tar jeg med bare noen temaer.

17.6.1 og 17.6.2 Logiske pekere

Hvis vi indekserer et ikke-nøkkelfelt kan, vi la pekeren være en *verdi* istedenfor fysisk, f.eks. slik at indeks for alder angir ansattnummer som har denne alderen. Ansattnummer må vel da også være indeksert på vanlig måte. Søking blir da tyngre – det blir to trinn istedenfor ett – men fysiske endringer i hovedfilen vil da ikke berøre indeksen for ikke-nøkkelfeltet.

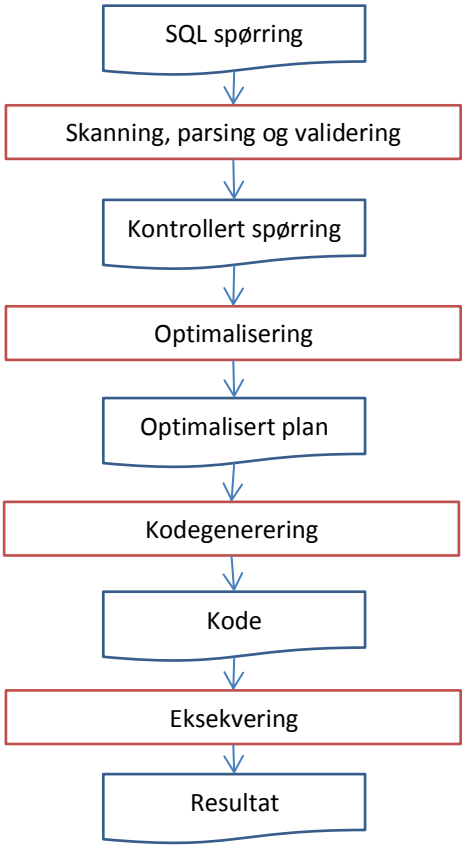
Man kan også bruke en hashet struktur på indeksen.

[2] Fra http://en.wikipedia.org/wiki/B%2B_tree

E&N 18 Algoritmer for spørringer og optimalisering

Kap. 18.1-18.6 er statarisk pensum. Resten er kursorisk pensum.

Vi brukere skriver våre spørringer i en form som vi finner mest "naturlig". Da SQL er et fjerde generasjons språk – 4GL – definerer vi *hvilket resultat vi vil ha* og ikke *hva som skal gjøres* for å produsere dette resultatet. Våre spørringer (og alt annet) må følgelig gjøres om til algoritmer og til eksekverbar kode. DBMS vil herunder forsøke å optimalisere algoritmen, så den eksekverer raskest mulig.



DBMS vil gjøre følgende:

1. Kontroll av teksten
    a. *Skanne* teksten, dvs. gå gjennom teksten og finne **"tokens"** som er gjenkjennbare ord f.eks. SQL-ord, kolonner og tabeller.
    b. *Parse* teksten, dvs. gå igjennom teksten igjen og sjekke **syntaksen**.
    c. *Validere* teksten, dvs. sjekke at alle **navn er gyldige**.

    Dette resulterer vanligvis i et *"query tree"* eller en *"query graph"* som beskriver spørringen.
2. *Query optimizer* optimaliserer så spørringen, dvs. bestemmer en fornuftig rekkefølge for de handlingene som spørringen krever.
    Dette resulterer i en *plan* som er mer eller mindre optimal.
3. *Query code generator* koder spørringen i henhold til planen. Denne koden er fortsatt i høynivå, tredje generasjons (3GL) språk. Den kan kjøres interpretert med en gang, eller kompileres og lagres til senere bruk (som trigger, stored function/procedure).
4. *Runtime database processor* eksekverer koden og genererer resultatet.

Optimalisering er her et relativt begrep. Full optimalisering til absolutt "beste" algoritme, vil kreve full kunnskap om databasens struktur og innhold, om filenes implementering og annet, og derfor er det ofte umulig å optimalisere fullstendig før spørringen faktisk er gjennomført. Det blir jo for sent. Isteden gjør "optimizer" så godt den kan.

18.1 SQL-spørringer til relasjonsalgebra

Boken bruker sitt eget eksempel (fra kapittel 5) med bl.a. disse to relasjonstabellene:

EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno

DEPARTMENT

Dname	Dnumber	Mgr_ssn	Mgr_start_date

Syntaks

I relasjonsalgebraen benyttes en syntaks som er slik[3]:

- ✓ Relasjonstabell: R
- ✓ Seleksjon: σ (gresk s kalt sigma)
  Seleksjonen $\sigma_{vilkår}(R)$ velger ut de radene av relasjonstabellen R som tilfredsstiller vilkåret. Resultatet er en relasjonstabell. Eksempel: $\sigma_{Dno=5}$ (EMPLOYEE)
- ✓ Projeksjon: π (gresk p kalt pi)
  Projeksjonen $\pi_{kolonner}(R)$ velger ut de nevnte kolonnene (adskilt med komma) fra relasjonstabellen R. Resultatet er en relasjonstabell. Eksempel: $\pi_{Fname,Lname}$(EMPLOYEE)
- ✓ Union: ∪
  $R_1 \cup R_2$ velger ut alle radene fra relasjonstabell R_1 og i tillegg alle radene i R_2 som ikke er i R_1 (eller sagt annerledes: Samtlige rader i de to tabellene men uten duplikater). De to relasjonstabellene må være union-kompatible, dvs. de må ha samme antall kolonner i samme rekkefølge og med samme domener. Resultatet er en relasjonstabell.
- ✓ Snitt: ∩
  $R_1 \cap R_2$ innebærer at man velger ut de radene som er helt like i de to relasjonstabellene, som må være union-kompatible. Resultatet er en relasjonstabell.
- ✓ Differanse (minus): –
  $R_1 – R_2$ innebærer at man velger ut alle radene i R_1 som ikke også finnes i R_2 (de finnes altså kun i R_1). De to relasjonene må være union-kompatible.
- ✓ Kartesisk produkt: x
  $R_1 x R_2$ innebærer at man lager alle mulige rader som består av én rad fra relasjon R_1 etterfulgt av én rad fra R_2. Resultatet er en relasjonstabell med alle kolonner fra R_1 og alle fra R_2.

Dette er de strengt nødvendige operasjonene. Man kombinerer med bruk av parenteser. Vilkårene uttrykkes med sammenlikninger og Boolsk algebra (AND, OR, NOT).

I tillegg har man gjerne lagt til – for enkelhets skyld, følgende

- ✓ Join: ⋈
  $R_1 \bowtie_{sammenheng} R_2$ innebærer at man lager kartesisk produkt bare for de radene i R_1 og R_2 som er relatert på den oppgitte måten. Ofte er det *natural join* der verdien av en fremmednøkkel i R_1 er lik verdien av en primærnøkkel i R_2. Eksempel: EMPLOYEE$\bowtie_{Dno=Dnumber}$DEPARTMENT
- ✓ Rename: ρ (gresk r kalt ro) gir nytt navn til relasjon eller kolonne eller et midlertidig navn til et mellomresultat.
- ✓ Tilordning: a←b innebærer at a tilordnes verdien b. Eksempel på rename og tilordning som vil gi etter- og fornavn på de ansatte i avdeling 5 (som er lærere):
  LÆRERE←$\sigma_{Dno=5}$(EMPLOYEE), RESULTAT← $\pi_{Fname,Lname}$(LÆRERE)
- ✓ Aggregeringsfunksjoner: F
  $F_{funksjon}$(*kolonne*) angir at funksjonen som er angitt skal brukes på den angitte kolonnen. Eksempel: $F_{count}(\pi_{SSn}$(EMPLOYEE)) som skal bruke aggregeringsfunksjonen *COUNT* på kolonnen *SSn* som er projisert (valgt ut) fra relasjonstabellen EMPLOYEE.

[3] Det samme er gjengitt i tabell 6.1 i læreboken.

Eksempel (fra boken)
Man har skrevet følgende spørring:

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

Den skal gi etternavn og fornavn for alle som er betalt høyere enn største lønn i avdeling 5.

Siden det her er en nøstet spørring, kan den deles i to. Den første, indre spørringen er

```
SELECT MAX(Salary) FROM EMPLOYEE WHERE Dno=5;
```

Dette gir følgende algebraiske uttrykk, der c er en skalar (enkeltverdi) konstant:

$c \leftarrow F_{max}(\sigma_{Dno=5}(EMPLOYEE))$

Den andre, ytre delen blir

```
SELECT Lname, Fname FROM EMPLOYEE WHERE Salary > c
```

som gir følgende algebra:

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

Nå kan optimizer optimalisere disse to algebraiske uttrykkene hver for seg.

18.2 Ekstern sortering

Ekstern sortering utføres når ikke hele filen får plass i RAM, ellers vil vi sortere i RAM da det er mye raskere. Særlig hvis filen er kontinuerlig, så den er rask å lese. Indre sortering (i RAM) er kjent fra programmeringsfaget og har mange algoritmer – det er skrevet hele bøker bare om det – og de er effektive i forskjellige situasjoner[4].

Ekstern sortering gjøres gjerne med *sort-merge algoritmen* som gjennomgår to faser: *Sorting phase*, der selve sorteringen foregår for en del av gangen, og *merging phase* der de sorterte delene slås sammen.

Filen som skal sorteres, deles først opp i biter som er små nok til å sorteres i RAM. Hver slik del kalles en *run*. En og en run leses til RAM, sorteres der og skrives til hver sin midlertidige fil. Dette er sorting phase.

Når alle delene er sortert og lagret i midlertidige filer, må de slås sammen to og to. Da leses de blokk for blokk (hvis det da ikke er plass til begge delene i RAM) og skrives etter hvert til en ny, midlertidig fil. Når alle delene er slått sammen slik, to og to, til halvparten antall filer av dobbelt størrelse, har man gjennomført en *pass*. Det kan være nødvendig med flere slik passes – for hvert pass blir det omtrent halvparten så mange midlertidige filer som er omtrent dobbelt så store som i forrige pass. (Det er bare hvis antall passes som skal flettes er partall, at det blir akkurat dobbelt så mange, og eksakt dobbelt størrelse bare ved fast postlengde.)

Når alt er lagret, sortert, i én pass, er filen ferdig sortert.

Dette krever mye lesing/skriving fra/til disk, og i tillegg tar sorteringen og flettingen tid i sentralenheten, så dette er gjerne en tung jobb. Man vil unngå den hvis man kan, f.eks. ved å gjøre mest mulig utvalg før sorteringen, og helst bruke en sortert indeks isteden.

Ekstern sortering av en fil som krever ni runs:

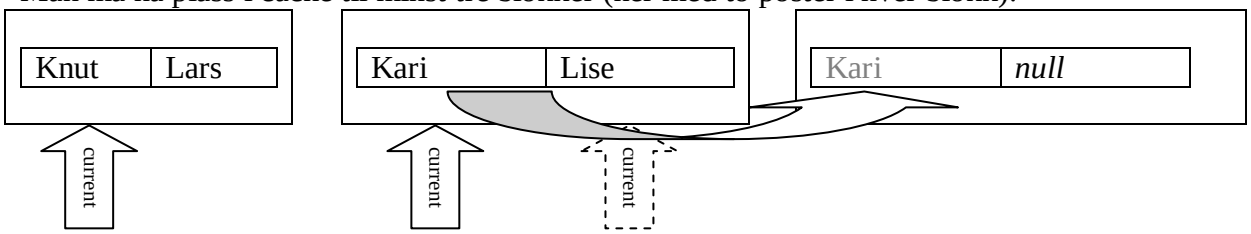
[4] En meget enkel oversikt for interesserte er http://www.softpanorama.org/Algorithms/sorting.shtml. Selve "bibelen" er Donald Knuth: "Art of Computer Programming, Volume 3: Sorting and Searching"

Sorting phase		Filen som skal sorteres							
		⇩ *Inndel i runs* ⇩							
		Run 1	Run 2	Run 3	Run 4	Run 5	Run 6	Run 7	Run 8	Run 9
		⇩ *Sorter hvert run for seg* ⇩							
		Run 1 sortert	Run 2 sortert	Run 3 sortert	Run 4 sortert	Run 5 sortert	Run 6 sortert	Run 7 sortert	Run 8 sortert	Run 9 sortert
Merging phase	**Pass 1**	⇩ *Flett to og to* ⇩								
		Run 1+2		Run 3+4		Run 5+6		Run 7+8		Run 9
	Pass 2	⇩ *Flett to og to* ⇩								
		Run 1+2+3+4				Run 5+6+7+8				Run 9
	Pass 3	⇩ *Flett to og to* ⇩								
		Run 1+2+3+4+5+6+7+8								Run 9
	Pass 4	⇩ *Flett to og to* ⇩								
		Ferdig sortert fil								

Hver enkelt fletting foregår slik (sortering i stigende rekkefølge)

1) Les en post fra hver fil.
2) Gjenta inntil en av filene tar slutt:
   Kopier den minste "posten" til ny fil og les en ny post fra den filen som det ble kopiert fra
3) Kopier alle resterende poster fra den filen som ikke er slutt, til ny fil.

Man må ha plass i cache til minst tre blokker (her med to poster i hver blokk):



Her kopieres Kari til den ferdige blokken, fordi Kari < Knut. Current flyttes til Lise, og ny sammenlikning/flytting gjennomføres. Når Lise etter hvert er skrevet til resultatblokken, må en ny blokk – med to nye poster – leses.

Beregninger

Anta at vi har plass til fem blokker i buffer, n_β=5. Filen er på 1024 blokker, b=1024. Antall initial runs er da $n_R = \frac{b}{n_\beta} = \frac{1024}{5} = 205$.

Degree of merging er antallet runs som kan merges i en *pass*.

$d_m = \min(n_\beta - 1, n_R) = \min(4,205) = 4$

Antall passes er $log_{d_m}(n_R)$ som her er $log_4(205) = 4$ (den største "firerpotens" som overstiger 205).

Antall runs for hver pass blir slik: 205→52→13→4→1.

18.3.1 Select

Det er her snakk om seleksjon av rader fra en tabell (eller fra en midlertidig resultattabell). Søking av *enkle utvalg* er søking der utvalgskriteriet er uten Boolsk algebra, altså bare med en enkel sammenlikning, f.eks. SELECT * FROM EMPLOYEE WHERE Lname > "P"; Boken forklarer seks forskjellige søkestrategier for et slikt utvalg.

1. Lineær søk ("brute force"). Alle postene hentes og sammenliknes med utvalgskriteriet. Det sier seg selv at dette er den tyngste strategien.

2. Binær søk er mulig hvis utvalgskriteriet gjelder en sorteringsnøkkel som filen er sortert etter.
3. Primærindeks[5] (eller hashnøkkel) kan brukes hvis søkekriteriet gjelder likhet med en verdi i en primærindeks.
4. Bruke en primærindeks til å hente flere poster, når kriteriet er "større enn" eller "mindre enn". Har vi først funnet den første, vet vi jo hvor resten er. Da må man ha spesiell algoritme for de tilfellene der sammenlikningsverdien ikke finnes (f.eks. hvis ingen har navn på "P" i eksemplet over).
5. Bruke en clustering[6] indeks til å hente flere poster.
6. Bruke en sekundærindeks[7] til å hente flere poster hvis søking gjelder likhet.

Videre lister boken tre strategier for henting av *komplekse utvalg* der det er anvendt Boolsk algebra:

7. Konjunktivt[8] utvalg (med AND) med bruk av en enkel indeks. Man velger først ut alle som tilfredsstiller første kriterium med en indeks, så sjekker man om den utvalgte posten også tilfredsstiller det andre kriteriet.
8. Konjunktivt utvalg med en sammensatt indeks – gitt at en slik indeks finnes og at kriteriene er likhet.
9. Konjunktivt utvalg med postpekere. Hvis kriteriene gjelder kolonner som er indeksert med direkte postpekere, kan man først velge ut de aktuelle pekerne fra hver indeks. Snittet mellom disse (de som finnes i begge) er det utvalget vi ønsker.

Hvor mange poster som blir hentet med et bestemt kriterium, kalles *selektivitet* og er et forholdstall i intervallet [0..1]. Dette kan man jo egentlig ikke vite før søket er gjennomført, men DBMS lagrer ofte et estimat på selektivitet. Hvis f.eks. et felt har 2000 forskjellige verdier jevnt fordelt over 16000 poster, så vil et søk på likhet omtrent returnere $\frac{16000}{2000} = 8$ poster og selektiviteten er omtrent $\frac{1}{2000} = 0{,}5‰$. Hvis filen øker til 30000 poster, vil vi da forvente å finne omtrent $30000 \cdot 0{,}5‰ = 15$ poster.

18.3.2 Join

Boken omtaler bare to joins, begge mellom to tabeller:

- ✓ EQUIJOIN der alle kriteriene er likheter (det kan være flere), og
- ✓ NATURAL JOIN som er som EQUIJOIN, men bare én av de sammenliknede kolonner utelates i svaret (typisk for fremmednøkkel/primærnøkkel).

Joins er alltid krevende, og innebærer i prinsippet følgende algebra:

$R_1 \bowtie_{A=B} R_2$ der A og B er kolonnenavn

Boken beskriver fire strategier:

1. Nøstet løkke join, der man for hver post *t* man henter i R_1 henter alle poster *s* i R_2 og ser om de tilfredsstiller likheten *t.A=s.B*.
2. Enkel løkke join, der man bruker først finner alle poster *s* i R_1 og bruker en indeks eller hash til å finne de matchende postene *t* i R_2 (der *t.A=s.B*).
3. Sort-merge join, krever at begge relasjonstabellene er lagret sortert og sorteringskriteriet inngår som utvalgskriterium. Da kan man hente postene i begge filene i sortert rekkefølge og matche dem én for én. Hvis filene ikke er sortert, kan de sorteres først.

[5] Jeg minner om at en primærindeks viser til det feltet som er brukt til å sortere en (fysisk) ordnet fil, der ordningsfeltet er unik nøkkel, dvs. en primær eller alternativ nøkkel.
[6] En clustering indeks viser til et ikke-unikt felt som filen er sortert etter. En slik indeks er *sparse* og viser til første post med den gitte verdien – de andre følger etter i filen.
[7] Sekundærindekser indekserer felt som ikke er brukt til sortering av filen. De kan være unike eller ikke, og sparse eller dense.
[8] Ordet er her brukt i Boolsk forstand og innebærer at to kriterier er koblet sammen med AND.

En variant har vi hvis det finnes en passende sekundærindeks for den ene eller begge filene. Da kan vi anvende dem, siden indeksene er sortert.

4. Hash join krever at begge filene først hashes til samme hashfil, med en kombinasjon av de to kolonnene som inngår i likheten. Etter denne hashingen – som kalles partioning phase – kan man gå igjennom den ene filen én gang – alle de matchende postene finnes da i samme bøtte. Det siste kalles da probing phase.

Resten av kapittel 15.3.2 er kursorisk.

18.4 Projeksjon og mengdeoperasjon

Det er vanligvis enkelt å velge ut en/flere kolonner av en tabell, men hvis verdiene skal være unike kan det innebære noe ekstra.

Relasjonsdatabaser er *helhetlige* dvs at alle operasjoner på relasjonstabeller resulterer i en ny relasjonstabell. Når da resultatet ikke inkluderer en nøkkel, må vi imidlertid godta dubletter (det bryter med entitetsintegriteten – resultatene er altså ikke "ekte" relasjonstabeller). Bare med DISTINCT rydder vi opp i det.

```
SELECT Lname FROM EMPLOYEE;
```

SSn	Lname
55	Hanssen
32	Ås
19	Hanssen

⇨

Lname
Hanssen
Ås
Hanssen

Hvis vi vil fjerne dubletter (av prinsipp eller med DISTINCT) kan vi

- ✓ Sortere resultatet – dublettene vil ligge etter hverandre
- ✓ Hashe resultatet – dubletter vil havne i samme bøtte

Kartesisk produkt innebærer å samle alle *n* rader med *m* andre rader, som oftest i en annen relasjonstabell. Resultatet gir $n \cdot m$ rader. Alle kolonner blir med, f.eks. *j* kolonner og *k* kolonner. Det krever mye lagringsplass og det kan bli enormt krevende å sortere, gruppere e.l.

UNION, SNITT og DIFFERANSE krever at relasjonstabellene er union-kompatible (se ovenfor), men vanligvis med forskjellig innhold. Her kan man bruke sort-merge og fjerne dubletter.

18.5.1 Aggregeringer

Standard aggregeringsfunksjoner er MIN, MAX, SUM, COUNT og AVG, men noen DBMS har flere (og i Oracle kan man lage sine egne).

1. MIN og MAX.
    a. Hvis det finnes en indeks for kolonnen, vil MIN være første post i indeksen, og MAX det siste. Hvis indeksen er et B-tre, må enten første eller siste post i nodene følges helt ut til blad.
    b. Hvis det ikke finnes en indeks, kreves full filgjennomgang.
2. COUNT, AVG og SUM.
    a. Hvis det finnes en dense indeks (der er alle postene med) kan disse beregnes bare vha indeksen.
    b. Hvis det finnes en sparse indeks, vil verdiene ligge i indeksen, men *antallet* må finnes på en annen måte:
    Hvis DBMS støtter COUNT DISTINCT, kan vi slippe å telle antall pekere. Da teller man antall *forskjellige* verdier og ikke antall poster. Ellers kan man

i. Med variabel rekordlengde i indeksen kan man telle antall pekere for hver indeksverdi
ii. Med fast rekordlengde i indeksen kan man telle antall pekere i pekerblokken

3. GROUP BY. Med gruppering blir det mer komplekst, fordi gruppene må finnes først. Det kan gjøres med sortering eller hashing og først deretter kan aggregeringen foretas for hver gruppe.

18.5.2 Outer joins

Et eksempel på outer join:

EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno

DEPARTMENT

Dname	Dnumber	Mgr_ssn	Mgr_start_date

```
SELECT Lname, Fname, Dname FROM EMPLOYEE LEFT OUTER JOIN DEPARTMENT ON Dno=Dnumber;
```

Her må man gjennomgå alle EMPLOYEE-poster *e* og for hver *e* enten

- ✓ finne riktig *Dname* ved hjelp av likhet *Dno=Dnumber*, eller
- ✓ fylle på med *null* (kalles *padding*)

Forhåpentlig er DEPARTMENT sortert eller indeksert etter *Dnumber*, ellers blir det tungt!

Note: Tidligere fantes ikke OUTER JOIN og man skrev det annerledes:

```
SELECT Lname, Fname, Dname FROM EMPLOYEE, DEPARTMENT WHERE (Dno=Dnumber) OR NOT
EXIST (SELECT * FROM EMPLOYEE, DEPARTMENT WHERE Dno=Dnumber);
```

Den siste ville sikre at også de postene i EMPLOYEE som ikke hadde motpost i DEPARTMENT kom med selvom Dno ikke var *null*. Hvis Dno var fremmednøkkel men *null* tillatt blir det enklere:

```
SELECT Lname, Fname, Dname FROM EMPLOYEE, DEPARTMENT WHERE (Dno=Dnumber) OR (Dno
is null);
```

18.6 Pipelining

Mange spørringer produserer temporære filer og det er tungt. Da kan det være bedre å "gjøre seg ferdig med én post av gangen". Da lar man hver post i første operasjon brukes som input til neste operasjon osv. Dette kalles pipelining. Algoritmen blir tyngre, men man slipper de temporære filene.

Eksempel:

```
SELECT Lname, Fname, Dname FROM EMPLOYEE, DEPARTMENT WHERE (Dno=Dnumber AND
Dno=5 AND Sex="F");
```

Her kan vi lage temporære filer og ta en operasjon av gangen:

1) Lage tempfil1 med bare ansatte som har Dno=5
2) Lage tempfil2 med bare ansatte fra tempfil1 som har Sex="F"
3) Lage tempfil3 med kartesisk produkt av tempfil2 og DEPARTMENT
4) Lage tempfil4 med bare de i tempfil3 som har Dno=Dnumber
5) Lage resultatfil med bare de tre angitt kolonnene

Isteden kan vi pipeline, f.eks. slik:

1. Les én EMPLOYEE. Ved slutt gå til 9
2. Hvis Dno<>5 OR Sex<>"F" så gå til 1
3. Start forfra i DEPARTMENT
4. Les én DEPARTMENT. Ved slutt gå til 1
5. Hvis Dno<>Dnumber så gå til 3
6. Skriv Lname, Fname, Dname til resultatfilen

7. Gå til 3
8. Gå til 1
9. Ferdig

Hvis det finnes ordninger og/eller indekser for noen av kolonnene, så utnytter vi naturligvis det.

18.7.1 Query trees og query graphs (kursorisk)

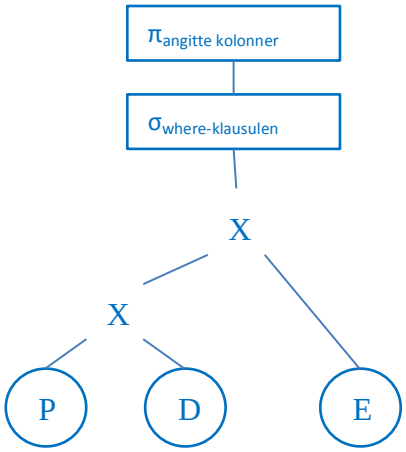
Boken har følgende eksempel kalt Q2:

```
SELECT P.Pnumber, P.Dnum, E.Lname, E.Address, E.BDate
FROM PROJECT P, DEPARTMENT D, EMPLOYEES E
WHERE p.Dnum=D.Dnumber AND D.Mgr_ssn=E.Ssn AND P.Plocation="Stafford";
```

Den tilsvarende algebraen er

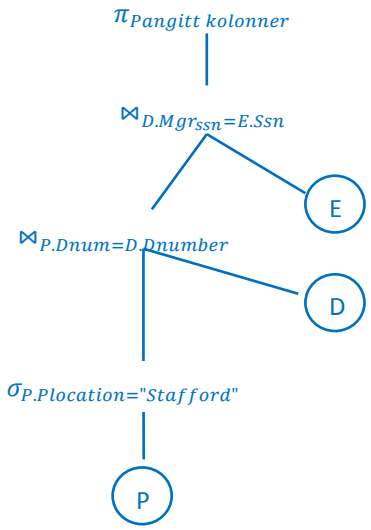
$\pi_{Pnumber,Dnum,Lname,Address,BDate}(((\delta_{Plocation="Stafford"}(PROJECT))$
$\bowtie_{Dnum=Dnumber}(DEPARTMENT)) \bowtie_{Mgr_{ssn}=Ssn}(EMPLOYEE))$

Hvis denne løses direkte, vil det først bli laget kartesisk produkt av P, D og E (project, department og employee). Deretter velges de radene som tilfredsstiller where-klausulen og til slutt velges de angitt kolonnene. Det kan vises slik i et tre:



Vi ser at det trengs fire operasjoner for å løse oppgaven. Dette er en første "oversettelse" og vil gi riktig resultat, men den er svært treg fordi kartesisk produkt lages to ganger først. Dette treet er helt uoptimalisert. Det vil bli tungt å eksekvere fordi alle – og hele – postene tas med i det kartesiske produktet PxDxE Det gir svært mange og svært store poster og dermed stor fil med få poster pr blokk. Den kan åpenbart optimaliseres.

Treet er *kanonisk*. Det vil forenklet si at operasjonene kan tas i en annen rekkefølge med samme resultat. (Man må allikevel sørge for at tilstrekkelig kolonner er med til at man kan gjennomføre utvalg etter where-klausulen.) I nedenstående tre, som er et forsøk på optimalisering, gjør man først et utvalg av poster i P. Utvalget kobles til D og E med equijoin (det er raskere enn først å lage kartesisk produkt og deretter velge ut fordi en equijoin kan "pipelines"). Til slutt velges de riktige kolonnene.



Dette er da bare én av mange mulige optimaliseringer.

De to trærne angir en rekkefølge for handlingene, mens en graf (*se lærebokens figur 18.4 c*)grafen ikke har noen rekkefølge. Siden optimizer uansett må legge inn en rekkefølge, brukes grafer nå lite.

(a) er optimalisert, og da blir PROJECT først "filtrert" så det blir vesentlig færre poster i de kartesiske produktene (avhengig av hvor mange som faktisk har Plocation="Stafford"). Det kan finnes mange optimaliserte trær som alle tilsvarer (b) – trikset er å velge den mest effektive.

Da brukes enten
- ✓ heuristiske regler ("tommelfingerregler), eller
- ✓ beregner "kostnaden" og velger den "billigste"

Ofte gjør optimizer begge deler.

Bokens figur 18.5 (*se figuren i læreboken*) viser optimalisering av følgende spørring:

```
SELECT lName
  FROM EMPLOYEE, WORKS_ON,PROJECT
  WHERE Pname='Aquarius' AND Pnumber=Pno
  AND Essn=Ssn AND Bdate>'1957.12.31';
```

Optimaliseringen gjøres med heuristiske regler. Forklaring av optimaliseringen:
(a) Viser det initiale, kalt *kanoniske*, treet. Det vil gi meget store kartesiske produkter og er derfor svært lite effektivt.
(b) Seleksjonene flyttes nedover. Dermed blir det færre poster som inngår i det kartesiske produktet.
(c) Den mest restriktive seleksjonen (som begrenser antallet poster mest) anvendes først (flyttes nedover).
(d) Kartesisk produkter erstattes med JOIN der det er mulig. En JOIN er bedre enn et kartesisk produkt, fordi bare de postene som tilfredsstiller JOIN-kriteriet kommer med.
(e) Projeksjoner flyttes nedover og tas så snart som mulig. Da blir postene mindre og følgelig de temporære filene mindre (og med flere poster pr blokk evt. færre blokker pr post).

Det avsluttende treet (e) vil eksekvere svært mye rasker enn det kanoniske, (a).

Generelt gjelder at seleksjon er bra – det gir færre poster. Restriktive seleksjoner er bedre enn mindre restriktive. Videre er projeksjoner bra – de gir færre kolonner, altså mindre poster.

Læreboken gjengir en rekke lover som gjelder for transformasjon av relasjonsalgebra. Dette er regler som optimizer bruker for å forbedre treet. I eksemplet ovenfor, har f.eks. optimizer anvendt loven om at en seleksjon konjunktiv seleksjon (med AND) kan brytes opp i en sekvens av seleksjoner – loven om kaskade av konjunktiv seleksjon.

18.7.3 Query Execution Plan (kursorisk)

Når treet er ferdig optimalisert, gjøres det om til en execution plan som inkluderer hvordan postene skal hentes (med indekser, hashing sortert fil eller full filgjennomgang) og om resultatet av hver operasjon skal mellomlagres ("materialized evaluation") eller pipelines ("pipelined evaluation").

18.8.1 Cost optimization (kursorisk)

Med heuristiske regler ble bare query-treet optimalisert. Det ble ikke tatt hensyn til størrelse på filene, tilgjengelige indekser o.l.

Med kostnadsoptimalisering beregnes en tenkt kostnad for hvert tre, og den "billigste" velges. Husk da at optimalisering også tar tid og kostnadsoptimalisering tar lengre tid enn den heuristiske. Kostnadsoptimalisering er derfor vanligst for kompilerte spørringer som antas å bli brukt igjen, og ikke så vanlig for interpreterte. De fleste DBMS vil tilby begge.

Kostnadsfaktorene er gjengitt i læreboken og gjelder:

- ✓ tilgang til eksternt lager
- ✓ lagringskostnad for midlertidige filer
- ✓ beregningskostnad (internt) f.eks. søking, sortering, merging og beregninger på kolonneverdier
- ✓ minnekostnad
- ✓ kommunikasjonskostnad

Et problem er hvilken vekt som skal legges på de forskjellige. For større databaser er det – som nevnt flere ganger tidligere – tilgang til eksternt lager som er det viktigste. En måte å få ned optimaliseringstiden på, er derfor å bare se på antall diskaksesser. Man beregner simpelthen antall blokker inn og ut. Man må ta hensyn til indekser o.a. og må også vurdere antall distinkte verdier i en kolonne og selektiviteten. Man bruker da gjennomsnittsberegninger.

Ved distribuerte databaser (databasen lagret på flere maskiner/steder) vil kommunikasjonskostnaden bli den viktigste. Man beregner antall bytes som sendes. Ved å velge bare en kostnadsfaktor, slipper man problemet med å velge vekting.

18.8.2 Kataloginformasjon i kostnadsberegninger (kursorisk)

Til hjelp med optimaliseringen, kan DBMS holde orden på ekstra informasjon om databasen. Noen eksempler på interessant informasjon:

- ✓ Filstørrelse (det er tyngre å lete sekvensielt i en stor fil), antall blokker
- ✓ Antall poster, poststørrelse, og antall poster pr blokk (blocking factor).
- ✓ Hvordan filen er organisert
- ✓ Alle indekser og antall nivåer i flernivåindekser
- ✓ Antall distinkte verdier for hvert attributt og selektivitet (andel av postene som tilfredsstiller et likhetskrav) – med disse to kan vi beregne gjennomsnittlig antall poster som vil bli returnert med likhet (alltid 1 for en primærnøkkel)

Noen av dette er enkelt å vedlikeholde fordi det endres sjelden, f.eks. informasjon om indeksene. Andre ting endres dynamisk, og er tunge å holde helt á jour. Optimaliseringen trenger imidlertid ikke

helt oppdaterte og eksakte tall. Derfor kan slike data oppdateres innimellom, til faste tider eller når serveren ellers er ledig.

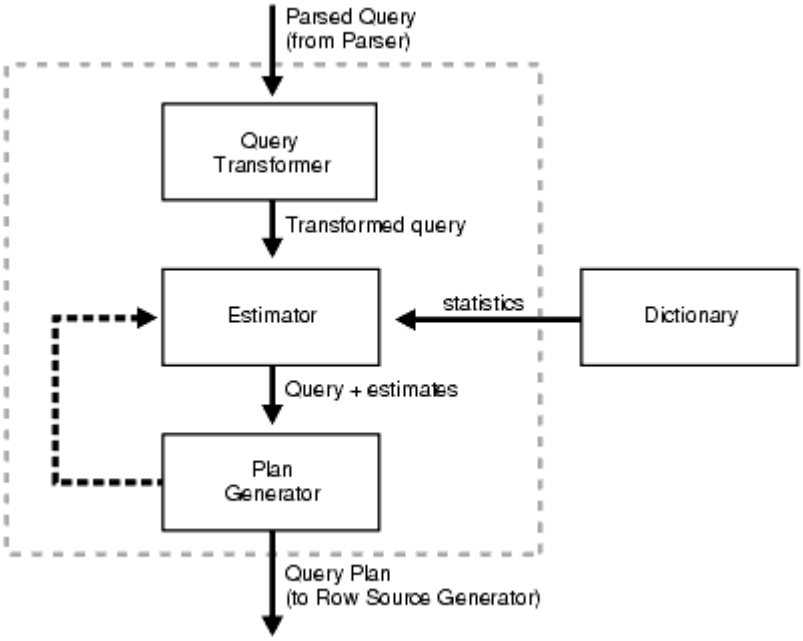
En spesiell mulighet som gjelder ikke-nøkkel attributter er å lage et histogram med angivelse av hvor mange det er av hver verdi, f.eks. antall menn og antall kvinner, eller antall som er ansatt i hver avdeling.

18.8.3 og 18.8.4 Eksempler (kursorisk)

I disse kapitlene gjengis er eksempler på noen kostnadsberegninger.

18.9 Kostnadsoptimalisering i Oracle (kursorisk)

Oracle tilbyr både heuristisk (som de kaller "rule-based") og kostnadsoptimalisering. Normalt brukes kostnadsoptimalisering automatisk[9]. Nedenstående figur er hentet fra dokumentasjonen:



Oracles estimator beregner tre mål for en spørring:

- ✓ selektivitet som er basert på innsamlet statistikk
- ✓ kardinalitet som er en beregning av antall rader
- ✓ kostnad som beregnes som en kombinasjon av I/O, CPU-bruk og minnebruk

Oracle prøver forskjellige execution plans men stopper når den beste hittil har lav kostnaden. Sålenge kostnaden er høy, prøves flere planer.

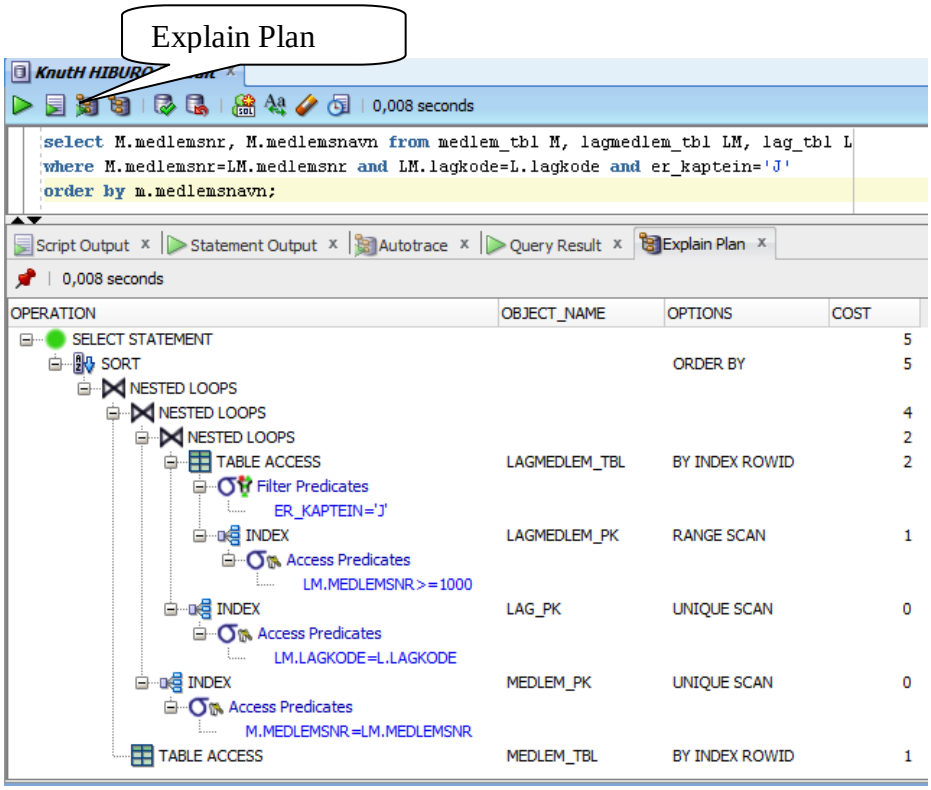
Hvis man mener å vite mer enn Oracle databasen om hva som lønner seg, kan man gi "hints". Syntaksen er temmelig kompleks, og det vil føre for langt å gå inn på det her. Dette eksemplet viser ifølge dokumentasjonen det meste som er tillatt:

[9] Du kan lese mye om dette i dokumentasjonen på
http://download.oracle.com/docs/cd/B28359_01/server.111/b28274/optimops.htm#i55044

```
SELECT /*+ LEADING(e2 e1) USE_NL(e1) INDEX(e1 emp_emp_id_pk)
           USE_MERGE(j) FULL(j) */
    e1.first_name, e1.last_name, j.job_id, sum(e2.salary) total_sal
  FROM employees e1, employees e2, job_history j
  WHERE e1.employee_id = e2.manager_id
    AND e1.employee_id = j.employee_id
    AND e1.hire_date = j.start_date
  GROUP BY e1.first_name, e1.last_name, j.job_id
  ORDER BY total_sal;
```

Jeg har skrevet hintene med fet skrift.

Den enkleste måten å se planen på, er å klikke knappen "Explain Plan" i Oracle SQL Developer:



Som du ser, viser Developer da execution treet med ekstra opplysninger om tilgangsmåte og beregnet kostnad. Du vil ellers kjenne igjen symbolene i treet. Det er litt overraskende at projeksjonen (velge ut to av kolonnene) ikke er med, men kanskje er det ikke nødvendig – isteden returneres bare to av kolonnene fra resultattabellen.

18.10 Semantisk optimalisering (kursorisk)

I semantisk optimalisering brukes databasens metadata (skranker og domener) til optimaliseringen. Noen eksempler:

- ✓ Det er angitt at ingen kan ha mer enn én ektefelle. Da er det heller ikke nødvendig å se etter flere ektefeller når først én er funnet.
- ✓ Hvis *null* ikke er tillatt, kan WHERE x IS NOT NULL fjernes.
- ✓ Hvis noe er deklarert unikt, er dubletter uaktuelt og kanskje kan man se bort fra DISTINCT.
- ✓ Hvis en skranke angir at en verdi skal være >=1000, har det ingen hensikt å søke etter verdier under 1000 selvom spørringen ber om det.

Man må imidlertid ikke glemme at det vil ta tid å lete igjennom alle reglene, så semantisk optimalisering har også en kostnad.

E&N 20 Transaksjoner

Tidligere (kapittel 18) har vi sett på optimalisering av spørringer o.l. Da var vi på et overordnet, logisk nivå med tabeller, JOIN, INSERT o.l. Slike overordnede operasjoner krever en fysisk gjennomføring i databasens filer. Det er disse fysiske operasjonene som er tema for dette kapittelet.

Databasens data og metadata endres over tid. Alle verdiene i databasen sett under ett, utgjør *databasens tilstand*. En tilstand varer en viss tid. Mens dataene endres, har databasen ingen tilstand (state), men er under endring (transition).

Databasen er *konsistent* hvis alle regler er overholdt, dvs. reglene for entitetsintegritet, domeneintegritet og referanseintegritet. Disse vil vanligvis være implementert som beskrankninger (constraints). I tillegg ønsker vi jo ikke at databasens data skal inneholde logiske selvmotsigelser, men det er ofte opp til brukerne. (Merk at begrepet *konsistent* ikke er det samme som *konsekvent*. Det siste innebærer å gjøre det samme hver gang, og det er det ikke noe krav om i databaser.)

En *transaksjon* (transaction) er en serie handlinger som bringer databasen fra én konsistent tilstand til en annen konsistent tilstand. Mens transaksjonen foregår, er databasen under endring (og har ikke nådd en tilstand), og kravet om konsistens gjelder da ikke. Transaksjonen må enten gjøres fullt ut, eller ikke i det hele tatt – de er altså *atomære*.

20.1 Introduksjon

Databaser er enten enbruker (single user) eller flerbruker (multiuser). Enbrukerdatabaser opptrer oftest som små, lokale databaser på egen PC. Flerbrukerdatabaser er det vanlige, og kjøres på en tjener med mange klienter som vil ha gjennomført endringer.

Flerbrukerdatabaser kjører flere prosesser løpende (en prosess er lik et program med eget minneområde – ikke å forveksle med tråder). Hvis det er flere prosessorer, kan prosessene kjøres parallelt, men vanligst er tidsdelt prosessering der prosessene startes og stoppes og kjører "på skift" (interleaved processing).

Uansett er databasens data lagret i filer på et ytre lager, på vanlig måte.

20.1.2 Read/write operasjoner

De grunnleggende operasjoner på en database er

read-item(X)
write-item(X)

der X er en navngitt item i databasen. X blir enten lest (read-item) eller skrevet/overskrevet (write-item) til/fra en programvariabel som her også kalles X.

En read-item(X) vil føre til at en blokk leses fra filene i databasen til bufferet. Blokken inneholder det navngitte item i databasen. Variabelen X henter sin verdi i blokken.

En write-item(X) krever flere deloperasjoner:

1. Gjennomfør read-item(X)
2. Endre data X i bufferet
3. Skrive bufferet tilbake til filen

Det vil vanligvis være flere buffere og det er ikke sikkert at deloperasjon 3 gjennomføres med en gang. Hvis ikke, kan endringen gå tapt om maskinen går ned og endringen vil være ukjent for andre brukere.

Læreboken har et eksempel i figur 20.2.

Vi kan tenke oss to transaksjoner T_1 og T_2. Den første vil flytte én elev fra klass 2B til 2C, den andre vil legge til én ny elev i klasse 2B. De to transaksjonene gjør følgende handlinger:

Transaksjon T_1
read_item(2B.antall)
2A.antall:=2B.antall-1
write-item(2B.antall)
read_item(2C.antall)
2B.antall:=2C.antall+1
write_item(2C.antall)

Transaksjon T_2
read_item(2B.antall)
2A.antall:=2B.antall+1
write_item(2B.antall)

For begge disse gjelder at de ikke må avbrytes halvveis – da blir databasen feil. Vi antar at det er 30 elever i hver klasse før oppdateringen begynner.

20.1.3 Hvorfor samtidighetskontroll?

Når flere klienter "samtidig" (dvs tidsdelt og uavhengig av hverandre) gjør endringer, oppstår det problemer:

1. Lost update = den som lagrer sist, vinner
2. Dirty read = leser data som ikke er endelige
3. Incorrect summary = dataene endres mens aggregeringen gjøres

Disse problemene må løses på en eller annen måte.

De to transaksjonene i forrige avsnitt, kan f.eks. flettes slik:

Transaksjon T_1	**Transaksjon T_2**
read_item(2B.antall)=30	
2A.antall:=2B.antall-1=29	
	read_item(2B.antall)=30
	2A.antall:=2B.antall+1=31
write-item(2B.antall)=29	
	write_item(2B.antall)=31
read_item(2C.antall)=30	
2B.antall:=2C.antall+1=31	
write_item(2C.antall)=31	

Dette gir "lost update" fordi *2B.antall* som er korrigert av T_1 blir overskrevet av T_2. Begge klassene ender med 31 elever.

Anta nå at T_1 feiler etter å ha skrevet *2B.ant*. Den må da settes tilbake til sin opprinnelige verdi. I mellomtiden har T_2 lest, korrigert og skrevet *2B.ant*. Oppdateringen til T_2 går tapt.

Transaksjon T_1	**Transaksjon T_2**
read_item(2B.antall)=30	
2A.antall:=2B.antall-1=29	
write-item(2B.antall)=29	
	read_item(2B.antall)=29
	2A.antall:=2B.antall+1=30
	write_item(2B.antall)=30
ABORT	
Restore(2B.antall)=30	

Dette er en "dirty read" fordi T_2 leser data som ikke er endelige. Det skulle blitt 31 i klasse 2B, men det blir bare 30 i begge.

Anta så at en transaksjon T_3 skal summere antallet i klasse 2A, 2B og 2C. Det flettes med T_1 som skal oppdatere antallet i 2B og 2C som ovenfor:

Transaksjon T_1	**Transaksjon T_3**
	read_item(2A.antall)=30
	sum:=2A.antall=30
read_item(2B.antall)=30	
2A.antall:=2B.antall-1=29	
write-item(2B.antall)=29	
	read_item(2B.antall)=29
	sum:=sum+2B.antall=59
	read_item(2C.antall)=30
	sum:=sum+2C.antall=89
read_item(2C.antall)=30	
2C.antall=2C.antall+1=31	
write_item(2C)=30	

Dette blir "incorrect summary" fordi T_3 leser *2B.antall* etter at T_1 har redusert det med 1, og *2C.antall* før T_1 har rukket å øke det med 1. T_3 får summen 89, men det skulle vært 90.

Læreboken viser et liknende eksempel i figur 20.3.

20.1.4 Recovery

Når det oppstår en feil, må systemet "ta seg inn" og fortsette operasjonene – vi ønsker jo at databasen skal være konsistent og tilgjengelig. For å få det til må DBMS logge alt som skjer, så den kan "rygge", fjerne oppdateringer og gjenta operasjonene senere.

Boken deler feiltypene i tre:

1. Transaction failure, f.eks. overflow, logiske feil, deling med null, avbrutt av brukeren eller databasen selv, data finnes ikke osv.
2. Media failure, f.eks. diskfeil og kommunikasjonssvikt
3. System failure, f.eks. system crash, strømbrudd, tyveri/vandalisme, virus.

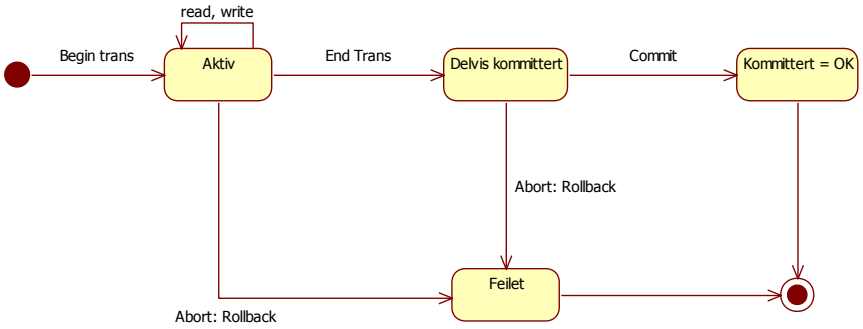
De tre typene er ikke så viktige, men transaksjonsfeil skjer i ett sett og derfor bør DBMS selv kunne gjenopprette databasen og rydde opp i feilen. De to andre feiltypene er sjeldne og krever menneskelig inngripen. Resten av kapitlet konsentrerer seg derfor om transaksjonsfeil.

20.2.1 Transaksjoners tilstander

Transaksjoner gjennomgår tilstander. En recovery manager holder orden på transaksjonens operasjoner, dvs.

- ✓ begin transaction = transaksjonen begynner – databasens tilstand antas konsistent
- ✓ read/write = det leses og skrives til/fra filer i databasen
- ✓ end transaction = transaksjonen er ferdig, alle read/write er gjennomført
- ✓ abort/rollback = det har skjedd en feil, databasens tilstand ved "begin transaksjon" må gjenopprettes
- ✓ commit = alt er OK, endringene gjennomføres og gjøres tilgjengelig for alle

Slik kan tilstandsdiagram for en transaksjon se ut (med UML-syntaks):



Mens transaksjonen er "aktiv", kan det skje feil som krever abortering, f.eks. at et item ikke finnes, divisjon med null osv. Da må alle endringer slettes, dvs. det må gjøres "rollback". Nødvendige opplysninger finnes i loggen. Hvis transaksjonen avslutter uten feil, kan det skje feil i tilstand "delvis kommittert", f.eks. at transaksjonen kolliderer med en annen transaksjon. Igjen må da transaksjonen rulles tilbake. Ellers gjennomføres en kommittering, og dataene skrives endelig til databasen.

20.2.2 Logg

Alt som skjer innen en transaksjon logges *til disk* (slik at den beholdes ved system crash). For å kunne finne igjen en transaksjon og knytte sammen handlingene, får hver transaksjon en systemgenerert ID. Det som logges er:

1. start, T. Transaksjonen T har startet
2. write-item, T, X, gammel verdi, ny verdi (Oracle kaller dem :OLD og :NEW). Den gamle verdien er det behov for ved tilbakerulling.
3. read-item, T, X. Denne er ikke strengt nødvendig for recovery, men logges ofte allikevel.
4. commit, T. Alt gikk bra med transaksjon T, *og* det er notert i loggen *og* loggen er skrevet til disk.
5. abort, T. Noe gikk feil – transaksjonen er fjernet *og* det er notert i loggen *og* loggen er skrevet til disk.

Merk at loggen ikke er ferdig før den er fysisk skrevet til disk. For å spare diskaksesser (trege!) er det allikevel vanlig å skrive loggen i et buffer, men før en commit er akseptabel, må bufferet være lagret. Dette kalles "force writing" av bufferet.

Recovery manager holder altså orden på denne loggen. I tillegg kan recovery manager gjennomføre

1. undo = recovery manager "rygger" gjennom loggen og skriver alle gamle verdier for en transaksjon tilbake til databasefilene
2. redo = recovery manager kjører forlengs gjennom loggen og utfører alle handlinger til en transaksjon på ny.

20.3 ACID

Ønskemålet for en transaksjon er at den skal være ACID, dvs.

Atomic	=	transaksjonen må gjøres helt eller ikke i det hele tatt
Consistency preserving	=	transaksjonen bringer databasen fra én, konsistent tilstand til en annen konsistent tilstand
Isolated	=	transaksjonen må ikke bli influert av andre, samtidige transaksjoner
Durable	=	transaksjonens endringer i databasen er permanente og ikke gå tapt ved et evt. system crash

Kravet om at transaksjonene skal være atomær, er recoverysystemets ansvar. Det gjennomføres gjennom logging og *undo* ved feil. Det kan også inntreffe at transaksjonen i seg selv er OK, men at den kolliderer med andre. Da kan det være aktuelt å først gjøre *undo* og deretter *redo* senere.

Kravene til konsistens vil DBMS selv ta seg av ved å kontrollere at alle endringer skjer i henhold til beskrankninger og andre kontrolltiltak (f.eks. at tabellnavn og kolonnenavn finnes). Vi forventer at de som skrev DBMS-programmet har håndtert dette. Vår utfordring er å skrive korrekte og tilstrekkelige beskrankninger.

Å holde transaksjonen isolert, er en oppgave for *concurrency control system*. En enkel måte å gjøre det på er at hver transaksjon utføres alene til den er fullt gjennomført – altså ingen parallell eksekvering av transaksjoner. Det kalles "serial schedule" – se senere. Dette er imidlertid umulig i praksis, da det medfører mye venting. Isteden må man "ta noen sjanser" og så heller *undo* og *redo* når feil oppstår.

Det er laget nivåer for isolasjon, men det legger vi ikke vekt på her.

20.4 Schedules

Schedule (plan) er rekkefølgen operasjonene gjøres i. Mer formelt innebærer det at *den enkelte operasjon alltid gjøres i den rekkefølgen som transaksjonen har bedt om, men transaksjonen flettes med andre transaksjoner*.

Hensikten er å unngå at en transaksjon må vente til den foregående er helt ferdig. På den måten øker databasens "hastighet". På den annen side innfører vi da nye muligheter for feil som må håndteres.

Det er *scheduler system* som lager schedules.

I det følgende brukes følgende notasjon:

r=read-item, w=write-item, c=commit, a=abort.
Indeksene angir transaksjonsnummeret.

Ovenfor (side 34) viste jeg et eksempel med oppdatering og summering av klasser (gjentatt her):

Transaksjon T_1	**Transaksjon T_3**
	read_item(2A.antall)=30
	sum:=2A.antall=30
read_item(2B.antall)=30	
2A.antall:=2B.antall-1=29	
write-item(2B.antall)=29	
	read_item(2B.antall)=29
	sum:=sum+2B.antall=59
	read_item(2C.antall)=30
	sum:=sum+2C.antall=89
read_item(2C.antall)=30	
2C.antall=2C.antall+1=31	
write_item(2C)=30	

Denne flettingen vil ha følgende schedule:

$S_c = r_3(2A), r_1(2B), w_1(2B), r_3(2B), r_3(2C), r_1(2C), w_1(2C)$

Du ser at transaksjonen T_1 og T_3 er "flettet". Da kan det oppstå *konflikt* mellom to operasjoner.

Det er konflikt mellom to operasjoner hvis *alle* disse tre kriteriene er oppfylt samtidig:

1. De to operasjonene hører til hver sin transaksjon,
2. de aksesserer det samme item X og
3. minst én av operasjonene er write-item(X).

I eksemplet S_c har jeg her merket konfliktene:

$S_c = r_3(2A), r_1(2B), \mathbf{w_1(2B)}, \mathbf{r_3(2B)}, \mathbf{r_3(2C)}, r_1(2C), \mathbf{w_1(2C)}$

Den første er en *mulig* konflikt, men den er faktisk OK fordi skrivingen skjer før lesingen. Den andre er en konflikt som skaper feil, fordi T_1 skriver en verdi som er i bruk i transaksjon T_3.

En schedule kan være *komplett*. Da er alle disse tre kravene oppfylt:

1. alle operasjonene til alle involverte transaksjoner, inkludert commit og abort, kommet med,
2. alle operasjonene til hver enkelt transaksjon står i den rekkefølge transaksjonen ber om og
3. det er bestemt en fornuftig rekkefølge for alle operasjoner i konflikt – andre operasjoner kan flettes fritt

Man vil altså gjerne lage en komplett schedule for alle de transaksjonene som er aktive i systemet. Deretter skal den gjennomføres inntil alle transaksjoner i flettingen enten er abortert eller kommittert. I praksis vil det komme nye transaksjoner til hele tiden mens en bestemt schedule utføres. Disse nye må også flettes inn i schedule, som derved må endres. Konsekvensen er en flytende schedule som endres hele tiden (hver gang en ny transaksjon kommer til).

20.4.2 Recovery-muligheter

Schedules er mer eller mindre vanskelig å redde fra feil. Noen er svært tunge og noen simpelthen umulige.

- ✓ *Recoverable schedule* innebærer at det *er* mulig å rette opp feil. Da er det aldri behov for å ta rollback på en transaksjon som er kommittert.
- ✓ *Cascading rollback* er recoverable, men krever rollbacks som er nødvendige fordi en *annen* transaksjon feilet. Schedules bør være *cascadeless*.
- ✓ *Strict schedules* er recoverable og cascadeless. De vil vente med å lese/skrive X til den transaksjonen som skrev X er kommittert. Da er recovery enkelt.

Man kunne altså ønske at alle schedules var *strict* men dessverre innebærer de *venting*.

20.5.1 Serialiserbarhet

- ✓ *Serial schedules* innebærer at alle operasjoner for transaksjon T1 gjøres før operasjonene i T2 osv. Da må T2 vente til T1 har kommittert. Det er håpløst i praksis, p.g.a all ventingen, f.eks. venting på I/O eller mens brukeren tar en beslutning. Siden vi må anta at transaksjoner er uavhengige (ref. ACID) blir det likegyldig hvilken transaksjon som tas først, men sluttresultatet blir ikke nødvendigvis det samme.
- ✓ *Non-serial schedules* er transaksjoner som er flettet.

En schedule sies å være *serialiserbar* (serializable) hvis den er *ekvivalent* med en eller annen seriell schedule. Siden alle serielle schedules er korrekte, vil også en serialiserbar schedule være det.

Ovenfor (side 34) viste jeg to transaksjoner som endret antallet i to skoleklasser. Hvis de utføres etter hverandre (serielt) gir de samme resultat:

Transaksjon T_1
read_item(2B.antall)=30
2A.antall:=2B.antall-1=29
write-item(2B.antall)=29
read_item(2C.antall)=30
2B.antall:=2C.antall+1=31
write_item(2C.antall)=31
Transaksjon T_2
read_item(2B.antall)=29
2A.antall:=2B.antall+1=30
write_item(2B.antall)=30

Transaksjon T_2
read_item(2B.antall)=30
2A.antall:=2B.antall+1=31
write_item(2B.antall)=31
Transaksjon T_1
read_item(2B.antall)=31
2A.antall:=2B.antall-1=30
write-item(2B.antall)=30
read_item(2C.antall)=30
2B.antall:=2C.antall+1=31
write_item(2C.antall)=31

Som det fremgår ender det med at 2B får 30 elever, mens 2C får 31 uansett hvilken rekkefølge de utføres i. De sies da å være *result equivalent* (neste avsnitt).

I boken finnes en tilsvarende figur 20.5.

Ekvivalens innebærer:

1. To schedules sies å være *result equivalent* hvis de til slutt resulterer i samme tilstand for databasen. *Eksempel i læreboken figur 20.6.*
2. To schedules sies å være *conflict equivalent* hvis rekkefølgen på to operasjoner i konflikt er den samme i begge schedules. Schedulen sies da også å være *conflict serializable*.
3. To schedules er *view equivalent* hvis de leser de samme dataverdiene. Dette er vanskelig å sjekke ut (algoritmen er "NP-hard" dvs. det er usannsynlig å finne en noenlunde effektiv algoritme).
4. To schedules kan være non-serializable, men allikevel korrekte når man vet noe mer om den aktuelle prosessen – f.eks. legge til/slette studenter i en klasse. Da er jo rekkefølgen uten betydning.

20.5.2 Sjekke conflict serializability

Det finnes algoritmer for å sjekke om to schedules er conflict serializable. En algoritme for slik sjekking er gjengitt *i læreboken på side 742*. Den innebærer å lage en presedensgraf (også kalt serialiseringsgraf). Relasjonen i grafen er retningsbestemt og angir rekkefølgen transaksjonene tas i. Grafen for de schedules *a* til *d* som er *gjengitt i lærebokens figur 20.5*, finnes i lærebokens figur 20.7. Der er schedule *c* problematisk, fordi den har en *syklus* som krever at T_1 må komme foran T_2 *og omvendt*. Ellers ser vi at schedule *d* tilsvarer schedule *a*. Den er følgelig serialiserbar.

Ta også en titt på bokens figur 20.8 der (a) = (e) har problemer p.g.a flere sykler.

Selvom det altså finnes slike algoritmer, så gjøres det ikke i praksis. Isteden anvendes *regler* som sikrer at schedulen blir serializable – se neste kapittel.

20.5.3 Regler

Hvis man lar transaksjonene gjøre som de vil, og så etterpå tester dem for serialiserbarhet, vil mange transaksjoner måtte rulles tilbake. Derfor er det mer praktisk å flette dem etter visse *regler*.

Når alle reglene anvendes på hver eneste transaksjon hver for seg, så sikrer man at schedule blir serialiserbar – og følgelig korrekte.

En vanlig teknikk (altså regel) er *låsing*. En kolonne, post, tabell o.l. kan "låses" slik at andre transaksjoner ikke kan bruke dem. De andre må da vente. Imidlertid oppstår det da nye problemer med

"dead-locks" o.l. (når T_1 venter på at T_2 skal bli ferdig med et item og omvendt). Vi skal se på låseteknikker i neste kapittel.

En annen mulighet er *timestamp* der hver transaksjon får et tidspunkt knyttet til seg for når de startet. Alle konflikter løses da ved at tiden bestemmer rekkefølgen for alle operasjoner i konflikt.

20.6 Transaksjoner i SQL

SQL har ikke kommandoen *Begin transaction*, men den gjelder fra det øyeblikk en SQL-setning starter eksekvering. *Commit* og *Rollback* (evt. *Abort*) finnes og avslutter en transaksjon på den ene eller den andre måten. Man kan angi at man bare vil lese (*read only*) eller både lese og skrive (*read write*). Videre kan man angi *Isolation Level* for hvor nøye man vil være med isolasjonskravet.

Oracle har mye slikt innebygget og automatisk – og den gjør det stort sett bedre enn vi klarer selv – men det er mulig å angi rollback punkter osv.

E&N 21 Samtidighetskontroll

Vi husker at transaksjoner skal være ACID (Atomic, Consistant, Isolated og Durable). For å få til dette – særlig isolasjon – samtidig som vi ikke vil gi slipp på flettet prosessering, må schedules være serialiserbare.

For å få til det, kan man innføre **regler** som gjelder for hver transaksjon og som sikrer at schedule blir serialiserbar. Reglene utgjør samlet en **protokoll**. *Samtidighetskontroll* blir da en viktig del av disse reglene. De sikrer at ikke to transaksjoner "samtidig" manipulerer samme data. Det er tre typer (minst):

a) Låsing
b) Timestamps (tidsstempling)
c) Multiversion
d) Validering i etterhånd – "optimistiske protokoller"

Det er viktig å vite hvor stor datamengde som er den minste, navngitte del (*X-en i write(X)*). Er det en feltverdi, en rad, en kolonne, en tabell? Og hva med indeksen som jo har dubletter av dataene?

21.1 Tofase låsing

Når data "låses", innebærer det at DBMS noterer seg at én transaksjon må ha dataene mer eller mindre for seg selv. Låser styrer en synkronisering av transaksjonene – én transaksjon må vente til en annen er ferdig (omtrent som en låst toilettdør). Man kan låse større eller mindre datamengde, og låsene kan være helt eller delvis (noen operasjoner kan fortsatt tillates for andre transaksjoner).

21.1.1 Låstyper

Binær lås er enten helt av eller helt på. Reglene for hver transaksjoner – med en slik lås – kan være:

1. *lock_item(X)* før *read_item(X) eller write_item(X)*
2. *unlock_item(X)* når alle *read_item(X)* og *write_item(X)* er utført

og dessuten noen opplagte:

3. ikke lås noe du allerede har låst
4. ikke lås opp noe du ikke har låst

Bare én transaksjon kan da ha item X låst av gangen. Andre som vil låse X settes i kø.

Dette blir ganske strengt – ofte kan man tillate *noe* tilgang parallelt. Binær låsing er følgelig lite brukt.

Flerverdilås (multiple mode locking) også kalt **read/write lock** har mulighet for gradering av låsen. Man kan enten

a) tillate andre å lese – read_lock(X) – kalt **shared** eller
b) kreve ekslusiv tilgang – write_lock(X) – kalt **exclusive**

En transaksjon som bare skal lese X, ber om *read_lock(X)*. En som også skal skrive X, ber om *write_lock(X)*. Da kan flere be om å *read_lock(X)* samtidig, og det er OK. Men hvis data er *read_locked* får ingen gjennomført *write_lock(X)* på samme item. De må vente i kø. Hvis køen skal være "rettferdig", må også *read_lock(X)* settes i kø hvis en *write_lock(X)* venter (altså hvis det er kø for X).

Regler for flerverdilås:

1. *read_lock(X)* eller *write_lock(X)* før *read_item(X)*
2. *write_lock(X)* før *write_item(X)*
3. *unlock(X)* når du er ferdig

og dessuten noen opplagte:

4. ikke lås opp noe du ikke har låst
5. ikke *write_lock(X)* hvis du allerede har (en eller annen) lås på *X* allerede.

Det siste ville ført til at transaksjonen ender med å vente på seg selv.

Det kan tillates at en transaksjon endrer låstype:

a. **Upgrade lock:** Fra *read_lock(X)* til *write_lock(X)* men bare hvis det kun er denne transaksjonen selv som har read_lock(X), ellers må oppgraderingen i kø.
b. **Downgrade lock:** Fra *write_lock(X)* til *read_lock(X)* når som helst.

21.1.2 To-fase låsing

Dessverre holder ikke reglene ovenfor alltid – en schedule kan bli ikke-serialiserbar selvom alle transaksjonene følger reglene. For å garantere serialiserbarhet, må man kreve **to-fase låsing** for alle transaksjoner. To-fase låsing innebærer at all låsing, inkludert evt. upgrade locks, gjøres *før* første unlock. Sålenge låsing pågår, sies transaksjonen å være i **expanding (=growing=first) phase**. Såsnart første unlock er gjennomført, kan bare nye unlocks gjennomføres og transaksjonen sies å være i **shrinking (=second) phase**. Derav navnet to-fase låsing.

I eksemplet i læreboken fig. 21.3 viser (a) de to transaksjonene, og (b) viser resultatet hvis de to transaksjonene kjører serielt og (c) viser at resultatet ikke blir like noen av de serielle. Reglene for flerverdilås følges, men ikke reglene for to-fase.

Siden to-fase låsing låser tidlig og holder låsen lenge, øker sjansen for venting. *Se lærebokens figur 21.4* der de samme transaksjonene gjennomfører to-fase låsing. Også sjansen for deadlocks (a venter på b som venter på a) øker.

Konservativ låsing er enda strengere. Da må hver transaksjon først deklarere alle items som skal leses (**read-set**) og skrives (**write-set**). Så må alle items låses før transaksjonen får lov til å gjøre noe som helst med dataene, ellers må den settes i kø og får heller ikke låse noe som helst. Da blir det enda mer venting, men ingen deadlocks.

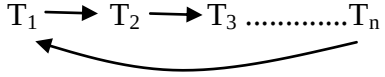
Rigorøs låsing bytter strategi, ved at transaksjonen ikke får lov til å låse opp før den kommitterer. Da er altså transaksjonen hele tiden i growing phase. Dette garanterer en strict schedule (jfr. kap. 20.4.2 – låsing opprettholdes helt til transaksjonen har kommittert).

Strict låsing er noe "slappere" enn rigorøs, ved at bare *write_locks* må vente på kommittering. *Read_locks* kan åpnes tidligere (men fortsatt bare i shrinking phase). Strict låsing kan gi deadlocks men garanterer strict schedule.

Selvom to-fase låsing garanterer serialiserbarhet, vil den virke innskrenkende, fordi noen serialiserbare schedules blir "forbudt".

21.1.3 Deadlocks, starvation

Deadlocks, som kanskje kan kalles "vranglås", innebærer at to eller flere transaksjoner venter på hverandre. Se nedenstående figur, der pilene betyr "venter på":



Ventingen er rekursiv, og T_n venter dessuten på en "eldre" transaksjon (begge deler er dårlige tegn). *I figur 21.5 fra boken*, ser vi også at de involverte låsene ikke behøver å gjelde samme item – poenget er at *transaksjonene* venter på hverandre.

Med konservativ to-fase kontroll, der alle låser gjøres først, unngår man deadlock. I praksis er det ikke så enkelt, fordi applikasjonene ikke vet på forhånd hvilke låser de trenger – det gjøres valg under eksekveringen. Andre, tilsvarende, er like upraktiske.

Med **timestamp** tilordnes hver transaksjon enten en økende teller eller et klokkeslett som er nøyaktig nok til at ingen kan bli like. Dette blir egentlig det samme, men husk at en teller "går rundt" eller gir feil når vi når maksimalt heltall.

Når to transaksjoner er i deadlock, aborteres den "yngste" (høyest nummer). Den restartes senere med samme timestamp, så den ikke mister sin plass i låsekøen. (Det hevdes å være to forskjellige strategier her, men de er *svært* like. Uansett må den som startet sist, vike – "age before beauty".) Etter at en yngre transaksjon er abortert til fordel for en eldre, vil alle transaksjonene bare vente på eldre, og da kan ingen rekursjon oppstå – følgelig heller ingen deadlocks. Schedule blir **deadlock free**.

Alternativt vil **no-wait strategy** aldri tillate venting. Isteden aborteres transaksjonen straks, og restartes etter en viss tid. Med "forsiktig venting" (**cautious wait**) sjekkes det om den som blokkerer venter på noe. Hvis ja, aborteres den sist ankomne, ellers venter man:

- ✓ **Abort T_3:** T3 vil låse noe som T_2 har låst, og T_2 venter på T_1
- ✓ **Vent T_3:** T_3 vil låse noe som T_2 har låst, og T_2 venter ikke på noen

Med "forsiktig venting" unngås deadlocks.

Istedenfor å unngå deadlocks, kan vi satse på å **oppdage deadlocks** og så gjøre noe med dem. En enkel måte å gjøre det på, er å tegne og vedlikeholde grafer. (Istedenfor grafer kan man lage en ordnet liste. Når en transaksjon må vente på en bak seg, er det en deadlock.)

Når en deadlock oppdages, må én transaksjon aborteres (den er **victim**). Man vil gjerne velge en som har gjort lite med databasen, men en approksimasjon er jo alltid å ta den "yngste".

Istedenfor å (a) unngå deadlocks og (b) oppdage deadlocks, kan man (c) vente bare en viss tid – deretter får transaksjonen **timeout**. Ved timeout aborteres transaksjonen og settes i kø på nytt. Deadlocks vil da løse seg selv etter hvert, men når man innfører timeouts, kan man jo også få timeouts av andre grunner.

Starvation oppstår når en transaksjon stadig må vente, eller aborteres og restartes, samtidig som de andre går normalt. Det kan skje hvis strategien er unfair. Enten må alle stå i én FIFO kø, eller – hvis noen gis prioritet – må prioriteten til en transaksjon økes i takt med ventetiden. (Mange legekontorer praktiserer det siste, tror jeg.) Med timestamp for hver transaksjon unngås starvation (prioriteten øker med "elde").

21.2. Timestamp ordering

Dette er et alternativ til låsing. Siden det ikke benyttes låsing, blir det heller ingen deadlocks.

Bakgrunn:

1. En rent seriell schedule er alltid riktig (dvs de riktige operasjonene gjøres i databasen – dataene kan jo allikevel være feil)
2. Hvis vi hadde kjørt rent serielt, så ville den transaksjonen som startet først, gjort alle sine transaksjoner før neste transaksjon
3. Når vi ønsker å flette transaksjoner (interleaved processing) sørger vi for at T_1 som startet først) får gjort sine operasjoner før T_2 som startet senere, men dette har kun betydning for operasjoner i **konflikt**.

Dette sikrer ikke bare at schedule er serialiserbar, men også at schedule tilsvarer en rent, seriell prosessering. Dermed blir schedule garantert riktig.

For å få dette til, bruker man følgende timestamps:

1. read_TS(X) er timestamp for den transaksjonen som sist leste X
2. write_TS(X) er timestamp for den transaksjonen som sist skrev X

Med disse to ekstra verdiene – som databasen må ta vare på – kan man sikre korrekt schedule på minst tre, forskjellige måter.

Måte 1: Basic Timestamp Ordering

Hver gang en transaksjon ber om read_item(X) eller write_item(X), sjekkes timestamp for transaksjonen TS(T_1) med de to read_TS(X) og write_TS(X). Hvis den riktige rekkefølgen blir brutt, så aborteres transaksjonen T_1 (f.eks. hvis én av dem er 2). Den restartes nå med ny timestamp (og faller derved bakover i køen). Hvis det da viser seg at T_2 har brukt verdier som T_1 har skrevet (som rulles tilbake), så må også T_2 aborteres. Vi får altså da en **cascading rollback**. Oppsummering av reglene:

1. T_1 ber om read_item(X) og en senere transaksjon har skrevet slik at write_TS(X) > TS(T_1): Aborter og sjekk kaskader.
2. T_1 ber om write_item(X) og en senere transaksjon har lest X, slik at read_TS(X) > TS(T_1): Aborter T_1 og sjekk kaskader.

Vi får ingen deadlocks, men mange undo og redo, og vi kan få starvation.

Måte 2: Strict Timestamp Ordering

Her brukes en strict schedule der en transaksjon T_1 som vil lese/skrive X som en senere transaksjon har skrevet, må vente til den andre transaksjonen har kommittert eller abortert. Det innebærer en simulert låsing, men den gir ingen deadlocks fordi all venting går bare én vei (som i strict schedule).

Måte 3: Thomas' Write Rule

Denne er kursorisk – se i boken.

21.3 Multiversion Concurrency Control

Her opprettholdes "gamle" verdier av data. Dette er default for Oracle, og forklares godt i Oracles dokumentasjon[10]. De sier det bl.a. der på denne måten: "All the data that the query sees, comes from a single point in time".

Databasen opprettholder altså mange versjoner av alle data, derav navnet "multiversion". I "temporal databases" er dette tatt helt ut, slik at alle data finnes i alle versjoner. I denne type database er det enkelt å gjennomfør multiversion concurrency control. I andre vil det koste ekstra lagringsplass og

[10] http://download.oracle.com/docs/cd/B28359_01/server.111/b28318/consist.htm#CNCPT020

ekstra prosessering for å finne "riktig" versjon. Boken beskriver to teknikker, men det går vi ikke dypt inn i her. De har regler for når det er nødvendig å lage en ny versjon av data. Denne teknikken kan fortsatt gi rollbacks (når T_1 vil lage en ny versjon av noe som en senere transaksjon allerede har lest).

21.3.2 Multiversion Two-Phase locking using Certify Locks

I denne teknikken brukes låsing med tre verdier: Read_lock(X), write_lock(X) og **certify_lock(X)**. Den siste tillates ikke sammen med hverken read_lock eller write_lock. Ved kommittering, kreves certify_lock på alle data som transaksjonen har write_lock på (dvs. en upgrade fra write_lock).

	Read	**Write**	**Certify**
Read	Tillatt	Tillatt	Ikke tillatt
Write	Tillatt	Ikke tillatt	Ikke tillatt
Certify	Ikke tillatt	Ikke tillatt	Ikke tillatt

Tre verdier (i tillegg til "ulåst"): Certify – eller én write sammen med mange read.

Når en transaksjon T_1 med denne teknikken oppnår en write_lock(X) som er ekskluderende for andre write_locks, blir det skapt en ny versjon X' av X. Andre transaksjoner kan fortsatt lese den kommitterte versjonen av X og T_1 kan skrive X'. Når T_1 vil kommittere, derimot, må den vente til ingen andre har read_lock(X). Først da kan den oppnå certicy_lock(X) og kommittere.

Denne teknikken kan gi deadlocks men ikke kaskader.

21.4 Optimistic Concurrency Control

Optimistisk samtidighetskontroll (også kalt validation concurrency control) virker slik at istedenfor å sjekke *før* operasjonen om den skal tillates, sjekker man *etterpå*. All oppdatering skjer da på kopier av data (altså *multiversion*). Vi får da tre faser for hver transaksjon:

1. **Read phase.** Data leses fra databasen, men endringer skrives til lokale kopier.
2. **Validation phase.** Det sjekkes om skriving av de lokale kopiene vil bryte med serialiserbarhet. I så fall aborteres og transaksjonen restartes senere.
3. **Write phase.** Dataene skrives fra de lokale kopier til databasen.

Det virker bra hvis transaksjonene ikke så ofte interfererer med hverandre, f.eks. når det er svært mye lesing og lite oppdatering.

21.1-21.4 Oppsummering

Vi ønsker jo flettet (interleaved) prosessering. Det gir imidlertid problemer med isolasjon (i ACID). For å avhjelpe dette vil vi ha en form for samtidighetskontroll. Vi kan ty til forskjellige teknikker:

a. Låsing
b. Timestamps
c. Multiversion samtidighetskontroll
d. Optimistic (validation) samtidighetskontroll

21.5 Granularity ("kornethet")

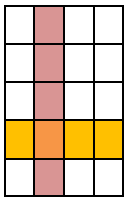
Alle samtidighetskontroller baserer seg på begrepet "item", kalt X ovenfor. Hvor stor denne X er, bør være åpent for diskusjon etter TANSTAFL-prinsippet. Vi kan f.eks. låse:

- ✓ hele databasen (aktuelt ved backup og restore)
- ✓ en hel fil (som kan ha data fra flere tabeller/indekser)
- ✓ en blokk[11] (minste adresserbare enhet)
- ✓ en tabell
- ✓ en rad
- ✓ en kolonne
- ✓ en verdi i en kolonne/rad

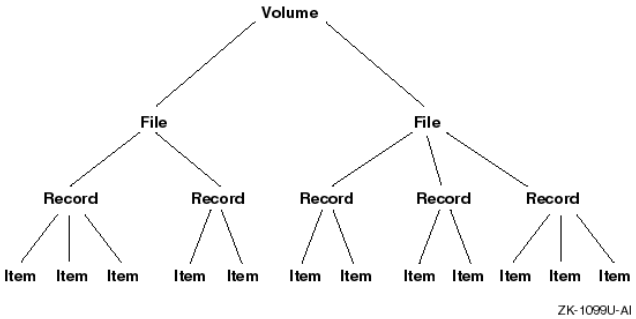
Noen trade-offs kan være:
Fin oppdeling gir stor grad av samtidighet (det er liten sjanse for konflikt når X er liten), men mange låser/timestamps/versjoner å holde orden på. Fin oppdeling gir følgelig stor overhead. Låsene krever også diskplass.

21.5.2 Multiple granularity

Databasen tilbyr ofte flere kornetheter. Da må låsesystemet også ta høyde for at en annen type konflikter, f.eks. når T_1 har låst en rad i en tabellen og T_2 vil låse en kolonne i samme tabell. Da er jo X forskjellig, men det er allikevel en konflikt. Slik kan det tegnes:



Noen ganger er dette enkelt.



Anta nå T_1 har låst en fil. En annen transaksjon T_2 vil låse en *Item*. Da kan T_2 enkelt bevege seg opp til roten og kontrollere om noe er låst på veien. I så fall må T_2 vente. Hvis derimot T_3 vil låse en *File* må samtlige underliggende noder traverseres i tillegg til alle noder opp til roten før man kan være sikker på at det ikke finnes låskollisjoner.

Boken har et tilsvarende eksempel i figur 21.7.

Man innfører da en ny type lås kalt **intention lock** for *underliggende* noder, som har tre verdier:

1. Intention-shared (IS): Shared lock er ønsket for en underliggende node.
2. Intention-exclusive (IX): En exclusive lock er ønsket for en underliggende node.
3. Shared-intention-exclusive (SIX): Denne noden er låst shared, men en ekslusiv lås er ønsket for en underliggende node.

[11] Det er ofte litt forvirring omkring begrepene. Jeg forstår det slik at *blokk* = en del av en fil, *page* = en del av en harddisk.

I tillegg finnes tre-verdi låser for den *aktuelle noden*, avhengig av hva som skal tillates samtidig:

4. Shared: Tillater andre å låse samme item shared.
5. Exclusive: Tillater ingen andre låser samtidighet.
6. Ingen lås.

Intention-låsene settes fra roten og ned til den ønskede node. Den indikerer hva slags lås transaksjonen ønsker for en underliggende node. Selve noden som transaksjonen vil låse, låses enten shared eller exclusive.

Basert på dette kan man lage regler for låsingen. Svært enkelt sagt får man ikke låse en node uten først å ha fått låst hele treet (inkludert roten) med tilsvarende intention lock. Man kan imidlertid gjerne ha flere transaksjoner som har samme intention (til og med en intention exclusive). Nodene over (nærmere roten) er egentlig ikke låst, transaksjonen har bare uttrykt – og fått registrert – et *ønske*. I tillegg gjennomføres to-fase låsing og en transaksjon får ikke låse opp noe, sålenge den har låser på underliggende noder. Med disse reglene kan man sikre mot kollisjoner *uten* å gjennomgå alle underliggende noder (fordi intention lock tas fra rot og nedover til noden ovenfor den man vil låse).

Boken har et eksempel med tre transaksjoner i fletting og bruk av intention locks. Det kan kanskje bidra til forståelsen av dette.

21.6 Concurrency control for indekser

Indekser skaper ekstra problemer, fordi de skal være "i takt med" data og fordi indekser i praksis er B-trær. Da starter all indeksering ved roten. Indeksen må følgelig låses på rotnivå. På den annen side kan låsene av nodene ovenfor noden droppes så snart riktig node er funnet – transaksjonen vil ikke bruke dem igjen.

21.7.1 Diverse, inkludert phantom records

En phantom record er en post som har blitt satt inn av en annen transaksjon T_1, mens en annen transaksjonT_2 holder på med alle poster av et slag som skulle inkludert den nye.

Eksempel: T_1 har låst alle studenter som tar fag INF315, fordi alle skal endres. "Samtidig" legger transaksjon T_2 til en ny student som tar faget. Det er tillatt, siden denne nye studenten ikke er låst. Den nye studenten blir imidlertid ikke oppdatert som den skulle av T_1.

Det er vanskelig å oppdage slike problemer, men én mulighet kan være å låse indeksen (hvis den er dense) da den nye posten skal inn der.

Et helt annet problem er brukerbeslutninger interaktivt. Brukeren tar beslutningen på grunnlag av data på skjermen, så de bør være låst exclusive. Men da må alle vente på denne brukeren som f.eks. tar lunsj. En mulighet er å programmere med låsing av finfordelt kornethet, men den vil fortsatt da hindre alle andre å låse på høyere nivå i treet. En annen mulighet er alltid å lese alle data pånytt (for å sjekke evt. endringer) når dataene fra brukeren skal skrives. Hvis noe er endret, må brukeren vises endringene og ta ny beslutning.

Flyselskapene låser f.eks. seter når de vises som ledige for en kunde, men låsen er tidsbegrenset. Jeg har opplevd at setene er tatt når jeg endelig fikk bestemt meg, men heldigvis har jeg også opplevd at jeg da tilbys *bedre* seter – de var opptatt da jeg sjekket første gang. Dette problemet øker med Internett, da det er håpløst å oppdatere alle nettsider i real time. Her kan det allikevel være håp: Mange Internett-spill oppdaterer nå klienten "samtidig", så teknikken er under utvikling.

Samtidighetskontroll i Oracle

Oracle foretar det meste av samtidighetskontrollen selv, men kan sette en bestemt metode, f.eks.

```
set transaction isolation level serializable;
```

Oracle – og mange andre – tillater også uttrykkelige låser:

```
lock table ansatt_tbl in row exclusive nowait;
```

En slik lås åpnes ved *commit/rollback*. Man kan låse både tabeller og views.

Lock mode kan være

1. **row shared**: Andre kan aksessere og låse andre rader , men ikke låse hele tabellen
2. **row exclusive:** Som (1) men andre kan ikke låse rader shared
3. **share:** Andre kan aksessere men ikke endre data og ikke låse shared
4. **exclusive:** Andre kan kun lese

Wait/*nowait* angir om transaksjonen skal vente ved evt. låsekonflikt (default er *nowait* og da returnerer transaksjonen straks med en feilmelding).

E&N 22 Database recovery

Rett som det er, så skjer det feil, f.eks. system crash og transaksjonsfeil. Da er det viktig at databasen kan "ta seg inn" = *recover* (egentlig "bli frisk igjen").

22.1.1 Noen algoritmer (oversikt)

Vanligvis skal vi oppnå recovery ved å sette databasen tilbake igjen til *seneste, konsistente tilstand*.

1. Ved svært store feil:
    a. Hent seneste backup
    b. Redo så langt som mulig i henhold til loggen – men bare transaksjoner som er kommittert.
2. Ved mindre skader:
    a. Undo i henhold til loggen
    b. Redo så langt som mulig

Det finnes minst flere teknikker for (2) ovenfor. De kan klassifiseres etter når ting lagres på disk:

i. Deferred update = NO_UNDO/REDO:
   Oppdater bare til buffere (e.l.) frem til commit point. Da først skrives fysisk til loggen og deretter fysisk til databasen. Da er det ikke nødvendig med UNDO, bare REDO.
ii. Immediate update = UNDO/REDO:
   Skriv endringer fysisk til disk, men *først* fysisk til loggen. Commit tillates før alle endringer er skrevet fysisk. Da kan det bli nødvendig både med UNDO og REDO.
iii. Variant av immediate update = UNDO/NO-REDO:
   Alle endringer skrives fysisk til disk, men først fysisk til loggen. Commit alltid etter at alle endringer er gjennomført fysisk. Da kan det være nødvendig med UNDO, men ikke med REDO.

22.1.2 Caching

For å unngå for mye venting på hardware (trege greier disse diskene og kommunikasjonsprotokollene ☺!), har DBMS buffere/caches der blokker lagres midlertidig i RAM. Blokker hentes dit og før en aksess til disk sjekkes det om det ønskede item allerede er i bufferet (eller rettere sagt om den blokken som inneholder itemet er der). DBMS har mange buffere og flere av dem kan være til samme fil. Det kan også være flere blokker i ett buffer. Bufferet har en "dirty bit" i bufferkatalogen, som forteller om noe i bufferet er endret. Da sies bufferet å være "dirty" – motsatt "clean".

Hvis bufferet er "dirty", må blokken i bufferet skrives til filen før en ny blokk kan hentes. Dette kalles "flushing" = "trekke ut" eller "tømme". Hvis et buffer er "clean" kan DBMS enkelt skrive over innholdet i det. DBMS kan velge å bruke bufferet med "eldst" innhold, FIFO, eller velge den som har vært brukt minst i det siste, LRU ("least recently used").

Når bufferet tømmes til disk, kan det kanskje skje tilbake der blokken ble hentet i sin tid = in-place updating, eller til et ledig sted = shadowing.

Forrige versjon av blokken kalles "before-image" BFIM og den nye er "after-image" AFIM. Med shadowing finnes altså begge fysisk, men med in-place updating bare "after-image".

22.1.3 Flere begreper (write-ahead, steal, force)

Her omtales tre begreper i forbindelse med tømming av buffere:

1. *Write-ahead logging*:
   Loggen er også en fil med buffer(e). Write-ahead logging innebærer at loggen skal fysisk skrives til disk, før AFIM får overskrive BFIM.
2. *Steal* og *no-steal*:

   a. Steal innebærer at fysisk skriving av en blokk tillates, selv om transaksjonene som har oppdatert den ikke har nådd commit.
   b. No-steal innebærer at fysisk skriving av en endret blokk må vente til alle transaksjoner som har oppdatert den, har nådd commit. Husk at man *kan* risikere at stadig nye transaksjoner oppdaterer samme buffer slik at man må vente lenge før dette kravet oppfylles. Det er også et problem med transaksjoner som gjør særlig mange endringer før de kommitterer. No-steal vil følgelig kreve mer bufferplass enn steal.
3. *Force* og *no-force*:
   a. Force innebærer at *alle* blokker i bufferet som er "dirty", skrives straks til disk.
   b. No-force innebærer ikke noe slikt krav. Da kan oppdaterte blokker være i RAM når andre trenger dem.

Vanligste strategi er steal og no-force, så da blir det stadig liggende blokker i RAM som er endret, men ikke tømt til disk.

22.1.4 Check-points
Fra tid til annen blir det nødvendig og/eller lønnsomt å "rydde opp" i bufrene. Alle "dirty" bufre skrives fysisk til disk og det logges et "check-point". Mens dette foregår, må alle transaksjoner holdes igjen. Etter check-point er det unødvendig å redo alle de writes som skjedde før check-point – de er jo allerede skrevet til disk.

22.1.5 Rollback
Dette er kursorisk, men vær sikker på at du forstår lærebokens figur 22.1 a) og b).

22.1.6 Handlinger som ikke endrer dataene
Hvis en transaksjon produserer rapporter og transaksjonen feiler med rollback, vil vi kanskje ikke at brukeren skal få de gale rapportene. Istedenfor å gi brukeren rapportene og så melde fra om feil, kan man spole rapportene, eller lagre dem som en batch-jobb, og evt. slette dem før de sendes hvis en rollback skjer. Eventuelt kan man slette rapporten bare hvis rollback har betydning for rapporten – men det kan være vanskelig å finne ut. Rapporten kan da sendes først ved suksessfull commit.

22.2 Deferred update
Dette innebærer no-steal og NO-UNDO/REDO.

22.3 Recovery ved immediate updates
Dette kapittelet utgår.

22.4 Shadow paging
Ovenfor er omtalt UNDO/REDO, NO-UNDO/REDO og UNDO/NO-REDO. Det finnes imidlertid også en NO-UNDO/NO-REDO. Da er det altså aldri aktuelt hverken med UNDO eller REDO. Det høres jo fint ut, men selvsagt har det sin pris!

Man ser da på databasen som noe *fysisk*, bestående av *n* blokker. Man lager en "katalog" med *n* plasser, der hver plass peker til en blokk.

Når en transaksjon begynner, blir denne katalogen kopiert i sin helhet til en "skyggekatalog" ("shadow directory"). Denne skyggen lagres på disk og *endres aldri*. Når transaksjonen skriver til disk (write_item(X)) og endrer f.eks. blokk 15, blir det laget en kopi av blokk 15 og endringen gjennomføres på kopien. Katalogen oppdateres til å peke på denne endrede kopien som dermed blir "current". Blokk 15 lagres på *ledig* diskplass (ingen overskriving).

A. Hvis noe går galt, frigjøres de endrede blokkene på disken for overskriving. Katalogen kastes og isteden hentes skyggekatalogen. Alt er nå tilbake til tilstanden før endringen.
B. Hvis alt går bra og transaksjonen gjennomfører commit, så kastes skyggekatalogen ved at dens diskplass frigjøres for overskriving. Endringen er nå definitivt gjennomført.

Ved flerbrukerdatabaser og flettede transaksjoner, må det også legges inn "check-points".

Ulempen er at databasens filer flyttes omkring på disk, så relaterte data ikke ligger fysisk samlet. Filene blir ikke kontinuerlige, hvilket forsinker. Man må også gjennomføre og håndtere en form for garbage collection for blokker (både i datafilen og skyggekatalogen) som fristilles.

22.5 ARIES

Dette kapittelet er kursorisk.
ARIES står for "Algorithm for Recovery and Isolation Semantics" og er en algoritme som gjennomfører steal/no-force.

22.6 Multidatabasesystemer

Noen systemer krever oppdatering av *flere* databaser, og de kan være forskjellige og ha forskjellige recover-teknikker. Da er det viktig at databasene holdes "i takt" og opprettholder konsistens også seg imellom.

Man setter da opp en egen *global recovery manager* – en form for koordinator.

Fase 1: Klar
Når en transaksjon er klar for commit, sender koordinatoren et signal av typen "prepare to commit" til alle involverte databaser. Hver og en force skriver loggen og returnerer signalet "ready to commit". Hvis ikke alt er OK svarer databasen "cannot commit". Dette er også default ved time-out.

Fase 2: Gjennomfør
a. Hvis minst én av databasene har svart "cannot commit" eller fått time-out, må transaksjonen feile og koordinator sender melding til *alle* om undo.
b. Hvis alle svarer "ready to commit", sender koordinator ordre om å gjennomføre commit til alle databasene. Skulle da en/flere få problemer under commit, kan den nå selv recover og transaksjonen har uansett lykkes.

Denne teknikken kalles *to-fase kommittering*. Bland ikke sammen med to-fase låsing (som var den der transaksjonen bare låser mer og mer før den begynner å bare låse opp igjen).

22.7 Katastrofer

Det må tas jevnlig backup av hele databasen. Det må naturligvis gjøres når databasen er konsistent, derfor må ingen transaksjoner holde på samtidig. I praksis settes databasen off-line mens det foregår, etter at køen av transaksjoner er satt på vent og alle kjørende transaksjoner har gjort seg ferdig (med abort eller vellykket commit).

Videre tas det *oftere* backup av loggen.

Ved alvorlige feil:
a. Første hentes seneste backup (restore)
b. Deretter kjøres seneste logg forlengs, men transaksjoner som ikke er kommittert kjøres ikke.
c. Hvis det fortsatt finnes loggfil på disk, som ikke er sikkerhetskopiert, kjøres også den på samme måte. Det kan bli behov for undo for noen (avhengig av steal osv.).

Noe går uansett gjerne tapt – de seneste transaksjonene kommer ikke med, og ingen transaksjoner som er ukommittert kommer med.

DEL 2A PRAKSIS – ORACLE

Vi skal se på hvordan man skaper en database i Oracle. Vi skal da bl.a. bruke mulighetene i Oracle til å lagre objekter med definerte felt og metoder, skape lagrede prosedyrer og funksjoner og triggere. Metodene og de lagrede prosedyrene/funksjonene programmerer vi i Oracles eget programmeringsspråk PL/SQL.

Til øvelsene og obligatoriske oppgaver, vil du trenge følgende programvare:

1. **Oracle:** Last ned og installer *Oracle SQL Developer* fra
   http://www.oracle.com/technetwork/developer-tools/sql-developer/overview/index.html
   Se appendiks A vedrørende installasjonen. Du skal *ikke* installere Oracle databasen på din egen maskin – du skal kjøre mot skolens Oracle databaseserver.

Note: Den oppgitte adressen er korrekt pr august 2012.

Brukere og skjemaer

Oracle utnytter klient/tjener teknologi. Forbindelsen er slik at Oracle ligger som tjener og klienten må ha et program som håndterer brukeren og kommuniserer med tjeneren. Denne kommunikasjonen går via en *Application Server* som validerer brukerens rettigheter, har kontakt med databaseserveren og utfører handlingene på vegne av klienten.

Oracle databasen er delt inn i *skjemaer*. Det er skjemaene som de fleste vil oppfatte som "databasen" og slik sett kan altså Oracle inneholde mange slike "databaser" (eg. *skjemaer*). Alle skjemaer og de objektene som finnes i dem, er "eid" av en bruker. Når en bruker opprettes, opprettes det også et tomt skjema med samme navn som brukeren. For å kunne gjøre noe med skjemaet sitt, må brukeren tilordnes rettigheter med en GRANT-setning. Gjennom GRANT kan brukere også få rettigheter til andres skjemaer og administrator har fulle rettigheter til alle skjemaer.

Som student vil du få rettigheter som utvikler. Det innebærer at du får alle rettigheter til "ditt eget skjema". Skjemaet har samme navn som ditt brukernavn og du er oppført som "owner" av det.

Note: Du kan lese mer om hvordan Oracle er strukturert i appendiks B.

Oracle 11g innebygde datatyper (i databasen)

Her er bare de aller mest aktuelle tatt med. Datatypene for Oracle 11g er beskrevet på http://download.oracle.com/docs/cd/B28359_01/server.111/b28318/datatype.htm#CNCPT012

Datatype	Description
VARCHAR2(size [BYTE | CHAR])	Variable-length character string having maximum length size bytes or characters. Maximum size is 4000 bytes or characters, and minimum is 1 byte or 1 character. You must specify size for VARCHAR2. BYTE indicates that the column will have byte length semantics; CHAR indicates that the column will have character semantics.
CHAR [(size [BYTE|CHAR])]	Fixed-length character data of length size bytes. Maximum size is 2000 bytes or characters. Default and minimum size is 1 byte. BYTE and CHAR have the same semantics as for VARCHAR2.
NUMBER[(precision [, scale]])	Number having precision p and scale s. The precision p can range from 1 to 38. The scale s can range from -84 to 127.
DATE	Valid date range from January 1, 4712 BC to December 31, 9999 AD. The default format is determined explicitly by the NLS_DATE_FORMAT parameter or implicitly by the NLS_TERRITORY parameter. The size is fixed at 7 bytes. This datatype contains the datetime fields YEAR, MONTH, DAY, HOUR, MINUTE, and SECOND. It does not have fractional seconds or a time zone.
CLOB	A character large object containing single-byte or multibyte characters. Both fixed-width and variable-width character sets are supported, both using the database character set. Maximum size is (4 gigabytes - 1) * (database block size).
BLOB	A binary large object. Maximum size is (4 gigabytes - 1) * (database block size).

BOOLEAN finnes ikke som datatype (det gjør det heller ikke i ANSI SQL). Isteden anbefales man å bruke en av disse:

- ✓ flagg_x char(1) default 'y' not null, check (flagg_x in ('y', 'n'))
- ✓ flagg_x number(1,0) default 0 not null, check (flagg_x between 0 and 1)

Disse kan man naturligvis tilpasse som man vil med 'T'/'F', -1/0, tre forskjellige verdier eller annet. Problemet med dette er at man må lage sin egen standard for databasen.

Beskrankninger (constraints)[12]

Vi ønsker å sikre at dataene som lagres, overholder visse regler (ofte kalt ”business rules”). I OOP sikrer man seg dette ved å skjule dataene (”private”) slik at alle endringer av dataene må gjøres gjennom tilgangsmetoder der det kontrolleres at dataene overholder reglene. Også konstruktøren bør benytte tilgangsmetodene når dataene initialiseres. Oracle tilbyr objekter, men det er ikke mulig å skjule dataene. Da har det liten hensikt å skrive tilgangsmetoder, fordi vi ikke kan garantere at de brukes – dataene kan endres ved direkte henvisning selv om det finnes tilgangsmetoder for dem. Av samme grunn holder det ikke å skrive egen konstruktør – en *update* bruker jo ikke konstruktøren. Det vi imidlertid kan gjøre, er å angi datatype – det begrenser jo verdiene litt. Vi kan f.eks. sikre at en tekst ikke er mer enn fem tegn eller at en verdi er et tall. Dette blir sjelden tilstrekkelig.

I Oracle kalles reglene for integritetsregler, og de kan sikres på to måter:
1) beskrankninger (*constraints*) på kolonnene i tabeller
2) triggere

Det er ikke mulig å legge beskrankninger på attributtene til et objekt, de må legges på tabellkolonner. Når vi har tabeller med objekter, må vi vente til tabellene er laget.
Beskrankningene må altså legges inn når vi skaper tabellene. Når vi bruker objekttabeller, må vi følgelig vente med beskrankningene til vi lager tabellene som skal lagre objektene.

Beskrankninger legges helst inn som *constraints*. Regler som er for komplekse til det, må enten lages som en trigger, eller som en lagret prosedyre som kalles fra en trigger. Det siste er mest aktuelt dersom flere triggere skal kalle samme prosedyre (nå eller i fremtiden). Triggeren skrives med PL/SQL

En særlig utfordring gjelder *collections*, fordi det ikke er tillatt å legge beskrankning på en slik samling. Vi skal f.eks. ha et antall fag som en del av studentobjektene, og hvert enkelt fag skal begrenses som FK til fagobjekter. Da må vi sikre at fag lagres i en navngitt tabell, som vi så kan legge beskrankninger på. Likevel kan vi ende med å måtte lage lagrede prosedyrer for å håndheve noen av beskrankningene.

Beskrankninger kan angis på to måter, som Oracle kaller *in-line* og *out-of-line*. *In-line* beskrankninger skrives sammen ved kolonne deklarasjonen. Da behøver man ikke angi hvilken kolonne det gjelder. *Out-of-line* føyes til etter kolonnedeklarasjonene, og da må man angi hvilken kolonne det gjelder. Hvis beskrankningen gjelder flere kolonner under ett, f.eks. en sammensatt primærnøkkel, må man nødvendigvis bruke *out-of-line*.

Beskrankninger kan gis et navn. Navnet vil da bli oppgitt i feilmeldinger. Uansett kan det være fornuftig å gi navn til beskrankninger som innebærer at det bygges en indeks, dvs. beskrankninger som inkluderer unike verdier (*unique* og *primary key*). Da må man bruke ordet *constraint*.

Syntaksen står sammen med oversikten over *create table* og fremgår også av nedenstående eksempler.

Ikke Null

Not Null angir at det ikke skal være tillatt med *null*-verdi i kolonnen. Hvis man motsatt vil eksplisitt angi at *null* er tillatt, kan man skrive bare *Null* – og det er dessuten default. I dette eksemplet kan a ikke være *null*, b og c tillates å være *null*:

[12] En god oversikt på nett: http://docs.oracle.com/cd/B10501_01/server.920/a96524/c22integ.htm#3493

```
create table eksempel_not_null (
  a number not null,
  b number     null,
  c number
);

insert into eksempel_not_null values (   1, null, null);
insert into eksempel_not_null values (   2,    3,    4);
insert into eksempel_not_null values (null,    5,    6);  -- feil
```

Unik

Med *unique* oppretter du en alternativ nøkkel. Vær oppmerksom på at *null* ikke oppfattes som en verdi, så flere *null* er ikke brudd på denne beskrankningen. Hvis du vil unngå det, må du bruke *not null* i tillegg.

```
create table eksempel_unique (
  a number unique,
  b number
);

insert into eksempel_unique values (4,   5);
insert into eksempel_unique values (6,   9);
insert into eksempel_unique values (null,9);
insert into eksempel_unique values (null,9);
```

To kolonner unike i kombinasjon:

```
create table eksempel_unique_2 (
  a number,
  b number,
  c number,
  unique (a,b)
);
```

Navngitt beskrankning:

```
create table eksempel_unique_3 (
  a number,
  b number,
  c number,
  constraint ab_unik unique (a,b)
);
```

Primærnøkkel

Primærnøkkel PK, kan beskrives som en kombinasjon av *unique* og *not null*, men i tillegg er det mulig å henvise til en PK med en fremmednøkkel FK:

```
create table eksempel_pk (
  a number primary key,
  b number
);
```

Fremmednøkkel

Fremmednøkkel sikrer at verdien gjenfinnes blant primærnøklene et annet sted (oftest en annen tabell). Du må oppgi hvilken tabell og kolonne det refereres til:

```
create table eksempel_fk (
  c number,
  d number references eksempel_pk(a);
```

Du kan også bestemme hva som skal skje hvis primærnøkkelverdien blir borte, slik at referanseintegriteten beholdes. Velg mellom *cascade* (sletter posten) og *set null* (setter FK til *null* – da må jo *null* være tillatt). Hvis du ikke bruker *on delete*, vil Oracle nekte sletting hvis det finnes relaterte poster, hvilket tilsvarer *on delete restrict* (men det er det ikke tillatt å skrive).

```
create table eksempel_fk_2 (
  c number,
  d number references eksempel_pk(a) on delete set null;

create table eksempel_fk_3 (
  c number,
  d number references eksempel_pk(a) on delete cascade;
```

Verdikontroll

Med *check* angir du lovlige verdier for kolonner:

```
create table eksempel_check (
  a number check (a between 0 and 100),
  b number
);
```

Et stort antall kontroller er tillatt i uttrykket som kontrolleres, også sammenlikning av flere kolonner:

```
create table eksempel_check_2 (
  liten number,
  stor number,
  check (liten <= stor)
);
```

Øving 1 i Oracle

I den elektroniske lærerressursen finnes "Øving 1 (Bil og eiere)" som faglærer kan gi deg. Du skulle nå ha tilstrekkelig innsikt i Oracle til å løse den. Du bør også studere løsningsforslaget.

Kort om PL/SQL

Note: I dette notatet forutsetter jeg at man kan programmere. Notatet ser derfor spesielt på syntaks.

Hva er PL/SQL

PL/SQL står for "Procedural Language extension to SQL". Det eksisterer ikke som eget produkt, men er en del av Oracle og brukes bare sammen med det. Oracle kan også programmeres med Java, C++ og andre, men da må koden skrives og kompileres utenfor Oracle, og legges ferdig kompilert inn i Oracle som en lagret prosedyre. PL/SQL kan være en mer integrert del av databasen, f.eks. – som vi skal se på – en del av en objekttype. Også PL/SQL kode må kompileres. Det gjøres på anmodning av Oracle databasen.

Den viktigste grunnen til at det passer å bruke PL/SQL, er at den er meget godt integrert i Oracle. F.eks. kan man direkte lage SQL-setninger i koden, og kompileringen optimaliseres for Oracle. Syntaksen er derfor litt spesiell, men man vil kjenne igjen de samme elementene som i andre språk. Språket er tredje generasjons prosedyrielt språk og er ikke objektorientert.

PL/SQL kan i brukes i forbindelse med skjemaer o.l., men her skal vi se på PL/SQL for lagrede prosedyrer, triggere og funksjoner

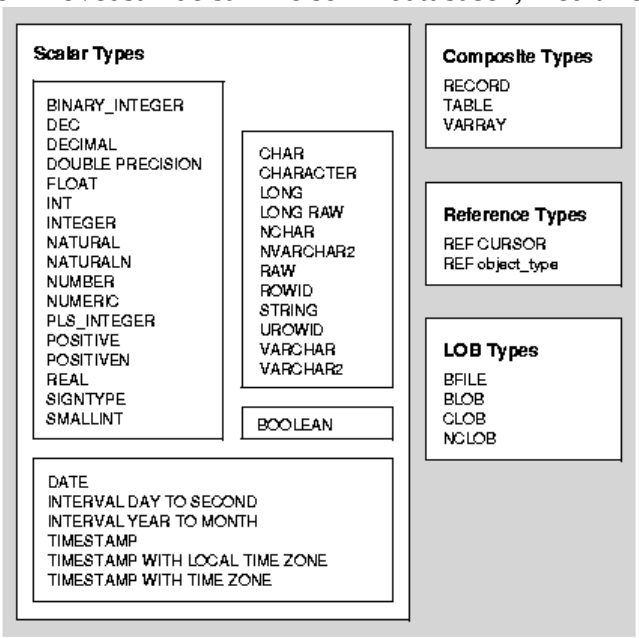
Oversikt over PL/SQL

Store/små bokstaver er uten signifikans utenfor strengkonstanter, men brukes ofte for å klart skille ut reserverte ord. Mer enn ett mellomrom er uten betydning (utenfor strengkonstanter). Linjeskift mellom ord er uten betydning, unntatt for in-line kommentarer som avsluttes ved første linjeskift.

Kommentarer over flere linjer begynner med /* og avsluttes med */. Kommentarer på slutten av en linje begynner med to tankestreker og gjelder da bare til første linjeskift.

Datatyper

Datatypene i PL/SQL er i hovedsak de samme som i databasen, med tillegg av BOOLEAN, altså:



Bruk VARCHAR2 (og ikke STRING). Ellers er INTEGER, NUMBER, CHAR og DATE vanlige.

Variable

En variabeldeklarasjon har følgende syntaks:

```
variable_name datatype [NOT NULL := value ];
```
f.eks.
```
DECLARE
   tmpnr number(6) not null;
   tmpnavn varchar2 (30);
```

Variable tilordnes direkte med kolon og likhetstegn:
```
studnr := 1234;
```

De kan også tilordnes fra en select-setning (mer om dette nedenfor):
```
SELECT column_name
   INTO variable_name
   FROM table_name
   [WHERE condition];
```

f.eks.
```
SELECT studentnavn INTO tmpnavn from tblStudent where studentnr = 4433;
```

Merk at dette krever at select-setningen returnerer bare én rad – vi kan jo ikke legge mange rader inn i én enkel variabel eller post.

Variables synlighet og levetid er innenfor den blokken der de er deklarert. Da blokker kan nøstes, kan variable altså være lokale.

Konstanter

Konstanter deklareres med følgende syntaks:
```
constant_name CONSTANT datatype := VALUE;
```

f.eks.
```
DECLARE
   maks_ant CONSTANT number (3) := 10;
```

Her må naturligvis verdien deklareres samtidig, og den kan ikke endres senere.

Poster

Det er også mulig å deklarere poster, hvilket er svært nyttig når man henter mer enn én kolonne fra databasen (men bare én rad):
```
TYPE record_type_name IS RECORD
   (first_col_name column_datatype,
   second_col_name column_datatype,
...);
```
f.eks.
```
TYPE stud_post IS RECORD
   (stud_nr number(6),
   stud_navn varchar2(30)
   );
```

Her er det mulig å utnytte databasen ved deklarasjonen, f.eks.
```
TYPE stud_post IS RECORD
   (stud_nr tblstudent.studentnr%type,
   stud_navn tblstudent.studentnavn%type
   );
```

Hvis tabellen *tblstudent* bare har disse to kolonnene, kan man skrive

```
stud_post tblstudent%rowtype;
```

Da får posten felter med samme navn som i databasen (*studentnr* og *studentnavn*).

Man får tilgang til feltene i posten med punktnotasjon:
```
stud_post.stud_nr := 9999;
```
og kan tilordne hele posten på en gang:
```
SELECT * INTO stud_post FROM tblstudent WHERE studentnr = 9999;
```

Arrays

Arrays deklareres som VARRAY og virker slik du er vant til. Deklarasjonen er uvant:
```
TYPE typenavn IS {VARRAY | VARYING ARRAY} (maks antall)
  OF elementtype [NOT NULL];
```

Her deklareres en varray for en kalender med 366 datoer:
```
DECLARE
  TYPE Calendar IS VARRAY(366) OF DATE;
```

Arrays brukes med indeks slik du er vant til. Det finnes en spesiell type arrays med flere verdier i hvert element, der ett av elementene er en indeks, men det tar jeg ikke her.

Tabeller og nøstede tabeller

Det er mulig å deklarere en variabel som tabell. Det finnes to typer av tabeller, nemlig *nested table* som er en tabell med records (kun én kolonne med poster i) og *associative arrays* som er indeksert. Her er en oversikt over egenskapene sammenliknet med varray[13]:

Has Ability To	Varray	Nested Table	Associative Array
be indexed by non-integer	No	No	Yes
preserve element order	Yes	No	No
be stored in database	Yes	Yes	No
have elements selected indidually in database	Yes	Yes	--
have elements updated indidually in database	Yes	No	--

Oracle sier det slik:

- Arrays in other languages become VARRAYs in PL/SQL.
- Sets and bags in other languages become nested tables in PL/SQL.
- Hash tables and other kinds of unordered lookup tables in other languages become associative arrays in PL/SQL.

Syntaksen slår du opp hvis du trenger det. Du finner det ofte under "PL/SQL collections".

Operatorer

Aritmetiske operatorer er som du er vant til (*, / osv.) men potenser skrives med **. Det finnes faktisk ingen heltallsdivisjon – alle tall konverteres til reelle tall før divisjonen og avrundes igjen etterpå – bruk evt. divisjon og kutt desimalene med funksjonen *trunk*.

Relasjonsoperatorene for sammenlikninger er som du er vant til (>, <, >= osv.), men det er vanlig å bruke != for ulik. Videre finnes LIKE, IS NOT og BETWEEN.

[13] http://www.developer.com/db/article.php/10920_3379271_4/Oracle-Programming-with-PLSQL-Collections.htm#TABLE2

Logiske operatorer er som du er vant til (AND, OR og NOT).

Konkatinering (slå sammen to strenger) gjøres med to loddrette streker: ||.

Det finnes *mengdeoperatorer* (UNION osv., men det får du neppe bruk for her) og IN.

Det er mulig å lage sine *egne operatorer* med CREATE OPERATOR.

Innebygde funksjoner

Det finnes et stort antall innebygde funksjoner for tallmanipulering, strengbehandling, behandling av tegn, datoer, konvertering av datatype osv.

Utskrift

Programmene eksekveres på tjeneren. I prinsippet vil altså utskrifter skje på tjeneren. Vi kan allikevel få se dem i SQL Developer ved å bruke DBMS_OUTPUT. Den har to metoder som du trenger:

- PUT(tekst)
- PUT_LINE(tekst)

Den første skriver strengen til bufferet uten linjeskift, den andre legger linjeskift til bakerst. Du kan også selv legge til linjeskift med NEW_LINE.

Bufferet vises først når koden er ferdig eksekvert.

PUT og PUT_LINE skal greit håndtere VARCHAR2, NUMBER og DATE. Alle andre datatyper må konverteres med funksjonen TO_CHAR().

Strukturer

Null-setning

Syntaksen krever ofte at det skal utføres minst én setning i strukturen. Mens du stubbe-programmerer, kan du da sette inn en setning som ikke gjør noe, men som gjør syntaksen lovlig og kan kompileres:

```
BEGIN
  null;
EXCEPTION
  null;
END;
```

Blokk

En PL/SQL blokk utgjør en samlet programdel (et element) og har en deklarasjonsdel, en utførelsesdel og en unntaksdel, med følgende struktur:

```
DECLARE -- valgfri
  variabeldeklarasjoner
BEGIN -- obligatorisk
  handlinger
EXCEPTION -- valgfri
  unntaksbehandling
END; -- obligatorisk
```

Algoritmeteoremet fastslår at enhver algoritme kan beskrives ved hjelp av språklige elementer (ord) og strukturelementene sekvens (én og én i gitt rekkefølge), seleksjon (valg) og iterasjon (løkker). Oracle har i tillegg det (ustrukturerte) kontrollelementet GOTO, men det bør du *ikke* bruke.

Sekvens

I Oracle eksekveres koden som vanlig setning for setning, ovenfra og nedover. Setningene avsluttes med semikolon. Hvis flere setninger skal grupperes, brukes BEGIN og END.

Seleksjon

Syntaksen er omfattende, men du kjenner den igjen fra andre språk:

```
IF utsagn er sant
THEN
  setning1;
  setning2;
ELSIF utsagn2 er sant THEN
  setning3;
ELSE
  setning4;
END IF
```

f.eks.

```
IF stud_nr < 1000 THEN
  stud_program := 'Årsstudiet';
ELSIF stud_nr > 2000 THEN
  stud_program := 'Bachelor';
ELSE
  stud_program := 'Master';
END IF;
```

Det finnes også en multiseleksjon med syntaksen

```
case variabel
  when verdi1 then setning1;
  when verdi2 then setning2;
  -- osv
else setning4; -- defaulthandling
end case;
```

Variabelen kan være av praktisk talt enhver datatype.

Det finnes også en variant av denne ("searched case"), som jeg ikke viser her.

Iterasjon

A. While loop

Teoretisk kan enhver iterasjon uttrykkes slik:

```
1. sett verdien for en variabel x, y..
2. sålenge x tilfredsstiller en betingelse
  3. gjør noe, inkludert å endre verdien av x, y..
```

PL/SQL har en *while-løkke* med slik syntaks:

```
sett variabel som inngår i betingelsen;
WHILE betingelse LOOP
  setning(er); -- inkludert endring av variabler som inngår i betingelsen
END LOOP;
```

While-løkker eksekverer som vanlig ingen, én eller flere ganger.

B. Exit loop

Hvis du bare vil skrive setningen som setter variabelen for betingelsen én gang, er det enklere med en *uthopp-løkke* (*exit loop*):

```
LOOP
  setning(er); -- inkludert endring av variabler som inngår i betingelsen
EXIT WHEN betingelse;
  setning(er);
END LOOP
```

Denne er svært vanlig ved henting av flere poster vha en *cursor* (se nedenfor), hvis vi ikke skal ha alle. F.eks.:

```
LOOP
   FETCH nr_cursor INTO tmp_nr;
   EXIT WHEN nr_cursor%NOTFOUND;  -- hopp ut når det er slutt på poster
   ...
END LOOP;
```

Den vil eksekvere det som står foran *EXIT* minst én gang.

C. For loop
Når antallet iterasjoner er kjent før løkken begynner, bruker du naturligvis heller en *for-løkke*. Her er syntaksen litt uvant:

```
FOR løkkevariabel IN [REVERSE] fra..til LOOP
   setning(er); -- her må ikke tellervariabelen endres!
END LOOP;
```

F.eks.

```
FOR i IN 1..3 LOOP
   setning(er); -- utføres tre ganger
END LOOP;
```

Mange språk kan angi STEP, men det er ikke tillatt i PL/SQL. Da tyr du til et lite trikk med å gange løkkevariabelen med STEP-faktoren, men husk at løkkevariabelen selv *ikke* kan endres inne i løkken:

```
FOR i IN 1..3 LOOP
   j := i * 2 -- j går nå i trinn på 2 og blir 2, 4 og 6
   setning(er); -- her kan j brukes istedenfor i
END LOOP;
```

OBS! Løkkevariabelen deklareres implisitt av FOR som INTEGER og vil eventuell overskygge andre, deklarerte variabler med samme navn. Løkkevariabelen er altså alltid lokal for for-løkken og er alltid en INTEGER.

Hente data fra databasen

En stor fordel med PL/SQL er at den sømløst kommuniserer med Oracle databasen. De mest aktuelle for PL/SQL er DML og TCL:

1. ***DML*** (Data Manipulation Language) som endrer data i tabellene, f.eks. SELECT, UPDATE, INSERT, DELETE, LOCK TABLE, CALL (kaller en lagret prosedyre). PL/SQL støtter alle disse unntatt *CALL*.
2. ***TCL*** (Transaction Control Language) som styrer transaksjoner, f.eks. SAVEPOINT, COMMIT, ROLLBACK og SET TRANSACTION. Alle disse er lovlige i PL/SQL (med bare få unntak).

Syntaksen for disse i PL/SQL er som i SQL.

Videre kan man i Oracle SQL bruke DDL og DCL, men det bør ikke være nødvendig å gjøre det i et PL/SQL-program – det tyder på at databasen har mangler. Slike mangler bør du heller endre direkte i databasen.

1. ***DDL*** (Data Definition Language) som skaper, sletter og endrer metadata, f.eks. CREATE, DROP, ALTER, RENAME. PL/SQL støtter de fleste av disse.
2. ***DCL*** (Data Control Language) som kontrollerer tilgang, f.eks. GRANT, DENY og REVOKE. Disse kan også brukes i PL/SQL.

Til slutt vil jeg nevne at Oracle også har kommandoer som vanligvis brukes av databaseadministrator. De kan *ikke* brukes i et PL/SQL-program (hvilket jeg finner naturlig):

1. ***Session Control*** som styrer sesjonen (hver pålogging til databasen skaper en sesjon)

2. ***System Control*** som styrer egenskapene til Oracle databasen.

Alle funksjoner i Oracle (som har flere funksjoner enn standard SQL), f.eks. AVG og SUM), kan brukes, og alle operatorer (f.eks. IS NULL, EXISTS).

Videre er det også en stor fordel at datatypene i PL/SQL er lik datatypene i Oracle databasen, både mht navn og intern representasjon – derfor er det ikke nødvendig med konverteringer. Det inkluderer datatypen *NULL*.

Cursor
Hvis du bare skal hente en skalar (én, enkelt verdi) eller bare én rad, kan du bruke en *implicit cursor*. Hvis du skal hente flere, må du bruke en *explicit (declared) cursor*.

A. Implicit cursor
Når en SQL-setning skal eksekveres, skapes en cursor automatisk. Cursoren peker til en "current" post. Dette kan bare brukes når databasen vil returnere (høyst) én rad. Her er *lonn_total* deklarert som en passende variabel:

```
SELECT SUM (lonn) INTO lonn_total
  FROM tblansatt WHERE avd_nr = 10;
```

Setningen vil føre til at databasen kalles og returverdien (en skalar) vil plasseres i variabelen *lonn_total*.

Vi kan hente flere verdier til hver sin variabel, f.eks. slik:

```
SELECT SUM (lonn), MAX(lonn) INTO lonn_total, lonn_max
  FROM tblansatt WHERE avdnr = 10;
```

Merk at variablene må stå i samme rekkefølge som i select-setningen og ha riktig type.

Vær oppmerksom på at hvis det viser seg at spørringen returnerer mer enn én rad, vil det oppstå en feil. Videre kan det oppstå feil andre steder hvis spørringen returnerer *NULL*. Slike feil må du teste for eller håndtere med EXCEPTION.

B. Explicit cursor
En explicit cursor deklareres i DECLARE-delen av blokken. Deretter må den åpnes (og senere lukkes) og kan så brukes med FETCH. Vanligvis vil jo FETCH foregå i en løkke, fordi det skal hentes flere poster.

Her er et eksempel (ikke helt realistisk for å vise muligheter):

```
DECLARE
   CURSOR cur_ansatt IS SELECT navn, lonn FROM tblansatt
     WHERE avdnr = 10;
   tmp_navn VARCHAR2(30); -- som navn i databasen
   tmp_lonn NUMBER(10,0); -- som lonn i databasen
BEGIN
    OPEN cur_ansatt;
LOOP
   FETCH cur_ansatt INTO tmp_navn, tmp_lonn;
   EXIT WHEN cur_ansatt%notfound;
   DBMS_OUTPUT.PUT_LINE('Ansatt: ' || tmp_navn || ' '
     || TO_CHAR(tmp_lonn));
END LOOP
   CLOSE cur_ansatt;
EXCEPTION
   IF cur_ansatt&ISOPEN THEN CLOSE cur_ansatt;
   DBMS_OUTPUT.PUT_LINE ('Det oppsto en feil');
END;
```

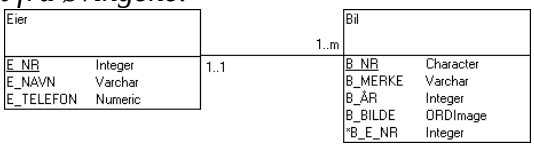
Siden vi i det ovenstående eksemplet skal gå igjennom *alle* postene som hentes med spørresetningen, er det enklere med en FOR-IN løkke. Da deklareres implisitt en post i FOR-setningen (jeg kaller den *tmp_ansatt*), så de to variablene deklareres ikke lenger. Legg merke til hvor mye enklere løkken blir:

```
DECLARE
   CURSOR cur_ansatt IS SELECT navn, lonn FROM tblansatt
     WHERE avdnr = 10;
BEGIN
   FOR tmp_ansatt IN cur_ansatt LOOP
     DBMS_OUTPUT.PUT_LINE('Ansatt: ' || tmp_ansatt.navn || ' '
     || TO_CHAR(tmp_ansatt.lonn));
   END LOOP
EXCEPTION
   DBMS_OUTPUT.PUT_LINE ('Det oppsto en feil');
END;
```

For-løkken åpner og lukker selv cursoren – det gir feil å skrive *OPEN* eller *CLOSE*. I praksis er den enklere å bruke enn en FETCH-løkke, og derfor brukes FETCH-løkker bare hvis det kanskje skal hoppes ut av løkken før alle radene er behandlet[14].

Skript

Note: *Jeg bruker her eksemplet fra øvingene:*



Et skript kan defineres som et program som kjøres interpretert. Fordelen er at det kan lages ad hoc av en bruker og kjøres uten å lagres eller kompileres. I Oracle kan vi lage skript og kjøre dem i SQL Developer eller et annet klientprogram. De kan lagres som en fil. Det er jo ikke mange brukere som er i stand til å lage slike skripts, men man kan lage et bibliotek av dem så de kan hentes og kjøres. I Oracle vil vi allikevel – av mange grunner – foretrekke å kompilere koden og lagre dem som lagrede prosedyrer og funksjoner. Da blir de tilgjengelige for alle og kan brukes i andre lagrede prosedyrer, i medlemsfunksjoner for objekter og i SQL.

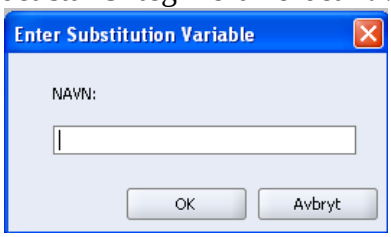
Her er et enkelt eksempel, der vi henter data fra brukeren og legger dem inn i Eiertabellen:

[14] Faktisk er det fullt lovlig å hoppe ut av en FOR-løkke også (med EXIT WHEN). Noen vil nok mene at det ikke er særlig "pent", men det er jo *praktisk*☺!

```
DECLARE
  nvn varchar(20);
  tlf number;
BEGIN
  nvn := '&navn';
  tlf := &telefon; /* &-tegnet virker bare i script */
  savepoint før_insert;
  insert into eier(E_NAVN, E_TELEFON) values (nvn, tlf);
  commit;
EXCEPTION
  when others then
  rollback to før_insert;
END;
/
```

Forklaring:

1) Vi starter med å deklarere to lokale variabel, *nvn* og *tlf*. Variable kan hete hva som helst, bare ikke reserverte Oracle-ord.
2) Variabelen *nvn* tilordnes (merk :=) verdien *&navn* med anførselstegn (”enkeltfnutter”) rundt. Når det står &-tegn foran ordet *navn*, betyr det at Oracle vil be brukeren oppgi navn. Det ser slik ut:



   Anførslene rundt *&navn* er ikke strengt nødvendige, men hvis ikke vi setter dem her, så må brukeren selv huske å ta dem med – ellers blir tilordningen feil.
3) Deretter gjør vi det samme med variabelen *tlf* men her slipper vi å tenke på anførselstegn.
4) Vi lager et lagringspunkt som vi kan ta *Rollback* til. Punktet må ha et navn (det kan være fristende med *Start*, men det er et reservert ord).
5) Vi setter verdiene inn i tabellen *eier*.
6) Vi gjennomfører endringen med *Commit*.
7) Hvis det skjer en eller annen feil (*Others* – kunne vært spesifisert med typer av feil), så tar vi *Rollback* til lagringspunktet vi satte opp. Da blir det ingen endringer i databasen.
8) Avslutter med ”/” som tegn på at prosedyren er ferdig og skal kjøres.

Det kan se litt kult ut at vi henter data fra brukeren, men husk at vanligvis kjøres PL/SQL på serveren, og der sitter det jo ingen brukere… Derfor kjører vi heller kompilerte prosedyrer og funksjoner som er lagret i databasen – se neste avsnitt.

Lagrede prosedyrer og funksjoner

Prosedyrer og funksjoner som skal lagres i databasen i kompilert form må jo ha et navn slik at de kan kalles. Videre er det da uaktuelt å bruke inputbokser for å hente dataene fra bruker, siden en lagret prosedyre kjører på serveren. Vi må følgelig bruke parametre.

La oss lagre en *prosedyre* under navnet *LEGG_INN_EIER*:

```
create or replace
PROCEDURE LEGG_INN_EIER
      (nvn in varchar2, tlf in number)
IS
/* Her kunne vi deklarert lokale variable */
BEGIN
  savepoint før_insert;
  insert into eier(E_NAVN, E_TELEFON) values (nvn, tlf);
  commit;
EXCEPTION
  when others then
  rollback to før_insert;
END;
/
```

Signaturen for denne prosedyren blir altså

```
LEGG_INN_EIER(varchar(20), number)
```

Ordet IN angir at dette parametret bare skal være input (tilsvarer *ByVal*). Legg også merke til at ordet DECLARE nå erstattes av ordet IS.

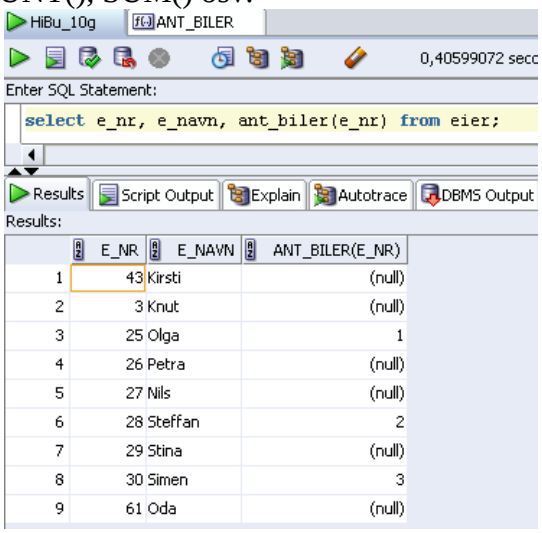
Den kompilerte versjonen av denne prosedyren lagres i databasen, og kan kalles fra en passende PL/SQL-blokk i skript eller med en passende trigger. Fra skript kan det se slik ut:

```
begin
  legg_inn_eier('Oda',12);
end;
/
```

La oss så lage en *funksjon* som sjekker hvor mange biler en gitt eier er registrert med. Vi kan lage en funksjon som returnerer antall biler til en gitt eier (angitt med E_NR) slik:

```
create or replace function ant_biler (nr in number)
  return number
is
  ant number;
begin
  select count(*) into ant from bil where b_e_nr = nr group by b_e_nr;
  return ant;
end ant_biler; -- etiketten er frivillig
```

Legg merke til hvordan det likner en prosedyre, men her kommer i tillegg en returverdi. Vi kan prøvekjøre denne funksjonen med en vanlig select. Nå bruker vi funksjonen helt på linje med innebygde funksjoner som COUNT(), SUM() osv:



	E_NR	E_NAVN	ANT_BILER(E_NR)
1	43	Kirsti	(null)
2	3	Knut	(null)
3	25	Olga	1
4	26	Petra	(null)
5	27	Nils	(null)
6	28	Steffan	2
7	29	Stina	(null)
8	30	Simen	3
9	61	Oda	(null)

Vi ser at det er seks eiere her som ikke har noen biler. Merk at resultatet blir *null* og ikke 0!

Jeg kjører nå

```
delete from eier where ant_biler(e_nr) is null;
```

og da slettes seks rader så det bare er tre eiere igjen:

```
E_NR                 E_NAVN               ANT_BILER(E_NR)
-------------------- -------------------- ----------------------
25                   Olga                 1
28                   Steffan              2
30                   Simen                3
```

Trigger

En trigger er en hendelse som databasen skal reagere på, og skrives slik:

```
create or replace trigger slettet_bil
  after delete on BIL
begin
  delete from eier where ant_biler(e_nr) is null;
end slettet_bil;
```

Tanken er at triggeren skal slå til etter at en bil er slettet. (Hvorfor ikke *før* bilen slettes? Holder det å sjekke bare når en bil *slettes*? Vi burde vel laget en trigger til for *update?*)

Når jeg nå kjører skriptet

```
delete from bil where b_nr = 'DA123456';
```

vil triggeren slå til og eieren Olga – som hadde bare denne bilen – slettes fra eier-tabellen.

Objekter med funksjoner

Skal det ha noen hensikt å deklarere objekter i Oracle databasen, må det være for å kunne bruke objektenes funksjoner. Ellers vil vanlige rader lagre det samme. Objektene og deres funksjoner må deklareres og defineres hver for seg. Det er også tillatt å lage *prosedyrer* til objekter, men i praksis har det lite for seg, for vi får ikke kalt dem – de er ulovlige i select-setninger fordi de ikke returnerer noe. Allikevel kan de ha noe for seg som lokale subprogrammer for objektet selv, f.eks. kan de kalles av en funksjon i samme objekt. Vi lager stort sett bare *funksjoner* til objektene.

Slik kan deklarasjonen se ut:

```
CREATE OR REPLACE TYPE bil_type AS OBJECT
   (bilnr INTEGER PRIMARY KEY,
   biltype VARCHAR2(30) NOT NULL,
   MEMBER FUNCTION tostring return VARCHAR2 (40)
   );
```

Legg merke til at funksjonen *tostring* ikke defineres her, den bare deklareres. Deretter må vi definere den i TYPE BODY:

```
CREATE OR REPLACE TYPE BODY bil_type AS
  MEMBER FUNCTION tostring RETURN VARCHAR2 (40)
  IS
  BEGIN
    RETURN TO_CHAR(bilnr) || ' ' || biltype; -- TO_CHAR er unødvendig
  EXCEPTION WHEN OTHERS THEN
    RETURN 'Feil i funksjonen tostring';
  END to_string;
END;
```

Konverteringen med TO_CHAR er unødvendig med de vanligste datatypene – PL/SQL konverterer selv til streng.

Vi kan nå bruke funksjonen i spørresetninger:

```
SELECT B.tostring() from tblbil B; -- funksjonsbruk krever tabell-alias
```

PL/SQL og nettet

Hvis Oracle skal benyttes til å produsere dynamiske nettsider basert på data fra databasen, må nettsiden bygges opp i sin helhet av programmet.

Note: Spesielt interesserte kan se hvordan det gjøres i appendiks C.

Collections (ADT, egendefinerte typer)

Oracle har støtte for egendefinerte typer, dvs. datatyper som du deklarerer selv. Du kan skape *Array*, *Object* og *Table* typer. Alle disse er *collections* (samlinger), dvs. at de inneholder flere data, og objektene har i tillegg *member functions* (metoder).

I SQL Developer kan du høyreklikke på *Type* og be om *NewType*, men det har lite for seg, da det bare gir et "skjelett", slik:

```
create or replace
type type1 as object
( /* TODO enter attribute and method declarations here */
);
```

Når du først har skapt typen, er det enklest å klikke på typenavnet i venstre kolonne, så SQL-teksten åpnes i eget vindu. Da kan du endre, kompilere osv. og du får bedre feilmeldinger.

A – Object type

Object type tilsvarer en klasse, og har minst én *column* (tilsvarer attributt) og kan ha *member functions* (tilsvarer metoder). *Object type* må først **deklareres**, dvs. vi forteller hvilke kolonner den skal ha med datatype, og hvilke medlemsfunksjoner/medlemsprosedyrer med innparametre og evt. returtype. Deretter må medlemsfunksjonen/prosedyren **defineres** med PL/SQL.

Deklarasjon av objekttype

Enkleste variant av syntaksen[15] for **deklarasjon** av en *object type* er slik:

```
CREATE OR REPLACE TYPE <objekttypenavn> AS OBJECT
  (
  <kolonnenavn1> <datatype1>,<kolonnenavn1> <datatype1>,osv...
  MEMBER FUNCTION (<innparameter1> <datatype1>,
    <innparameter2> <datatype2>,osv…)
    <metodenavn> RETURN <returdatatype>
  );
```

For eksempel:

```
create or replace
TYPE rektangel_type AS OBJECT
  (
  nr number,
  bredde number,
  lengde number,
  MEMBER FUNCTION kvm RETURN number
  );
```

Hvis objekttypen skal ha flere medlemsfunksjoner/prosedyrer er det bare å legge inn flere, adskilt med komma på samme måte som flere kolonner. Prosedyrer deklareres med MEMBER PROCEDURE og de har selvsagt ingen returverdi.

[15] Full syntaks i dokumentasjonen
http://docs.oracle.com/cd/B19306_01/server.102/b14200/statements_8001.htm#SQLRF01506

Definisjon av metodene

Deretter må du skrive **definisjonen** av medlemsfunksjonene/prosedyrene:

```
CREATE OR REPLACE TYPE BODY <objekttypenavn> AS
  MEMBER FUNCTION <metodenavn> RETURN <returdatatype>
  IS
    <deklarasjoner>
  BEGIN
    <kode>
    RETURN <returverdi>;
  EXCEPTION WHEN OTHERS THEN
    <kode>
    RETURN <returverdi>;
  END <metodenavn>;
END;
```

Her kan du også legge inn flere medlemsfunksjoner/prosedyrer etter hverandre og du har tilgang til objektets attributter (de er brukt i eksemplet).

Eksempel:

```
create or replace
TYPE BODY rektangel_type AS
  MEMBER FUNCTION kvm RETURN number
  IS
  BEGIN
    RETURN bredde * lengde;
  EXCEPTION WHEN OTHERS THEN
    RETURN -1;
  END kvm;
END;
```

Objekttabell (lite fleksibelt, men enkelt)

I en objekttabell lagres data av objekttypen som en hel rad. Da kan tabellen ikke ha andre kolonner – det skapes kolonner som tilsvarer attributtene i objekttypen. Dette betyr at tabellen er som en vanlig relasjonstabell, men samtidig håndterer den radene som objekter. Syntaksen er slik:

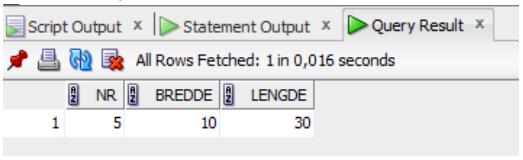
```
create table rektangel_tabell of rektangel_type;
```

Det dannes nå en tabell med tre kolonner (nr, bredde og lengde) og en insert er rett frem:

```
insert into rektangel_tabell values (5,10,30);
```

En vanlig spørresetning viser nå

```
select * from rektangel_tabell;
```



Allikevel oppfattes hver rad som et objekt, så man kan bruke punktnotasjon og objektets funksjoner

```
select R.kvm() from rektangel_tabell R;
```

Tabell med objekter i en kolonne (mer fleksibelt, men tyngre)

Man kan også lage en tabell der en kolonne inneholder objektene. Den lages på helt vanlig måte, idet datatypen oppgis som den skapte/definerte objekttypen. Dette skaper da en tabell der én av kolonnene har data av objekttypen. Tabellen kan ha andre kolonner også. Det kan være kjekt at tabellen også kan ha andre kolonner, men det gjør syntaksen for DML tyngre.

Eksempel:

```
create table rektangel_tabell (kode char(1), rektangel rektangel_type);
```

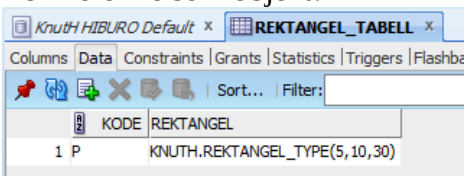
Legge objekter inn i tabellen

Et objekt tar bare én kolonne i tabellen. Ettersom objektet har flere attributter, må dataene eksplisitt gjøres om til et objekt av riktig type ved innlegging. Det gjør vi med typens *constructor* som har samme navn som typen selv (du kan lage din egen konstruktør som en funksjon i typedeklarasjonen). Jeg får ikke til å legge inn verdier direkte med Data-fanen i SQL Developer, så her må det brukes *insert*-setninger, f.eks.:

```
insert into rektangel_tabell values ('P',rektangel_type(5,10,30));
```

Hente objekter fra tabellen

Hvis vi ser på tabellens data, ser vi én kolonne som objekt:



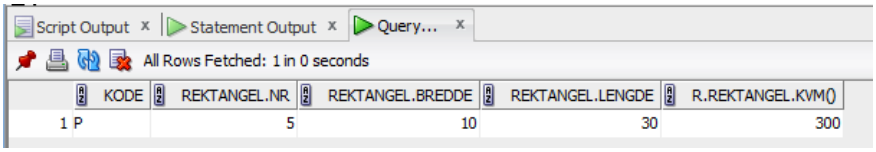
Tilsvarende hvis vi bruker en "rett-fram" *select*-setning:

```
select * from rektangel_tabell;
```

Hvis vi vil hente attributtene i objektene, må vi for det første bruke en alias for tabellnavnet, og for det andre bruke punktnotasjon ("sin"). Vi kan bruke funksjoner, hvis vi tar med parentesene. F.eks.

```
select R.kode, R.rektangel.nr, R.rektangel.bredde, R.rektangel.lengde,
R.rektangel.kvm() from rektangel_tabell R;
```

som gir dette resultatet:



Det kan være fornuftig å lage en funksjon *tostring* for alle objekter, så man enkelt kan skrive

```
select R.rektangel.tostring()"Rektangeldata" from rektangel_tabell R;
```

B – Arv

Bare objekttyper (og ikke tabeller) kan arve egenskapene til et annet objekt. Subtypen kan legge til felt, legge til metoder og endre metoder. For at en objekttype skal kunne brukes som supertype, må den være deklarert *not final*.

Statiske, abstrakte og konstante elementer

- ✓ Metoder kan legges til med andre parametre (*overload*) og overstyres med samme parametre (*override*). Overloading er helt vanlig og krever ingen ekstra tiltak, men de må ha en annen *signatur* (navn, returtype og rekkefølge av parametertypene). Overriding krever bruk av ordet *OVERRIDES*.
- ✓ Vi kan lage abstrakte objekttyper med uttrykket *not instantiable*.
- ✓ Vi kan deklarere metoder (dvs. prosedyrer og funksjoner) som *final* og da kan de ikke overstyres i subtyper.
- ✓ En objekttype som er deklarert *final* kan ikke brukes til arv – den utgjør en bladnode i arvehierarkiet.
- ✓ Metoder som merkes *static* er objekttypemetoder og kalles ved å oppgi klassenavnet foran metodenavnet. Det er ikke tillatt med *static* attributter.
- ✓ Lag aldri en type som er både *final* og *not instantiable* – den er helt ubrukelig.

Her har jeg f.eks. brukt det jeg kunne komme på.

1) Lager en abstrakt objekttype *BOK_TYPE* med en abstrakt metode *to_string*:

```
create or replace type bok_type as object
  (ISDN number,
  tittel varchar2(20),
  not instantiable member function to_string
  return varchar2 -- abstrakt metode
  )
not instantiable -- abstrakt objekttype
not final; -- arv er tillatt
```

2) Lager en objekttype FAGBOK_TYPE som arver fra BOK_TYPE og overstyrer funksjonen *to_string*:

```
create or replace type fagbok_type under bok_type
  (fag varchar2(10), -- nytt attributt
  overriding member function to_string return varchar2 --overstyring
  )
instantiable -- konkret
final; -- kan ikke arves
```

Her må også den arvede funksjonen *to_string* defineres:

```
create or replace type body fagbok_type
is
overriding member function to_string
  return varchar2 -- overriding den abstrakte funksjonen i bok
  is
  begin
    return isdn||' '||tittel ||' '||fag;
  end;
end;
```

3) Lager en objekttype *ORDBOK_TYPE* som arver fra *BOK* og overstyrer funksjonen *to_string*:

```
create or replace type ordbok_type under bok_type
  (spraak varchar2(10),
  static procedure gjør_noe, -- objekttypemetode (tilsv. klassemetode)
  overriding member function to_string
  return varchar2 -- overriding
  )
instantiable -- konkret
final; -- kan ikke arves
```

Her må også den arvede funksjonen *to_string* defineres, samt prosedyren *gjør_noe*:

```
create or replace type body ordbok_type
is
  overriding member function to_string
  return varchar2
  is
    begin
    return isdn||' '||tittel||' '||spraak;
    end;
  static procedure gjør_noe
  is
    begin
      null; -- kom ikke på noe fornuftig å gjøre!
    end;
end;
```

Tabellen skapes som nevnt ovenfor:

```
create table bok_tabell (bok bok_type);
```

For å legge inn data, gjør vi også som før:

```
insert into bok_tabell values (fagbok_type(1234,'databaser','db2'));
insert into bok_tabell values (ordbok_type(4433,'cappelen blå','engelsk'));
```

Når vi skal hente ut data med *select* må vi tenke oss litt om. Vi kan ikke uten videre velge alle kolonnene, for noen av dem finnes ikke i subtypene. Her kommer *to_string* til sin rett, for den finnes i alle tre typene:

```
select B.bok.to_string() from bok_tabell B;
```

Denne gir:

```
B.BOK.TO_STRING()
---------------------------------
1234 databaser db2
4433 cappelen blå engelsk
2 rows selected
```

Hvis vi vil ha tak i en, bestemt kolonneverdi som bare finnes i én av subtypene, må vi gjøre om alle objektene til den ene subtypen. Dermed vet kompilatoren at kolonnen finnes.

```
select B.bok.ISDN, treat (B.bok as fagbok_type).fag,
treat (B.bok as ordbok_type).spraak from bok_tabell B;
```

Dette gir følgende:

```
BOK.ISDN   TREAT(B.BOKASFAGBOK_TYPE).FAG  TREAT(B.BOKASORDBOK_TYPE).SPRAAK
---------- ------------------------------ --------------------------------
1234       db2
4433                                      engelsk
2 rows selected
```

Jeg er usikker på om denne teknikken kan brukes i PL/SQL – versjon 10g ga exception.

C – Varray type

Note: Varrays har den fordelen at vi kan unngå joins, og de kan holde orden på en rekkefølge, f.eks. være sortert. De er imidlertid begrenset i størrelse, og man må vite det maskimale antallet (men det kan unngås ved at varrayen utvides med et trigger ved behov).

Varray er en samling (*collection*) av verdier. Oracle kaller arrays for VARRAY (variable array) fordi antall elementer i arrayen kan variere. Vi må oppgi maksimalt antall elementer.

Vi vil ha en array med plass til to tekster:

```
create or replace type farge_array as varray(2) of varchar2(10);
```

Note: Normalt ville jeg nok heller lagt farger som attributt i objektet – dette er bare for å få brukt varray.

Denne arrayen kan vi så bruke i en tabell. F.eks. kan vi legge til fargene til i rektageltabellen, slik at hvert rektangel kan ha inntil 2 farger:

```
alter table rektangel_tabell add (farge farge_array);
```

Vi har nå fått følgende struktur på tabellen:

Columns | Data | Constraints | Grants | Statistics | Triggers | Flashback | Dependencies | Details | Indexes | SQL

Actions...

Column Name	Data Type	Nullable	Data Default	COLUMN ID	Primary Key	COMMENTS
REKTANGEL	REKTANGEL_TYPE	Yes	(null)	1	(null)	(null)
FARGE	FARGE_ARRAY	Yes	(null)	2	(null)	(null)

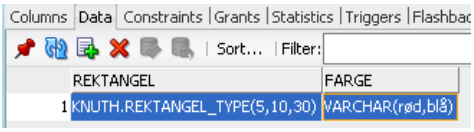
La oss nå slette alle rektanglene:

```
delete (select * from rektangel_tabell);
```

og legge inn én ny, med to fagnr (merk igjen bruken av typenes konstruktør):

```
insert into rektangel_tabell values
  (
  rektangel_type(5, 10, 30),
  farge_array('rød','blå')
);
```

Dataene vises nå slik:



For å velge nummer og farger, kan vi nå skrive:

```
select R.rektangel.nr, R.farge from rektangel_tabell R;
```

som gir følgende resultat:

```
REKTANGEL.NR      FARGE
----------------- ------------------
5                 VARCHAR(rød,blå)
1 rows selected
```

Elementene i arrays referes med indeks på vanlig måte, f.eks. farge(1), **men det er kun tillatt i PL/SQL og ikke i SQL-setninger**. Det finnes ferdige funksjoner som `COUNT`, `DELETE`, `EXISTS`, `EXTEND`, `FIRST`, `LAST`, `LIMIT`, `NEXT`, `PRIOR` og `TRIM`. Noen av dem er prosedyrer, andre er funksjoner, og ingen av dem er tillatt i SQL-setninger. Det er derfor sterkt behov for en *tostring*-funksjon når man bruker varrays. Der kan man "bla" igjennom arrayen, og bygge opp en passende streng for fargene.

D: Table type (nøstede tabeller)

Table type er en samling rader. Det er altså mulig å ha kolonner der hver verdi er en tabell. Det kalles nøstet tabell og kan høres besnærende, men det mange grunner til at dette ikke er særlig aktuelt.

Note: Du kan lese mere om nøstede tabeller i appendiks D.

Øving 2 i Oracle

I den elektroniske lærerressursen finnes øvelse 2 ("Egendefinerte typer og objekttabeller"). Den kan læreren gi deg tilgang til. Du bør gjøre denne øvelsen og studere løsningsforslagene før du begynner arbeidet med den obligatoriske oppgaven.

DEL 2B PRAKSIS – PERSISTENS AV OBJEKTER (MED C#)

Vi skal se på to objektorienterte måter å sikre persistens av objekter på, og vi skal benytte C# til programmeringen.

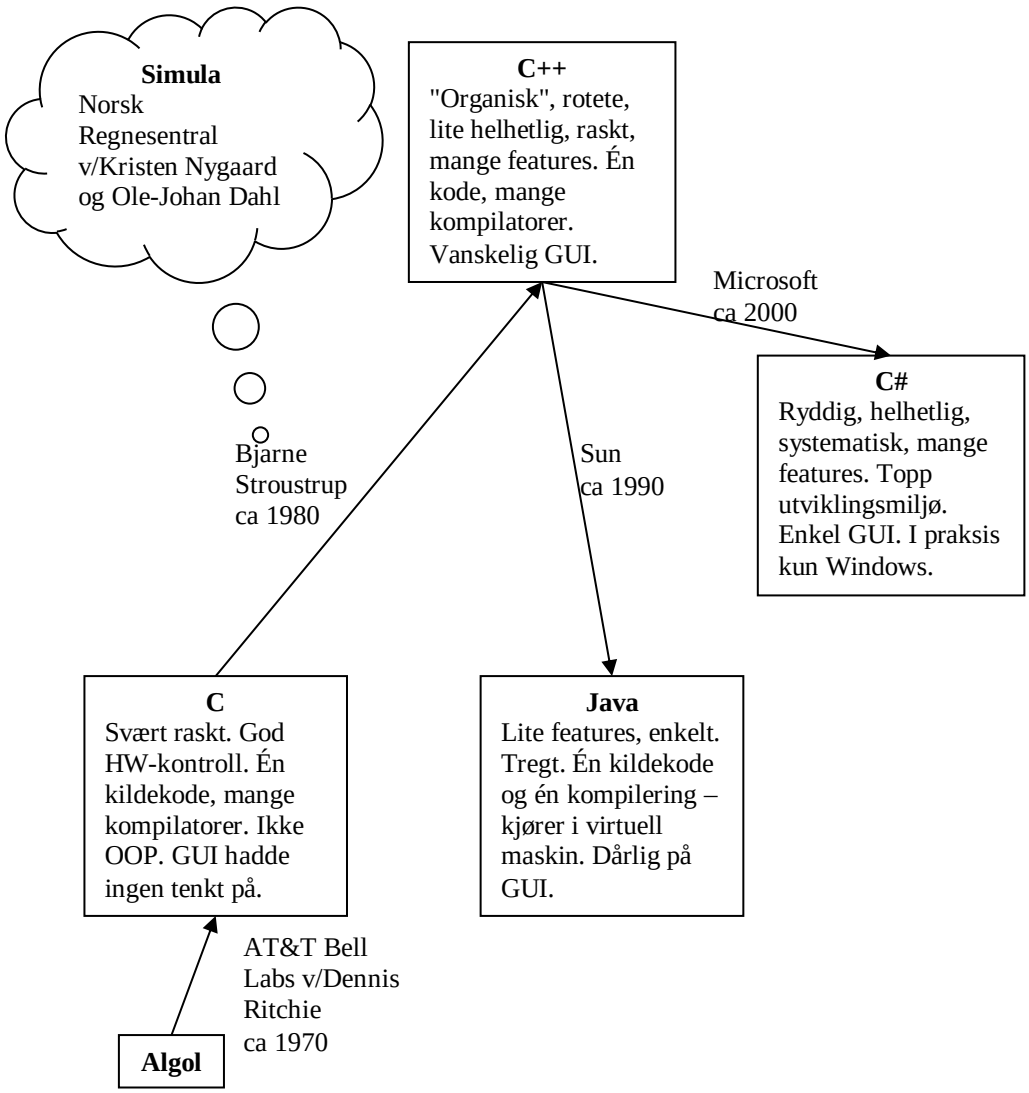
Til øvelsene og obligatoriske oppgaver, vil du da trenge følgende programvare:

1. **C#:** Last ned og installer *Visual C# Express 2010* fra http://www.microsoft.com/visualstudio/en-us/products/2010-editions/visual-csharp-express (få også med evt. security updates)
2. **db40:** Last ned og installer *db4objects* http://www.db4o.com/DownloadNow.aspx.
3. **Prevayler:** Last ned og unzip *Bamboo.Prevalence* fra http://sourceforge.net/projects/bbooprevalence/files/

Note: De oppgitte adressene er korrekte pr august 2012.

Litt historie

Da C# skal benyttes som eksempelspråk, kan det være nyttig å vite litt om C-familiens historie.



Simula ble laget ved Norsk Regnesentral av Kristen Nygaard (systemerer) og Ole-Johan Dahl, (programmerer). Det var ferdig i begynnelsen av 1960-årene og implementert på Univac stormaskiner. Det inneholdt det meste som vi forbinder med OOP i dag, f.eks. klasser/objekter, arv, virtuelle metoder, meldinger og garbage collection. Det regnes helt klart som det første objektorienterte språk. Det fikk liten utbredelse utenfor akademiske miljøer.

En gang rundt 1970 (ferdig ca. 1973) laget AT&T Bell Labs språket C. Det var tett koblet til Unix fra samme sted, som hadde den fordelen at det eksekverte i deler og sendte data fra del til del, såkalt "piping" (ref. "pipelining" i dagens Oracle). I 1973 var C blitt sterkt nok til at Unix kunne omskrives til språket C – det var tidligere skrevet i Assembler. C var svært raskt, og ga tett HW kontroll. Det ble derfor godt likt av ingeniører og fikk stor spredning hos dem. Man skriver koden én gang og kan kompilere til mange O/S med tilpassede kompilatorer. Man bruker teksteditorer som utviklingsverktøy.

Rundt 1980 ble dansken Bjarne Stroustup inspirert av Simula, og ledet utvidelse av C til C++ ("C inkrementert"). Bjarne beskrev utviklingen som "organisk vekst", dvs. at nye features ble lagt til etter hvert. Dermed ble språket noe "rotete" dvs. lite helhetlig (det du kan gjøre ett sted, kan du ikke

nødvendigvis gjøre et annet, liknende sted). Det var fortsatt raskt, og var fullt objektorientert. Det hadde mange features. Du skriver koden én gang og kan kompilere til mange O/S med tilpassede kompilatorer. Utviklingsverktøy er enkle, nærmest rene teksteditorer men med enkel "kompilering".

I begynnelsen av 1990-årene hadde Sun laget ferdig Java. Det er et subsett av C++ og følgelig betydelig enklere, og er fullt objektorientert. Grafiske brukergrensesnitt er tunge å lage eller krever komplekse editorer. Du skriver koden én gang, og kompilerer én gang til "byte code". Kjøring skjer da i en O/S-tilpasset, virtuell maskin, JVM. Java er treg. Det har vært vanskelig å finne brukervennlige utviklingsverktøy for Java, men nå kommer de for fullt, f.eks. NetBeans.

Rundt år 2000 kom Microsoft med C# (egentlig C♯ som i musikk – altså C++ med en halvtone ned). Dette er også et subsett av C++, men mange flere features enn Java og eksekverer raskt. Det er ryddet opp i forhold til C++, og fremstår som mye mer helhetlig. Tanken er at man skal skrive kode én gang og kompilere for flere O/S, men i praksis kjøres C# bare under Windows. Editoren er suveren og grafisk brukergrensesnitt svært enkelt å lage. Java-entusiaster har ment at C# bare er en liten og helt unødvendig utvidelse av Java, men jeg har programmert dem alle og mener at C# er *nesten* som C++. Java ligger langt under.

C# for dem som kan Java

De som kan Java og har programmert objektorientert også med VB, vil ikke ha store problemer med C#. De vil kjenne igjen det meste av koden og nesten alt i programmeringsmiljøet.

Det er noen mindre forskjeller, som man oppdager etter hvert. Særlig ser man at det er flere features, mer fleksibilitet.

Ett, enkelt eksempel på fleksibilitet kan være parameteroverføringen. I Java er alle parametre obligatoriske (de må oppgis). Det kan føre til at man må lage mange, overstyrte versjoner av en funksjon for å kunne kalle f.eks. *f(x,y,z)* eller *f(x,y)*. I C# kan man bestemme om parametre skal være obligatoriske eller frivillige (*optional*) og lager da bare én *f(x,y,optional z)*. Begge deler virker greit, men C# er enklere å lage. I C# velger vi mellom inn, ut og inn-ut parametre. Førstnevnte er en verdi (ikke referanse) som skal brukes – endringer av verdien inne i funksjonen får ingen effekt i den kallende modul. Den andre må være en referanse som ikke må ha verdi ved kallet, men får verdi ved retur. Den tredje er en referanse som har verdi ved kallet og kan ha endret verdi etterpå. I Java er alle metodeargumenter i Java "pass-by-value", altså finnes det kun inn-parametre[16].

Et annet eksempel er at C# har flere muligheter for å angi synlighet – om enn ikke like mange som i C++.

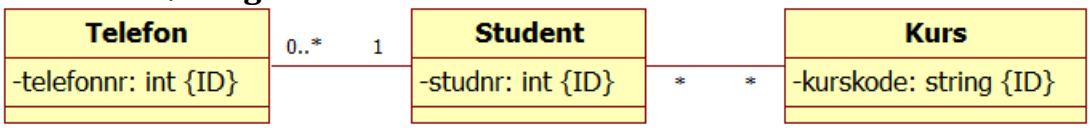
C# skiller ikke mellom interface og klasse og regner bruken av begge som arv. En interface ses da på som en abstrakt klasse.

[16] Hvis argumentet er et objekt sendes en kopi av referansen. Metoden har da allikevel tilgang til objektets attributter og kan endre dem hvis de bare er mutable (kan endres).

Litt bakgrunnsstoff om persistens av objekter

Jeg viser til min omfattende rapport notat om dette[17] for dem som vil ha en mer grundig innføring og drøfting. *Den er imidlertid for omfattende for dette kurset.*

Modell til drøftingene



Hvorfor persistens

Objekter må være i RAM når de skal endres eller utføre handlinger. Men RAM er for ustabilt miljø og objektene må lagres på ytre lager for å eksistere mellom sesjoner, ved maskinavbrudd osv. De må sikres *persistens* (lat. *persistere* av *sistere* dvs å stå fast og *per* som intensiverer, altså å stå virkelig godt fast).

Det er ikke alle objekter som trenger persistens. De andre er *transiente* (lat. *transiens* dvs. overgang). Det er typisk *entitetsobjektene* som modellerer vår Universe of Discourse som er persistente, mens kontroll- og grenseobjekter gjerne er transiente (det kan selvsagt være unntak fra denne hovedregelen).

Det reiser seg to, strategiske valg:

1. Når skal objektene lagres? Vi kan
    a. Vente til programmet avsluttes og lagre alle objektene da. Det kan bli svært sjelden på en server og er risikabelt.
    b. Lagre noen av gangen i bolker f.eks. når maskinen er ledig. Det er mindre risikabelt men vi kan fortsatt miste endringer.
    c. Lagre alle endringer umiddelbart og helst ikke bekrefte det til brukeren før det er gjort med suksess. Det er klart det sikreste men også mest ressurskrevende.
2. Når skal objektene hentes til RAM? Vi kan
    a. hente alle objekter når programmet åpnes (*eager loading*). Det vil fylle opp RAM og gjør objektene utilgjengelig for andre klienter som også trenger dem. Det er svært enkelt.
    b. hente objektene i blokker ved behov, f.eks. alle objekter av en gitt klasse. Det sparer RAM og kan gi andre tilgang til de samme objektene når vi har endret og lagret dem igjen (og fjernet dem fra "vår" RAM). Dette er mer fleksibelt i flerbrukermiljøer, men mer tungvint. Det reiser også problemstillingen om hvor mange objekter som skal med. Skal f.eks. bare kursene hentes, eller skal studentene som kursene henviser til også hentes? Og hva da med studentenes telefoner?
    c. Hente ett og ett objekt ved behov (*lazy loading*). Dette er mest fleksibelt men også mest tungvint. Det krever også at man holder orden på nøyaktig hvilke objekter som finnes i RAM. Hvis man f.eks. henter et kursobjekt, så vil listen over studenter som tar kurset være *null*. Skyldes det at kurset ikke har studenter eller er de bare ikke hentet ennå?

Objektene i RAM er identifisert ved sin RAM-adresse. Når vi skriver

```
Student stud1 = new Student();
```

skapes et studentobjekt i RAM og dets adresse legges inn i variabelen *stud1* som er en *referanse* til objektet (og ikke objektet selv). Når objektet lagres på ytre lager, mister vi denne RAM-adressen. Neste gang objektet hentes til RAM vil det få en ny plassering (på *heap*). Ettersom objektene refererer til hverandre, mister vi dermed alle *assosiasjoner* mellom dem. Vi må følgelig ha et identifiserende attributt lagret sammen med objektet på ytre lager.

[17] Knut W. Hansson: "Persistens av Objekter", Rapporter fra Høgskolen i Buskerud nr 88, 2012.

Da kan vi

1. Velge å bruke et attributt som objektet uansett har, f.eks. kundenummer, fakturanummer, telefonnummer, studentnummer fødselsnummer osv.
2. Legge til et (kunstig) attributt i klassen, som vi sikrer har unik verdi. Dette attributtet kan vises eller undertrykkes i henvendelser til brukeren, men brukes internt.
3. Legge til en generert, unik verdi bare når objektet lagres.
4. Bruke ett eller flere attributter i en hashingalgoritme og anvende denne verdien som identifikator. Alle objekter har funksjonen `public virtual`[18] `int GetHashCode()` som returnerer et heltall. Microsoft vil imidlertid ikke garantere at den returnerer forskjellig verdi for alle objekter, så den må vi eventuelt overskrive selv.

Når et objekt refererer til et annet objekt, er det uansett RAM-adressen det henviser til (i alle fall hvis vi programmerer objektorientert). Det er altså *ikke* her snakk om å bruke en verdi som "fremmednøkkel" når objektet er i RAM. Når objektet *lagres* må derimot assosiasjonen gjøres om til en "fremmednøkkel" som henviser til det andre objektets ID-verdi.

Når et objekt omgjøres til en strøm av tegn eller bytes, sies det å bli *serialized*. Når den hentes fra en strøm av tegn/bytes og omgjøres til objekt igjen, heter det at den blir *deserialized*. Du kan skrive koden for dette selv, eller overlate det til innebygde objekter/funksjoner.

Et system av flere objekter som refererer til hverandre, kalles en *graf*. En spesiell form for graf er hierarki, men det forekommer sjelden i objektorienterte programmer.

Måter å persistere på

Mulighet A: Ingen persistens
Det kan selvsagt tenkes systemer som ikke trenger persistens, f.eks. systemer for prosesstyring. Det går jeg ikke inn på her.

Mulighet B: "Flat", sekvensiell tekstfil
Dette har dere gjort i flere tidligere kurs.

Objektene omgjøres til en sekvens av tegn – en tegnstrøm. Naturlig er å bruke kommalimitert (CSV) format.

Det reiser seg en problemstilling om man skal benytte én eller flere filer. Vi kan lagre hele grafen i én fil, bare vi kan vite hvilken objekttype (klasse) som hver post representerer. De fleste vil nok allikevel velge å lagre hver klasse for seg i en egen fil, *studentfil*, *telefonfil* osv.

Man må spesielt velge hvordan man skal lagre assosiasjonene. Vi kan f.eks. lagre hvert studentobjekt med en liste "fremmednøkler" til kurs og til telefoner. Det vil for det første kreve et annet skilletegn enn komma mellom "fremmednøklene" til kurs og til telefoner. For det andre er spørsmålet hva vi gjør med kursobjektenes referanser til studenter. Vi kan også lagre hvert kursobjekt med en liste med "fremmednøkler" til studentene som tar kurset, men dette er jo redundant. På den annen side er det nødvendig ved lazy loading, da vi ellers ikke kan vite hvilke studenter som tar kurset når et kurs lastes. Helt tilsvarende gjelder for studenter og telefoner. Personlig foretrekker jeg å lagre mange-til-mange assosiasjoner i en egen fil, for å unngå denne redundansen.

[18] Virtuelle funksjoner er funksjoner som forventes redefinert i subklasser. Man kan heller ikke stole på at ferdige klasser har tilstrekkelig god hashingalgoritme – man må skrive sin egen.

Når et objekt legges til, kan det legges til bakerst i filen "sin". Når det endres eller slettes, må filen endres "midt inni". Det er svært tungt med sekvensielle filer.

Hvis flere klienter laster, endrer og lagrer de samme objektene, må de ha skrivetilgang til samme fil. Det skaper også betydelige problemer. Hvis filen bare leses går det allikevel greit. I det siste tilfelle kan filen endres av og til, og filen settes da midlertidig utilgjengelig for andre. Et eksempel kan være en samling poststeder som stort sett bare skal leses til RAM (for deserialisering til postobjekter). Det går også greit hvis filen tar vare på objekter som er unike for hver bruker (f.eks. nettleserens bokmerker) og hver bruker får sin egen fil.

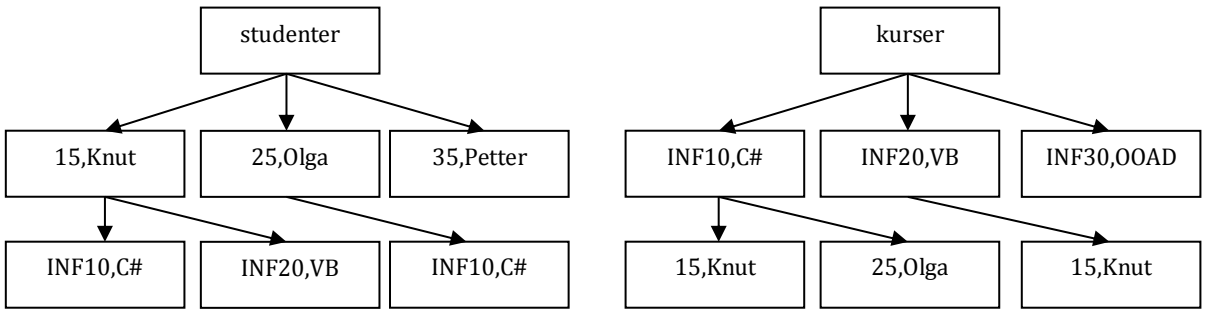
En fordel med kommalimiterte filer er at de kan åpnes med andre programmer og endres manuelt. De kan også enkelt importeres til andre programmer.

Mulighet C: Binær form

Dette har dere gjort med Visual Basic og kanskje også med Java?

Istedenfor å serialisere objektene til en sekvensiell strøm tegn, kan vi serialisere til en bytestrøm. Egne klasser hjelper oss med dette. En ulempe er at *BinaryFormatter* bare kan serialisere ett objekt. Vi må følgelig legge alle objekter som skal lagres inn i én og samme samling.

Når objekter serialiseres, tas også objekter som det refereres til med (*eager* strategi). I vårt tilfelle vil det innebære at serialisering av alle studentobjektene vil ta med de kursene som studentene henviser til. Fortsatt vil det da finnes kurs som ikke blir lagret, fordi de ikke har studenter. Hvis man da i tillegg serialiserer kursobjektene (for å få med alle), vil de ta med mange – men ikke alle – studentobjekter. De vil danne to, separate grafer:



Problemet er at når kursene lagres med studenter, blir studentene lagret som *nye objekter* selv om de er lagret fra før. Det er naturligvis ikke ønskelig. Det kan unngås ved uttrykkelig å implementere grensesnittet *ISerializable* med metoden *GetObjectData()* som brukes i serialiseringsprosessen. Man kan også merke attributter som man ikke vil ha med som *[NonSerialized]* (Java: *transient*). Det løser imidlertid ikke problemet her, for hvis referansene *kurs.kursdeltakere* og tilsvarende for studentobjektene merkes *[NonSerialized]*, så kommer sammenhengene mellom ikke objektene med i det hele tatt.

Løsningen på dette er å samle alle studentobjekter og kursobjekter i én struktur. Denne strukturen serialiseres og da kommer alt med. Serialiseringsobjektene er "smarte nok" til ikke å lagre studentobjekter/kursobjekter dobbelt.

De serialiserte objektene lagres i sekvensielle filer. De problemene jeg omtalte ovenfor for lagring i sekvensielt tekstfil gjelder derfor også her. I tillegg kommer at det ikke er mulig å endre innholdet i filen – alle objektene må lagres pånytt hvis ett blir endret (og vi vil sikre persistens med en gang). De er også helt umulige å endre manuelt og vanskelig å importere til andre programmer.

Mulighet D: XML-fil
Dette har dere også gjort tidligere.

Objektene kan lagres som XML og egne klasser finnes til det. Det er minst to forskjellige. Den enkleste, *XmlSerializer*, stiller følgende krav til objektene som skal serialiseres:

1. Attributtene må være *public*. Det ødelegger altså innkapslingen.
2. Den kan bare serialisere ett objekt og det må være et *objekt* – den klarer altså ikke å serialisere en *struct* (struct brukes mye i C-språkene – det tilsvarer et objekt uten metoder, altså en post).
3. Klassene behøver ikke merkes *[Serializable]*.
4. Alle referanser blir også serialisert (*eager*), men den håndterer ikke sirkelreferanser (det gir feilmelding når objektet oppdager det).

Vi har sirkelreferanser når objekt A refererer til B som direkte eller indirekte refererer til A igjen. Det er ganske vanlig. Hvis referansene er *not null* går det allikevel greit. F.eks. kan vi la være å lagre referansene fra student til telefoner og bare lagre telefonenes referanse til student. Deretter kan vi enkelt bygge opp igjen studentenes referanser til telefonene ved å gå igjennom alle telefonene.

En mer kompleks klasse, *DataContractSeriliazer*, håndterer selv sirkelreferansene. Den har en temmelig kompleks konstruktør, der man angir hvordan den skal virke.

Da også XML-filen er sekvensiell, oppstår de samme problemene som med tekstfiler og binære filer som er omtalt ovenfor. En stor fordel er imidlertid at filen kan endres manuelt og leses til mange andre programmer. En meget viktig fordel er også at XML-filen har standardisert format (mens flate tekstfiler har hvert sitt format som må være kjent hvis andre skal bruke den).

Mulighet E: Relasjonell eller objektrelasjonell database
Også dette har dere programmert tidligere i flere kurs.

Dette er nok en svært vanlig metode. Man bruker jo kjent teknologi og får bra styring på flerbruker-problematikken (hvis det gjøres riktig). Det er i praksis ingen forskjell på relasjonell og objektrelasjonell database i denne sammenheng.

Man må gjøre om strukturen i det objektorienterte programmet så det passer til en relasjonsdatabase, såkalt *mapping*. Det har vi hatt i et tidligere kurs. Når da et objekt skal lagres i databasen, må det serialiseres. I praksis medfører det at man selv må programmere serialiseringen, og deserialiseringen. Det er ganske tungt og kan fort gi feil. Man må bruke egne klasser til å holde kontakt med databasen.

Hvis alle objektene lastes ved programstart – eager strategi – vil de måtte settes "opptatt" (dvs. låses) i databasen. Da vinner man ikke noe i flerbrukersammenheng. Man må altså bruke lazy strategi. Det gir mye ekstra programmering for å holde orden på hva som er i RAM til enhver tid, og de må hurtigst mulig ut derfra igjen så postene i databasen kan fristilles for andre. En mulighet er å ha alle objektene i RAM bare på serveren. Da gir låsingen i databasen ingen problemer. Det man allikevel oppnår i forhold til sekvensielle filer, er at man enkelt kan oppdatere ett objekt (uten å måtte lagre alle) i databasen. For øvrig vil en sekvensiell fil gjøre samme nytten (og er mye enklere å programmere).

Vi kan også merke oss at datatypene ikke er like i programmet og i databasen. F.eks. har ikke Oracle Boolske verdier, men det bruker vi naturligvis gjerne i programmet. Videre har et heltall ikke samme representasjon i Oracle som i C# osv.

Det er komplisert og blir fort feil når man benytter relasjonsdatabase til persistens av objekter. Jeg synes det er rat at dette allikevel er så populært. Jeg tror det kan ha sammenheng med at relasjonsdatabaser er den klart mest brukte databasetypen og godt kjent både i virksomhetene og fra

utdanningen, samtidig som objektorientert programmering blir stadig vanligere. Objektdatabaser er en relativt ny teknologi som først nå begynner å bli moden og feilfri, og de er ikke så kjente hverken i virksomhetene eller fra utdanningen.

Entity Framework og andre muligheter

Det finnes noen verktøy som kan bidra til å mappe objektorienterte systemer til relasjonsdatabaser og gjøre det mye enklere å lagre/hente data. Wikipedia har en grei oversikt på http://en.wikipedia.org/wiki/List_of_object-relational_mapping_software#.NET. Disse er av minst to typer:

1. De tilbyr verktøy som genererer XML-fil som beskriver mappingen. Utgangspunktet er enten programmet (da skapes databasen fra et generert sql-skript) eller databasen (da skapes klasser tilsvarende databasens tabeller).
2. De tilbyr klassebibliotek som gir persistens gjennom arv (f.eks. kan studentklassen gjøres persistent ved å være en subklasse av en klasse *DatabaseObject* som har de nødvendige metodene for lagring i databasen).

En svært interessant mulighet er Entity Framework[19]. Det tilbyr mye hjelp med mapping. Mappingen lagres i en XML-fil. Senere benytter rammeverkets objekter denne XML-filen når objekter skal lagres/hentes. Der vil det f.eks. kunne fremgå at ett objekt er lagret i flere tabeller (som må kobles når objektet hentes) og hvilke objekter som ett objekt refererer til. Man behøver ikke selv å skrive SQL-setningene hverken for henting, lagring eller oppdatering. Samtidighet ordnes "bak kulissene".

Rammeverket kan ta utgangspunkt i en egen entitetsmodell, i koden eller i en eksisterende database. Slik kan deler av XML-filen se ut (fra den refererte nettsiden):

```
<ComplexType Name = "Addr">
    <Property Name = "Street" Type = "String" Nullable = "false" />
    <Property Name = "City" Type = "String" Nullable = "false" />
    <Property Name = "Country" Type = "String" Nullable = "false" />
    <Property Name = "PostalCode" Type = "Int32" />
</ComplexType>
<EntityType Name = "Customer">
    <Key>
        <PropertyRef Name = "Email" />
    </Key>
    <Property Name = "Name" Type = "String" />
    <Property Name = "Email" Type = "String" Nullable = "false" />
    <Property Name = "Address" Type = "Addr" />
</EntityType>
<Association Name = "CustomerAndOrders">
    <End Type = "Customer" Multiplicity = "1" />
    <End Type = "Orders" Multiplicity = "*">
        <OnDelete Action = "Cascade"/>
    </End>
</Association>
```

Det synes klart at Entity Framework nok er et verktøy for fremtiden. Dessverre har jeg ikke kunnet prøve det ut.

Mulighet F: Objektdatabase og andre NoSQL lagringssystemer

Dette antar jeg er nytt for dere. Dette skal vi følgelig bruke tid på gjennom et eksempel.

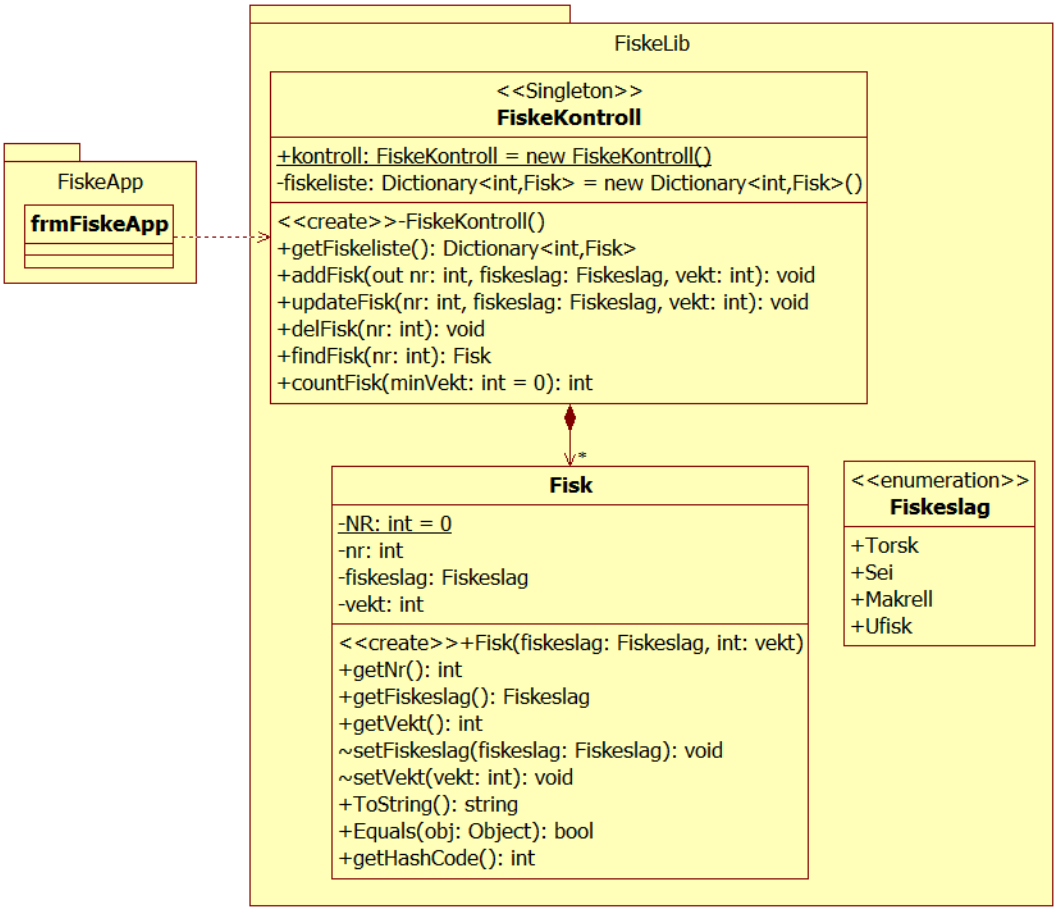
[19] Se f.eks. http://en.wikipedia.org/wiki/ADO.NET_Entity_Framework

Vi skal se på to, konkrete muligheter: *db4objects* (en objektdatabase) og *Bamboo Prevayler* (som egentlig ikke er en database men sikrer persistens på annen måte). De to benytter helt forskjellige strategier for å oppnå persistens. Begge er fri programvare (FLOSS).

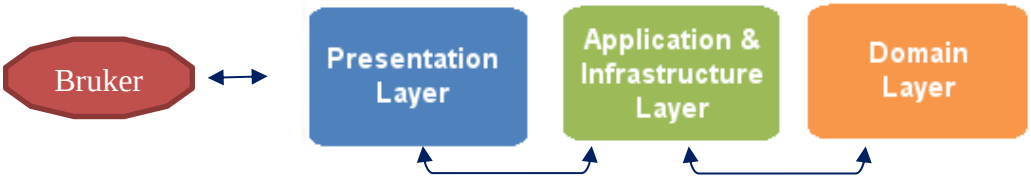
Note: Du kan evt. lese litt mer om NoSQL databaser i appendiks E.

Demonstrasjon av objektpersistens med NoSQL teknikker

Læreren har – i den elektroniske lærerressursen – fått eksempelprogrammet "FiskeDemo" i tre deler: Uten persistens, med persistens i databasen db4objects og persistens med Prevayler. Dette kan læreren gjøre tilgjengelig så du kan laste det ned, studere det og prøvekjøre det. Eksempelprogrammet "FiskeDemo" er også kommentert i et eget dokument som læreren kan gi deg tilgang til.



Systemet er lagt opp etter modellen DDD (Domain Driven Design) men litt tilpasset med tre lag:



Presentasjonslaget utgjør da grensesnittet mot brukeren. Applikasjons- og infrastrukturlaget håndterer *alle* tilstandsendringer i domenelaget, og domenelaget utgjøres av entitetsobjektene. Jeg tenker meg at systemet deles i to: En klientdel og en tjenerdel. Tjeneren kan da kjøre som en, enkelt, frittstående server, med flere klienter tilknyttet fra andre maskiner. Vi skal ikke her konkret lage en slik løsning, men tenke oss at det er slik det gjøres. Begge verktøyene for persistens som vi skal anvende, støtter en slik ekte klient/tjener-løsning med kommunikasjon over TCP/IP.

Øving i objektpersistens

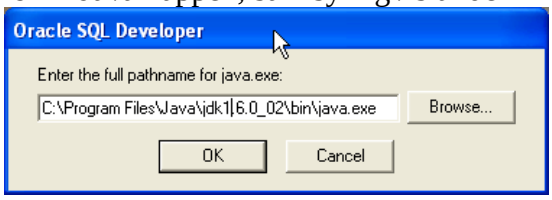
I den elektroniske lærerressursen finnes øvelsen ("Hittegods"). Den kan læreren gi deg tilgang til. Du bør gjøre denne øvelsen og studere løsningsforslagene før du begynner arbeidet med den obligatoriske oppgaven.

APPENDIKS A
INSTALLASJON OG KONFIGURERING AV ORACLE SQL DEVELOPER

Bildene nedenfor er fra Oracle SQL Developer 3.0 fra mars 2011.

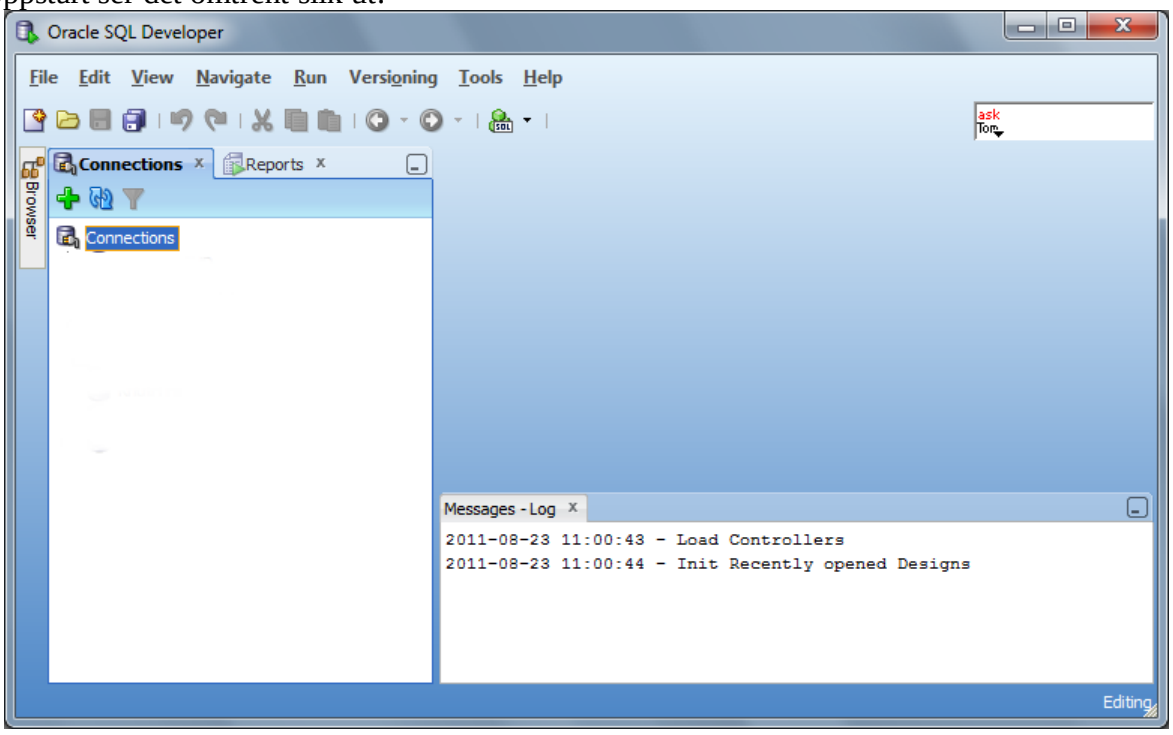
- ✓ Sjekk om du har JDK 1.6.0_11 eller bedre (sannsynligvis har du allerede JDK 7). Det kreves for å få brukt programmet. Evt. må du hente og installere tilstrekkelig JDK-versjon. Det holder *ikke* med bare Java run time eller Java VM – det kreves Java.exe. Hvis du ikke har tilstrekkelig JDK installert, henter du Developer *med* Java (versjon 1.6.0_11 følger da med) eller henter/installerer bedre Java-versjon selv, *før* installasjon av Developer.
- ✓ Last ned riktig zip-fil (med/uten Java, evt. for Mac/Linux) fra http://www.oracle.com/technetwork/developer-tools/sql-developer/downloads/index.html
- ✓ Følg installasjonsinstruksjonene i kapittel 1.2 på http://download.oracle.com/docs/cd/E18464_01/doc.30/e18458.pdf

Første gang du starter SQL Developer, blir du spurt om du skal "migrere" (dvs. at du har lastet ned ny versjon og vil ha data overført). Det har du antakelig ikke. Du kan også få et spørsmål om hvor Java ligger. Merk at det er Java JDK (*java.exe*) som skal angis – det holder ikke med Java Run time eller Java VM. Du finner den under *bin* i Javamappen, sannsynligvis under *Program Files*.

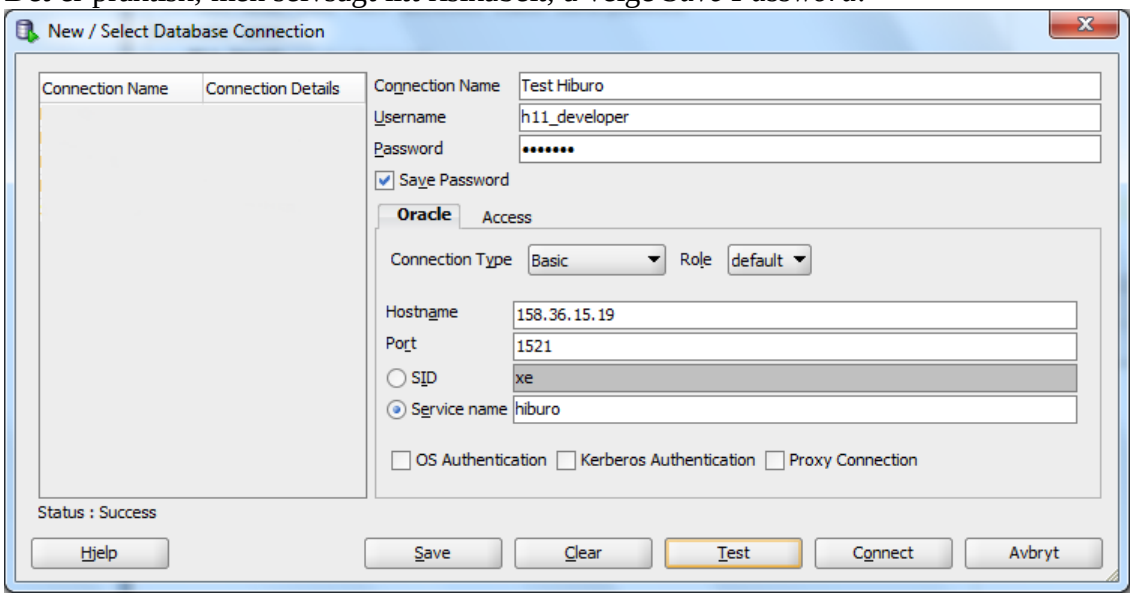


Hvilke filer du vil assosiere med SQL Developer kan du bestemme siden under "Tools"-menyen.

Ved oppstart ser det omtrent slik ut:



Du må lage minst én "Connection". Du kan også lage flere, og du navngir dem fritt. Klikk plusstegnet ”New connection” og fyll ut. Du bruker selvsagt eget brukernavn/passord.
Det er praktisk, men selvsagt litt risikabelt, å velge *Save Password*.



Kjør ”Test” og se etter ”Success” eller feilmelding. Deretter kan du klikke ”Connect” og SQL Developer er klar til bruk. Hvis du brukte testbrukeren, må du lage en ny connection for ditt eget brukernavn/passord – og slette testbrukeren – før du gjør noen øvelser.

Sjekk at det virker ved å gjøre øvelse 1.

APPENDIKS B
KORT OVERSIKT OVER ORACLES STRUKTUR

Filer

Oracle har forskjellige typer av filer:

1. **Datafiles** lagrer data og knyttes til bare én database. De kan utvides automatisk ved behov. En gruppe datafiler kalles et *tablespace*.
2. **Control files** lagrer fysiske metadata, f.eks. databasenavn, navn og adresse til data og redo-filene, tidspunkt for databaseopprettelsen o.a. Control files kan *multiplexes* dvs. speiles. De holdes automatisk a jour.
3. **Redo log files** logger alle endringer i data. Det muliggjør *rolling forward* og kan også multiplexes.
4. **Andre filer**:
    a. **Archive log files** er backup av tidligere redo log files
    b. **Parameter files** er backup av konfigurasjonsdata
    c. **Trace files** logger alt som skjer (dvs. alle handlinger/hendelser). Det brukes bl.a. av DBA til tuning. En subtype er **alert file** som logger feil og brukes til vedlikehold.
    d. **Backup files** er en full kopi av databasen på et gitt tidspunkt (snap shot). Kopiering til backup file kan automatiseres og brukes til recovery slik
        i. Først restore siste backup
        ii. Deretter roll forward redo log files

Logisk databasestruktur

1. **Data block** er minste, adresserbare enhet. Blokkstørrelsen settes i bytes med DB_BLCO_SIZE når databasen skapes. Det kan også angis opptil seks blokkstørrelser som brukes av forskjellige filer.
2. **Extent** er et fast antall blokker som lagres kontinuerlig (contiguous).
3. **Segment** er en mengde extents. Det er forskjellige typer segmenter:
    a. **Data segment**
    b. **Index segment**
    c. **Temporary segment** for midlertidige data produsert under utførelsen av en SQL-setning. Den slettes (eller egentlig fristilles til ny overskriving) etter utførelsen.
    d. **Rollback segment** er det stort sett slutt på – det gjøres nå annerledes.

    Oracle utvider automatisk segmentene med en ny extent når de er fulle.

Databasen deles også logisk i **table spaces** (minst én). De grupperer relaterte data. De kan være

1. **Smallfile tablespace** (default) for inntil 1024 filer
2. **Bigfile tablespace** utnytter 64 bits adressering og kan bli opptil 8 exabytes (exa = 10^{18} = "trillioner" = en milliard milliarder. De utgjør bare én fil med data tilsvarende 1024 filer.

Tablespaces kan settes off-line, dvs. utilgjengelig for brukerne, f.eks. under administrasjon av databasen.

Det er mange table spaces i en database, f.eks.

1. Bigfile
2. SYSTEM med datadictionary, stored procedures og triggere
3. SYSAUX med tillegg til SYSTEM
4. UNDO der hver transaksjon tilordnes ett segment
5. Default temporary tablespace

Skjema
Oracle benytter begrepet *object* om tabeller, views og indekser. Det er ingen sammenheng mellom skjema og table space (det kan være flere skjema pr tablespace og omvendt).

1. **Tables** inneholder data, delt i *rows* og *columns*
2. **Indexes**
3. **Views**
4. **Clusters** er grupper av tabeller som lagres sammen fysisk. Det er tabeller som ofte brukes sammen, f.eks. fordi de har samme kolonner (FK/PK eller annet). Clusteret kan hashes.
5. **Triggere**
6. **Prosedyrebiblioteker**
7. **Java** (klasser, kilder og ressurser)
8. **Bruker**
9. **Roller**
10. ... og annet

Data dictionary
Data dictionary er read only og vedlikeholdes automatisk av Oracle. Den inneholder data om

1. **logisk struktur** (skjema)
2. **fysisk struktur** (allokert og brukt plass)
3. **brukere** (navn og rettigheter, evt. roller dvs. en gruppe rettigheter)
4. **integritetsregler** (domener, FK/PK, kaskader o.l.)
5. **revisjonsdata** (hvem/når er skjemaet endret)
6. **andre generell data**

Data dictionary lagres i SYSTEM table space og kan bare leses med SELECT.

Prosesser/minne
Serveren kjører en lang rekke parallelle prosesser. Mange av dem ligger fast i minnet. Jeg går ikke nærmere inn på det her.

Det må også kjøres et program på klienten, som kommuniserer med serveren.

Utilities
Oracle tilbyr en rekke utilities (frittstående programmer som utfører handlinger på databasen).

Språk

1. **SQL**. Oracle aksepterer ANSI/ISO SQL-99-core, og SQL:2003. Vider har Oracle sine egne utvidelser.
2. **Kommandoer** til databasen, f.eks.
    a. **Connect/disconnect**
    b. **Describe**
    c. **Open/close, mount/dismount, shutdown**
    d. **Backup/restore**
    e. **osv.**
3. **Triggers** som "fyrer av" en prosedyre ved gitte hendelser, f.eks. insert, update eller delete.
4. **Scripts** dvs rene sekvenser av handlinger. Disse er knyttet til klienten – én og én setning sendes til serveren for utførelse. Oracle tilbyr mange halvferdige scripts, som du retter opp og kjører. (Server Side Scripts dreier seg om scripts som inngår i dynamiske nettsider.)
5. **Prosedyrielle språk**
    a. **PL/SQL** er Oracles eget programmeringsspråk. PL/SQL kan kjøres *interpretert*, f.eks. fra et script, eller *kompileres* og lagres på serveren som en *stored procedure*. Det kjøres da ved direkte kall eller via et trigger.

    b. **Java** kompileres og kjøres på server på samme måte som PL/SQL (Oracle har altså en JVM).
6. **Bibliotheksklasser for databasetilgang** for
    i. **C og C++**
    ii. **Java**
    iii. **.NET** gjennom ODBC, også med en egen "data provider"
    iv. **COBOL** med en precompiler
    v. **FORTRAN**

Andre begreper

Oracle har mange andre elementer som tilsammen gjør det til en fleksibel database. Her er noen av dem:

1. **Objekter** med attributter og metoder (stored procedures/functions eller pekere til eksterne programmer). Objekter kan være datatype for en kolonne som da følgelig ikke har atomære verdier.
2. **Collections** som er en samling av elementer, f.eks.
    a. **arrays/varrays**
    b. **table**
3. **LOBS** (Large Objects)
    a. **CLOB** = Character Large Object – en sekvens av tegn
    b. **NCLOB** = National Character Large Object
    c. **BLOB** = Binary Large Object
    d. **BFILE** = binære data lagret i en ekstern fil. Disse er read only og bare random access
    e. **RAW** og **LONG RAW** er binære data som Oracle ikke kan konvertere ved flytting til et annet O/S
4. **ROWID** er en pseudokolonne i alle tabeller og blir ikke listet ved SELECT * eller DESCRIBE. De er enten
    a. **fysiske**, dvs. adressen til hver post
    b. **logiske** dvs primærnøkkelen til posten – brukes i indekser

    Disse får du ikke endret selv – de vedlikeholdes automatisk av Oracle. Du får listet dem hvis du uttrykkelig ber om det, f.eks. SELECT rowid, * FROM... ROWID lagres ikke i databasen og de er dynamiske. Du kan følgelig ikke bruke dem til identifikasjon, fordi de kan bli endret (av andre) før neste kall.
5. **Inheritance** gjelder mellom typer, inkludert objekttyper. Objektene kan være virtuelle, ha konstanter, ha virtuelle data, ha funksjoner osv., mye som i OOP.
6. **User aggregate functions (UDAG)** er brukerdefinerte aggregeringer av typen AVG, MAX o.l.

APPENDIKS C
ORACLE OG HTML

OBS! Vår server har ikke http-server installert, så dette får vi ikke forsøkt.

I praksis kan vi jo ikke vente at brukere sitter med SQL Developer og oppdaterer databasen derfra. PL/SQL kan også lage HTML-sider dynamisk. Den ene måten å gjøre det, er å legge HTML-koder inn i prosedyren, slik at prosedyren skaper en webside for retur. Da må vi lage *hele* websiden, f.eks. slik:

```
create or replace
procedure ALLE_BILER
  (fra_aar number)
/*Lister alle biler fra det angitt året*/
is
begin
  htp.p('<HTML>');
  htp.p('<HEAD>');
  htp.p('<TITLE>KNUTH: Alle registrerte biler</TITLE>');
  htp.p('</HEAD>');
  htp.p('<BODY>');
  htp.p('<H1>Alle biler registrert hos KNUTH</H1>');
  htp.p('<TABLE BORDER = "1" WIDTH = "90%">');
  htp.p('<TR>');
  htp.p('<TD>Bilnr</TD><TD>Bilmerke</TD><TD>År</TD>');
  htp.p('</TR>');
  for post in
      (select b_nr, b_merke, b_år, b_e_nr from bil
      where b_år >= fra_aar
      order by b_nr)
  loop
    htp.p('<TR>');
    htp.p('<TD>'||post.b_nr||'</TD>'); /* tegnet || gir konkatinering */
    htp.p('<TD>'||post.b_merke||'</TD>');
    htp.p('<TD>'||post.b_år||'<TD>');
    htp.p('</TR>');
  end loop;
  htp.p('</TABLE>');
  htp.p('</BODY>');
  htp.p('</HTML>');
  end;
```

Denne prosedyren krever ett argument (et NUMBER), og vil liste alle biler som har år >= det angitte argumentet. Legg videre spesielt merke til for-løkken der *post* er én bestemt post som selectsetningen returnerer. Da kan vi legge inn kolonner fra posten med syntaksen ||*post.b_nr*|| osv. Hver iterasjon vil altså legge inn data fra én post i en tabellrad (<TR>…</TR>).

Når brukeren skal kalle denne prosedyren, man han/hun skrive

```
http://158.36.31.24:80/<pslql-katalog>/<data-acces-descriptor>/ALLE_BILER
```

Merk at porten for http er 80, og at vi må angi subkataloger. plsql-katalogen heter vanligvis *psl* og data-asccess-descriptoren kalles også DAD og er et katalognavn som man angir når man starter http-serveren.

Vi kan også bruke PL/SQL embedded i HTML. Da kan det se omtrent slik ut:

```
<%@ page language = "PL/SQL" %>
<%@ olsql procedure = "alle_biler" %>
<HTML>
<HEAD>
<TITLE>KNUTH: Alle registrerte biler</TITLE>
</HEAD>
<BODY>
<H1>Alle biler registrert hos KNUTH</H1>
<TABLE BORDER="1" WIDTH="90%">
<TR>
<TD>Bilnr</TD><TD>Bilmerke</TD><TD>År</TD>
</TR>
<%
  for post in
    (select b_nr, b_merke, b_år, b_e_nr in bil order by b_nr) loop
%>
    <TR>
    <TD><%= post.b_nr %></TD>
    <TD><%= post.b_merke %></TD>
    <TD><%= post.b_år %><TD>
    </TR>
<% end loop; %>
</TABLE>
</BODY>
</HTML>
```

Denne koden må lastes til Oracle med programmet *psploader* og det må – tror jeg – gjøres på serveren. Det har jeg heller ikke fått prøvd.

APPENDIKS D
TABLE TYPE (NØSTEDE TABELLER)

Note: Vi kan oppnå omtrent det samme som nøstede tabeller ved å lage to tabeller og henvise fra den ene til den andre med FK, og spesifisere kaskade ved sletting av hovedtabellen. Den eneste mulige fordelen med nøstede tabeller som jeg kan se, er at hovedtabellen ikke behøver PK (da Oracle automatisk genererer AK i den nøstede kolonnen). De færreste Oracle-eksperter ser noen overbevisende grunner til å bruke nøstet tabell istedenfor tradisjonelle tabeller med joins. Det eneste er at den eksekverer raskere, da man unngår joins. En stygg ulempe med nøstede tabeller er dessuten at beskrankninger på den relaterte (sub)tabellen blir vanskeligere og delvis umulig – se nedenfor.

Table type brukes til å nøste tabeller. Også nøstede tabeller er en samling (*Collection*). Det gir mulighet for å ha en *tabell* som kolonne i en annen tabell. Den har ingen øvre grense for antall verdier (les: rader), men ved sletting kan rader bli stående tomme. De kan ikke aksesseres med indeks, men de enkelte radene kan hentes ut (se nedenfor). Nøstede tabeller har bare én kolonne – hvis du trenger flere må du først lage et objekt som blir datatypen for den ene kolonnen.

For å få til dette uten å bryte med relasjonsprinsippet om atomære kolonneverdier, lager Oracle en ny tabell som er "underlagt" hovedtabellen. Den nøstede tabellen som tilhører radene i hovedtabellen lagres altså i en egen tabell som Oracle kaller en *store table*. I hovedtabellens kolonne for den nøstede tabellen, legger Oracle automatisk en AK for hver rad. I *store table* får radene denne verdien som FK.

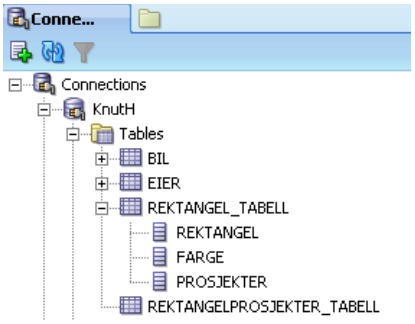
Vi vil lage en tabell med referanser (et nummer) til prosjekter der rektangelet har vært i bruk:

```
create or replace type prosjekt_tabell_type as table of number;
```

Denne skal inn i rektangeltabellen som ny, nøstet kolonne:

```
alter table rektangel_tabell
  add (prosjekter prosjekt_tabell_type)
  nested table prosjekter
  store as rektangelprosjekter_tabell;
```

Legg merke til at vi må oppgi *kolonnenavn* og *kolonnetype* som vanlig. I tillegg må vi fortelle hvilket *tabellnavn* den nøstede tabellen (*store table*) skal ha. Legg også merke til at Oracle nå har laget tabellen "rektangelprosjekter_tabell". Den får du ikke lov å se på direkte, så klikking på den gir feilmelding. Tilgang til denne *store table* skal jo gå gjennom rektangeltabellen.



La oss nå slette de postene vi måtte ha i rektangeltabellen

```
delete (select R.* from rektangel_tabell R);
```

og så legge inn en ny post:

```
insert into rektangel_tabell values
  (
  rektangel_type(5,100,300),
  farge_array('rød','blå'),
  prosjekt_tabell_type(91,92,93)
  );
```

Legg spesielt merke til hvordan vi legger inn tre rader i den nøstede tabellen.

Hvis vi vil velge ut alle rektangler med prosjektnummer, må vi lage et kartesisk produkt av rektangeltabellen og prosjekttabellen:

```
select R.rektangel.nr, R.rektangel.kvm(), P.*
  from rektangel_tabell R, table(prosjekter) P;
```

Legg merke til måten vi henter den nøstede tabellen på med *table* som jo er konstruktøren for en tabell. Det finnes ingen tilsvarende funksjon for å hente ut elementer i en *varray*.

Slik blir resultatet:

```
REKTANGEL.NR     R.REKTANGEL.KVM()  COLUMN_VALUE
---------------- ------------------ ---------------------
5                30000              91
5                30000              92
5                30000              93
3 rows selected
```

Det er mulig å legge ***beskrankninger på den underordnede tabellen***, men ofte virker det ikke etter hensikten. F.eks. kan man ønske å sikre at et prosjekt bare tas med én gang for hvert rektangel. Hvis vi legger en unik-beskrankning på prosjektnummer, er resultatet at et prosjekt bare kan forekomme én gang i undertabellen. Da kan et rektangel ikke brukes i et prosjekt hvis andre allerede har gjort det. Det var ikke hva vi ønsket. (I en ekstern tabell ville vi hatt fremmednøkkel til rektangelet og tatt det med i unik-beskrankningen og det ville løst problemet.)

APPENDIKS E
NOSQL DATABASER

Objektdatabaser er et av flere eksempler på en teknologi som er "på vei inn" – NoSQL databaser. De beskrives som "post relational data management" systemer. Et hovedproblem med relasjonsdatabaser med dagens teknologi, er modellens strenge struktur. Man må på forhånd definere hvilke og hva slags data som skal lagres. Det er vanskelig å forutsi hvilke data som skal lagres og hvilken struktur de har. Dataene har forskjellige formater, f.eks. bilder, multimedia, dokumenter, nettsider, tweets, facebook entries, tekstmeldinger osv. og formatene endrer seg stadig og nye kommer til. Dataene kan også bli lagret flere steder f.eks. i en mobil-app, på PCen og på en server og alle versjonene må være like. Uansett format og sted, så skal dataene lagres og de skal være søkbare. Mange løsninger på problemet har vært forsøkt med utvidelser/endringer i relasjonsdatabasene, men de angriper bare symptomene og ikke problemet.

Flere selskaper har vært nødt til å skape sine egne – ikke-relasjonelle – løsninger for å møte problemene de hadde. Eksempler er Google (Big Table) og Amazon (Dynamo) som begge har data som svært dårlig lar seg strukturere inn i en relasjonsdatabase.

Bare ytterst få virksomheter har ressurser til å lage sine egne databasehåndteringssystemer. Det har derfor etterhvert kommet mange NoSQL databasehåndteringssystemer, både frie og kommersielle som gjør det enklere å lage slike databaser.

NoSQL databaser kan deles i

1. **Dokumentbaserte databaser.** Her er hver post et "dokument", dvs. en autonom samling data som er komplett selvbeskrivende, f.eks. XML, HTML, pdf og MS Word. Hvert "dokument" får en unik ID, men ellers er det ingen fast struktur på dataene og følgelig heller ikke noe skjema. Dokumentene spres på mange servere og klienter.
2. **Grafdatabaser.** Det som lagres er primært assosiasjoner, f.eks. veisystemer og sosiale systemer (facebook). Assosiasjonene er mange til mange og sirkulære (dine venner er mine venner osv). Navigasjon skjer via assosiasjonene. Assosiasjonene er strukturerte, f.eks. kan en vei krysse eller munne ut i en annen vei. Veier kan også knyttes til steder og begynne, slutte, gå igjennom eller passere steder.
3. **Nøkkelverdidatabaser ("key value store").** Dataene lagres i par: Første angis navnet på verdien (nøkkel), deretter angis verdien. I tillegg knyttes dataparet til en identitet. Siden hver post er komplett beskrevet behøver ikke postene ha samme struktur[20]. F.eks. kan begge disse postene lagres selv om de har helt forskjellig struktur:
   ID:kunde1 (navn:Knut, telefon:22224444, arbeidsgiver:HiBu)
   ID:kunde2 (fnr:12345, alder:32)
   Data lagres som datatyper eller objekter.

   En form for slik nøkkelverdidatabase er **objektdatabaser** (også kalt objektorienterte databaser). Her er det objekter som lagres. Objektene får en unik ID som nøkkel – gjerne automatisk – og verdien er objektet. Objektene kan "pakkes ut" ved å bruke klassen der objektets medlemmer er beskrevet. Disse databasene er følgelig spesielt godt egnet til å sikre persistens av objekter.

[20] I en relasjonsdatabase måtte vi ha laget en tabell *kunde(navn,telefon,arbeidsgiver,fnr,alder)* med mange lagrede nulls. Hvis en ny kunde hadde noe annet av interesse, måtte tabellen utvides og alle eksisterende rader ville fått null i den nye kolonnen. Svært tungvint og en fryktelig sløsing med plass. Det vil også kreve algoritmer og prosessortid for å konvertere mellom objekter og rader i databasen.

APPENDIKS F
BINÆRSØK I EN ARRAY

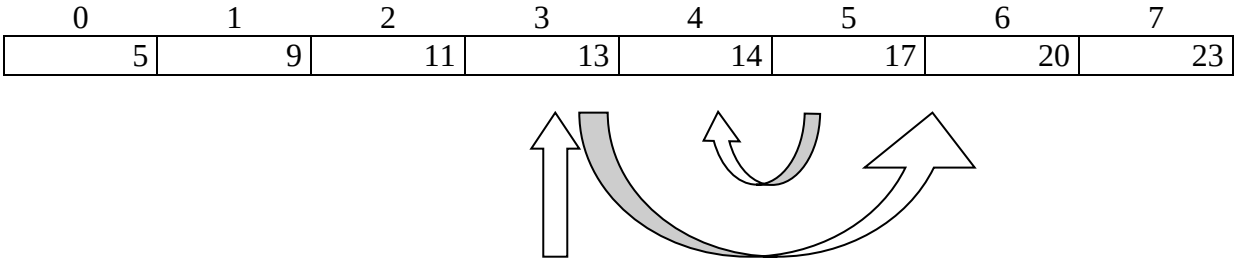
Skrevet i Visual Basic 2008 Express. Dette er repetisjon som studentene kan lese selv.

```
Public Class ArrayBinarySearch
    Dim tall(100) As Integer 'arrayen vi skal søke i

    Private Sub butFind_Click(ByVal sender As System.Object, _
            ByVal e As System.EventArgs) Handles butFind.Click
        Dim lav As Integer = 0 'laveste postnr
        Dim høy As Integer = 100 'høyeste postnr
        Dim prøv As Integer 'indeksen for den posten vi prøver
        Dim søk As Integer = CInt(txtSøk.Text) 'tallet vi søker
        Do While høy >= lav
            prøv = (lav + høy) \ 2
            If søk < tall(prøv) Then 'er evt. i nedre halvdel
                høy = prøv - 1
            ElseIf søk > tall(prøv) Then 'er evt. i øvre halvdel
                lav = prøv + 1
            ElseIf søk = tall(prøv) Then 'er funnet
                MsgBox(CStr(søk) & " er funnet som post nr " & CStr(prøv))
                Exit Sub
            End If
        Loop 'Kommer vi ut her, finnes ikke posten
        MsgBox(CStr(søk) & " finnes ikke")
    End Sub

    Private Sub Form1_Load(ByVal sender As System.Object, _
            ByVal e As System.EventArgs) Handles MyBase.Load
        'Fyller arrayen med 0, 2, 4...200 og den er sortert stigende
        For i As Integer = 0 To 100
            tall(i) = i * 2
        Next
        txtSøk.Focus()
    End Sub
End Class
```

Eksempel:



Vi leter etter tallet 15:

A: Verdien må evt. finnes mellom element 0 og element 7. Prøver først midt i, i element $\frac{0+7}{2} \approx 3$

B: 13 er for lite. Prøver da midt mellom element 3 og element 7, dvs element 5.

C: 17 er for stort, prøver da midt mellom element 5 og element 3, dvs element 4 som er 14. Vi kan nå vite at tallet 15 ikke finnes i arrayen.

(Her gikk det med tre sammenlikninger, men det kan bli fire, f.eks. hvis også 17 er for lite, da må vi kanskje prøve både element 6 og element 7.)

Høgskolen i Buskerud
Postboks 235
3603 Kongsberg
Telefon: 32 86 95 00

www.hibu.no

ISSN 1893-2398 (online)

HØGSKOLEN
i Buskerud