SQL99s objektmodel

teori og praksis

Speciale for

Rasmus Borch

Datalogisk Institut, Københavns Universitet, Juni 2000 - Februar 2001

Abstract

Specialet har til formål at analysere og vurdere SQL99s objektmodel, både set fra en teoretisk og en praktisk side. Objektmodellen omfatter alle egenskaber, som kan forventes af en objektorienteret model på nær indkapsling. Dog kan indkapsling på attributniveau simuleres ved hjælp af rettigheder og på tabelniveau ved hjælp af views. De øvrige objektorienterede egenskaber er ikke væsentligt mere komplekse at bruge end de tilsvarende løsninger i SQL92, dog kan OIDer give problemer med ad hoc forespørgsler og søgehastighed. Oracle og DB2 er meget langt med deres implementation af standarden, men Informix er en smule bagud, grundet dens mangel på identitetsbaserede referencer. DB2 ligger tættest på standarden syntaksmæssigt. Når objektorienterede databaser skal designes er det vigtigt, at man gør sig forskellen klar mellem referencer og nedarvning. Sidstnævnte kan give problemer med typemigrering. Objektmodellen vil formentligt have størst brugbarhed ved implementation af nye systemer. Det anbefales at være påpasselig med brugen af de objektorienterede egenskaber, hvis man ikke tidligere har haft erfaring med disse.

Indholdsfortegnelse

1 Forord

Denne rapport er et speciale på Datalogisk Institut, Københavns Universitet (DIKU). Den er udarbejdet af Rasmus Borch i perioden juni 2000 til februar 2001 under vejledning af Troels Andreasen, Roskilde Universitetscenter og Jyrki Katajainen.

Specialets titel er ”SQL99s objektmodel – teori og praksis”. Teoridelen består af en analyse af objektmodellen i SQL99, som beskriver standarden for objekt-relationelle databasesystemer. I praktikdelen bliver tre implementationer af objekt-relationelle databasesystemer analyseret. Konkret drejer det sig om Oracle 9i, Informix Universal Server 9.1 og IBM DB2 Universal Database 7. Afslutningsvis overvejes kort de nye faciliteters eventuelle styrker og svagheder i forbindelse med objektorienteret systemudvikling.

Specialets opbygning er ændret en smule i forhold til arbejdsbeskrivelsen. Dette skyldes, at visse emner som f.eks. multipel klassifikation viste sig at have en mere naturlig placering i et andet afsnit. Specialet omfatter alle emner, som er nævnt i arbejdsbeskrivelsen.

1.1 Forudsætninger

Specialets indhold forudsætter et godt kendskab til relationelle databaser og databasesystemer, herunder SQL92, samt en god forståelse af begrebet objektorientering f.eks. i forbindelse med et objektorienteret programmeringssprog. Erfaring med systemudvikling vil også være en fordel. Til kode eksemplerne vil programmeringssprogene C++ og Java fortrins blive benyttet.

1.2 Typografi

Specialet er skrevet med Times New Roman, 12 punkt. Engelske betegnelser skrives med *kursiv*. Hvis ord eller sætninger særligt bør understreges, markeres dette med understregning.

Første gang et nyt centralt begreb introduceres i teksten, skrives det med **fed** skrift. Citater eller kode, som skal fremhæves, skrives også med **fed** skrift. Kode og reserverede ord skrives med `Courier New`. I kode tilstræbes det at navngive variable med småt, reserverede ord med STORT og brugerdefinerede typer med Stort begyndelsesbogstav. Dette kan ikke gøres helt generelt, eftersom visse programmeringssprog, som f.eks. Java og C++, kræver, at reserverede ord skrives med småt.

Når navnet på en brugerdefineret type, variabel eller værdi forekommer i teksten vil det være skrevet med Arial.

1.3 Litteraturhenvisninger

To former for henvisninger vil blive brugt. I begge tilfælde symboliserer #### udgivelsesåret.

- Generel henvisning til relevant tekst vil være på formen ”[Forfatter(ere), ####]”. Den præcise beskrivelse af litteraturen kan findes i afsnit 11 ”Litteratur”, som er sorteret efter forfatternavn.
- Omtale af forfatter med henvisning til relevant tekst vil være på formen ”…Forfatter(ere) [####]…”. Henvisningen kan igen findes i afsnit 11 ”Litteratur” under formen ”[Forfatter(ere), ####]”.

I visse tilfælde vil også et afsnits- eller delnummer blive inkluderet i henvisningen.

2 Indledning

Objektorientering har vundet stort indpas indenfor mange datalogiske discipliner, såvel analyse og design som i programmering. Databasesystemerne har stærke relationer til både design og programmering. Det er derfor nærliggende også at gøre databasesystemerne objektorienterede. Dette kan grundlæggende gøres på to måder:

1. Tage udgangspunkt i de eksisterende relationelle systemer og tilføje dem objektorienterede egenskaber. Disse kaldes for **objekt-relationelle databasesystemer**.
2. Tage udgangspunkt i den ”rene” objektorienterede model og skabe helt nye systemer herudfra. Disse kaldes ofte for **objekt-orienterede databasesystemer**[1].

Begge retninger har mange tilhængere, og foreløbigt er det svært at spå om hvilken, der vil blive den dominerende på sigt. De objekt-relationelle databasesystemer har en stor fordel, idet producenterne allerede har størstedelen af markedet med deres relationelle systemer. De objekt-relationelle egenskaber vil derfor automatisk blive stillet til rådighed for mange systemudviklere efterhånden, som systemerne opgraderes.

Det bør også nævnes, at der findes en tredje gruppe, som ikke mener, at objektorientering nødvendigvis er vejen frem for databasesystemerne.

SQL (*Structured Query Language*) er det sprog, som er det primære for relationelle databasesystemer. **SQL99** er den nyeste standard for sproget. Før den blev vedtaget som standard, havde den arbejdsbetegnelsen **SQL3**. Den er afløser for den ældre SQL92 standard. SQL99 indholder en lang række udvidelser og forbedringer i forhold til SQL92. Én af disse er tilføjelsen af en objektmodel. Det er nok en af de mere centrale udvidelser i forhold til SQL92, men absolut ikke den eneste. Standarderne har i mange år haltet efter databasesystemerne, men SQL99 gør meget for at bringe den op på siden af og i visse tilfælde foran den nuværende teknologi. Dog indeholder den flere aspekter, som allerede har været hverdag for databaseudviklere og -administratorer igennem længere tid, f.eks. brugerroller og udløsere (*triggers*)[2].

Betegnelsen SQL99 vil fremover blive brugt, når der henvises til ISO/IEC 9075:1999 [ISO, 1999]. Betegnelsen SQL:1999 benyttes også i forbindelse med standarden, men af hensyn til symmetrien vælges SQL99. SQL3 vil henvise til standarden før den blev vedtaget. SQL3 har løbende været under udvikling. Hvis der i specialet nævnes, at en egenskab var ”med i SQL3”, bør dette læses som ”med i en eller flere udgaver af

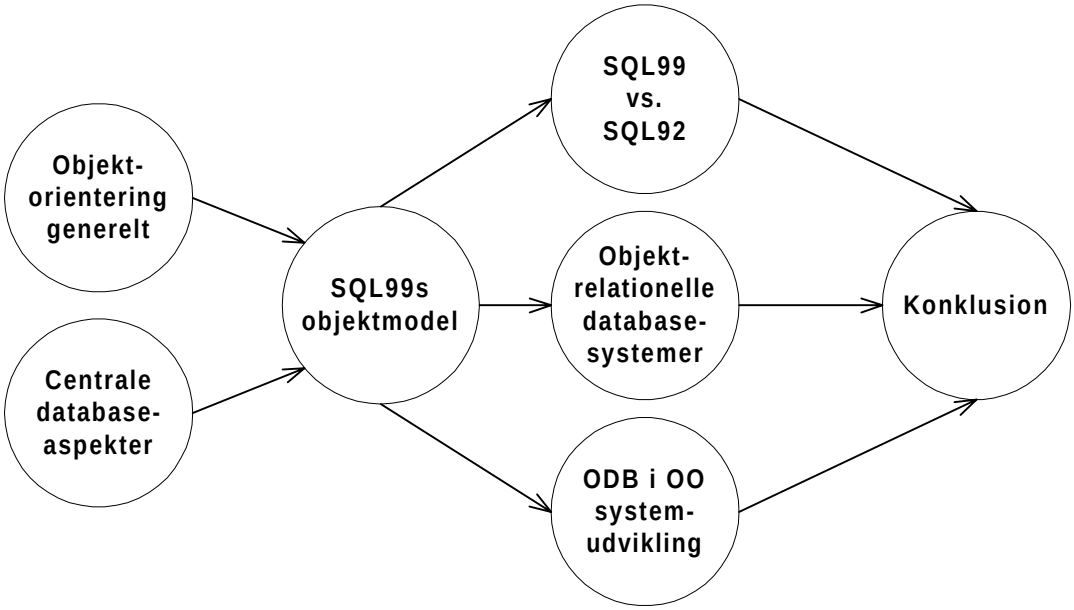
[1] Betegnelsen ”objektorienterede databaser” bruges også som fællesnævner for begge paradigmer. For at skelne vil jeg benytte begrebet ”rene” objektorienterede databaser.

[2] Hændelsesstyrede metodekald i databasen.

SQL3". Beskrivelserne af SQL3 er hentet fra Ullman og Widom [1997] og Elmasri [2000].

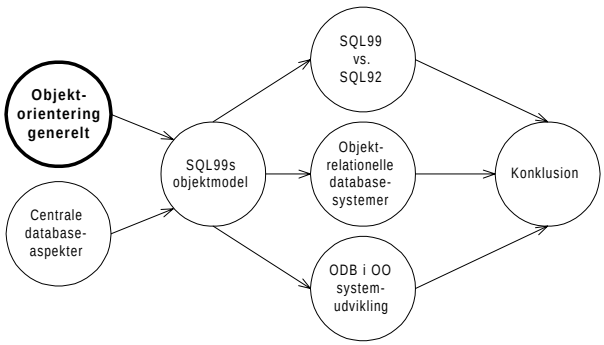
I specialet vil jeg udelukkende koncentrere mig om objektmodellen. De øvrige ændringer og tilføjelser vil ikke blive behandlet. Det kan diskuteres hvilke egenskaber, som hører ind under objektmodellen. I næste afsnit vil jeg definere mere præcist hvilke, jeg mener, der hører ind under en objektmodel.

Specialets opbygning kan illustreres på nedenstående måde:



Figuren skal læses fra venstre mod højre. En cirkel repræsenterer ét hovedafsnit i specialet. Pilene henviser til de øvrige afsnit som i høj grad afhænger af afsnittet. Således afhænger afsnittet "SQL99s objektmodel" i høj grad af "Objektorientering generelt" og "Centrale databaseaspekter". Konklusionen afhænger også i høj grad af afsnittet "SQL99s objektmodel", men dette sker for en stor del gennem de tre mellemliggende afsnit "SQL99 vs. SQL92", "Objekt-relationelle databasesystemer" og "ODB i OO systemudvikling".

3 Objektorientering generelt



Først vil jeg præcisere, hvad jeg mener med en objektmodel. Som navnet antyder er en **objektmodel** en model for objektorientering. En **model** kan betragtes som en teoretisk, begrebsmæssig konstruktion. Den benyttes ofte til at modellere et system. Det kan være et konkret system, som eksisterer i den virkelige verden, eller et teoretisk system. I dette tilfælde er der tale om et teoretisk system, nemlig det objektorienterede paradigme. Det er vigtigt at skelne mellem selve modellen og dens **implementation**. En implementation er et fysisk system, som simulerer en model.

> Eksempel:
> En objektmodel er en teoretisk, begrebsmæssig konstruktion for det objektorienterede paradigme. Modellen kan have en eller flere implementationer i form af f.eks. objektorienterede programmeringssprog og objektorienterede databaser.

Objektorienteret modellering betyder, at **objekter** er den centrale byggesten i modelleringen. Til sammenligning har E-R modellen entiteter og relationer som sine centrale elementer. Objektorientering er ikke noget nyt begreb, heller ikke i databasesammenhæng. Jasmine, et rent objektorienteret databasesystem, var på tegnebordet allerede i midt 80'erne. Derfor kan man undre sig over, at begrebet ikke er bedre forankret i teorien, end det er. Hvis man siger "det er objektorienteret", så har de fleste en mere eller mindre klar opfattelse af, hvad det betyder. Problemet er blot, at denne opfattelse kan variere fra person til person. Det, som er objektorienteret, for den ene, er ikke nødvendigvis objektorienteret for den anden. Denne uenighed kan forårsages af manglende kendskab og erfaring med objektorientering. Men ser man bort fra den årsag, så kan uenigheden også skyldes forskellige grundlæggende ideologiske forestillinger.

For at undgå eventuelle misforståelser og uklarheder vil jeg i dette afsnit beskrive, hvilke egenskaber, som jeg mener, en model for objektorientering bør omfatte. Eksempler og argumenter vil i en vis udstrækning trække på de objektorienterede modeller i programmeringssprog. Dette skyldes, at modellerne generelt er accepterede som værende "objektorienterede". De er derfor en god referenceramme til at

præcisere, hvad objektorientering omfatter. Den opfattelse, jeg har fået, af objektorientering er hovedsageligt baseret på de objektorienterede modeller i programmeringssprogene C++, Java og Delphi. I forbindelse med objektorienterede databaser, findes et forslag fra folkene bag **ODMG** [Atkinson, 1993]. Dette forslag beskriver et objektorienteret paradigme for databaser. Forslaget ligger meget tæt op ad den klassiske objektmodel, som bliver præsenteret i dette afsnit.

Afsnittet er opdelt i to afsnit. Først beskrives og begrundes den valgte terminologi. Derefter gennemgås den objektmodel, som jeg betragter som værende den klassiske objektmodel.

3.1 Terminologi

Et af problemerne med objektorientering er, at der ikke er en fælles terminologi at gå ud fra. Derfor vil jeg så vidt muligt forsøge at undgå de uklare begreber, som ofte benyttes i objektorienteret sammenhæng. Den valgte terminologi, som resten af specialet vil benytte, vil blive beskrevet og begrundet i det efterfølgende. Udvalgte begreber er samlet i appendix B.

Det er en helt grundlæggende datalogisk færdighed at kunne skelne mellem værdi, variabel og type. Det virker umiddelbart enkelt, men selvom man har arbejdet med datalogi i adskillige år, kan det ske, at man blander rundt på dem. Det er derfor heller ikke overraskende, at det samme gør sig gældende indenfor objektorientering; dog i større udstrækning. Jeg vil i dette afsnit definere begreberne værdi, operation, type, variabel og objekt.

3.1.1 Værdi

Indledningsvis er det vigtigt at skelne mellem **værdier** og **indkodninger**. Indkodninger repræsenterer værdier. En værdi er svær at definere præcist, men det kan beskrives på følgende måde:

> En værdi er et unikt element, som er uafhængigt af tid og rum.

Hvis to værdier er lig med hinanden[3], så er det den samme værdi. Ligeledes ændres en værdi aldrig. Hvis den gjorde det, ville den være en anden værdi.

[3] Det er ikke helt enkelt at definere ”lig med” operatoren mellem to værdier præcist. Vi har alle en fornemmelse af dens virkemåde, men hvordan den defineres er svært. Det er derimod langt nemmere at sammenligne to indkodninger af værdier.

Eksempel:
Værdien ”fire” er altid den samme. Hvis den ændres til f.eks. ”fem”, så er der tale om en ny værdi. Værdien ”fire” kan indkodes på et utal af måder, f.eks. 4, 100_2, fire og 2+2.

Af hensyn til læsbarheden, vil jeg fremover benytte ordet ”værdi” for både selve værdien og dens indkodning. I de tilfælde, hvor det er essentielt at skelne, vil jeg eksplicit skrive ”indkodning”.

Værdier kan være sammensatte. Det vil sige, at de er sammensat af andre værdier. En værdi kunne f.eks. være ordet ”kat”. Det er sammensat af værdierne ”k”, ”a” og ”t”.

En værdi alene giver ikke meget mening. Det er nødvendigt også at have en betydning for værdien. Denne fremkommer som en naturlig del af modelleringsprocessen.

Eksempel:
Hvis man f.eks. ønsker at modellere en persons vægt, kan dette gøres med et heltal, som indikerer vægten i kilo. Min vægt er 94 kilo, så i en implementation af modellen vil værdien ”94” have betydningen ”Vægten for Rasmus Borch”. Er implementationen foretaget på en binær computer, vil den formentligt blive indkodet som 1011110.

3.1.2 Operation

En operation er nært beslægtet med en matematisk funktion. Den foretager en mapning fra én mængde af værdier til en anden mængde af værdier. Dog adskiller den sig på følgende punkter.

- En operation kan have sideeffekter. Det vil sige, at resultatet af operationen ikke er begrænset til den returnerede værdi.
- Som et resultat af ovenstående kan en operation returnere mere end én værdi.
- En operation behøver ikke at returnere nogen værdier.

En operation kan defineres på følgende måde.

> A og B er to vilkårlige mængder af værdier (inklusiv den tomme mængde). En operation, F, er en mapning af værdier fra A til B:
>
> $F(A) \rightarrow B$
>
> F kan producere sideffekter. Det vil sige, at den kan påvirke værdier, som ligger udenfor A og B.

Hvis både A og B er den tomme mængde, har operationen udelukkende sideeffekter.

3.1.3 Type

En af de udbredte definitioner på ”type” er følgende:

> En type er en mængde af værdier.

Ovenstående definition kan udvides således, at en type også omfatter en mængde af **operationer**, som opererer på værdierne i typen.

> En type er en mængde af værdier, tilknyttet en mængde af operationer.

Denne form for operationer kan enten producere værdier af typen eller tage argumenter af typen (eller begge dele). Sådanne operationer behøver ikke at være lukkede med hensyn til typen.

> Eksempel:
> Man kan betragte alle positive heltal inkl. 0, $\mathbf{N_0}$, som værende en type. Til denne type kan knyttes oprationen $\mathbf{N_0}$-Addition, som kan tage to argumenter af typen $\mathbf{N_0}$ og generere en ny værdi af typen $\mathbf{N_0}$. Ligeledes kan $\mathbf{N_0}$-Subtraktion tilknyttes. Denne tager ligeledes to argumenter af typen $\mathbf{N_0}$, men genererer en ny værdi af typen **Z** (mængden af alle heltal). Typen kunne også tilknyttes operationen $\mathbf{N_0}$-Minimum, som ikke tager nogen argumenter, men returnerer den mindste værdi i $\mathbf{N_0}$.

Det lidt uklare begreb ”tilknyttet” er brugt bevidst. Hvorledes en operation tilknyttes en type er implementationsafhængigt. Det er normalt for objektorienterede sprog, at deres syntaks eksplicit binder operationer til en type, men der er intet til hinder for, at samtlige operationer, som benyttede et argument eller producerede en værdi af typen, automatisk blev tilknyttet typen. En operation kan i princippet godt være tilknyttet flere typer på én gang. Antag at `T` og `U` er to forskellige typer. En operation, som tager et argument af typen `T` og returnerer en værdi af type `U`, kunne både være tilknyttet `T` og `U`.

Næsten samme definition kommer til udtryk på en lidt anderledes måde i Gammas bog ”Design Patterns” [1995], som definerer en ”type” således:

> ”A **type** is a name used to denote a particular interface. We speak of an object as having the type ”Window” if it accepts all requests for the operations defined in the interface named ”Window”. An object may have many types, and widely different objects can share a type. …” (fed skrifttype fra originalteksten)
>
> [Gamma, 1995]

Det er måske ikke så heldigt, at en type defineres, som værende et navn, men den grundlæggende mening er, at typen identificeres ved dens operationer. Værdierne i typen er så de værdier, som operationerne enten kan tage som argument eller kan

returnere. Man kan sige, at begrebet er defineret bagvendt. Hvis to mængder af værdier har samme operationer, så må det være den samme type, men det kan man nu ikke konkludere generelt. Det forholder sig således i mange typer, men det præsenterede typebegreb skal forstås helt generelt.

> Eksempel:
> Type A er mængden af værdierne {bil, både, hus}.
> Type B er mængden af værdierne {skat,moms,afgift}.
> Til begge mængder knyttes den samme operation, $\in$, som afgør om et vilkårligt element er med i mængden. Selvom de to typer har samme operationer, kan de ikke betragtes som værende den samme type.

Det forhold, at typer og operationer er knyttet sammen, betragtes ofte som en grundlæggende objektorienteret egenskab ved en model. Dog har ikke-objektorienterede programmeringssprog som f.eks. C og Pascal også en tæt knytning af typer og operationer i forbindelse med de prædefinerede typer. Hvis et sprog skal kvalificere sig til betegnelsen "objektorienteret", så må det være et krav at nye typer og operationer også kan knyttes sammen.

3.1.4 Variabel

Jeg vil benytte følgende definition af ordet **variabel**:

> En variabel kan betragtes som en pladsholder, hvor en værdi kan opbevares. Typen på variablen afgør hvilke værdier, som den kan opbevare. En variabel er tilknyttet en adresse eller et navn, som identificerer den.

Det, som tydeligst adskiller en variabel fra en værdi, er, at en variabel har en adresse. Variable tildeles ofte et navn som f.eks. x. Hvilken betydning, som x har, afhænger af konteksten. Den kan enten angive selve variablen eller den værdi, som variablen indeholder.

> Eksempel:
> Betragt følgende programlinier skrevet i C:

```
1: int x;
2: int y;
3: x = 1;
4: y = x;
```

> I linie 3 angiver "x" adressen på x og i linie 4 angiver "x" indholdet af x.

Der er en væsentlig forskel mellem **datalogiske** og **matematiske variable**. En datalogisk variabel kan variere, en matematisk variabel er konstant. Når en ubekendt, f.eks. X, optræder i et ligningssystem, vil alle forekomster af X angive den samme værdi. X kan ikke ændre værdi på noget tidspunkt. Dette forhold er f.eks. anvendt i funktionsprogrammeringssprog. Her findes der ikke variable i datalogisk forstand.

Kun konstanter kan angives [Bird & Wadler, 1988]. Alle henvisninger til ”variable” i dette speciale bør læses som ”datalogiske variable”.

3.1.5 Objekt

Ordet **objekt** er formentligt det mest brugte og mest uklare begreb i det objektorienterede paradigme. Det bliver benyttet både i betydningen type, værdi og variabel.

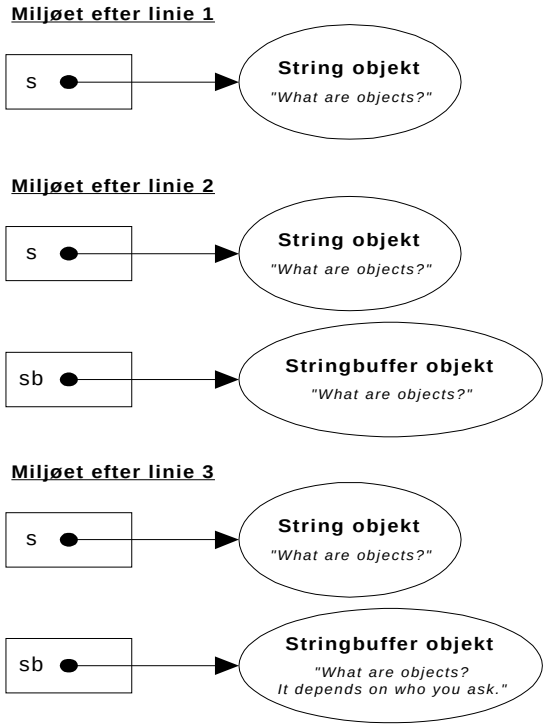
Nedenstående Java eksempel vil blive brugt til at illustrere diskussionen.

Eksempel:

```
1: String s = new String(”What are objects?”);
2: StringBuffer sb = new StringBuffer(s);
3: sb.append(”\nIt depends on who you ask.”);
```

Som en indledende bemærkning kan det nævnes at både typen String og typen StringBuffer er subtyper af den globale supertype i Java ved navn Object. Det sker altså at ordet ”objekt” benyttes som betegnelse for en type. Argumentet for, at det også bruges med betydningerne værdi og variabel, kræver lidt grundigere forklaring.

Selvom Java ikke umiddelbart ser ud til at indeholde referencer, er både s og sb i virkeligheden begge referencer til objekter af typerne String og StringBuffer henholdsvis. Miljøet efter hvert af de tre linier er illustreret nedenfor.



Som vist kan objekter af typen StringBuffer ændre den værdi, som den indeholder. Den kan så at sige skifte tilstand. Den slags objekter kaldes for **mutérbare objekter**

(*mutable objects*). Man kan derfor betragte den som en variabel. Det vil sige en pladsholder for en værdi. Selvom værdien ændres, er StringBuffer objektet det samme (det har samme adresse).

Objekter af typen String kan ikke ændre deres værdi. De er **ikke-mutérbare objekter** (*immutable objects*). Den eneste måde at ændre indholdet på er ved at nedlægge objektet og skabe et nyt. Derved er der reelt tale om en nyt objekt, da det ikke er sikret at have samme adresse som det forrige. De kan derfor betragtes som værende værdier.

Problemet med ovenstående er, at i C++ og Java har begge slags objekter adresser – noget som værdier ikke har (ifølge afsnit 3.1.1 ”Værdi”). Jeg kan forestille mig tre forskellige måder at løse dette problem på:

1. Begge slags objekter er variable. Nogle variable kan ikke ændre deres tilstand med henblik på at ”ligne” værdier. Det, som gør dem ikke-mutérbare, er, at variablens type ikke har tilknyttet nogen operationer, som kan ændre den lagrede værdi.
2. Adressen på de ikke-mutérbare objekter svarer til den adresse, som f.eks. også er associeret med en konstant streng i C. En konstant streng i C kan angives alle steder, hvor et argument af typen *char** er forventet.

   Eksempel:
   ```
   strcpy(s,”En konstant streng som (char*) argument”);
   ```

   Adressen er blot et resultat af den indkodning, som er foretaget af værdien, så den må stadig betragtes, som værende en værdi.
3. Objekter af begge slags svarer ikke direkte til variable og værdier, men er en konstruktionsmæssig overbygning til dem begge. De kan således bestå af en eller flere værdier og variable. De objekter, som udelukkende består af værdier, kaldes ikke-mutérbare objekter.

Hvilken forklaring man foretrækker er nok mest en smagssag. Personligt bryder jeg mig ikke om forklaring nr. 3, da den ikke kan afmystificere begrebet, men blot understrege at objekter er noget nyt og mærkeligt, som er svært at forstå. Det, mener jeg, ikke er tilfældet. Forklaring nr. 1 holder sig strengt til det forhold, at kun variable har adresser. Men det giver problemer med f.eks. de konstante C strenge som beskrevet i forklaring nr. 2. De er tydeligvis ikke variable, men har en adresse alligevel. Forklaring nr. 2 bevarer det pæne forhold mellem mutérbare/ikke-mutérbare objekter og variable/værdier. At de fleste implementationer så vælger at indkode alle objekter med adresser, behøver ikke at rokke ved det grundlæggende fundament. Man kan sige, at en værdi aldrig har en adresse, men dens indkodning kan have.

Min definition på objekter er derfor følgende:

> Alle objekter er enten mutérbare eller ikke-mutérbare. Mutérbare objekter svarer til variable. Ikke-mutérbare objekter svarer til værdier.

Fordi begrebet "objekt" ikke har en alment accepteret definition, giver det risiko for misforståelser eller blot generel uklarhed. Jeg vil derfor fremover undgå at benytte det med undtagelse af mere generelle vendinger som "objektorienteret" og "objektmodel". Derimod vil jeg benytte begreberne type, værdi og variabel, som defineret tidligere.

3.1.6 Type vs. klasse

Forskellen på typer og **klasser** er et af de centrale punker i objektorientering, hvor der hersker delte meninger om den præcise betydning. Date og Darwen [1998] siger, at der ikke er nogen forskel; klasser er lig med typer. Ullman og Widom [1997] skelner mellem typer uden operationer og typer med operationer. Hvis der ikke er nogen tilknyttede operationer, så er der tale om en "type". Hvis der er tilknyttet operationer, så er der tale om en "klasse". Ikke alle definitioner er lige præcise. Nedenstående er taget fra et tidligt forslag til en standard for objektorienterede databasesystemer [Atkinson, 1993].

> "The notion of class is different from that of type. Its specification is the same as that of a type, but it is more of a run-time notion. It contains two aspects: an object factory and an object warehouse. The object factory can be used to create new objects, by performing the operation new on the class, or by cloning some prototype object representative of the class. The object warehouse means that attached to the class is its extension, i.e., the set of objects that are instances of the class."
>
> [Atkinson, 1993]

Denne skelnen minder en del om Javas forskel på klasser og primitive typer. De primitive typer kan bruges ad hoc, hvorimod værdier af de mere komplekse typer og brugerdefinerede typer først skal oprettes på kørselstidspunktet. Desuden er den lavet med objektorienterede databasesystemer for øje. Derfor introduceres f.eks. typens *extension*, som indeholder samtlige værdier af typen, som er oprettet på et givent tidspunkt.

Jeg vil fremover holde mig til Date og Darwens definition, som ikke skelner mellem klasser og typer. Dette skyldes hovedsageligt, at jeg ikke ser nogen god grund til at skelne mellem disse to koncepter.

3.1.7 Operationer vs. metoder

Operationer associeret med en type kaldes også for metoder, funktioner, medlemsfunktioner eller håndtag (*handles*). Ordet ”operationer” er valgt for at illustrere, at typer, som f.eks. heltal med tilhørende aritmetiske operationer, også er objektorienterede typer. De er blot så indarbejdede, at de i visse programmeringssprog som Java har fået særstatus som primitive typer. Her fremstår programmeringssprog som C++ forbilledligt, idet overstyring af operationer (*operator overload*) giver mulighed for at udvide sprogets syntaks med nye typer.

Der er ikke den store risiko for tvetydighed indenfor dette begreb, men for at bevare konsekvensen vil jeg holde mig til ordet ”operation”.

3.2 Den klassiske objektmodel

Dette afsnit beskriver **den klassiske objektmodel**. Denne er nok den mest gængse opfattelse af, hvad objektorientering er. Derfor vil den blive omtalt som den ”klassiske”. Som sagt er afgrænsningen svag, og det kan derfor variere fra person til person, hvad vedkommende mener, at modellen skal omfatte. Jeg vil forsøge kun at inkludere de helt grundlæggende egenskaber.

3.2.1 Subtyper

Modellen har et **typehierarki**, som sin base. Hvis man har en type, T, så kan en anden type, S, betragtes som værende en **subtype** af T. Dette kræver, at dens mængde af mulige værdier er en delmængde af Ts mulige værdier, samt at den understøtter samtlige operationer, som T understøtter. Den må gerne understøtte operationer, som T ikke understøtter.

> Eksempel:
> En subtype, $\mathbf{N}_{16\text{-bit}}$, til typen $\mathbf{N}_0$ kunne f.eks. være heltallene mellem 0 og 65535 med operationerne Addition og Subtraktion[4]. Mængden af mulige værdier er en ægte delmængde af $\mathbf{N}_0$s og den indeholder samtlige operationer, som $\mathbf{N}_0$ typen gør.

Lidt mere formelt, kan man udtrykke det på følgende måde:

- V(T) er mængden af mulige værdier i typen T
- F(T) er mængden af operationer på T
- S << T angiver at S er en subtype af T

$S << T \Leftrightarrow V(T) \supseteq V(S) \wedge F(T) \subseteq F(S)$.

[4] Som beskrevet behøver operationerne ikke at være lukkede med hensyn til typen (afsnit 3.1.6 ”Type vs. klasse”).

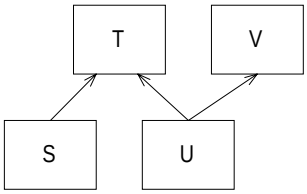
Det er værd at reflektere lidt over ovenstående definition. Den siger, at en type kan være sin egen subtype. Dette sker i tilfældet hvor V(T) = V(S) og F(T) = F(S). Men kan man betragte enhver type som værende dens egen subtype? Spørgsmålet er en smule filosofisk og er derfor svært at diskutere, men grundlæggende er det bare et definitionsspørgsmål. I den ovenstående definition er enhver type en subtype af sig selv.

Det faktum, at antallet af operationer rent faktisk kan stige for subtyper, kan virke en smule forvirrende. Det bryder med sub/super begreberne, idet subtypen på dette område kan være mere omfattende end supertypen. I stedet for at kalde S for en subtype af T kunne den betegnes som en **afledt type** af T. Jeg vil dog holde mig til betegnelserne sub- og supertyper, da jeg mener, at de er så generelt udbredte, at misforståelser ikke vil opstå.

3.2.2 Typehierarkier

En type kan have flere subtyper, og disse kan igen have flere subtyper. Hver subtype kan ligeledes have flere supertyper. Derved fremkommer der en acyklisk orienteret graf, hvor hver knude svarer til en type, og kanterne har orientering fra subtypen til supertypen. Denne graf udgør typehierarkiet.

Eksempel:
Følgende typehierarki indeholder fire typer, S, T, U og V. S har T som sin supertype og U har både T og V som sine supertyper.



Man kan sagtens forestille sig adskillige distinkte typehierarkier, men i visse programmeringssprog vil der være én supertype, som alle subtyper vil have som deres mest generelle supertype[5]. I f.eks. Java hedder denne globale supertype Object. Visse programmeringssprog (f.eks. Java og Delphi) lægger en yderligere begrænsning på subtyperne, idet en given type højst kan have én supertype. Dette gøres ofte ved at implementere subtyperne med ”enkelt nedarvning”. Det mere generelle typehierarki, som ikke sætter begrænsninger på antallet af supertyper for en subtype, implementeres som regel med ”multipel nedarvning”.

[5] Ved en ”mest generel” supertype forstås en type, som ikke har nogen supertyper.

I dette speciale vil der ikke blive skelnet mellem enkelt og multipel nedarvning, med henblik på at afgøre om et system kan kaldes objektorienteret eller ej. Det vigtige er, at subtype begrebet er understøttet og giver mulighed for polymorfi.

3.2.3 Polymorfi

Polymorfi benyttes i forskellige sammenhænge. Disse er nært beslægtede, men alligevel ikke helt ens. For at skelne mellem dem vil de blive omtalt som henholdsvis polymorfe operationer og polymorfe typer.

Polymorfe operationer er operationer, som er delvist uafhængige af argumenternes typer. F.eks. vil det være muligt at skrive en liste-sammenkædnings-operation uden at tage højde for typen af elementerne i listen. F.eks. angives elementer blot som havende typen α. Denne kan så være en vilkårlig type, når operationen kaldes med en given liste. Man kan betragte α, som værende en **typevariabel**. Det vil sige en variabel, hvis type er en mængde af typer. Denne form for polymorfi findes blandt andet i C++ (ved hjælp af *templates*), men optræder oftest i forbindelse med funktionsprogrammeringssprog (som f.eks. ML og Miranda). I forbindelse med databasesystemer, kan aggregeringsfunktionen `COUNT()` nævnes. Denne tæller antallet af tupler i en relation, uden at være afhængig af deres typer.

I forbindelse med objektorientering, er det ofte polymorfe typer man henviser til. Alle typer i et typehierarki er polymorfe. Det vil sige, at de altid kan opfattes som værende en af deres supertyper. Hvis `S` er en supertype af `T`, så er det muligt at benytte værdier af typen `S` i alle udtryk, hvor en værdi af typen `T` er forventet. Dette kan lade sig gøre, eftersom `S` per definition, som minimum, skal indeholde de samme operationer, som `T` gør.

> Eksempel:
> I tilfældet med de to heltalstyper, $\mathbf{N_0}$ og $\mathbf{N_{16\text{-}bit}}$, vil det altid være muligt at benytte en værdi af typen $\mathbf{N_{16\text{-}bit}}$ (subtypen) i et vilkårligt udtryk, hvor en værdi af typen $\mathbf{N_0}$ (supertypen) ville kunne forekomme.

Visse sprog, som f.eks. Java, har en global supertype, `Object`, som næsten alle typer nedarver fra[6]. Selvom Java ikke understøtter polymorfe operationer direkte, kan det simuleres ved at benytte argumenter af typen `Object`. Disse kan så være vilkårlige subtyper i Java.

På samme måde kan man betragte de polymorfe typer, f.eks. α, som værende en global supertype. Dog adskiller de sig lidt fra `Object` i Java, idet `Object` kan være en vilkårlig type. α er én specifik type, som kan være en vilkårlig type. Forskellen illustreres bedst med et eksempel:

[6] Primitive typer som heltal, tegn osv. er undtaget

Eksempel:

Polymorf operation i Java:

```
int foobar(Object a, Object b, Object c) { /* operation */ };
```

Polymorf operation i ML:

```
function foobar(a: α, b: α, c: β) : int = /* operation */;
```

Eksemplet viser, at i "ægte" polymorfe operationer er det muligt at angive, at to parametre (a og b) skal have samme vilkårlige type. I den simulerede polymorfi i Java er det kun muligt at angive, at en parameter kan være vilkårlig (dog ikke de primitive typer).

3.2.4 Værdier

En type, T, er en mængde af mulige værdier, tilknyttet en mængde af operationer. Hvis en værdi, v, er med i typen T, siges det, at v har typen T. Det er muligt, at værdien også er med i en eller flere subtyper af T, men grundet polymorfi kan værdien optræde alle steder, hvor en værdi af supertypen er forventet.

En værdi, v, har altid en **mest specifik type**. Det vil sige den type, T, som v er en værdi af, og hvor v ikke er en værdi af nogle af Ts subtyper. Når man generelt taler om typen af en værdi, er det som regel den mest specifikke type, der menes.

Eksempel:
Antag et typehierarki med følgende tre heltalstyper, $\mathbf{N_0}$, $\mathbf{N_{16\text{-}bit}}$ og $\mathbf{N_{8\text{-}bit}}$. Her ville den mest specifikke type for værdien 256 være $\mathbf{N_{16\text{-}bit}}$.

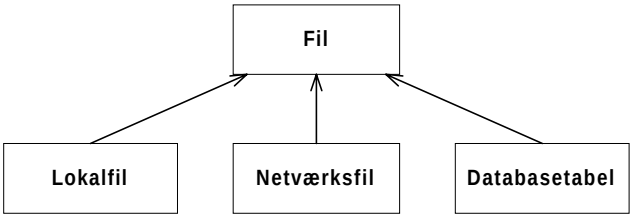
3.2.5 Abstrakte typer

En **abstrakt type** er en type, som ikke kan have nogen værdier, der har typen som sin mest specifikke. Abstrakte typer benyttes i nedarvning, hvor en række subtyper skal have en fælles supertype for at knytte dem sammen, og det ikke giver mening at have værdier af supertypen.

Abstrakte typer er ofte defineret ved, at en eller flere af de tilknyttede operationer ikke er implementeret endnu (C++), men i visse sprog angives det eksplicit (Java). Det er normalt, at abstrakte typer ikke implementerer nogen af sine operationer. Dette overlades så til subtyperne. Derved fungerer den abstrakte supertype som et slags interface. Hvis et program udelukkende benytter den abstrakte types operationer, vil enhver af dens subtyper kunne bruges af samme program. Dette er en af de grundlæggende designstrategier, som anbefales af Gamma:

> "Program to an interface, not an implementation".
>
> [Gamma, 1995]

Eksempel:
Antag følgende typehierarki af filer:



Typen Fil vil typisk være en abstrakt type. Det er nødvendigt, at angive hvordan filen skal hente sine data. Hvis applikationen generelt benytter filer af typen Fil, vil det være i stand til at benytte alle tre slags filer. Det vil endda være muligt, at tilføje nye former for filer uden at programmet skal ændres.

3.2.6 Operationer

Operationer i det objektorienterede paradigme kan grundlæggende opdeles i to grupper: **statiske** og **dynamiske**. Statiske operationer knytter sig til selve typen, og dynamiske operationer knytter sig til værdier af typen. Statiske operationer benyttes ofte til at generere nye værdier, hvor dynamiske bearbejder eksisterende værdier. Fælles for de dynamiske operationer er, at de alle har et skjult argument, nemlig den værdi som operationen er knyttet til. Dette argument betegnes ofte *this* eller *self*. På dansk vil jeg bruge udtrykket **selv-referencen**.

Eksempel:
Et kald til operationen intValue på en værdi af typen Integer i Java returnerer selve værdien af heltallet[7]. Syntaksen for udførelsen af operationen er følgende:

```
Integer syv = new Integer(7);
if (syv.intValue() == 6)
  /* denne linie udføres ikke */;
```

Selvom operationen intValue umiddelbart ikke ser ud til at have nogen argumenter, har den et skjult argument, nemlig selv-referencen til syv. Ovenstående operationskald har derfor samme betydning som følgende:

```
Integer syv = new Integer(7);
if (Integer.intValue(syv) == 6)
  /* denne linie udføres ikke */;
```

I det første eksempel er intValue en dynamisk operation, fordi den er knyttet til værdiens type (i dette tilfælde Integer). I det andet eksempel er den en statisk operation, fordi den er knyttet til selve typen Integer. Uanset hvilken type syv har, udføres den samme

[7] I Java findes der for hver primitiv type (heltal, tegn etc.) en tilsvarende omsluttende type (*wrapper*), som udvider den primitive types funktionalitet. For den primitive type int hedder den omsluttende type Integer.

operation, nemlig den som er tilknyttet typen Integer. Det er derfor, at den betegnes som værende statisk.
Som sagt er betydningen af de to eksempler den samme. Dette skyldes, at værdien syv netop har typen Integer som sin mest specifikke. Havde den i stedet været Double som i følgende eksempel, så havde betydningen ikke været den samme:

```
Double syv = new Double(7.0);
if (syv.intValue() == 6)
  /* denne linie udføres ikke */;
```

Dette skyldes, at intValue operationen, som bliver udført i dette eksempel, er den, som er tilknyttet typen Double og ikke typen Integer. Hvilken operation, som udføres, bestemmes derfor af syvs mest specifikke type. Da denne ikke nødvendigvis er kendt på oversættelsestidspunktet, skal valget af operationen derfor være dynamiskt – heraf navnet.

To slags operationer har særstatus i objektorienteret sammenhæng. Det er en **konstruktør** og en **destruktør** (*constructor* og *destructor*). Konstruktøren for en type udføres, hver gang en ny indkodning af en værdi af typen skabes. Den står for initialisering af indkodningen af værdien. Tilsvarende udføres destruktøren, hver gang en indkodning af en værdi af typen skal nedlægges. Den burde foretage en pæn oprydning af de eventuelle ressourcer, som indkodningen har brugt.

C++ eksempel:

```
string *s;
s = new string(”ABC”);     // ”string” er konstruktøren. Ny streng
                           // indeholdende ”ABC” oprettes.
delete s;      // Strengen fjernes/deallokeres af destruktøren
               // for string (~string). ”s” eksisterer stadig.
```

En destruktør er dynamisk, fordi det ikke giver mening at kalde den uden en værdi, som skal nedlægges. En konstruktør kan opfattes både som værende dynamisk og statisk. Initielt er den ikke tilknyttet nogen bestemt værdi og kan derfor betragtes som værende statisk. Dog skabes den nye værdi af konstruktøren, som den derved bliver knyttet til. I alle objektorienterede programmeringssprog har konstruktører en selvreference, som peger på den nyligt oprettede værdi. Derved kan de betragtes som værende dynamiske.

3.2.7 Indkapsling

En anden central egenskab ved den klassiske objektmodel er **indkapsling**. En type kan skjule sin interne repræsentation, således at værdier af typen kun kan behandles igennem et fast defineret interface.

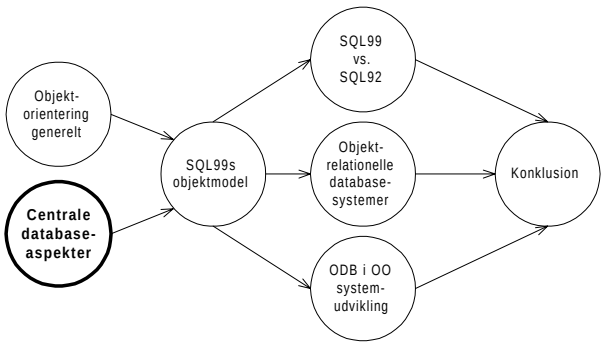
Subtyper har mulighed for at **redefinere** operationer, som er defineret af dens supertyper. Derved kan subtyper tilpasse deres operationer til deres egen interne repræsentation.

Normalt findes der to indkapslingsniveauer: **privat** (*private*) og **offentligt** (*public*). Attributter og operationer, som er erklæret private, kan kun benyttes i typens interne repræsentation og er således ikke med i det **virkefelt** (*scope*), som værdien af typen befinder sig i. Derimod er offentlige attributter og operationer også synlige udenfor typens interne virkefelt.

3.2.8 Referencer

Referencer (også kaldet pegere, hægter eller pointere) har ikke nogen direkte tilknytning til type/subtype begrebet, men begrebet er alligevel centralt for objektorientering. Objekter i programmeringssprog er ofte dynamiske, og det er derfor nødvendigt, at kunne referere dynamisk til deres lokation, eftersom den kan ændre sig med tiden. Objekter kan også kombineres til meget komplekse datastrukturer, og referencer er nødvendige for at kunne opbygge strukturerne dynamisk. Referencer er derfor en vigtig type i objektorientering.

4 Centrale databaseaspekter



Der er flere centrale aspekter ved databaser, som bør omtales nærmere. Ikke alle har nogen direkte tilknytning til det objektorienterede, men de er alligevel relevante at få med. Afsnittene er opdelt i objektorienterede og relationelle emner. Følgende emner vil blive diskuteret:

Relationelle databaseaspekter

1. Forskellen på databaser og databasesystemer.
2. Eftersom specialet vil holde sig til de tre begreber variable/værdi/type, hvorledes passer disse så med de grundlæggende konstruktioner i relationelle databaser: relationer, tupler, domæner og attributter.
3. Definitionen på en entitet.
4. Transaktioners rolle i databasesystemer

Objektorienterede databaseaspekter

5. Hvorledes påvirkes identifikationen af en given værdi, når den objektorienterede model introduceres.
6. Forskellen på objektorienteret og objekt-relationel.

4.1 Relationelle databaseaspekter

4.1.1 Databaser og databasesystemer

En **database** er en logisk samling af data, som eksisterer i længere tid. Disse er typisk relaterede, men det er ikke noget krav. En database består typisk af relationer, indekser, brugere med tilknyttede rettigheder, views med mere. Alle de metadata, som en database indeholder, som beskriver tabeller, brugere, views etc., kaldes for **databaseskemaet**. Dertil kommer de faktiske data, som databasen indeholder. Et **databasesystem** er en applikation, som administrerer en eller flere databaser [Ullman & Widom 1997].

4.1.2 Variabel, værdi, type vs. relation, tupel, domæne, attribut

Som beskrevet i forrige afsnit vil jeg undgå at benytte begreber som klasser, objekter og instanser, da betydningerne af disse ofte varierer. I stedet vil jeg benytte begreberne variable, værdier og typer, som er defineret i forrige afsnit og som generelt ikke misforstås. Det er derfor nærliggende, at beskrive hvorledes disse begreber forholder sig til de gængse begreber i relationelle databaser: relationer, tupler, domæner og attributter. Dette er kun for interessens og helhedens skyld. Alle fire nævnte begreber er præcist definerede og giver normalt ikke anledning til uklarhed.

Domæne: Et domæne svarer til en type. Domænet afgør hvilke værdier en attribut kan tildeles.

Attribut: Attributter svarer til variable. En attribut kan tildeles en værdi og virker derfor som en ”pladsholder”. Typen på attributten er domænet.

Tupel: En tupel er en værdi. Den er sammensat af værdierne i de enkelte attributter. Typen på denne værdi kaldes rækketypen (*row type*) i SQL99 og SQL3.

Relation: En relation svarer til en variabel. Den kan indeholde en mængde af tupler med en given rækketype. Relationens type er derfor mængden af delmængder (*powerset*) af mængden af samtlige tupler med den givne rækketype.

Det er vigtigt at holde sig for øje, at tupler ikke er en samling attributter, men en samling værdier. Attributterne optræder i relationen som pladsholdere for tuplernes værdier.

Når en relation skabes i en relationsdatabase i SQL92 med `CREATE TABLE`, oprettes rækketypen og relationsvariablen samtidig. Som det vil fremgå af analysen af SQL99s objektmodel, har den nye standard inkluderet mulighed for at adskille disse to. Derved bliver det muligt at oprette rækketypen for sig og derefter en eller flere relationsvariable af den type.

> **Note:** Date og Darwen [1998] introducerer begrebet *relvar*, som netop har til hensigt at understrege at relationer er variable. Dog foretager de den skelnen at selve *indeholdet* i en relvar er en relation. Det vil sige relationen er mængden af værdier i relationsvariablen. I dette speciale vil enhver henvisning til relationer hentyde til relationsvariable. Hvis der er tale om indholdet vil dette fremgå af teksten. Ordet ”tabel” vil ligeledes blive brugt som synonym for relationsvariabel.

4.1.3 Entiteter

Databaser rummer ofte repræsentationer af virkelige ting, personer, steder etc. Det er vigtigt skelne mellem disse **entiteter** og deres tilsvarende **repræsentation** i databasen. Dette svarer meget til den tidligere diskussion af forskellen på værdier og indkodninger, men entiteter og værdier svarer ikke helt til hinanden. F.eks. er værdier uafhængige af tid og rum. Værdien ”2” har til alle tider været den samme. Dette gælder ikke for entiteter, som f.eks. kunne være nærværende forfatter. Entiteter

behøver dog ikke at have en fysisk tilstædeværelse, selvom de ofte har. Et eksempel kunne være den danske lovgivning. Den er repræsenteret på papir og på elektronisk form, men selve loven er ikke fysisk og ændrer sig over tid.

Begrebet vil hovedsageligt blive brugt i afsnit 6 ”SQL99 vs. SQL92” og afsnit 8 ”ODB i OO systemudvikling”.

4.1.4 Transaktioner

Transaktioner er en central egenskab for databasesystemer og bør derfor nævnes. Grundlæggende er der fire krav til databasesystemets behandling af transaktioner. Disse betegnes ofte ACID, hvilket er et akronym for følgende egenskaber:

Atomic	Flere kommandoer kan udføres som én samlet. Hvis én fejler, fejler de alle (alt eller intet).
Consistent	Integriteten i databasen skal være vedligeholdt før og efter en transaktion udføres. Mens transaktionen udføres, kan integriteten godt kompromitteres.
Isolatable	Alt arbejde udført af en transaktion er usynligt for alle andre transaktioner indtil den afslutter med *commit*.
Durable	Arbejde afsluttet med *commit* skal overleve også selvom systemet lider et nedbrud umiddelbart efter.

Ingen af disse krav ændres ved indførelse af en objektmodel i databasesystemet. De er underlagt de samme krav og restriktioner, som hidtil har været gældende.

4.2 Objektorienterede databaseaspekter

4.2.1 Referencer i databaser

Den relationelle model tager sit udgangspunkt i mængdelæren. Det har derfor været centralt at den samme værdi (tupel) ikke kan forekomme mere end én gang i en tabel (relation). I visse tilfælde har det været ønskeligt at kunne indsnævre antallet af mulige tupler yderligere. Dette kan gøres ved at angive hvilke attributter i tuplerne, som tilsammen skal være unikke. Denne delmængde af attributter kaldes for en nøgle. En tupel kan have flere alternative nøgler[8]. Nøgler er centrale for referencerne i den relationelle model. Når en reference refererer til en tupel, angives værdierne for en af tuplens nøgler. Referencerne betegnes derfor **værdibaserede referencer**.

[8] Én af nøglerne benævnes primærnøglen. Retningslinierne for valget af primærnøglen blandt de mulige kandidater er ikke dækket af den relationelle model.

Eksempel:
Primærnøglen er dobbelt understreget.

Fornavn	Efternavn	adresse	telefonnr
Jens	Jensen	Jernbanegade	11223344
Hans	Jensen	Jernbanegade	11223344

I eksemplet må der ikke findes to personer, som har samme for- og efternavn, som bor på samme adresse.

I den klassiske objektmodel findes primærnøgler ikke. Værdier er ikke identificeret ved deres indkodning (eller dele heraf). Det er muligt at have værdier, som ligner hinanden indkodningsmæssigt, uden at de derved er den samme værdi. Det skyldes, at værdier i objektorienterede programmeringssprog i stedet identificeres ved deres placering i den aktive hukommelse. Eftersom objektmodellerne er inspireret af den virkelig verden, giver dette god mening. I vores dagligdag vil vi ikke have problemer med at håndtere to ens glas, som står på et spisebord. Selvom de er ens i næsten alle henseender, kan vi nemt skelne dem ved deres placering: ”*Det er mit glas til højre*”.

I objektorienterede databaser kan man ikke benytte værdiens placering. Det kan være, at den bliver flyttet på grund af opdatering eller omstrukturering. I stedet identificerer man værdierne med en objektidentifikator (OID), som genereres af systemet, som oprettede værdierne. Denne OID benyttes blandt andet, når en reference, skal referere til en værdi. Referencerne betegnes derfor **identitetsbaserede referencer**.

Eksempel:
Samme værdi forekommer to gange. De identificeres af OIDen, hvis format er systemspecifikt.

OID	fornavn	Efternavn	adresse	telefonnr
000000000001	Jens	Jensen	Jernbanegade	11223344
000000000002	Jens	Jensen	Jernbanegade	11223344

4.2.2 Objektorienteret vs. objekt-relationelt

Den objekt-relationelle model bygger på foreningen af den objektorienterede model og den relationelle model. Den mest grundlæggende forskel på disse to modeller er, at tupler (værdier) i den relationelle model ikke har nogen påhæftet identifikator eller adresse. De er udelukkende identificeret ved deres indhold. Derimod er værdier i den objektorienterede model identificeret ved deres OID. Disse to modeller virker umiddelbart ikke forenelige. Den objekt-relationelle model giver dog både mulighed for værdier med og uden OID og skaber derved ikke en egentlig sammensmeltning af de to forskellige modeller, men snarere en ramme som samler dem i én model. Denne ramme placerer relationen som den centrale byggesten i modellen uanset hvilken slags værdier, som man benytter (med eller uden OID).

Når man betragter implementationerne af de omtalte modeller, er billedet et andet. Til forskel fra den relationelle model, så har de fleste af dens implementationer, f.eks. Oracle, SQL Server og Sybase Adaptive Server, allerede mulighed for at sætte adresser på tupler i form af `ROWIDs`[9]. Den grundlæggende forskel på værdier med og uden OIDer er derfor ikke så tydelig i implementationerne, som den er i modellerne. Dette er opsummeret i nedenstående tabel.

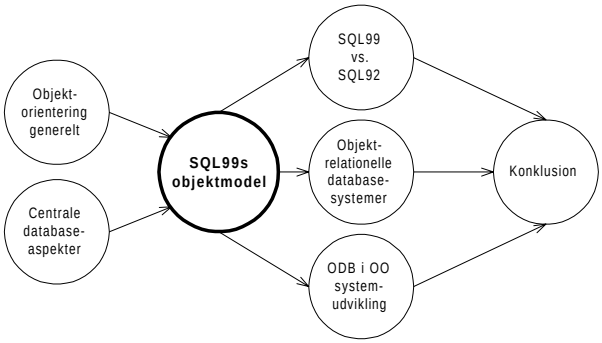
	RELATIONELT	**OBJEKT-RELATIONELT**	**OBJEKTORIENTERET**
Model	Ingen OIDer	Både med og uden OIDer	OIDer
Implementation	`ROWIDs`	`ROWIDs` og OIDer	OIDer

Som det kan ses, optræder de største forskelle på rækken ud for ”Model”. Her skal den objekt-relationelle model forene to helt forskellige modeller. Når det drejer sig om implementationerne, så ligger forskellen hovedsageligt imellem `ROWIDs` og OIDer, hvilket må siges at være noget mindre. Det er måske dette forhold som gør, at den objekt-relationelle model virker mere problematisk for teoretikere end for pragmatikere.

SQL99 sætter blandt andet standarden for den objekt-relationelle model. Denne indeholder både en relationel model og en objektorienteret model. Sidstnævnte er emnet for dette speciale. Jeg vil derfor benytte ordet objektorienteret frem for objekt-relationelt, da det ikke er hele den objekt-relationelle model, som jeg analyserer. Den relationelle model kan og bør dog ikke helt udelades og vil blive medtaget i det omfang, det er relevant.

[9] Hver række tildeles automatisk et unikt nummer som identificerer den.

5 SQL99s objektmodel



Objektmodellen i SQL99 er en af de meget centrale tilføjelser til SQL standarden. I dette afsnit vil den blive gennemgået og sammenlignet med den klassiske objektmodel, som blev præsenteret i forrige afsnit.

5.1 Indledning

Som beskrevet i indledningen tager objektmodellen i SQL99 udgangspunkt i den eksisterende teori og praksis omkring relationelle databaser. Den bygger derfor på de samme grundprincipper, hvoraf den mest centrale er relationen. De ”rene” objektorienterede databaser afskaffer relationstypen som den centrale struktur, og indfører i stedet brugerdefinerede strukturerede typer.

5.1.1 Terminologi

Standarden er skrevet på engelsk. Den introducerer nogle få nye betegnelser, men disse kan forholdsvis nemt oversættes til dansk, uden at det bliver meningsforstyrrende. For at øge specialets læsevenlighed vil teksten derfor blive holdt på dansk, så vidt det er muligt.

5.1.2 Argumentation

Nedenstående gennemgang af SQL99 er lavet på baggrund af selve SQL99 standarden fra ISO [1999]. Da denne nærmere er en lovtekst end en lærebog, er informationen ofte givet på meget komprimeret form og fyldt med referencer til andre sektioner i standarden. Ligeledes kan den bevidst være uklar på visse områder, hvis der ikke er vedtaget noget konkret på det givne område. Citater er medtaget fra standarden, hvor det skønnes nødvendigt og interessant. De øvrige steder vil der være henvisninger til de relevante sektioner i standarden.

5.1.3 Afgrænsning

Målsætningen er at beskrive, hvordan objektmodellen i SQL99 er opbygget. Eftersom modellen griber ind i andre dele af standarden, kan det være svært at afgrænse

præcist. Objektmodellen har konsekvenser i store dele af standarden, men det er ikke alle ændringer, som er lige centrale for modellens funktionalitet. Den efterfølgende gennemgang vil gå i dybden med de dele af standarden, som er centrale for objektmodellen. Specielt vil **kollektionstyper** som mængder, lister, multimængder og *arrays* ikke blive gennemgået, da de ikke er centrale elementer i objektmodellen. De er komplekse datastrukturer, hvilket ikke specielt er forbundet med objektorientering. Se eventuelt afsnit 8.1.4 ”Komplekse typer” for en grundigere omtale af dette emne. Dog er de kort beskrevet i Appendix C, da de også er en del af SQL99s udvidede typemodel, som også omfatter objektmodellen.

5.2 Brugerdefinerede typer

SQL99 introducerer begrebet **brugerdefineret type** (*user defined types*). Det er disse brugerdefinerede typer, som indeholder hovedparten af de objektorienterede faciliteter i SQL99. En brugerdefineret type kan opbygges på forskellige måder. Enten tager den sit udgangspunkt i en eksisterende type, eller også er den en struktureret type bestående af en eller flere attributter hver med sin type.

> ”The representation of a user-defined type is expressed either as a single data type (some predefined data type, called the *source type*), in which case the user-defined type is said to be a *distinct type*, or as a list of attribute definitions, in which case it is said to be a s*tructured type*.”
>
> [ISO, 1999: del 2, afs. 4.8]

De distinkte typer (*distinct types*) kan benyttes til nærmere at specificere funktionaliteten af en eksisterende type, hvorimod en struktureret type (*structured type*) kan benyttes til at modellere mere komplekse typers funktionalitet. Både distinkte og strukturerede typer kan have tilknyttet operationer.

Den overordnede grammatik for brugerdefinerede typer er følgende:

```
<user-defined type definition> ::= CREATE TYPE
                    <user-defined type body>
<user-defined type body> ::= <user-defined type name>
                    [ <subtype clause> ] [ AS <representation> ]
                    [ <instantiable clause> ] <finality>
                    [ <reference type specification> ] [ <cast option> ]
                    [ <method specification list> ]
```

[ISO, 1999: del 2, afs. 11.40]

Følgende er et eksempel, som viser det grundlæggende i brugerdefinerede typer. `FINAL` beskrives nærmere i afsnit 5.2.2 ”Uklarhed omkring afsluttede typer”.

Eksempel:
```
CREATE TYPE CPRType AS
   CHAR(10)
FINAL;

CREATE TYPE PersonType AS
(  cpr         CPRType,
   fornavn     CHAR(40),
   efternavn   CHAR(40)
)
NOT FINAL;
```

CPRType er en distinkt type, og PersonType er en struktureret type. Som det kan ses af grammatikken, er det reserverede ord AS samt repræsentationen af typen valgfri. Dette skyldes, at typer kan erklæres forud (*forward*), sådan som det kendes fra f.eks. C og Pascal. Dette er nødvendigt for at kunne definere gensidigt refererende typer.

5.2.1 Distinkte typer vs. domæner

Umiddelbart kunne distinkte typer godt minde en del om domæner, som kendes fra SQL92. De er begge baseret på en prædefineret type (herefter kaldet basistypen). De adskiller sig dog på en række væsentlige områder [Date, 2000] [Gulutzan & Peltzer, 1999].

1. Distinkte typer er underlagt stærk typekontrol. To distinkte typer, A og B, som begge har T som basistype, kan ikke udveksle data.
2. Distinkte typer arver kun sammenligningsoperationerne fra deres basistype. Alle øvrige skal selv defineres.
3. Det er muligt at definere CAST operationer for en distinkt type. Disse angives eksplicit og giver mulighed for at konvertere typen til eller fra andre typer (begge veje skal defineres). Dog skabes implicit to CAST operationer, når typen oprettes. Disse konverterer mellem den distinkte type og basistypen (en hver vej). Det er muligt at lave brugerdefinerede CAST operationer, som også er implicitte og derfor ikke behøver at blive angivet, hvor en konvertering er nødvendig.

En distinkt type er ikke en subtype af sin basistype, idet den distinkte type ikke nødvendigvis understøtter samtlige operationer, som basistypen gør.

5.2.2 Uklarhed omkring afsluttede typer

Det er ifølge standarden obligatorisk at angive om en brugerdefineret type er FINAL eller NOT FINAL (se afsnit 5.2.9 ”Afsluttede typer”). Der synes, at være en del uklarhed omkring dette emne i forbindelse med strukturerede typer. Distinkte typer kan kun være FINAL. Dette nævnes i den sekundære litteratur, f.eks. Gulutzan og Peltzer [1999] og Date [2000], samt i standarden:

> ” If… [a user defined type] …defines a distinct type, then:
>
> a) …
>
> b) …
>
> c) FINAL shall be specified.”
>
> [ISO, 1999: del 2, afs. 11.40]

Ligeledes nævner den samme sekundære litteratur, at valget står frit for mellem `FINAL` og `NOT FINAL` med hensyn til strukturerede typer. Dog synes dette ikke at passe med standarden. Den foreskriver, at strukturerede typer altid skal være `NOT FINAL`:

> ”6) If… [a user defined type] …specifies a structured type, then:
>
> a) …
>
> b) NOT FINAL shall be specified.”
>
> [ISO, 1999: del 2, afs. 11.40]

Umiddelbart må man tro på standarden, da den jo per definition er standarden, men det lader til, at der er en fejl eller mangel på dette punkt. Det giver ikke mening, at strukturerede typer altid skulle være `NOT FINAL`. Med mindre at det netop er for, at angive om typen er struktureret eller distinkt. Denne holdning nævnes af Gulutzan og Peltzer, men affejes uden grundig argumentation:

> ”… (There is a belief that the [final] clause is for distinguishing distinct types from structured types. We don’t believe that. We repeat that a distinct type is a UDT defined with ”AS <predefined type>”, all other UDTs are structured types.)”
>
> [Gulutzan & Peltzer, 1999: s. 528]

Hvis det eneste formål med angivelsen er at skelne mellem distinkte og strukturerede typer, kunne dette være løst noget enklere og mere intuitivt. F.eks. med følgende syntaks:

```
CREATE DISTINCT TYPE …
CREATE STRUCTURED TYPE …
```

Den endelige konklusion må forblive åben. Jeg vil i de efterfølgende eksempler holde mig til standardens forskrift. Emnet vil blive omtalt igen under gennemgangen af de tre konkrete systemer (Oracle, Informix og DB2) i afsnit 7.2 ”Typer”.

5.2.3 Brugerdefinerede typer og rækketyper

Tidligere beskrivelser af SQL99 standarden (SQL3), havde en klar skelnen mellem brugerdefinerede typer og såkaldte **rækketyper** (*row types*). Rækketyperne skulle bruges til at definere typer på tabeller, mens brugerdefinerede typer skulle bruges til at definere typer på attributter [Ullman & Widom, 1997] [Elmasri, 2000]. Dette ser

standarden nu ud til at gøre op med. I stedet benyttes brugerdefinerede typer til både tabeltyper og attributtyper.

Nedenstående er et uddrag af grammatikken for tabel definition (centrale elementer er fremhævet):

<table definition> ::= CREATE [<table scope>] TABLE
<table name> <table contents source>
[ON COMMIT <table commit action> ROWS]
<table contents source> ::= <table element list> |
OF <user-defined type> [<subtable clause>]
[<table element list>]
[ISO, 1999: del 2, afs. 11.3]

Definitionen på <user-defined type> er følgende:

<user-defined type> ::= <user-defined type name>
[ISO, 1999: del 2, afs. 6.1]

En brugerdefineret type betegnes altså med dens navn, som det er normalt i de fleste programmeringssprog.

Ovenstående betyder, at `CREATE TYPE` syntaksen nu også benyttes til tabeller. I de tidligere udgaver af SQL99, blev syntaksen `CREATE ROW TYPE` benyttet. Her var en kraftig skelnen mellem tabeltyper og attributtyper. At samme `CREATE TYPE` notation også bruges til attributtyper ses ved, at grammatikken for en attribut definition kræver en <data type>, som blandt andet kan være en <user-defined type>:

<attribute definition> ::= <attribute name> **<data type>**
[<reference scope check>] [<attribute default>]
[<collate clause>]
[ISO, 1999: del 2, afs. 11.41]

<data type> ::= <predefined type> | <row type> |
<user-defined type> | <reference type>
| <collection type>
[ISO, 1999: del 2, afs. 6.1]

Brugerdefinerede typer kan altså benyttes både som definitioner for tabeller og attributter. Spørgsmålet er så hvad <row type> dækker over (også angivet som en <data type> i ovenstående grammatik)? <row type> har følgende grammatik:

<row type> ::= ROW <row type body>
<row type body> ::= <left paren> <field definition>
[{ <comma> <field definition> }…] <right paren>
[ISO, 1999: del 2, afs. 6.1]

Definitionen minder om det oprindelige forslag til definitionen af rækketyper. Et eksempel på grammatikken er følgende:

Eksempel:
```
ROW ( cpr CHAR(10), fornavn CHAR(40), efternavn CHAR(40) )
```

Definitionen benytter <field definition> hvilket kun adskiller sig fra <attribute definition>, idet sidstnævnte kan have en default værdi tilknyttet. Ovenstående eksempel på en rækketype kan jævnfør grammatikken optræde i alle sammenhænge hvor en type er forventet[10]. Nedenstående eksempel viser et eksempel på denne brug:

Eksempel:
```
CREATE TABEL person
(  cpr          CHAR(10),
   navn         ROW ( fornavn     CHAR(40),
                      efternavn   CHAR(40)  ),
   adresse      ROW ( vej         CHAR(40),
                      postnr      INT,
                      bynavn      CHAR(40)  )
)
```

I eksemplet benyttes det reserverede ord `ROW` som type-konstruktør for to **unavngivne** typer. Det vil sige, at den er med til at definere typens struktur på samme måde som f.eks. *record* i Pascal og *struct* i C. `ROW` benyttes også som værdi-konstruktør, når nye rækker skal indsættes:

Eksempel:
```
INSERT INTO person VALUES
        (”1234561234”,
         ROW ( ”Hans”, ”Hansen” ),
         ROW ( ”H.Hansensvej 1”,1234,”Hanslev”) )
```

Denne brug af `ROW` er bl.a. beskrevet af Gulutzan og Peltzer [1999] og [Ramakrishnan & Gerhke, 2000]. Ovenstående tabel kunne også have været lavet ved hjælp af brugerdefinerede typer i stedet for rækketyper. Desuden er brugerdefinerede typer

[10] Dog kræver det at Feature T051 er inkluderet. Se eventuelt Appendix A om SQL99 standardens generelle opbygning.

langt kraftigere udtryksmæssigt, eftersom de også kan have tilknyttet operationer. Standarden har følgende forklaring på begrebet (fed skrift tilføjet):

> ”**The rows of a table have a type, called ‘‘the row type’’**; every row of a table has the same row type, which is also the row type of the table. A table that is declared to be based on some structured type is called a ‘‘typed table’’; its columns correspond in name and declared type to the attributes of the structured type. Typed tables have one additional column, called the ‘‘self-referencing column’’ whose type is a reference type associated with the structured type of the table.”
>
> [ISO, 1999: del 1, afs. 4.3]

En rækketype er altså også typen på en række i en typedefineret tabel (f.eks. en tabel som er defineret ud fra en brugerdefineret type). Det havde måske været mere oplagt, at en række i en tabel, som er defineret ud fra en type `T`, har typen `T`. Men der skelnes mellem attributter af typen `T` og tabelrækker af typen `T`. Grunden til dette kan være at tabelrækker har en implicit **selvrefererende kolonne**[11] (*self-referencing column*). Derved er de to typer ikke identiske, selvom de er skabt ud fra samme definition. En anden grund kan være ønsket om at bevare skellet mellem rækker og attributter, som den objekt-relationelle model ligger op til.

5.2.4 Typehierarkier

Det er muligt at danne et typehierarki ved hjælp af nedarvning. Kun enkelt nedarvning er understøttet [ISO, 1999: del 2, afs. 4.8.3]. Der findes ikke nogen fælles supertype, som alle typer automatisk er en subtype af.

Subtyper genereres med det reserverede ord `UNDER`.

Eksempel:

```
CREATE TYPE PersonType AS
(  cpr         CPRType,
   fornavn     CHAR(40),
   efternavn   CHAR(40)
)
NOT FINAL;

CREATE TYPE MedarbejderType UNDER PersonType AS
(  stilling    CHAR(40),
   lokalnummer SMALLINT,
   leder       CPRType
)
NOT FINAL;
```

[11] Beskrives nærmere i afsnit 5.4.1 ”Repræsentation af referencer”.

5.2.5 Typetest på køretidspunktet

I objektorienterede modeller er typen af værdien i en variabel ikke altid kendt på oversættelsestidspunktet. Det er derfor normalt, at tillade typecheck på køretidspunktet. Dette understøtter SQL99 også. Det er muligt at angive en hel liste af typer, samt at begrænse typechecket til én specifik type uden at inkludere dennes eventuelle subtyper.

Nedenstående tabel viser eksempler på udtryk og deres sandhedsværdi.

Eksempel:

- `S << T`
- v er en værdi med typen `S`
- n er `NULL`
- `U` er en type udenfor typehierarkiet med `T` og `S`

UDTRYK	SANDHEDSVÆRDI
`v IS OF (T)`	TRUE
`v IS NOT IF (T)`	FALSE
`v IS OF (T, U)`	TRUE
`v IS OF (U)`	FALSE
`v IS OF (ONLY S)`	TRUE
`v IS OF (ONLY T)`	FALSE
`n IS OF (T, U)`	UNKNOWN

5.2.6 Typekonvertering

SQL99 bruger stærk typekontrol ved navngivne brugerdefinerede typer. Det er således ikke muligt at sammenligne værdier af en navngiven brugerdefineret type med værdier af andre typer (hverken brugerdefinerede eller prædefinerede). Det er dog muligt at oprette brugerdefinerede **konverteringer**, som kan omgå den strenge typekontrol. Dette gøres med `CREATE CAST` kommandoen.

Eksempel:

```
CREATE CAST( PersonType AS VARCHAR(90) )
   WITH FUNCTION PersonType2VarChar90(PersonType)
   AS ASSIGNMENT;
```

Angivelsen `AS ASSIGNMENT` er valgfri. Hvis den er medtaget kan typekonverteringen foretages implicit.

Eksempel:

```
strengVar = CAST( personVar AS VARCHAR(90) );      -- eksplicit
strengVar = personVar;                             -- implicit
```

Den brugerdefinerede konverteringsoperation skal have formen:

```
FUNCTION funktionsnavn(FraType) RETURNS TilType;
```

Unavngivne typer er kompatible med alle andre unavngivne typer, som de er strukturelt ækvivalente med.

Eksempel:
Attributterne a og b er strukturelt ækvivalente. Derfor er de kompatible.

```
CREATE TABLE t1
( a     ROW(  a1 INT, a2 INT ) );

CREATE TABLE t2
( b     ROW(  b1 INT, b2 INT ) );

SELECT a
FROM t1, t2
WHERE t1.a = t2.b; -- lovligt
```

Hvis der er tale om typekonvertering fra en supertype til en subtype, kan konverteringen foretages eksplicit ved hjælp af `TREAT` operatoren.

Eksempel:

```
kundeVar = TREAT( personVar AS KundeType );
```

Ovenstående giver fejl, hvis værdien af variablen personVar ikke er af typen KundeType. Konvertering fra subtype til supertype kræver ingen særlig syntaks, da dette er indeholdt i polymorfibegrebet.

Eksempel:

```
personVar = kundeVar; -- lovligt på grund af polymorfi
```

5.2.7 Polymorfi

Polymorfi i brugerdefinerede typehierarkier er fuldt understøttet.

> ”If T_a is a subtype of T_b, then a value in T_a can be used wherever a value in T_b is expected. In particular, a value in T_a can be stored in a column of type T_b, can be substituted as an argument for an input SQL parameter of data type T_b, and can be the value of an invocation of an SQL-invoked function whose result data type is T_b.”
>
> [ISO, 1999: del 2, afs. 4.8.3]

Det vil derfor være muligt at indsætte værdier af typen S i attributter, hvis den erklærede type T er en supertype til S. Dog begrænser ovenstående sig ikke til polymorfi på attributniveau. Der er tale om en vilkårlig type. Det vil derfor også være muligt at indsætte en række af typen S i en tabel af typen T. Typedefinerede tabeller omtales senere i afsnit 5.3 ”Typedefinerede tabeller”.

5.2.8 Abstrakte typer

Abstrakte typer omtales ikke direkte i standarden, men i syntaksen for de brugerdefinerede typer er angivet to mulige annoteringer: `NOT INSTANTIABLE` og `INSTANTIABLE`. De er ikke beskrevet nærmere, men det må antages, at det er abstrakte typer de giver mulighed for.

- `NOT INSTANTIABLE` svarer til abstrakt
- `INSTANTIABLE` svarer til ikke abstrakt

Default er `INSTANTIABLE` [ISO, 1999: del 2, afs. 11.40].

5.2.9 Afsluttede typer

En type kan erklæres som værende **afsluttet** (*final*). Dette betyder, at den ikke kan have subtyper, og at andre typer derfor ikke kan nedarve fra den. Der er ikke noget belæg for i objektmodellen at indføre en skelnen mellem typer, som er afsluttede eller ej, men det er der når typesystemet skal implementeres. Her kan det give en hastighedsforbedring på køretidspunktet. Hvilke optimeringer der kan foretages afhænger af den konkrete implementation, men en typisk fordel er følgende.

Kald til operationer i en given type `T`, kan i mange tilfælde ikke *linkes* på oversættelsestidspunktet. Har `T` en subtype `S`, som redefinerer den kaldte operation, kan man ikke altid på oversættelsestidspunktet afgøre, hvilken udgave af operationen, som skal udføres. Det kommer an på hvilken ”mest specifik type”, den involverede værdi har på kørselstidspunktet. Derfor må der laves en såkaldt dynamisk linkning, som er langsommere end den tilsvarende statiske. Havde `T` været afsluttet og derved ikke havde haft nogen subtype `S`, så kunne kaldet være blevet foretaget med statisk linkning.

Man skulle tro, at man på oversættelsestidspunktet kunne afgøre om en type har nogen subtyper, men det er ikke altid tilfældet. I visse objektmodeller er det muligt at indlæse værdier af nye typer på kørselstidspunktet. I disse modeller er det således muligt at udbygge typehierarkiet på kørselstidspunktet, og man kan derfor ikke bestemme på oversættelsestidspunktet om en type har nogen subtyper. Men hvis typen er erklæret som værende afsluttet, så kan man være sikker på, at der ikke tilføjes nye subtyper. Samme situation findes i f.eks. Java, som også understøtter afsluttede typer.

5.2.10 Operationer

Både distinkte og strukturerede typer kan have operationer tilknyttet. Kun **operationshovedet** erklæres sammen med den brugerdefinerede type. Dette beskriver operationens interface (navn, argumenter, etc.). **Operationsimplementationen** angives særskilt. Denne indeholder den konkrete implementation af operationen.

Det er muligt at angive operationer, som er statiske og dynamiske. Dette gøres med de reserverede ord `STATIC` og `INSTANCE`[12]. Dynamiske operationer kaldes i standarden *instance methods*. Selv-referencen i de dynamiske operationer kaldes for `SELF`.

Eksempel:

```
CREATE TYPE CPRType AS
   CHAR(10)
INSTANTIABLE
FINAL
INSTANCE METHOD Modulus11Check() RETURNS BOOLEAN
   LANGUAGE C
   PARAMETER STYLE GENERAL
   DETERMINISTIC
   NO SQL;

CREATE TYPE PersonType AS
(  cpr         CPRType,
   fornavn     CHAR(40),
   efternavn   CHAR(40),
   telefonnr   CHAR(8)
)
NOT INSTANTIABLE
NOT FINAL;

CREATE TYPE MedarbejderType UNDER PersonType AS
(  stilling    CHAR(40),
   gage        SMALLINT,
   leder       REF(MedarbejderType)
)
INSTANTIABLE
NOT FINAL
INSTANCE METHOD ArbejderUnder
   (IN leder MedarbejderType AS LOCATOR) RETURNS BOOLEAN
   LANGUAGE SQL
   PARAMETER STYLE SQL
   DETERMINISTIC
   READS SQL DATA;
```

De fire forskellige beskrivelser af operationen har følgende betydninger:

- `LANGUAGE SQL`: Angiver hvilket sprog implementationen af operationen er skrevet i. Mulige sprog[13]: `ADA`, `C`, `COBOL`, `FORTRAN`, `MUMPS`, `PASCAL`, `PLI`, `SQL`.
- `PARAMETER STYLE`: Angiver hvordan argumenter overføres. Mulige måder: `SQL`, `GENERAL`

[12] Bemærk at operationer benytter det reserverede ord `INSTANCE`. Typer benytter `INSTANTIABLE` i forbindelse med abstrakte typer.

[13] ”MUMPS” er Massachusetts General Hospital Utility Multi-Programming System. ”PLI” er PL/1.

- `DETERMINISTIC`: Angiver om operationen er deterministisk eller muligvis ikke er deterministisk. Med deterministisk menes, at for et givet sæt parametre vil operationen altid give det samme resultat. Hvis den er nondeterministisk kan den give forskellige resultater for samme værdier som argumenter. Dette kan bruges af databasesystemet til at optimere en række udførelser af samme operation. Hvis flere operationskald vil have samme argumenter, og operationen er deterministisk, kan systemet nøjes med at udføre operationen én gang. Mulige angivelser: `DETERMINISTIC`, `NOT DETERMINISTIC`
- `READ SQL DATA`: Angiver i hvor høj grad operationen bruger SQL. Angivelsen sætter grænsen for, hvad operationen må foretage sig, men en operation som både må læse og skrive SQL data, behøver ikke at gøre det. Mulige niveauer: `NO SQL`, `CONTAINS SQL`, `READ SQL DATA`, `MODIFIES SQL DATA`.

Argumentet leder i operationen ArbejderUnder er erklæret som værende et `IN` argument. Det vil sige, at eventuelle rettelser i værdien ikke skal afspejle sig i den oprindeligt overførte værdi. Andre muligheder er `OUT` og `INOUT`. Argumentet er også forsynet med `AS LOCATOR` angivelse. Det betyder, at værdien skal referenceoverføres i stedet for værdioverføres. Således vil leder være en reference til Medarbejder-værdien, hvilket forbedrer køretiden. Dette er hovedsageligt tiltænkt store typer som f.eks. `BLOB` og `CLOB`[14]. Referenceoverførte argumenter er normalt den måde programmeringssprog implementerer `OUT` argumenter på. Det vil sige, at ændringer skal afspejle sig i den oprindelige værdi, som operationen blev kaldt med. Men i SQL99 er det muligt at angive brugen af referenceoverførte argumenter, som ikke skal opdatere den oprindelige værdi.

Operationer kan enten være **interne** eller **eksterne**. Interne operationer udføres af selve databasesystemet. Eksterne operationer kaldes igennem operativsystemet. Disse kan være skrevet i f.eks. C eller Pascal. Nedenstående eksempel viser, hvorledes en ekstern operation i Pascal kan defineres.

Eksempel:
```
INSTANCE METHOD Modulus11Check() FOR CPRType
   RETURNS BOOLEAN
   LANGUAGE PASCAL
   EXTERNAL NAME modulus_11_check
   PARAMETER STYLE GENERAL;
```

Selve implementationen af operationen angives separat. Den indeholder ikke noget, som har betydning for objektmodellens indhold, så den vil ikke blive gennemgået.

[14] *Binary Large OBject* og *Character Large OBject*. De er begge subtyper af LOB (*Large Object*).

5.2.10.1 Sammenligningsoperationer

For at databasesystemet skal være istand til at sammenligne kolonner, som har brugerdefinerede typer, er det nødvendigt, at specificere hvorledes denne sammenligning skal foretages. Dette kan gøres på tre måder. De har alle visse begrænsninger i forhold til hvilke typer, som de kan benyttes med. Den generelle syntaks ser således ud:

> ”<user-defined ordering definition> ::= CREATE ORDERING FOR
> <user-defined type> <ordering form>
> <ordering form> ::= <equals ordering form> |
> <full ordering form>
> <equals ordering form> ::= EQUALS ONLY BY
> <ordering category>
> <full ordering form> ::= ORDER FULL BY <ordering category>
> <ordering category> ::= <relative category> | <map category> |
> <state category>
> <relative category> ::= RELATIVE WITH
> <relative function specification>
> <map category> ::= MAP WITH <map function specification>
> <state category> ::= STATE [<specific name>]”
> [ISO, 1999: del 2, afs. 11.54]

Som det fremgår, angives det om ordningen kun må bruges til lighedsoperationer (`EQUALS ONLY`) eller til ordninger generelt (`ORDER FULL`). De tre typer har følgende betydning:

`RELATIVE WITH`: Den angivne operation tager to argumenter af typen <user-defined type> og returnerer et heltal, som angiver forholdet mellem de to værdier. Eftersom beregningen af den returnerede værdi er brugerdefineret, kan der ikke siges noget generelt om den. Dog vil databasesystemet altid tolke svaret på følgende måde: En positiv værdi betyder, at første argument er størst, en negativ værdi at andet argument er størst og nul at de er ens (svarende til `strcmp` i C). Denne ordning kan kun angives for maksimale supertyper (det vil sige typer, som ikke har nogen supertyper).

`MAP WITH`: Operationen tager kun ét argument af typen <user-defined type> og returnerer en værdi af en prædefineret type. Den returnerede værdi vil typisk være et hash af argumentværdien, men dette bestemmes igen af den konkrete implementation. Modsat relative ordning, kan mapning tildeles subtyper. Dette kræver dog, at samtlige supertyper til typen også har mapninger defineret.

`STATE`: Hvis tilstandsordning vælges, oprettes en automatisk sammenlignings-operation, som returnerer `TRUE` eller `FALSE`. Sammenligningen baseres på en sammenligning af hver enkelt attribut i <user-defined type>. Hvis alle attributter har samme værdi returneres `TRUE`, ellers returneres `FALSE`. Gulutzan og Peltzer [1999]

skriver, at operationen aldrig kan returnere UNKNOWN. Dette virker som et brud med de gængse regler for SQLs treværdislogik. En kendt værdi sammenlignet med en ukendt værdi (NULL), giver sandhedsværdien UNKNOWN. Måske er de to ens, måske ikke. Standarden nævner ikke noget om dette, andet end at operationen har følgende definition:

```
”CREATE FUNCTION SNUDT.EQUALS
            ( UDT1 UDTN, UDT2 UDTN )
  RETURNS BOOLEAN
  SPECIFIC SN
  DETERMINISTIC
  CONTAINS SQL
  STATIC DISPATCH
  RETURN
   ( TRUE AND
     SPECIFICTYPE (UDT1) = SPECIFICTYPE (UDT2) AND
     UDT1.C 1 = UDT2.C 1 AND
     .
     .
     .
     UDT1.C n = UDT2.C n )”
```

[ISO, 1999: del 2, afs. 11.54]

Som det fremgår her returnerer funktionen BOOLEAN, men dette burde ikke forhindre at NULL returneres. Alle værdier i en database kan angives som værende NULL, med mindre NOT NULL er specificeret. Hvorvidt en operation kan returnere NULL er uklart. Både standarden [ISO, 1999] og Gulutzan og Pelzer [1999] beskriver ikke det område. Dog mener jeg, at det må være højst sandsynligt, at det kan lade sig gøre, men det kan ikke afgøres med sikkerhed.

5.2.11 Indkapsling

Følgende uddrag fra standarden beskriver, at attributter i brugerdefinerede typer er indkapslede, og at der kun er adgang til dem gennem særlige observatøroperationer (*observer functions*):

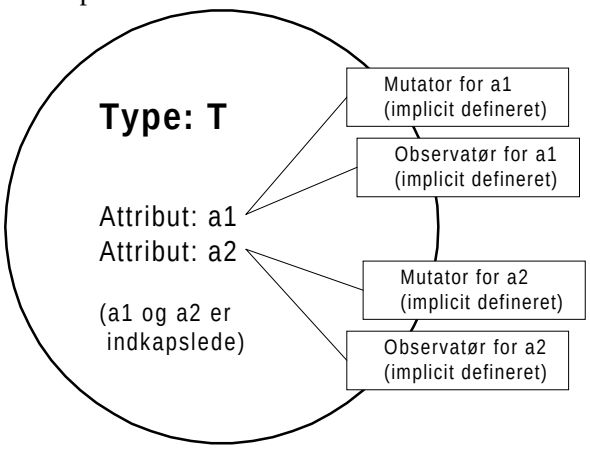
> ”Attribute values are said to be *encapsulated*; that is to say, they are not directly accessible to the user, if at all. An attribute value is accessible only by invoking a function known as an *observer function* that returns that value. An instance of a structured type can also be accessed by a *locator*.”
>
> [ISO, 1999: del 1, afs. 4.6.4.1]

Næste uddrag beskriver, at observatøroperationer implicit defineres for hver attribut i de brugerdefinerede typer:

> "**observer function**: An SQL-invoked function *A* implicitly defined by the definition of attribute *A* of a structured type T. If *V* is some value in *T* and the declared type of *A* is *AT*, then the invocation of *A(V)* returns some value *AV* in *AT*. *AV* is then said to be the value of attribute *A* in *V*."
>
> [ISO, 1999: del 2, afs. 3.1.5]

Tilsvarende defineres implicit en mutatoroperation (*mutator*) for hver attribut [ISO, 1999: del 2, afs. 3.1.5]. Dog er der ingen omtale af forskellige indkapslingsniveauer, som f.eks. privat og offentlig. Hvis der implicit defineres både en observatør- og mutatoroperation for hver attribut og denne tilsyneladende er offentlig, så mister indkapslingen sin betydning. Præcis dette emne er også uklart i Ullman og Widoms [1997] beskrivelse, men det blev antydet at indkapslingsniveauerne skal styres med rettigheder i databasesystemernes indbyggede sikkerhedssystem. Standarden har holdt sig til denne løsning, idet grammatikken for rettigheder [ISO, 1999: del 2, afs. 10.5] giver mulighed for at tildele eller fjerne `EXECUTE` rettigheder for udvalgte operationer. På den måde understøtter standarden indkapsling, men gør det dynamisk, således at indkapslingen afhænger af, hvilken kontekst variable af typen benyttes i.

Eksempel:



Typen T oprettes med to attributter, A1 og A2. Disse er automatisk indkapslede, således at deres virkefelt (*scope*) er begrænset til typens interne repræsentation. Samtidig oprettes implicit to operationer til typer for hver attribut. De benyttes til at ændre og aflæse attributtens værdi. Man kan ikke forhindre at disse operationer oprettes, så de må siges at bryde med indkapslingen. Dog tillader databasesystemets rettighedsstyring, at man kan fjerne `EXECUTE` rettighederne fra operationerne, således at ingen (eller kun nogen) kan udføre dem. Hvis attributten skal være fuldstændigt indkapslet og hverken må læses eller opdateres udefra, fjerner man alle `EXECUTE` rettigheder til den tilhørende operationer.

5.3 Typedefinerede tabeller

I SQL92 var erklæringen af typen og oprettelsen af tabellen samlet i samme SQL kommando. SQL99 tilføjer en ny måde, som tillader at typen først defineres i form af en brugerdefineret type og tabellen derefter oprettes. Derved kan en type benyttes til flere tabeller.

Eksempel:
```
CREATE TABLE maend OF PersonType;
CREATE TABLE kvinder OF PersonType;

CREATE TABLE medarbejdere OF MedarbejderType;
```

Det er også muligt at skabe tabeller i et hierarki, således at tabellerne danner et slags typehierarki. Værdier i en given undertabel er implicit også med i dennes supertabeller.

Eksempel:
```
CREATE TABLE personer OF PersonType;
CREATE TABLE medarbejdere OF MedarbejderType UNDER personer;
```

I ovenstående eksempel er det et krav, at MedarbejderType er en direkte subtype af PersonType. Det vil sige, at MedarbejderType er defineret som værende `UNDER PersonType`.

For hver række i medarbejdere vil der altid være en tilsvarende række i personer. Hvis en række slettes i medarbejdere eller personer vil alle tilsvarende rækker i den anden tabel også blive slettet. Eftersom rækker findes i alle passende tabeller, vil en forespørgsel på f.eks. personer også give resultater, som faktisk er medarbejdere. Dette er helt i tråd med den objektorienterede tankegang.

5.3.1 Indkapsling af typedefinerede tabeller

Typedefinerede tabeller har ingen indkapsling. Derved kan typedefinerede tabeller behandles som almindelige tabeller. Det er muligt at indsætte, forespørge og opdatere data direkte. Dette kræver dog de tilsvarende rettigheder til tabellen, men adgangen er så at sige ”alt eller intet” [Date, 2000] [Gulutzan & Pelzer, 1999].

Dog kan indkapslingen simuleres ved hjælp af *views*. De kan begrænse hvilke attributter, som er synlige udefra. Den direkte adgang til selve tabellen kan så fjernes ved hjælp af rettighedsstyring.

5.4 Referencer

Som nævnt skelnede den tidligere udgave af standarden (SQL3) mellem brugerdefinerede typer og rækketyper. Dette skel er nu fjernet, og brugerdefinerede typer bruges nu også som typer på tabeller. Det er derfor heller ikke overraskende, at referencer nu peger på brugerdefinerede typer i stedet for rækketyper:

> <reference type> ::= REF <left paren> <referenced type>
> <right paren> [<scope clause>]
> <scope clause> ::= SCOPE <table name>
> <referenced type> ::= **<user-defined type>**
> [ISO, 1999: del 2, afs. 6.1]

`SCOPE` angivelsen er bibeholdt fra tidligere. Den er en valgfri tilføjelse til `REF` definitionen, som begrænser referencen til kun at henvise til rækker i én bestemt tabel.

5.4.1 Repræsentation af referencer

Den traditionelle måde at lave referencer på ved hjælp af fremmednøgler kan erstattes med referencer, som i dette eksempel:

Eksempel:
```
CREATE TYPE MedarbejderType UNDER PersonType AS
(  stilling    CHAR(40),
   lokalnummer SMALLINT,
   leder       REF(MedarbejderType)
)
NOT FINAL;
```

I eksemplet henviser referencen til sin egen type, men den kunne lige så godt have henvist til en anden brugerdefineret type.

Det er en generelt udbredt metode i databasemodellering, at tupler identificeres med et kunstigt unikt nummer (f.eks. kundenummer, adressenummer etc.). Dette kan så benyttes som fremmednøgle i andre tabeller, som refererer til rækken. På mange måder er dette lettere. Dels slipper man for at overveje, hvad primærnøglen rent faktisk består af, dels undgår man at fylde sine tabeller med lange fremmednøgler. Ulempen er at numrene i sig selv ikke giver nogen mening. Dette er ofte acceptabelt, da nummeret hurtigt kan slås op i den relevante tabel. Emnet diskuteres nærmere i afsnit 6.1.1.1 ”Motivation for den valgte løsning”.

Næsten samme metode er brugt i forbindelse med typedefinerede tabeller. Hver tupel i tabellen får tildelt et unikt nummer kaldet en objekt identifikator (*object identifier*). Dette forkortes ifølge Stonebraker og Brown [1999] **OID**. Standarden omtaler kun

OIDer som selvrefererende kolonner[15]. OIDer svarer til *self* referencen i dynamiske operationer (se afsnit 3.2.6 ”Operationer”). OID er generelt et accepteret begreb og samtidig væsentligt kortere end selvrefererende kolonne. Derfor vælger jeg at benytte OID fremover.

Hvorledes referencer skal repræsenteres, er ikke defineret i standarden. Den foreskriver kun, at hver række i en typedefineret tabel skal have tilknyttet en unik identifikator på *N* byte, hvor *N* er implementationsafhængigt. I øvrigt skelner standarden mellem tre former for referencer. Når en brugerdefineret type defineres, angiver man hvilken slags reference, som skal benyttes, når der refereres til værdier af typen. De tre referencetyper er:

- brugerdefineret (*user-defined representation*)
- afledt (*derived representation*)
- systemgenereret (*system-defined representation*)

OIDer af typen **brugerdefineret** betyder, at man selv angiver typen, som skal ligge til grund for referencen. F.eks. er det muligt at skrive følgende:

Eksempel:
```
CREATE TYPE PersonType AS
(  cpr         CPRType,
   fornavn     CHAR(40),
   efternavn   CHAR(40)
)
NOT FINAL
REF USING INTEGER;
```

Dette betyder, at rækker af typen `PersonType` vil blive identificeret med et heltal. Dette heltal skal suppleres sammen med de øvrige data, når en ny række af denne type skal indsættes i databasen. Enhver prædefineret type kan benyttes, men ikke brugerdefinerede typer. Denne form kaldes også for brugergenererede OIDer [Gulutzan & Pelzer, 1999]. Værdien af OIDen skal nemlig angives af brugeren, hver gang en ny række indsættes.

Afledte OIDer benytter følgende syntaks (kun den relevante linie er gengivet):

```
REF FROM (CPR)
```

Det er muligt at angive flere attributter adskilt med komma. Referencen bliver i dette tilfælde konstrueret ud fra de angivne attributter i typen – præcis som en sammensat primærnøgle. Kravene til attributterne er også de samme som for primærnøgler (`UNIQUE`, `NOT NULL`).

[15] Grunden til at ordet ”OID” undgås er formentligt at ”Object identifier” er betegnelsen i SQL99 for den streng, som de alle databasesystemer bør indeholde, som beskriver i hvor høj grad de er kompatible med standarden. Se eventuelt Appendix A.

Den sidste slags reference er **systemgenererede**. Syntaksen er følgende:

```
REF IS SYSTEM GENERATED
```

I dette tilfælde bliver referencerne systemgenereret, og det er derfor op til den konkrete implementation at bestemme, hvad referencen skal bestå af. Hvis intet angives, når en type defineres, bliver referencetypen sat til systemgenereret.

For alle tre referencetyper gælder det, at referencerne skal være unikke og ikke `NULL`. Hvorvidt, referencerne skal være lokalt unikke (indenfor samme database) eller globalt unikke (i alle databaser i hele verden[16]), er ikke helt klart i standarden. Umiddelbart virker det som om, at referencen blot skal være lokalt unik.

> "A table *BT* whose row type is derived from a structured type *ST* is called a *typed table*. Only a base table or a view can be a typed table. A typed table has columns corresponding, in name and declared type, to every attribute of *ST* and one other column *REFC* that is the self-referencing column of *BT*; let *REFCN* be the <column name> of *REFC*. The declared type of *REFC* is necessarily REF(*ST*) and the nullability characteristic of *REFC* is *known not nullable*. If *BT* is a base table, then **the table constraint 'UNIQUE(REFCN)' is implicit** in the definition of *BT*." (understregning tilføjet)
>
> [ISO, 1999: del 2, afs. 4.16.2]

Standarden betegner det som et *table constraint*. Så det lader til, at kravet kun gælder indenfor en given tabel. Dog kan man nemt konstruere eksempler, hvor lokalt unikke referencer vil skabe tvetydighed.

Eksempel:

```
CREATE TYPE A AS
(  id    INTEGER     )
FINAL
REF IS FROM (id);

CREATE TABLE t1 OF A;
CREATE TABLE t2 OF A;

CREATE TABLE t3
(  rf    REF(A)      );
```

Værdier af typen `A` identificeres ud fra `id`. Der findes to tabeller af typen `A`. Hvis `t3` indeholder en reference til en værdi, som findes i både `t1` og `t2`, så vil det ikke være muligt at afgøre, hvilken værdi der oprindeligt blev refereret til. Man kan umiddelbart forestille sig tre løsningsmuligheder:

[16] I princippet bør et OID være globalt unikt. Hvis en værdi flyttes fra en database til en anden burde den ikke ændre OID, da den så per definition ville være en anden værdi.

1. Referencer skal være globalt unikke.
2. Ovenstående tilfælde tillades kun hvis rf har defineret et `SCOPE`.
3. Referencer bliver af systemet tilføjet yderligere oplysninger, som f.eks. tabelnavn.

System genererede OIDer må forventes at være globalt unikke, og vil derfor ikke kunne give tvetydighed.

5.4.2 Redundant referenceerklæring

Der synes at være redundans i erklæringen af den ønskede form for OID. Som beskrevet kan dette angives på definitionstidpunktet for typen. Dog kan samme angivelse forekomme i forbindelse med oprettelse af typedefinerede tabeller[17]. Her er det muligt både at navngive OID kolonnen, samt at specificere dens type. Det er således muligt at skrive følgende ifølge SQL99s syntaks:

```
CREATE TYPE MyType
(  a     INT )
FINAL
REF IS SYSTEM GENERATED;

CREATE TABLE myTable OF MyType
(  REF IS oid DERIVED      --<-- dette burde ikke være tilladt!
);
```

Det har ikke været muligt at finde en omtale af dette i standarden, men det må antages, at den specificerede OID type i tabeldefinitionen skal svare til den valgte for tabeltypen.

5.4.3 Dereferering

Den store fordel ved referencer er, at man nemt kan navigere rundt mellem sammenkædede data ved hjælp af **sti-udtryk**.

Eksempel:

Definitioner:

```
CREATE TYPE MedarbejderType UNDER PersonType AS
(  stilling    CHAR(40),
   lokalnummer SMALLINT,
   leder       REF(MedarbejderType)
)
NOT FINAL;

CREATE TABLE medarbejdere OF MedarbejderType;

CREATE TABLE salgsafdeling
(  salgsmedarbejder  REF(MedarbejderType) SCOPE medarbejdere );
```

[17] Dog kun for basistabeller (tabeller uden supertabeller).

Forespørgsel:
```
SELECT salgsmedarbejder->fornavn
FROM salgsafdeling
WHERE salgsmedarbejder->leder->efternavn = ”Hansen”;
```

Dereferering bliver som vist udført med operatoren **->** (sammensat af ”minus” og ”større end”). Denne benyttes, når man ønsker at angive en operation eller attribut i den refererede værdi, som vist i eksemplet med ”salgsmedarbejder**->**fornavn”. Ønsker man i stedet at få fat i hele den refererede værdi, benyttes operationen `DEREF`. Havde man i eksemplet skrevet `SELECT DEREF(salgsmedarbejder)`, så havde forespørgslen returneret hele den refererede værdi fra tabellen medarbejdere.

Det kan forekomme, at den værdi, som en reference peger på, er blevet slettet. Hvis en sådan reference derefereres, returneres `NULL`.

5.5 Diskussion

I dette afsnit vil jeg kommentere standardens objektmodel. Hvert afsnit vil være inddelt i to underafsnit: Teoretisk og Praktisk. I det teoretiske afsnit anskues aspektet fra en akademisk, teoretisk synsvinkel. I det praktiske afsnit vælges en praktisk, implementationsbevidst synsvinkel.

5.5.1 Objektmodellens omfang

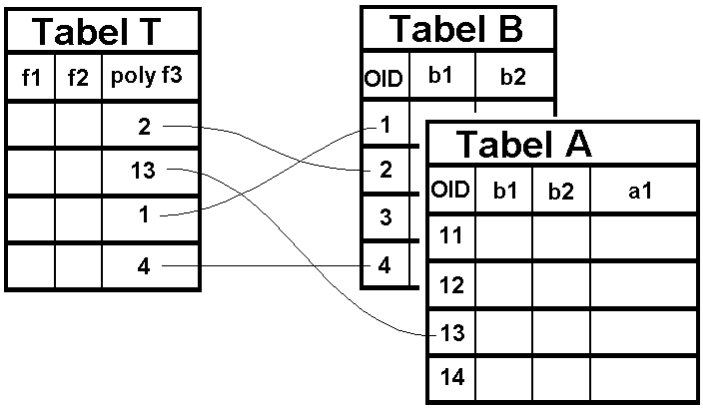
5.5.1.1 Teoretisk

Objektmodellen i SQL99 er meget omfattende. Subtyper, polymorfe typer, sammenknytning af typer og operationer samt referencer er alle omfattet af standarden. Den må derfor siges, at have en næsten fuldt udbygget objektmodel. Det eneste, som mangler, er indkapslingen på attributniveau.

5.5.1.2 Praktisk

At standarden inkluderer polymorfi på attributniveau har formentligt været overvejet en ekstra gang, da det kræver en del ændringer i databasesystemernes måde at behandle data og typer på. Ikke alene kan værdier have variabel størrelse, men de kan også udvides løbende. I SQL92 var det muligt at specificere f.eks. variable strenge med `VARCHAR(`*n*`)`, men disse kan maksimalt have en længde på *n* karakterer. Nu er det muligt at have attributter, som har typen `T`, som har størrelse *n*. `T` kan så nedarves til `S`, som har størrelsen *n*+*m*. På grund af polymorfi skal det være muligt at lagre værdier af typen `S` i attributter af typen `T`. Det giver et praktisk problem. En løsning ville være at tilføje et ekstra lag af indirektion, således at attributten i virkeligheden var en reference til værdien. Denne kunne så gemmes i en implicit tabel for værdier af netop dens type.

Eksempel:



Tabellen T indeholder en attribut, poly f3, som har typen B. A er en subtype af B. I dette tilfælde kunne databasesystemet automatisk oprette to tabeller: en af typen A (Tabel A) og en af typen B (Tabel B). Når en værdi af den respektive type blev indsat i poly f3, ville den blive indsat i den tilsvarende tabel. Kun OIDen ville optræde i selve tabellen T.

Prisen for løsningen er forringet udførelseshastighed, men ligger meget tæt op ad den måde ODMG foreslår at ODL/OQL skal implementeres på (ved hjælp af *extents*) [Cattell & Barry, 1997].

Indkapsling, referencer og sammenknytning af typer og operationer ligger ikke fjernt fra de eksisterende databasesystemers funktionalitet.

- Indkapsling: Sådan som den er defineret i SQL99, vil den blive styret med det eksisterende rettighedssystem.
- Identitetsbaserede referencer: Ligger forholdsvis tæt op af værdibaserede referencer baseret på kunstnøgler og systemgenererede rækkenumre (*row ID*).
- Sammenknytning af typer og operationer: Dette får først større praktisk betydning, når subtyper og polymorfi inddrages. Ellers er det blot eksplicit definerede og navngivne virkefelter (i stil med *namespaces* i C++).

5.5.2 SQL99 vs. ODMG

5.5.2.1 Teoretisk

Grænsen mellem ODMGs forslag til en standard for objektorienterede databasesystemer [Cattell & Barry, 1997] og SQL99 standarden for objekt-relationelle databasesystemer har hidtil været klart afgrænset. ODMG er fortaler for at typer (hvilket de kalder klasser) skal være den primære ”byggeklods” i databaserne. Omvendt fastholder fortalere for det objekt-relationelle (som f.eks. Stonebraker og Brown [1999]), at relationen fortsat skal være det centrale. Hver har sine argumenter,

men grundlæggende drejer det sig om, at ODMG ønsker en revolution, hvorimod f.eks. Stonebraker ønsker en evolution.

Denne grænse er imidlertid blevet mindre tydelig med den endelige udformning af SQL99 standarden, idet brugerdefinerede typer nu både benyttes til at definere typerne på attributter og tabeller. Desuden kan der opbygges typehierarkier med tabeller. På den måde bliver typerne den centrale "byggeklods", når en database skal designes. Dette er et stort skridt i retning af ODMGs foreslåede standard. Dog ligger SQL99 stadig meget tæt op ad SQL92 og er da også langt mere bagud kompatibelt med denne end ODMGs forslag.

ODMG har et meget anderledes definitionssprog, Object Definition Language (ODL), som adskiller sig meget fra det traditionelle Data Definition Language (DDL), som SQL standarden indeholder. Dog står der nævnt i ODMG version 2.0, at enhver SQL92 forespørgsel skal kunne forstås af de objektorienterede databasesystemer. Med andre ord skal SQL92 *Data Manipulation Language* (DML) være en delmængde af *Object Query Language* (OQL). Kravet om understøttelse af SQL92 DML, er sikkert medtaget for at kunne vinde indpas hos den store mængde af SQL-programmører, som ikke nødvendigvis kender OQL. OQL ligger da også meget tæt op ad SQL, men har visse udvidelser. En del af disse er medtaget i SQL99 (f.eks. polymorfi og dereferering), så forskellen er blevet væsentligt mindre.

5.5.2.2 Praktisk

ODMG har svært ved at slå igennem, da det er en helt ny teknologi. Det vil tage noget tid før teknologien vil blive opfattet som værende lige så robust som den relationelle. De rent objektorienterede databasesystemer har desuden det problem, at de kræver et vist kendskab til objektorientering. Det er begyndt at være mere og mere udbredt, men to af de mest brugte programmeringssprog igennem tiderne er COBOL og C. Hvis man kunne måle hvor stor en procent del af verdens software, som er objektorienteret, så ville det formentligt være under 20%. Langt det meste software på mainframes er skrevet i COBOL eller PL/1, og på UNIX-maskinerne er det C, som anvendes mest. De objektorienterede sprog vinder stadig frem, og hvis man udelukkende ser på software udviklet indenfor de seneste fem år, så er det formentligt langt mere end 20%, som er mere eller mindre objektorienteret.

5.5.3 Indkapslingen

5.5.3.1 Teoretisk

Normalt angives indkapslingsniveauerne for en type på definitionstidspunket (f.eks. Java, C++ og Delphi). Det sikrer, at typen har et ensartet interface til sine omgivelser. I C++ har man mulighed for at åbne for indkapslingen overfor visse andre typer

(kaldet *friends*), således at de kan få adgang til private attributter og operationer. Dog er denne lempelse statisk. Det vil sige, at den ikke kan ændre sig på køretidspunktet.

I SQL99 simuleres indkapslingen på attributniveau ved hjælp af rettigheder. Det vil sige, at man skal udføre en passende kombination af `GRANT` kommandoer for at definere indkapslingen. Dette adskiller sig radikalt fra den traditionelle måde. Ved at benytte rettigheder opnår man en mere dynamisk håndtering, som i visse situationer kan være brugbar. F.eks. kan man implementere forretningslogik såsom *"alle må læse firmaets salgsstatistikker, men kun salgsafdelingen må opdatere dem*". Dette gøres ved kun at give `EXECUTE` rettigheder til de relevante mutatoroperationer til brugere i salgsafdelingen.

Den oprindelige hensigt med indkapslingen er dog ikke at administrere brugerrettigheder eller forretningslogik. Indkapsling er essentielt for objektorientering, fordi det skjuler den interne repræsentation af indkodningerne af værdierne. Det sikrer, at typen kan præsentere et interface for omgivelserne, som specificerer hvad værdier af typen kan og ikke hvordan de gør det. Hvordan de gør det kan i realiteten ændre sig, uden at omgivelserne behøver at tage sig af det. Netop dette er en af de store styrker i objektorienteret systemudvikling: del og hersk. Faktisk er begrebet *friends* i C++ også et brud på den objektorienterede tankegang. Dog er det ikke helt så grelt, eftersom det er statisk, men set i forhold til teorien om objektorientering burde det ikke være tilladt.

Hvorfor er indkapslingen så defineret på denne måde i SQL99? Der kan være flere grunde. For det første kan det være et ønske om at benytte det allerede eksisterende sikkerhedssystem, eftersom indkapslingsniveauerne kan betragtes som en form for rettighedsniveauer. Derved undgår man at have to slags sikkerhedsmekanismer. For det andet kan det skyldes, at man forestiller sig at langt største delen af typerne, hovedsageligt vil bestå af offentlige attributter og operationer. Derved bliver administrationen ikke så uoverkommelig.

Den manglende indkapsling af typedefinerede tabeller bryder direkte med den objektorienterede model. I de tidligere udkast til standarden (SQL3) blev typedefinerede tabeller defineret ud fra rækketyper. Disse var heller ikke indkapslet. Dette forhold er essensen i den objekt-relationelle model. Det er på dette punkt, at den egentlige forskel mellem den objektorienterede model og den objekt-relationelle model træder frem. Relationen er stadig den centrale byggesten og den har særstatus med hensyn til indkapslingen.

Som tidligere omtalt kan indkapslingen på tabelniveau simuleres ved hjælp af *views* (afsnit 5.3.1 "Indkapsling af typedefinerede tabeller"). Dog havde jeg foretrukket, at indkapslingsniveauerne privat og offentlig var direkte understøttet. Disse kunne f.eks. angives på nedenstående måde.

Eksempel:
```
CREATE TYPE IndkapsletType AS
(  a1    INT PRIVATE,
   a2    INT PUBLIC,
   a3    INT -- default er PUBLIC
);
```

Hvis en attribut angives at være `PRIVATE`, skulle dette resultere i at ingen observatør- og mutatoroperationer blev oprettet for attributten. Den ville derfor kun være tilgængelig for typens interne repræsentation. En forespørgsel på en tabel af typen `IndkapsletType` ville kun returnere kolonnerne `a2` og `a3`, eftersom `a1` er indkapslet. Databasesystemets rettighedssystem kunne stadig bruges til at implementere forretningslogik og generelle sikkerhedsregler.

5.5.3.2 Praktisk

Et praktisk problem med indkapslingsmetoden er, at den er meget kompleks at håndtere. Rettighedsstyring i en database med et komplekst databaseskema er i forvejen en kompliceret affære, som nemt kan føre til fejl. Antag at en given tabel har fem felter. Samles disse i en brugerdefineret type, oprettes der en mutator- og en observatøroperation til hver. Det giver ti operationer per tabel. Et produktionssystem kan nemt indeholde over 100 tabeller. Det giver 1000 operationer, der skal tildeles rettigheder for. De fleste vil formentligt være tilgængelige for alle, men hvis man f.eks. ønsker at implementere forretningslogik ved hjælp af indkapslingen, så kan det blive meget omstændeligt. Administration af indkapslingen kræver desuden et godt kendskab til det konkrete databasesystems brugere og roller. Dette har hidtil været håndteret af databaseadministratoren, men nu skal systemudvikleren også have kendskab til det, for at kunne definere den korrekte indkapsling.

Jeg mener, at de praktiske og teoretiske uhensigtsmæssigheder gør, at indkapslingen er et reelt anklagepunkt mod standarden.

5.5.4 Referencerne

5.5.4.1 Teoretisk

SQL99 gør det muligt at få stor indflydelse på OIDens format. Som beskrevet understøtter den tre variationer:

1. Brugerdefineret: Fungerer som en kunstig primærnøgle som f.eks. kundenummer. Man vælger selv typen og angiver OID værdien hver gang en værdi indsættes.
2. Afledt: I princippet det samme som en primærnøgle. OIDen sammensættes af en del af værdiens indhold.
3. Systemgenereret: Systemet genererer OIDen.

De to første metoder, brugerdefineret og afledt, ligger meget tæt op ad den traditionelle måde at lave nøgler på i relationelle databaser. Den tredje, systemgenereret, understøttes allerede af flere systemer (f.eks. Sybase og NonStop SQL) i form af rækkeidentifikatorer (`rowID`). Det vil sige numre, som automatisk tildeles værdierne, når de indsættes. Dog vil disse systemgenererede OIDer typisk være mere komplekse. F.eks. benyttes Oracles OID til blandt andet at beskrive værdiens placering i databasen. Det sikrer, at værdier hurtigt kan lokaliseres, men det gør det besværligt at flytte værdierne, da alle referencer til værdien så skal opdateres.

Set fra en teoretisk vinkel er implementeringen af referencer udmærket. Dog skal man passe på ikke at undervurdere kompleksiteten af identitetsbaserede referencer (*pointers*). *Pointerchasing* har sine svage sider. De er meget sårbare overfor fejl og gør systemet uoverskueligt. Én fejl kan være, at den refererede attribut (i ovenstående tilfælde navn) skrives forkert, og referencen derved læser fra den forkerte tabel. I stedet for f.eks. navn skulle der måske stå knavn (kunde navn). Havde attributten været fundet ved hjælp af en *join* med eksplicit tabelangivelse, så havde fejlen været fanget. Med sti-udtryk er det kun ét bogstav som adskiller de to udtryk.

De programmeringssprog, som nyder størst fremgang i øjeblikket, er dem, som forsøger at skjule deres brug af referencer (f.eks. Java og Visual Basic). Disse sprog er lettere at anvende, da en hel del faldgruber bliver elimineret. Simplicitet er en stor styrke i systemudvikling og er også en af grundene til, at relationsdatabasen har fået den udbredelse, som den har. At introducere pointers kan være et skridt i den forkerte retning. Dette diskuteres nærmere i afsnit 5.6 ”Dates kritik”.

5.5.4.2 Praktisk

Standarden burde have haft en mere præcis specifikation af formatet for de systemgenererede OIDer. Som minimum burde standarden foreskrive retningslinier for OIDerne. F.eks. ville det have været en fordel, hvis følgende spørgsmål havde været besvaret (min personlige holdning er angivet i parentes og gælder kun systemgenererede OIDer):

1. Kan OIDer være afhængige af værdiens placering? *(nej)*
2. Hvis en værdi *kopieres* ud af databasen, er kopien så at betragte som en ny værdi med et nyt OID? *(ja)*
3. Hvis en værdi *flyttes* ud af databasen, får den så en ny OID når den indsættes igen? *(ja, flytte svarer til at kopiere og slette)*
4. Kan to forskellige værdier i forskellige databaser have samme OID? *(ja[18])*

[18] Det kan hævdes at OIDer bør være globalt unikke, hvilket er praktisk muligt ved at benytte systemgenererede OIDer som delvist er baseret på f.eks. databasesystemets netværks ID.

Endnu et problem med referencerne er, at de nemt kan skabe meget uoverskuelige datasammenhænge, hvis referencer frit kan referere til tupler i forskellige tabeller. Der er delvist taget højde for dette ved at indføre `SCOPE` notationen, hvor en reference kan begrænses til en eller flere tabeller, men den er ikke obligatorisk. Det er derfor muligt at skabe vilkårligt komplekse referencer i databasen, hvilket vil gøre det umuligt for databasesystemet at optimere eventuelle forespørgsler. Netop denne optimering er også en meget central del af de relationelle databasesystemers succes. I modsætning til tidligere generationer af databasesystemer, flyttede de relationelle systemer ansvaret for hastigheden væk fra programmøren og over til databasesystemet. Programmøren skulle blot specificere hvad han ønskede fundet, ikke hvordan han ønskede det fundet. Eftersom en stor del af systemudviklerne, som benyttede databaser, ikke nødvendigvis var eksperter i forespørgselsoptimering, gjorde det arbejdet langt nemmere, hurtigere og mere stabilt[19]. Med de nye referencer kan ansvaret for et hurtigt svar, nemt blive skubbet tilbage til systemudvikleren, hvilket ville være et skridt tilbage. Hastigheden behøver dog ikke altid at blive værre. Hvis programmøren er i stand til at lave en bedre optimering, end systemet ellers ville have kunnet, så opstår problemet ikke. Det kræver dog en vis programmeringserfaring at lave sådanne optimeringer.

5.5.5 Stærkt typekontrol

5.5.5.1 Teoretisk

Den stærke typekontrol, som er blevet indført med de brugerdefinerede typer, har længe været et ønske hos tilhængerne af den rene relationelle model (f.eks. Date [2000]). Stærk typekontrol er generelt et nyttigt redskab i systemudvikling, da det kan hjælpe til at undgå en lang række klassiske fejl. De distinkte typer kan nu løse den opgave, som domæner oprindeligt var tiltænkt.

Dog mener jeg ikke, at der er behov for at skelne mellem `CAST` og `TREAT`. Efter min mening gør de det samme, nemlig typekonverterer. Selvom `TREAT` ikke ændrer på værdiens mest specifikke type, ændrer man stadig den type, som værdien skal optræde med i konteksten. `TREAT` svarer til at skrive følgende i C:

Eksempel:

```
short shrt;
long lng;
lng = -32768;
shrt = (short) lng;
```

[19] Problemet opstår kun når systemet af en eller anden årsag *ikke* vælger at løse forespørgslen på den mest hensigtsmæssige måde. Så kan det være en kamp uden lige at overbevise det om, at det tager fejl, og at det bør løse problemet på en anden måde.

Argumentationen er følgende:

1. short kan betragtes som værende en subtype af long. Alle værdier i short er indeholdt i long og alle operationer på long værdier fungerer også med short værdier.
2. Variabel lng af typen long tildeles en værdi af typen short (den mest specifikke type for værdien -32768 er i C short).
3. Værdien af variablen lng typekonverteres til sin mest specifikke type (short) og tildeles derefter til en variabel af denne type (shrt).

Operationen i sidste linie er i følge C terminologi en typekonvertering, men svarer fuldstændigt til en `TREAT`. Jeg ser ingen grund til at skelne mellem dem. Deres primære funktionalitet er begge at ændre typen på en værdi.

Man kunne argumentere for, at `TREAT` er indkodningsbevarende, hvorimod `CAST` er indkodningsændrende. Men dette passer ikke på distinkte typer. Disse har præcis samme indkodning som deres basistype, men kræver alligevel `CAST` i stedet for `TREAT` for at kunne konverteres.

5.5.5.2 Praktisk

Bagsiden af medaljen ved stærk typekontrol er, at det kræver mere arbejde. Før de kan bruges, skal der oprettes en passende mængde `CAST` operationer, som kan konvertere til de typer, som den skal kunne udveksle værdier med. I stedet for initielt at måtte alt (som med domæner), må man nu intet – på nær udveksling med basistypen. Men når alt er defineret, har man også et bedre og mere sikkert system.

En distinkt type baseret på f.eks. en numerisk basistype vil kun arve sammenligningsoperationerne (>, <, = etc.). Man kunne derfor tro at al aritmetik således skulle implementeres igen for typen, hvis den skulle bruges i beregninger! Dette er (heldigvis) ikke tilfældet, grundet de to implicitte `CAST` operationer. Således vil følgende udtryk (`x` og `y` har den distinkte type `T` med basistypen `INT`):

```
x = y + 1;
```

blive opfattet af systemet som følgende:

```
x = CAST( CAST( y AS INT ) + 1 AS T);
```

De implicitte konverteringer mellem den distinkte type og basistypen gør det nemmere at bruge. Desværre er de også med til at nedbryde den klare forskel fra domænerne. Umiddelbart ligner de distinkte typer domæner, og dette er ikke særlig hensigtsmæssigt set fra et indlæringssynpunkt.

5.6 Dates kritik

Date [2000] har fra starten set skeptisk på konceptet om objektorienterede databasesystemer. Han har peget på en række punkter, som han mener er uhensigtsmæssige. I sin syvende udgave af "An Introduction To Database Systems" fremstiller han de to mest grundlæggende fejl ved de objektorienterede databasesystemer, sådan som han ser dem. Disse betegnes henholdsvis "*The First Great Blunder*" og "*The Second Great Blunder*". Jeg vil i dette afsnit beskrive og diskutere de to "fejl", samt vurdere deres berettigelse i forbindelse med SQL99.

5.6.1 "The First Great Blunder"

Dette anklagepunkt omhandler forholdet mellem relationer og klasser. I såvel SQL99 som i de "rene" objektorienterede databaser sidestilles de to. Date mener, at det er en grundlæggende fejl, hovedsageligt fordi en relation er en variabel, og en klasse er en type.

> "… **What concept is it in the relational world that is the counterpart to the concept *object class* in the object world?**
> … there are two equations that can be, and have been, proposed as answers to this question:
> • domain = object class
> • relvar = object class
> We now proceed to argue, strongly, that the first of these equations is right and the second is wrong.
> In fact, of course, the first equation is *obviously* right, since object classes and domains are both just types. Indeed, given that relvars are *variables* and classes are *types*, it should be immediately obvious too that the second equation is wrong (variables and types are not the same thing); for this very reason. *The Third Manifesto* [3.3] asserts categorically that relvars are not domains. Nevertheless, many people, and some products, have in fact embraced the second equation—a mistake that we refer to as **The Great Blunder** (or, more precisely. The *First* Great Blunder, since we will meet another one later). …" (kursiv og fremhævet skrift i originalen)
>
> [Date, 2000]

Date bruger betegnelsen **relvar**, fordi betegnelsen **relation** ofte bruges både om selve tabellen (variablen) og om dens indhold (værdierne). For at skelne kaldes værdierne for "relationen" og variablen for "relvar".

Han har ret i at variable og typer ikke kan sammenlignes, men påstanden "*relvar = object class*" er formentligt ikke det, som fortalerne for objektorienterede databaser påstår. Når en tabel oprettes med SQL angives type og variabel på samme tid. Typen

er derfor ”navneløs”. Når en type oprettes i objektorienterede databasesystemer, som f.eks. Jasmine, så angives både typens navn (klassenavnet), samt navnet på typens *extent*. Dette indeholder samtlige oprettede værdier af klassen, og kan derfor sidestilles med relationsvariablen. Den sammenligning som foretages er derfor ikke ”*relvar = object class*” men derimod følgende:

relation **type**	svarer til	object **class**
relation **variable**	svarer til	object class **extent**

At alle klasser har et tilhørende *extent*, bliver ofte ikke fremhævet særlig meget i litteraturen omkring de ”rene” objektorienterede databaser. I det oprindelige forslag til objektorienterede databaser [Atkinson, 1993] omtales *extent* heller ikke som noget centralt for databaserne. De kan benyttes, hvis det har en betydning, ellers kan de ignoreres.

> ” We do not, however, feel that it is necessary for the system to automatically maintain the extent of a type (i.e., the set of objects of a given type in the database) or, if the extent of a type is maintained, for the system to make it accessible to the user. Consider, for example, the rectangle type, which can be used in many databases by multiple users. It does not make sense to talk about the set of all rectangles maintained by the system or to perform operations on them. We think it is more realistic to ask each user to maintain and manipulate its own set of rectangles. On the other hand, in the case of a type such as employee, it might be nice for the system to automatically maintain the employee extent.”
>
> [Atkinson, 1993]

Den lidt vage behandling af begrebet skyldes, at de ”rene” objektorienterede databaser ikke har relationen som noget centralt element. Her er det kun klasserne, som er de centrale byggesten. Derfor kan begrebet nemt overses, men det ændrer ikke på, at *extents* er det tætteste man kommer på en relation i det objektorienterede paradigme.

De uklare definitioner af de centrale begreber er formentligt årsagen til misforståelserne Date og ODMG imellem, men helt personligt ser jeg også Dates påstand som værende en smule bevidst fordrejet. Det virker som om, at han generelt ikke ønsker at bryde med den relationelle model. Det er der sikkert mange gode grunde til ikke at gøre, men ”*The First Great Blunder*” er, så vidt jeg kan se, ikke en af dem.

5.6.2 ”The Second Great Blunder”

Den anden centrale fejltagelse, som Date mener at de objektorienterede databasesystemer begår, er at blande pegere (pointers) og relationer sammen. Hvor

hans forrige anklage går på sammenblandingen af typer og variable (klasser og relationsvariable), så baserer han denne på variable og værdier.

Hans anklage er, at introduktionen af OIDer bryder med det grundlæggende koncept i de relationelle databaser. I stedet for at benytte primærnøgler kan tupler nu identificeres ved hjælp af deres OID. Date mener, at det er grundlæggende forkert at give tupler OIDer eftersom OIDer svarer til adresser. Tupler er værdier, og værdier har per definition ikke adresser.

Som omtalt tilbage i afsnit 3.1.5 ”Objekt” er mit bud, at værdier ikke kan have adresser, men deres indkodninger kan godt. Dog er det helt klart et brud med den relationelle model, eftersom tuplernes identifikation skifter fra at være værdibaseret til identitetsbaseret. Date nævner også at en øget kompleksitet vil følge, når pegere benyttes i stedet for værdibaserede referencer. Dette anklagepunkt er jeg helt enig i. En af grundede til at de relationelle databaser har vundet så stort indpas er formentligt, at de er forholdsvis nemme er benytte og forstå. De udviklere, som arbejder med databaser, har det sjældent som deres primære fagområde. Det er nærmere en service, som de har brug for i deres applikationer. Kræver denne service alt for meget uddannelse og har den for stor kompleksitet, vil det uden tvivl være bremsende for produktiviteten.

5.7 Delkonklusion

Brugerdefinerede typer og subtyper er godt understøttet. Det er muligt at lave komplette typehierarkier både på tabelniveau og på attributniveau. Dog er kun enkel nedarvning understøttet.

Det kan diskuteres, hvorvidt indkapsling er understøttet af objektmodellen eller ej. Standarden påstår selv, at den understøtter indkapsling, men kræver samtidig to implicit definerede operationer på hver attribut, som bryder med indkapslingen. Disse kan så styres ved hjælp af rettigheder, men løsningen er ikke god. Den er besværlig at administrere og kræver unødigt meget viden om databasesystemets rettighedssystem med brugere og roller af den systemudvikler, som skal definere indkapslingen.

På tabelniveau findes ingen indkapsling, men dette skyldes at SQL99 er en objekt-relationel model, hvor tabeller har særstatus. Det er på dette punkt, at modellen klart adskiller sig fra den objektorienterede model. Dog mener jeg, at en vis form for indkapsling, godt kunne have været understøttet, eventuelt som et alternativ. Indkapslingen kan alligevel simuleres ved hjælp af *views*.

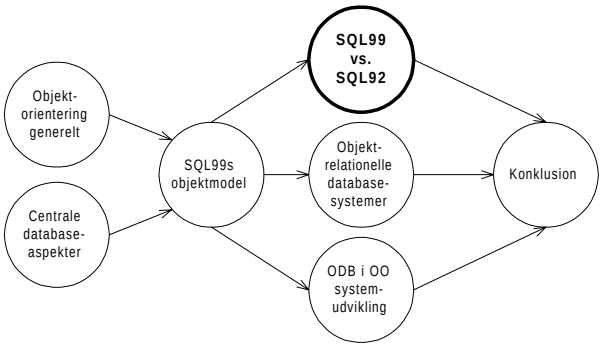
Referencerne er generelt godt understøttet, men den nuværende definition indeholder visse faldgruber, eftersom der hersker en del uklarhed omkring systemgenererede OIDer. Det er desuden muligt at lave meget komplekse systemer af referencer, som kan forhindre en optimal hastighed ved forespørgsler, med mindre at programmøren er i stand til at løse problemet selv.

De centrale egenskaber for SQL99 er opsummeret her:

- Distinkte typer
- Strukturerede typer
- Unavngivne strukturerede typer
- Stærk typekontrol
- Subtyper
- Abstrakte typer
- Referencer
- OIDer: system genererede / afledte / brugerdefinerede
- Operationer
- Selv-reference i typens operationer
- Brugerdefinerede sammenligningsoperationer
- Brugerdefinerede typekonverteringer
- Typedefinerede tabeller
- Tabelhierarkier
- Polymorfi på tabelniveau
- Polymorfi på attributniveau
- Simuleret indkapsling ved hjælp af rettigheder

Objektmodellen nærmer sig meget den model, som benyttes af ODMG. Dog vil det formentligt kræve en del af de nuværende relationelle databasesystemer, før de kan understøtte alle faciliteterne, herunder specielt polymorfi på attributniveau.

6 SQL99 vs. SQL92



I dette afsnit vil udtrykskraften i SQL99 blive analyseret i forhold til SQL92. Seks typiske SQL92 konstruktioner vil blive gennemgået og sammenlignet med en tilsvarende løsning i SQL99. Løsningen i SQL99 vil tilstræbe at benytte de objektorienterede faciliteter. Eftersom SQL92 er en delmængde af SQL99[20], ville samtlige seks SQL92 konstruktioner kunne udføres i SQL99. Dog har denne sammenligning til formål at illustrere eventuelle styrker og svagheder i den objektorienterede model. Eksemplerne er på ingen måde komplette og viser ikke samtlige faciliteter ved SQL99s objektmodel. F.eks. gennemgås polymorfi i forbindelse med attributter ikke. Dette er en klar styrke i SQL99, men det har ingen pendant i SQL92s funktionalitet.

Et gennemgående eksempel vil blive brugt i afsnittet.

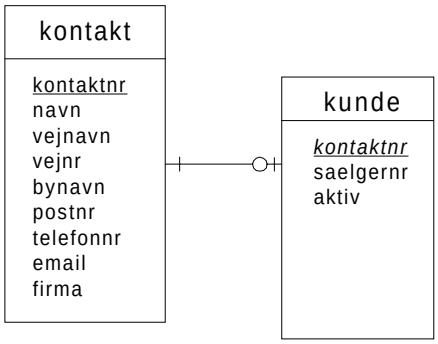
6.1 Oprettelse af tabeller

6.1.1 SQL92

Når en ny tabel oprettes i SQL92 angives både dens type og navnet på relationsvariablen. Eventuelle referencer til rækker i andre tabeller skabes ved hjælp af fremmednøgler.

Tabellerne vil blive brugt som basis for de efterfølgende eksempler. De er skabt efter dette design:

[20] SQL99 indeholder visse inkompatibiliteter med SQL92, men disse er ganske få. Se eventuelt appendix D i [Gulutzan & Pelzer, 1999].



Understregningangiver primærnøgle

Kursiv angiver fremmednøgle

angiver en "1 til 0 eller 1" reference

Tabellerne oprettes på følgende måde:

```
CREATE DOMAIN TelefonNrType NUMERIC(10,0);

CREATE TABLE kontakt
(  kontaktnr INT,
   navn      VARCHAR(60) NOT NULL,
   vejnavn   VARCHAR(60),
   vejnr     VARCHAR(20),
   bynavn    VARCHAR(60),
   postnr    NUMERIC(4,0),
   telefonnr TelefonNrType,
   email     VARCHAR(40),
   firma     SMALLINT,
   PRIMARY KEY (kontaktnr)
);

CREATE TABLE kunde
(
   kontaktnr INT REFERENCES kontakt(kontaktnr) ON DELETE CASCADE,
   saelgernr INT REFERENCES kontakt(kontaktnr) ON DELETE SET NULL,
   aktiv     SMALLINT,
   PRIMARY KEY (kontaktnr)
);
```

Ovenstående er et eksempel på et udsnit af et databaseskema for en virksomhed. Den indeholder en tabel af personer eller firmaer, som firmaet har kontakt til (medarbejdere, kunder, freelancere, aktionærer etc.). Hver kategori vil typisk have en separat tabel med yderligere oplysninger, men kun tabellen kunde er medtaget for overblikkets skyld. Basen kunne have været designet på et utal af andre måder, som hver ville have haft sine styrker og svagheder.

6.1.1.1 Motivation for den valgte løsning

Denne løsning er ikke den pæneste set fra et teoretisk synspunkt, fordi den benytter kunstige nøgler (kontaktnr). Det havde rent teoretisk været pænere at skabe en

primærnøgle for tabellen kontakt, som var en passende kombination af dens attributter, f.eks. navn, adresse og firma. Dog benyttes kunstnøglemetoden oftere i praksis. Dette skyldes flere grunde:

1. De er lettere at lave end at skulle udtænke en passende unik kombination af attributter, som alle attributterne er afhængige af.
2. Fremmednøgler bliver simplere, eftersom komplekse (sammensatte) primærnøgler ikke skal kopieres med rundt i databaseskemaet.
3. Som en konsekvens af punkt 2 bliver tabellerne mindre (færre attributter). Dette gør dem mere overskuelige og formentligt også en smule hurtigere.
4. Forespørgsler bliver kortere og enklere. Specielt når to eller flere tabeller samles i et join.
5. Hvis attributter som indgår i en primærnøgle skal rettes, kræver det at rækken slettes og indsættes igen.
6. Ændring af typen på en attribut, som indgår i en primærnøgle, vil påvirke alle tabeller i databasen, hvor der refereres til tabellen.

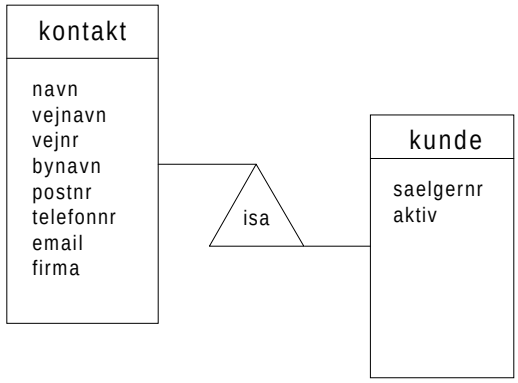
En af de få ulemper ved metoden er, at systemet fyldes med intetsigende id-numre, som kræver yderligere opslag i databasen før de giver mening. Dog er det mit klare indtryk, at langt de fleste databaseskemaer, som designes i dag, benytter kunstnøglerne, da fordelene opvejer ulemperne.

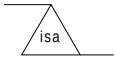
Ikke alle id-numre kan klassificeres som værende kunstnøgler. F.eks. er ordrenumre og varenumre ikke kunstnøgler. Det skyldes, at nummeret også har en betydningen udenfor databasen. De repræsenterer entiteter. Det gør f.eks. et adressenummer ikke. Det skal omsættes til den relevante adresseinformation, før det giver mening udenfor databasen. Dog nyder de samme fordele som kunstnøglerne. Man kan samle begge slags nøgler under betegnelsen **skalare nøgler**. Det vil sige nøgler, hvis værdier kan repræsenteres med en enkelt skalar værdi.

Metoden ligger meget tæt op ad de nye identitetsbaserede referencer, som benyttes i SQL99. Ved at vælge de skalare nøgler frem for de ”pæne” sammensatte primærnøgler, er det min overbevisning at SQL92 vil klare sig bedre i sammenligningen med SQL99. Var de skalare nøgler ikke blevet brugt, så havde ovenstående seks fordele kunne tilskrives SQL99s objektorienterede funktionalitet, hvilket ville have givet et forkert billede.

6.1.2 SQL99

I modsætning til SQL92 har SQL99 kunstnøglemetoden indbygget. Attributten kontaktnr kan nu erstattes af OIDer. Det objektorienterede design ser således ud:





angiver en subtype (notation lånt fra E/R modellering)

Tabellerne oprettes på følgende måde. Bemærk at definitionen på typerne og tabellerne nu er adskilt.

```
CREATE TYPE TelefonNrType AS NUMERIC(10,0) FINAL;

CREATE TYPE KontaktType AS
(  navn        CHAR(60),
   adresse     ROW ( vejnavn   VARCHAR(60),
                     vejnr     VARCHAR(20),
                     bynavn    VARCHAR(60),
                     postnr    NUMERIC(4,0)
                   ),
   telefonnr   TelefonNrType,
   email       CHAR(40),
   firma       BOOLEAN     -- ny prædefineret type i SQL99
) NOT FINAL;

CREATE TABLE kontakt OF KontaktType
(  REF IS kontaktID,          -- OIDen erklæres
   navn WITH OPTIONS NOT NULL -- navn angives som værende NOT NULL
);

CREATE TYPE KundeType UNDER KontaktType AS
(  saelgernr   REF(KontaktType) SCOPE kontakt,
   aktiv       BOOLEAN
) NOT FINAL;

CREATE TABLE kunde OF KundeType UNDER kontakt;
```

Hver række i tabellerne kunde og kontakt får automatisk genereret et OID. Dette gemmes som rækkens første attribut. F.eks. har tabellen kontakt følgende felter:

KontaktID	navn	adresse				telefonnr	email	firma
		vejnavn	vejnr	bynavn	postnr			

Bortset fra det sammensatte adressefelt, ligger dette meget tæt op ad SQL92, hvis man benytter kunstnøgler.

Det helt nye her er subtabellerne. Det er muligt at skabe et hierarki af tabeller. Kravet er, at hierarkiet svarer til det tilsvarende typehierarki, som tabellerne er baseret på. Det vil sige, kontakt kunne ikke være en subtabel af kunde, eftersom det er kunde, som nedarver fra kontakt. En forespørgsel på supertabellen kontakt vil også returnere alle kunder. Dette vil blive beskrevet nærmere i et eksempel 6.3.2 ”Forespørgsel med polymorfi”.

Som det er vist, er det muligt at påhæfte attributterne diverse `OPTIONS`, når tabellen oprettes. Det er således muligt at lave flere tabeller ud fra samme type, som har forskellige begrænsninger (*constraints*), default værdier etc. for attributerne. Det kan sågar angives, at en attribut skal være primærnøgle (dog ikke hvis tabellen er en subtabel).

Bemærkning: Eksemplet i SQL92 er konstrueret således at SQL99 løsningen både kunne demonstrere nedarvning og referencer. Eksemplet skal ikke forstås således, at nedarvning er den nye måde at lave referencer på. Tværtimod bør man overveje grundigt, hvornår man skal bruge nedarvning, og hvornår man skal bruge referencer. Dette emne diskuteres nærmere i afsnit 8.2 ”Objektorienteret design”.

6.1.3 Vurdering

De nye faciliteter kan ikke vurderes alene på dette eksempel. Referencer og tabelhierarkier viser først deres styrke i forespørgselseksemplet. Det lader ikke til, at de nye faciliteter i SQL99 komplicerer tabeloprettelsen væsentligt. Det er klart, at med nye muligheder skabes også større kompleksitet, og SQL99 syntaksen er da også en del mere uoverskuelig end SQL92. Havde typerne også haft operationer tilknyttet, havde syntaksen været væsentligt mere kompleks.

Dog mener jeg, at de unavngivne strukturerede typer, som f.eks. adresse kolonnen, er en syntaksmæssig forbedring i forhold til SQL92. Det åbner mulighed for at gruppere attributterne i logiske enheder. Visse tabeller har en tendens til at indeholde mange attributter. Med denne gruppering vil det være muligt at øge læsbarheden. Dog er ændringen ikke rent syntaksmæssig. Ønskes adressen udvalgt ved en forespørgsel, skal blot én attribut angives i stedet for fire. Man er også sikret, at man får alle relevante attributter med.

6.2 Indsættelse af værdier

6.2.1 SQL92

Indsættelse af værdier i tabeller i SQL92 er forholdsvis enkelt. Det er muligt at angive alle værdier direkte, eftersom referencerne (fremmednøglerne) også er værdibaserede i modsætning til REFs i SQL99, som er identitetsbaserede.

```
INSERT INTO kontakt
          (kontaktnr,navn,vejnavn,vejnr,bynavn,postnr,
           telefonnr,email,firma)
   VALUES (100,”Hans Hansen”,”Østergade”,”1”,”Østerby”,2222,
           11223344,”hans@hansen.dk”,0);

INSERT INTO kontakt
          (kontaktnr,navn,vejnavn,vejnr,
           bynavn,postnr, telefonnr,email,firma)
   VALUES (101,”Jens Jensen”,”Ny Østergade”,”1, st.tv.”,
           ”Ny Østerby”,4444,55667788,”jens@jensen.dk”,0);

INSERT INTO kunde
          (kontaktnr,sælgernr,aktiv)
   VALUES (101,100,1); -- J.J. er kunde med H.H. som sælger
```

6.2.2 SQL99

Samme eksempel med SQL99s objektorienterede faciliteter ser således ud:

```
INSERT INTO kontakt
          (navn,
           adresse,
           telefonnr,email,firma)
   VALUES (”Hans Hansen”,
           ROW(”Østergade”,”1”,”Østerby”,2222),
           11223344,”hans@hansen.dk”,FALSE);

INSERT INTO kunde
            (navn,
             adresse,
             telefonnr,email,firma,saelgernr,aktiv)
  SELECT ”Jens Jensen”,
         ROW(”Ny Østergade”,”1”,”Ny Østerby”,4444),
         55667788,”jens@jensen.dk”,FALSE,kontaktID, TRUE
  FROM kontakt
  WHERE navn = ”Hans Hansen”;
```

En af de afgørende forskelle i forhold til SQL92 er, at referencen til ”Hans Hansen” ikke kan angives direkte, men skal udtrækkes ved hjælp af en SELECT-kommando.

Det sammensatte adressefelt konstrueres med det reserverede ord ROW.

Bemærk at telefonnumrene kan angives direkte som numeriske værdier. Dette skyldes de to implicitte CAST operationer, som konverterer mellem TelefonNrType og NUMERIC(10,0). Effekten er, at syntaksen for indsættelse af værdier i domæner og distinkte typer er ens.

6.2.3 Vurdering

Indsættelsen af ”Jens Jensen” ved hjælp af en `SELECT`-kommando er lidt mere kompleks, men gør samtidig, at han kan oprettes med én `INSERT` kommando. Dette må siges at være mere intuitivt, end at skulle oprette ham to steder. Når ”Jens Jensen” indsættes i tabellen kunde, oprettes han automatisk også i kontakt.

En tabel, med mange referencer eller med referencer som kræver komplekse forespørgsler, vil kunne generere meget komplekse `INSERT` kommandoer. Man må derfor forvente at brugen af referencer til en vis grad vil komplicere indsættelsen af nye data.

6.3 Udvælgelse af værdier

Dette afsnit vil blive opdelt lidt anderledes end de forrige. For at illustrere forskellene er det nødvendigt at lave flere forskellige forespørgsler. Afsnittet er derfor opdelt efter disse.

6.3.1 Forespørgsel med referencer

I dette eksempel vises en forespørgsel, som trækker data på tværs af tabellerne kunde og kontakt. Forespørgslen skal løse opgaven: ”Find samtlige aktive kunder som har Hans Hansen som sælger”.

6.3.1.1 SQL92

Forespørgslen kan i SQL92 løses ved hjælp af *join*:

```
-- k=kunde, kk=kundekontakt, sk=saelgerkontakt
SELECT k.navn
FROM kunde k, kontakt kk, kontakt sk
WHERE k.kontaktnr = kk.kontaktnr AND k.saelger = sk.kontaktnr AND
        sk.navn = ”Hans Hansen” AND k.aktiv = 1;
```

Netop i dette eksempel giver kunstnøgledesignet bagslag. Det er nødvendigt at *joine* tabellen kontakt med sig selv for at afgøre hvilke kunder, som har ”Hans Hansen” som tilknyttet sælger. Havde vi brugt primærnøgler, som udelukkende var sammensat af værdibærende attributter, så havde sælgerens navn formodentligt været en del af nøglen. Derved havde man umiddelbart kunnet afgøre om en kunde havde en given sælger tilknyttet, uden at det krævede yderligere opslag. Havde man designet databasen anderledes, havde man sparet en *join* her. Derved havde forespørgslen formentligt kørt hurtigere, fordi kun et tabelopslag ville have været nødvendigt.

6.3.1.2 SQL99

I SQL99 kan man drage nytte af referencerne til at lave en kortere løsning med sti-udtryk:

```
SELECT k.navn
FROM kunde k
WHERE k.saelger->navn = ”Hans Hansen” AND k.aktiv;
```

Referencen k.sælger kan nemt derefereres ved hjælp af operatoren **->**. Derved undgår man at lave et *join* mellem de to tabeller. Attributten k.aktiv indeholder enten sandhedsværdien TRUE eller FALSE. Hvis attributten er NULL er dens sandhedsværdi UNKNOWN.

6.3.1.3 Vurdering

Forespørgslen i SQL99 er meget enklere end den tilsvarende i SQL92. Dog er selve referencerne ikke årsagen til, at den er enklere. Årsagen ligger i den egenskab, som er blevet tilføjet, nemlig **->** operatoren. Der er intet til hinder for, at samme operator også kunne fungere ved hjælp af referenceintegriteten mellem fremmednøgler og primærnøgler. Databasen i SQL92-design indeholder al den nødvendige information, for at samme syntaks kunne fungere der (bortset fra at SQL99 understøtter typen BOOLEAN).

Hvis OIDerne direkte angiver, hvor en række er placeret, kunne man forestille sig, at SQL99s sti-udtryk ville være hurtigere end den tilsvarende løsning med *join*. Dog er dette ikke nødvendigvis tilfældet. Det skyldes en helt generel ulempe ved referencerne, som kan illustreres med følgende forespørgsel mellem to uafhængige tabeller:

Eksempel:
```
CREATE TYPE Type1 AS (a INT);
CREATE TABLE t1 OF Type1;

CREATE TYPE Type2 AS (b REF(Type1));
CREATE TABLE t2 OF Type2;

SELECT b
FROM t2
WHERE t2.b->a = 1;
```

I eksemplet oprettes to uafhængige tabeller. Tabellen t2 består af referencer til rækker i t1. I forespørgslen ønskes en liste over alle referencer, som peger på en række i t1, som indeholder værdien ”1”. Lad os antage at antallet af rækker i de to tabeller er 10 for t1 og 10.000 for t2[21]. Forespørgslen tager udgangspunkt i t2, men i det tilfælde hvor der kun findes en eller to rækker i t2, som rent faktisk peger på en række i t1 med værdien ”1”, så bliver svartiden meget dårlig. Alle 10.000 rækker i t2 skal gennemlæses og referencen slås op, før systemet kan afgøre, om rækken skal med eller ej. Et normalt relationelt system ville have mulighed for at tage den mindste tabel

[21] Konstruktionen er ikke så urealistisk endda. Hvis t2 er en ordretabel og t1 er en stamdatatabel, kunne forholdet sagtens forekomme.

først og så foretage opslaget den anden vej. Dette ville dog kræve, at der var et indeks på fremmednøglen i t2. Men det afgørende er, at det er muligt at optimere forespørgslen med et indeks. Det er det ikke med de identitetsbaserede referencer i SQL99, eftersom en given række ikke ved hvem, som refererer til den.

Problematikken er ikke ny. De **hierarkiske databasesystemer** har samme problem. Her er alle data ordnet i hierarkier, og det er kun muligt at foretage søgninger fra et givet punkt i hierarkiet og ned efter. En forespørgsel i stil med ”hvilke knuder har underliggende knuder, som opfylder et givent kriterie” ville kræve at samtlige aktuelle knuder blev gennemlæst og deres underliggende knuder undersøgt. Den relationelle model gav en ny måde at designe databaseskemaet på, som indeholdt bidirektionale referencer. Det var således muligt at foretage forespørgsler begge veje med en hurtig svartid. Dette var en af årsagerne til at de relationelle databasesystemer blev mere populære end de hierarkiske. Med introduktionen af referencer er de relationelle systemer på vej til at lave samme fejl som de hierarkiske. Det er selvfølgelig stadig muligt at foretage *join*-operationer mellem tabeller, men det kræver at databasen designes ved hjælp af værdibaserede referencer. Identitetsbaserede referencer bør kun benyttes i de tilfælde, hvor man er sikker på, at man næsten aldrig har brug for at gå mod referencens retning. Det kræver med andre ord en vis forudseenhed, hvilket ikke er nemt. I de relationelle systemer kan problemet nemt løses ved blot at lægge et indeks på fremmednøglen. Så kan systemet foretage opslag begge veje med optimal hastighed, og dette kræver ikke nogen form for forudseenhed.

6.3.2 Forespørgsel med polymorfi

I begge modeller kan kunder også betragtes som kontakter. Denne form for polymorfi er simuleret i SQL92 designet, men er direkte understøttet i SQL99. Forespørgslen som skal løses er ”List samtlige kontakter. Hvis en kontakt også er en kunde, så skal den kundespecifikke information også medtages”.

6.3.2.1 SQL92

Til at løse denne opgave benyttes et `NATURAL LEFT OUTER JOIN`, som foretager et join på de to kontaktnr kolonner. `LEFT` sikrer at alle rækker i kontakt medtages uanset om de har en tilsvarende række i kunde. Hvis de ikke har, indsættes `NULL` i kolonnerne saelger og aktiv.

```
SELECT *
FROM kontakt NATURAL LEFT OUTER JOIN kunde;
```

Løsningen er rimelig enkel, men kræver at man har nogenlunde styr på de forskellige former for *join*.

6.3.2.2 SQL99

Grundet den indbyggede polymorfi er alle kunder også med i tabellen kontakt. Derfor burde det være nok at udvælge fra denne tabel.

```
SELECT *
FROM kontakt;
```

Præcist hvad resultatet af ovenstående forespørgsel bliver, er ikke helt klart. Det er sikkert nok, at rækker fra både kontakt og kunde medtages. Men spørgsmålet er hvilke kolonner, der medtages. Man kan forestille sig to resultater.

1. Alle kolonner som findes i kontakt medtages.
2. Hver række i resultatet får samtlige kolonner med, som er specificeret for dem. Det vil sige kunder får også saelger og aktiv kolonnerne med. Dette vil resultere i en resultatmængde, hvor alle rækker ikke har samme antal kolonner – såkaldte flossede rækker (*jagged rows*).

Det er ikke lykkedes at finde et sted i standarden, som omtaler dette punkt. Den sekundære litteratur bevæger sig også udenom [Date, 2000] [Gulutzan & Pelzer, 1999]. Eftersom standarden ikke omtaler emnet, må man gå ud fra at løsning nummer 1 er den korrekte. Et så afgørende brud med den relationelle model, som løsning 2 er, ville formentligt være beskrevet grundigt.

Hvis løsning nr. 1 antages at være rigtig, løser forespørgslen ikke opgaven. En alternativ løsning kunne være:

```
SELECT *
FROM ONLY(kunde)
UNION
SELECT k.*, NULL, NULL
FROM ONLY(kontakt) k;
```

SQL99 introducerer et nyt reserveret ord `ONLY`, som angiver at kun rækker, som har tabellens type som sin mest specifikke, skal medtages.

6.3.2.3 Vurdering

Vurderingen af SQL99 i dette spørgsmål afhænger af hvilken af de to løsninger i forrige afsnit, som er den korrekte. Enten er løsningen enkelt og elegant (første forslag) eller også er den lidt mere kompliceret end den tilsvarende for SQL92 (andet forslag).

For at udnytte det objektorienterede paradigmes styrker, som f.eks. polymorfi, er det nødvendigt, at det implementeres fuldstændigt. Halve løsninger vil ofte ikke give de fordele, som er nødvendige, for at skiftet har sin fulde berettigelse.

6.4 Sletning af værdier

6.4.1 SQL92

Sletning af en given række i SQL92 er enkelt, hvis man har værdien af primærnøglen.

```
DELETE FROM kontakt
WHERE kontaktnr = 101;
```

”Jens Jensen” slettes som kontakt. Hans tilsvarende række i tabellen kunde slettes også på grund af at `ON DELETE CASCADE` blev angivet, da kunde tabellen blev oprettet.

6.4.2 SQL99

Det er ikke muligt at indtaste OIDer direkte i en forespørgsel. Man må derfor benytte nogle af de andre felter i tabellen. Problemet med denne løsning er, at man ikke kan være sikker på, at det kun er den rigtige række, som slettes. Hvis flere f.eks. hedder ”Jens Jensen”, og man bruger dette som eneste søgekriterie, så kan man nemt også få slettet andre rækker.

```
DELETE FROM kontakt
WHERE navn = ”Jens Jensen”;
```

Dette problem opstår kun, hvis OIDerne er systemgenererede som i dette tilfælde. Havde de enten været sammensat af attributter (*composite*) eller brugerdefineret (*user-defined*), så havde det ikke været noget problem. Se eventuelt afsnit 5.4.1 ”Repræsentation af referencer”.

”Jens Jensen” er både en kontakt og en kunde. Derfor er det muligt at slette ham i begge de tilsvarende tabeller. Uanset hvilken man vælger, forsvinder han automatisk fra den anden også. Ovenstående kunne altså også have været udført således:

```
DELETE FROM kunde
WHERE navn = ”Jens Jensen”;
```

6.4.3 Vurdering

Som det kan ses, er der ikke den store forskel. Dog er det besværligt at sikre sig at kun den rigtige række bliver slettet, hvis ikke man har OIDen. Den kan ikke indtastes direkte, så den skal trækkes ud af databasen med en forespørgsel. Det er muligt at definere en primærnøgle for tabellen. Dette vil gøre det enklere at sikre sig, at kun den korrekte række slettes, men grundlaget for OIDen forsvinder, hvis en primærnøgle alligevel er nødvendig.

Problemet her er ikke selve sletning af rækken. Det er udvælgelsen af den korrekte række. Præcis det samme anklagepunkt gør sig gældende i forbindelse med almindelige forespørgsler. Her vil det samme problem opstå, hvis man forsøger at udvælge en bestemt række.

Man skal være omhyggelig med at definere de korrekte begrænsninger på attributterne i SQL92. Havde kontaktnr i tabellen kunde brugt `ON DELETE SET NULL` i stedet for `ON DELETE CASCADE`, så ville "Jens Jensen" ikke være blevet slettet fra tabellen kunde, hvilket ville være uhensigtsmæssigt. Den slags problemer undgår man i SQL99, da man ikke kan fjerne en tupel "halvt". Enten fjernes den fra begge tabeller, eller også fjernes den ikke. Det er en styrke i denne sammenhæng, men giver problemer i andre. Dette beskrives i næste afsnit om "Opdatering af værdier".

6.5 Opdatering af værdier

En simpel `UPDATE` i SQL99 lider under præcis de samme problemer, som blev illustreret i forrige afsnit for `DELETE`. I dette afsnit vil jeg i stedet koncentrere mig om en lidt anderledes opdatering af data end blot en `UPDATE` kommando.

I stedet for at lave en "værdiopdatering" vil jeg lave en "typeopdatering". En entitet repræsenteret i databasen skifter type, og de tilhørende opdateringer skal foretages. Der er ikke tale om at ændre på selve databaseskemaet, men om en omstrukturering af data. Opgaven er følgende: "Hans Hansen har skiftet job og går derfor fra at være sælger til at være kunde". Entiteten "Hans Hansen" skifter type fra SaelgerType til KundeType. Dette kaldes også for typemigrering.

6.5.1 SQL92

Migreringen foregår ved at oprette "Hans Hansen" i tabellen kunde og slette ham fra tabellen saelger[22].

```
DELETE FROM saelger
WHERE saelgernr = 100;

INSERT INTO kunde
        (kontaktnr,saelgernr,aktiv)
  VALUES (100,NULL,1); -- H.H. er kunde med ukendt sælger
```

6.5.2 SQL99

Typemigreringen i SQL99 minder en del om fremgangsmåden i SQL92, men der er nogle afgørende forskelle. Problemet er, at SQL99 ikke understøtter dynamisk typemigrering. Således har værdien "Hans Hansen" typen KontaktType, som sin mest specifikke type, og dette kan ikke ændres. Det er derfor nødvendigt at slette ham som kontakt og genindsætte ham som kunde.

[22] Denne er ikke tidligere blevet defineret, men dens struktur er ikke særlig afgørende for eksemplet. Attributten "saelgernr" er primærnøglen, som refererer til kontaktnr i tabellen "kontakt".

```
DELETE FROM kontakt
WHERE navn = ”Hans Hansen”;

INSERT INTO kunde VALUES
          (navn,
           adresse,
           telefonnr,email,firma)
   VALUES (”Hans Hansen”,
           ROW(”Østergade”,”1”,”Østerby”,2222),
           11223344,”hans@hansen.dk”,FALSE);
```

Syntaksmæssigt ser det ikke ud til at have den store forskel fra SQL92, men problemet er, at ”Hans Hansen” nu tildeles et andet OID, end han havde tidligere, fordi det er systemgenereret. Alle andre steder, som har referencer til ”Hans Hansen”, skal således også opdateres i databasen.

6.5.3 Vurdering

Typemigrering er et generelt problem for systemer med typehierarkier. Hvis de ikke understøtter dynamisk typemigrering, er det nødvendigt at slette og genoprette repræsentationen af entiteten i databasen. Dette er meget uhensigtsmæssigt, hvis man benytter systemgenererede OIDer. De ”rene” objektorienterede databaser har samme problem. Der er ingen let løsning udover dynamisk typemigrering. I en tidligere udgave af ODMGs forslag (version 1.2 [Cattell, 1995]) var dynamiske typer medtaget som et ønske til fremtidige udgaver af ”standarden”[23]. Dette punkt blev dog droppet i version 2.0 af den foreslåede standard [Cattell & Barry, 1997]. Dynamisk typemigrering diskuteres nærmere i afsnit 8 ”ODB i OO systemudvikling”.

6.6 Normalformer

Normalformer og relationelt databasedesign er centrale emner i den relationelle model. I forbindelse med sammenligningen af SQL92 og SQL99 kunne det være nærliggende at lave en tilsvarende sammenligning af relationelt design og objektorienteret design. Dog adskiller disse to modeller sig så grundlæggende fra hinanden, at en sådan sammenligning ikke ville være hensigtsmæssig. I forbindelse med SQL99 og SQL92 er der heller ikke tale om to helt adskilte paradigmer. Objektmodellen i SQL99 kan betragtes som en overbygning til SQL92. Et design kan derfor både indeholde elementer fra den relationelle model og fra objektmodellen. De to er ikke gensidigt udelukkende, og foreningen vil resultere i et objekt-relationelt design.

Specifikt omkring normalformer kan to egenskaber i SQL99s objektmodel dog nævnes, som bryder med disse.

[23] I citationstegn, eftersom den ikke er vedtaget officielt.

1. Indførelsen af identitetsbaseret identifikation i form af OIDer bryder med det traditionelle værdibaserede nøglebegreb.
2. Komplekse attributtyper (f.eks. brugerdefinerede typer og ARRAY) bryder med 1. normalform.

Dog tillader SQL99, at OIDer afledes af attributterne i den brugerdefinerede type. Derved ændres der ikke væsentligt på designet i forhold til SQL92.

En anden forskel er referencerne. Disse giver en lidt anderledes måde at designe databasen på. Dog skal man være varsom med at benytte dem, som vist i et tidligere eksempel. De kan føre til meget lange svartider på forholdsvis enkle forespørgsler.

Den største ændring er brugen af nedarvning i databasedesignet. Dog skal man igen være varsom, eftersom visse operationer bliver mere komplicerede, som f.eks. typemigrering.

Afslutningsvis skal det bemærkes at objektorienteret design kræver mindst lige så meget træning som relationelt design. De ekstra egenskaber, som stilles til rådighed, kan være en hjælp i designet, men de kan også bruges uhensigtsmæssigt og give flere problemer end de løser. Dette vil blive behandlet nærmere i afsnit 8.2 ”Objektorienteret design”.

6.7 Delkonklusion

Visse ting bliver lettere, hvis man ønsker at benytte brugerdefinerede typer frem for almindelige tabeller i sit databasedesign. Specielt forespørgsler, som kan drage fordel af dereferering, vinder en hel del. Dog kræver dereferingsoperatoren ikke de nye objektorienterede faciliteter for at kunne fungere. Den kunne lige så vel have opereret ved hjælp af referencer, som bygger på primær- og fremmednøgler.

Operationerne kan også blive lidt mere intuitive i SQL99, f.eks. indsættelse af værdier, som repræsenterer en given entitet. Her kan subtyper gøre, at alle data vedrørende entiteten kan indsættes på én gang.

Dog fremkommer der en del ulemper ved brugen af systemgenererede OIDer. Typemigrering og udvælgelse af specifikke rækker uden brug af OIDer kan give problemer. Sidstnævnte kan bedst omgås ved at tilføje en primærnøgle til tabellen, hvorved den identitetsbaserede identifikation mister noget af sin værdi. Et andet og større problem er selve derefereringen af de identitetsbaserede referencer. Disse indfører nogle af de samme begrænsninger, som var de hierarkiske systemers grundlæggende svaghed. De kræver derfor ekstra omtanke at bruge, hvilket formentligt vil mindske deres generelle brugbarhed.

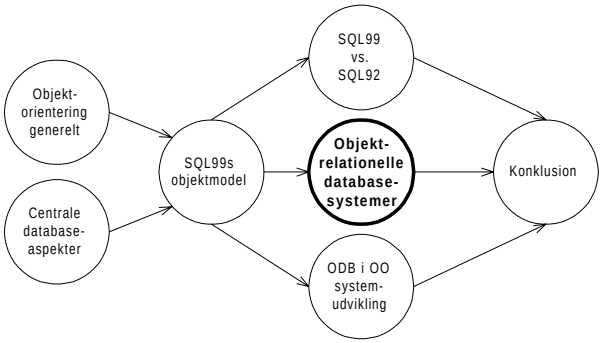
Som overordnet konklusion på sammenligningen af SQL99 med SQL92 er min vurdering, at de medtagne eksempler ikke alene kan begrunde valget af SQL99 frem for SQL92. Sammenligningen har på ingen måde været komplet, men formålet med

dette afsnit var at sammenligne centrale operationer i SQL92 med tilsvarende objektorienterede operationer i SQL99. SQL99 tilbyder en række nye faciliteter, hvoraf nogle ikke har nogen pendant i SQL92, f.eks. polymorfi på attributniveau. Benyttet med omhu vil disse uden tvivl kunne bidrage med fordele til databasedesignet.

Indlæringsmæssigt vil de nye faciliteter formentligt gøre det en smule mere krævende at sætte sig ind i SQL. Det er min erfaring, at SQL92 har en lidt speciel indlæringskurve. Man bliver hurtigt rimelig god, og kan lave simple forespørgsler fra én eller to tabeller. Men på et tidspunkt begynder det at blive væsentligt mere kompliceret: indlejrede forespørgsler, *outer joins*, gruppering etc. Her begynder det at kræve, at man forstår det relationelle grundlag, som SQL er baseret på. Når dette niveau er forstået, kommer resten forholdsvis nemt. Med introduktionen af de objekt-relationelle egenskaber, følger endnu et fagområde, som skal tilegnes, før databasen kan benyttes af såvel systemudviklere som databaseadministratorer (forudsat at de objekt-relationelle egenskaber benyttes).

Som tidligere nævnt skal en del af forklaringen på SQLs store succes formentligt findes i dens enkelthed. Har man ikke brug for særlig avancerede forespørgsler men blot skal hente og gemme data på en forholdsvis enkel måde, så kræver sproget ikke meget indlæring. Introduktionen af de objekt-relationelle egenskaber vil gøre det muligt at hæve kompleksitetsniveauet væsentligt. Det kan være en fordel i udviklingsmiljøer, hvor den ekstra funktionalitet er påkrævet, men i andre sammenhænge kan det blot føre til længere udviklingstider og større risiko for fejl. Det er derfor vigtigt, at man designer sin database med omhu og ikke benytter de avancerede egenskaber uden at have et reelt behov for dem.

7 Objekt-relationelle databasesystemer



Næsten samtlige af de store databaseleverandører har taget det objekt-relationelle paradigme til sig og er begyndt at inkorporere det i deres produkter. Dette gælder f.eks. *Oracle 9i* (Oracle Corporation), *Informix Universal Server 9.1* (Informix Software Inc.) og *IBM DB2 Universal Server 7* (International Business Machines Corporation). De tre produkter vil herefter blive omtalt som henholdsvis **Oracle**, **Informix** og **DB2**. For at sætte standarden i perspektiv vil jeg i dette afsnit gennemgå deres respektive objekt-relationelle egenskaber og sammenligne dem med standardens forskrifter. Selvom de objekt-relationelle udvidelser er forholdsvis nye, er alle tre produkter meget omfattende. En detaljeret analyse af ét eller to af produkterne vil derfor være for omfattende for dette kapitel. Jeg har i stedet valgt at lave en bredere gennemgang af alle tre, for derved at skabe et overblik over *state of the art* indenfor de eksisterende objekt-relationelle databasesystemer. Formålet med afsnittet er således at give en fornemmelse for produkternes placering i forhold til hinanden og standarden. Til det formål vil udvalgte eksempler fra forrige afsnit blive implementeret i hvert af de tre systemer (antaget at de understøtter den pågældende egenskab).

7.1 Dokumentation

Arbejdet med at analysere de tre systemer er i DB2 og Informix's tilfælde baseret på deres respektive dokumentation. Dokumentationen for Oracle 9i er endnu ikke udkommet i sin endelige udgave. Analysen er baseret på dokumentationen for Oracle 8i, samt korrespondance med Martin Jensen (Technical Consulting Manager – Technical Strategic Solutions) fra Oracle Danmark, som venligst har besvaret en lang række spørgsmål, omkring de nye egenskaber i Oracle 9i.

Analysen er foretaget så præcist og korrekt som muligt. Dog må jeg tage det forbehold, at visse egenskaber, som i dette afsnit fremstår som manglende ved et givet produkt, godt kan være understøttet alligevel – eventuelt på en alternativ måde eller i form af en *work-around*. Det er i sagens natur oplagt, at manglende egenskaber ikke udstilles i dokumentationen. Ofte nævnes de relevante emne ikke, som f.eks. `MAP` og

`ORDER` operationer i DB2. Finder man, hvad man søger, så ved man, at man er færdig. Finder man det ikke, så kan man ikke afgøre, om det er fordi det ikke findes, eller fordi man bare ikke har fundet det.

Hvis det viser sig, at egenskaben alligevel findes i én form eller en anden, så bør dens manglende optræden i dette speciale opfattes, som en opfordring til at forbedre dokumentationen på det givne område.

7.2 Typer

Alle tre produkter understøtter (ikke overraskende) brugerdefinerede typer. DB2 og Informix understøtter både distinkte og strukturerede typer. Oracle understøtter kun strukturerede typer. Inden produkternes brugerdefinerede typer beskrives i deltaljer, vil jeg først gennemgå problematikken omkring afsluttede typer.

7.2.1 Afsluttede typer

Som tidligere beskrevet i afsnit 5.2.2 ”Uklarhed omkring afsluttede typer” hersker der uklarhed omkring den præcise brug af ordet `FINAL` i forbindelse med brugerdefinerede typer. Standarden synes at foreskrive, at alle distinkte typer er afsluttede og at alle strukturerede typer ikke er afsluttede. Dette er f.eks. Gulutzan & Pelzer [1999] ikke enig i.

Nedenstående tabel viser hvorledes afsluttede typer bruges i de tre systemer for henholdsvis distinkte og strukturerede typer. F.eks. angiver ”Afsluttede” ud for Informix og distinkte typer, at distinkte typer i Informix altid er afsluttede.

TYPER \ PRODUKTER	ORACLE	INFORMIX	DB2
Distinkte typer		Afsluttede	Afsluttede
Strukturerede typer	Ikke afsluttede	Ikke afsluttede	*Brugerdefineret*

Kun DB2 giver mulighed for eksplicit at vælge, om en struktureret type er afsluttet eller ej. Dette gøres ved at angive `NOT FINAL`, hvis den ikke skal være afsluttet. Det er ikke muligt at angive `FINAL` eksplicit, men hvis intet angives, betragtes typen som værende afsluttet.

7.2.2 Distinkte typer

7.2.2.1 Eksempler

Oracle er ikke medtaget i dette afsnit, da den som beskrevet ikke understøtter distinkte typer.

SQL99

```
CREATE TYPE TelefonNrType AS NUMERIC(10,0) FINAL;
```

Informix

```
CREATE DISTINCT TYPE TelefonNrType AS NUMERIC(10,0)
```

Informix har valgt at tydeliggøre forskellen mellem distinkte og strukturerede typer ved at lade ordet `DISTINCT` indgå i typeerklæringen. Dette virker som et fornuftigt valg. Grunden til at samme syntaks ikke er medtaget i standarden kunne være, at ordet allerede er reserveret i forbindelse med `SELECT DISTINCT`.

Alle operationer nedarves fra basistypen. Dette bryder med standardens forskrift om, at kun sammenligningsoperationerne nedarves. Dette må siges at være et fornuftigt valg, da brugen af de distinkte typer ellers ville være blevet alt for besværlig.

Distinkte typer i Informix kan baseres på alle former for typer (f.eks. rækketyper). Dette er anderledes end standarden, som foreskriver, at kun prædefinerede typer kan benyttes som basistyper.

DB2

```
CREATE DISTINCT TYPE TelefonNrType AS NUMERIC(10,0)
  WITH COMPARISONS
```

På samme måde som hos Informix, nedarves alle operationer fra basistypen. Dog kræves det, at `WITH COMPARISONS` angives, hvis denne understøtter sammenligningsoperationer (f.eks. heltal og strenge gør; BLOB og CLOB gør ikke).

I DB2 kan distinkte typer kun baseres på prædefinerede typer, som standarden foreskriver.

7.2.2.2 Vurdering

Distinkte typer i Informix er mere generelle end de tilsvarende i DB2. Begge nedarver samtlige operationer fra basistypen. Selvom dette er en afvigelse fra standarden, må den alligevel siges at være et fornuftigt valg. Som distinkte typer er beskrevet i SQL99, vil de være meget besværlige at benytte.

7.2.3 Strukturerede typer

7.2.3.1 Eksempler

Alle tre systemer understøtter strukturerede typer. For at illustrere brugen af brugerdefinerede operationer medtages en simpel operation, som returnerer navn og adresse konkateneret. Selve implementationen af operationen foretages i et senere afsnit (7.5 ”Operationer”). Operationshovedet er medtaget her, for at vise i hvor høj grad operationer og typer knyttes sammen i de tre systemer. SQL99 foreskriver, at operationerne erklæres sammen med typen, og at de derefter implementeres separat.

SQL99

```
CREATE TYPE KontaktType AS
(  navn        CHAR(60),
   adresse     ROW ( vejnavn   VARCHAR(60),
                     vejnr     VARCHAR(20),
                     bynavn    VARCHAR(60),
                     postnr    NUMERIC(4,0)
                 ),
   telefonnr   TelefonNrType,
   email       CHAR(40),
   firma       BOOLEAN
)
NOT FINAL
REF IS SYSTEM GENERATED
   INSTANCE METHOD postadresse() RETURNS VARCHAR
   LANGUAGE SQL  -- valgfri
   DETERMINISTIC -- valgfri
   CONTAINS SQL  -- valgfri
; -- end create type
```

Oracle

```
CREATE OR REPLACE TYPE AdresseType AS OBJECT
(  vejnavn   VARCHAR(60),
   vejnr     VARCHAR(20),
   bynavn    VARCHAR(60),
   postnr    NUMERIC(4,0)
);

CREATE OR REPLACE TYPE KontaktType AS OBJECT
(  navn        VARCHAR(60),
   adresse     AdresseType,
   telefonnr   NUMERIC(10,0),
   email       CHAR(40),
   firma       BOOLEAN

   MEMBER FUNCTION postadresse RETURN VARCHAR
);
```

Oracle benytter en lidt anden syntaks end SQL99. Angivelsen `OR REPLACE` er valgfri. Angives den ikke, resulterer kommandoen i en fejl, hvis typen allerede eksisterer. Som udvikler er det behageligt at have den slags kommandoer, således at man ikke først skal slette og derefter oprette typen.

Strukturerede typer kan ikke skabes ad hoc, som det er muligt i SQL99 ved hjælp af `ROW`. Derfor er det nødvendigt at skabe en ny navngiven type (`AdresseType`).

Hvilken form for OID man ønsker, angives ikke i forbindelse med typedefinitionen. Dette gøres først, når en tabel af typen oprettes.

Som det fremgår af terminologien, betragter Oracle ordet ”objekt” som værende en type (`CREATE TYPE … AS OBJECT`). Dette er blot endnu et eksempel på den uklare definition af begrebet objekt.

Informix

```
CREATE ROW TYPE KontaktType
(  navn         VARCHAR(60),
   adresse      ROW ( vejnavn   VARCHAR(60),
                      vejnr     VARCHAR(20),
                      bynavn    VARCHAR(60),
                      postnr    NUMERIC(4,0)
                ),
   telefonnr    NUMERIC(10,0),
   email        CHAR(40),
   firma        BOOLEAN
);
```

Denne syntaks minder om rækketyperne fra SQL3. SQL3 skelnede mellem rækketyper og Abstrakte Datatyper (ADTer[24]) [Ullman & Widom, 1997], men disse blev slået sammen til strukturerede typer i SQL99. Funktionaliteten af Informix’s rækketyper svarer til rækketyperne i SQL3. Det er hverken muligt at indkapsle attributterne eller knytte operationer til typerne.

Det er ikke muligt at angive typen på OIDen i forbindelse med typedefinitionen. OIDer i Informix vil blive behandlet nærmere i de efterfølgende afsnit 7.3 ”Typedefinerede tabeller” og afsnit 7.4 ”Subtyper, tabelhierarkier og referencer”.

[24] ADTerne fra SQL3 minder meget om brugerdefinerede typer i SQL99. Dog kunne ADTer ikke bruges som tabeltyper. Kun kolonner kunne have en ADT.

DB2

```
CREATE TYPE AdresseType AS
(  vejnavn     VARCHAR(60),
   vejnr       VARCHAR(20),
   bynavn      VARCHAR(60),
   postnr      NUMERIC(4,0)
)
MODE DB2SQL;

CREATE TYPE KontaktType AS
(  navn        VARCHAR(60),
   adresse     AdresseType,
   telefonnr   NUMERIC(10,0),
   email       CHAR(40),
   firma       SMALLINT
)
NOT FINAL
REF USING VARCHAR(16) FOR BIT DATA
MODE DB2SQL
   METHOD postadresse()
   RETURNS VARCHAR
   LANGUAGE SQL
   DETERMINISTIC
   CONTAINS SQL
; -- end create type
```

DB2s syntaks virker mere kompleks end de to øvrige. Dog er mange af de angivne karakteristika for operationen postadresse valgfrie. De er medtaget for at vise, hvor tæt de ligger på standardens syntaks. IBM har tydeligvis haft stor indflydelse på den endelige udformning af SQL99.

Dog har DB2 stadig et par afvigelser:

1. `BOOLEAN` er ikke understøttet.
2. `MODE DB2SQL` skal angives. Dette angiver at syntaksen er specifik for DB2. Det er ikke muligt at angive andet end `DB2SQL`, og det er påkrævet at det medtages.
3. OIDens type kan kun angives som værende én prædefineret type svarende til brugerdefinerede OIDer i SQL99. Default er `VARCHAR(16) FOR BIT DATA`.
4. DB2 understøtter ikke `SYSTEM GENERATED` og `DERIVED` OIDer.

På samme måde som hos Oracle er det ikke muligt at lave ad hoc rækketyper. Disse skal skabes med en `CREATE TYPE` kommando.

7.2.3.2 Vurdering

Informix lader til at være lidt bagefter de to andre. Syntaks og semantik ligner meget tidligere udgaver af SQL3. Dog havde denne også understøttelse af ADTer, hvilket

Informix mangler. Oracle understøtter de vigtigste funktioner, men DB2 ligger tættest op af SQL99.

7.2.4 Uigennemsigtige typer

Informix understøtter en tredje brugerdefineret type, som ikke er med i standarden. Denne kaldes for uigennemsigtige typer (*opaque types*). Denne slags type er blandt andet beskrevet af Stonebraker og Brown [1999]. Stonebraker var en af de tidlige fortalere for objekt-relationelle databasesystemer. Han var desuden hovedarkitekt bag Illustra, et rent objekt-relationelt databasesystem. Illustra blev senere opkøbt af Informix og dens objekt-relationelle egenskaber blev flyttet over i Informix databasesystemet.

Uigennemsigtige typer giver mulighed for at skabe brugerdefinerede typer, som er helt igennem brugerdefinerede. De benytter ikke databasesystemets prædefinerede typer på nogen måde. Databasesystemet oplyses om størrelsen af værdierne af typen samt eventuelt ekstra metainformation. Definitionen på en uigennemsigtig type kan se således ud:

```
CREATE OPAQUE TYPE VeryLargeInt(INTERNALLENGTH=VARIABLE)
```

Derefter skal der registreres en række operationer skrevet i C, som kan håndtere værdierne af typen.

Uigennemsigtige typer vil ikke blive beskrevet nærmere, eftersom de ikke er med i SQL99.

7.3 Typedefinerede tabeller

7.3.1 Eksempler

Alle tre systemer understøtter typedefinerede tabeller, dog uden at to systemer er helt ens.

SQL99

```
CREATE TABLE kontakt OF KontaktType
(  REF IS kontaktID SYSTEM GENERATED,
   navn WITH OPTIONS NOT NULL
);
```

Oracle

```
CREATE TABLE kontakt OF KontaktType
(  navn NOT NULL
)
OBJECT ID SYSTEM GENERATED;
```

Oracle giver mulighed for to slags OIDer. Enten `SYSTEM GENERATED` (som vist) eller `PRIMARY KEY`. Sidstnævnte svarer til afledte (*derived)* OIDer i SQL99. Det er kun

muligt at definere typen på OIDen på dette tidspunkt, modsat SQL99 hvor typen bestemmes på `CREATE TYPE` tidspunktet.

Informix

```
CREATE TABLE kontakt OF TYPE KontaktType
( CHECK(navn IS NOT NULL)
);
```

Umiddelbart skulle man tro, at Informix kun understøttede systemgenererede OIDer, eftersom syntaksen ikke giver mulighed for at vælge. Dog har det ikke været muligt at finde nogen information om emnet i dokumentationen. Næste afsnit om ”Subtyper, tabelhierarkier og referencer” vil indikere at rækkerne slet ingen OIDer har.

Det er ikke muligt at angive begrænsninger (*constraints*) på individuelle attributter, når tabellen oprettes. Dette skal gøres når rækketypen defineres. Dog kan samme funktionalitet simuleres med begrænsninger på tabelniveau (som f.eks. `CHECK`).

DB2

```
CREATE TABLE kontakt OF KontaktType
(  REF IS kontaktID USER GENERATED
   navn WITH OPTIONS NOT NULL
);
```

Som det kan ses ligger DB2s syntaks meget tæt op ad SQL99. Igen er visse DB2-specifikke valgmuligheder udeladt, men standardens syntaks er i høj grad overholdt. Det er selvfølgelig en begrænsning, at den kun understøtter brugerdefinerede OIDer. Det er dog muligt, at få DB2 til at generere unikke værdier til en OID-kolonne. Dette gøres ved at benytte den indbyggede operation `GENERATE_UNIQUE()`, som skaber et nyt unikt OID af typen `VARCHAR(13) FOR BIT DATA`. Dette kan så benyttes som OID, forudsat at en passende type er valgt for OID-kolonnen (default typen `VARCHAR(16) FOR BIT DATA` er kompatibel). Indsættelsen af en systemgenereret værdi kunne f.eks. se således ud:

```
INSERT INTO kontakt VALUES
   (KontaktType(GENERATE_UNIQUE()), ”Hans Hansen”, …);
```

7.3.2 Vurdering

De tre systemer ligger mere lige her. Dog adskiller Oracle og DB2 sig stadig fra Informix. De understøtter begge OIDer, hvilket Informix formentligt ikke gør (se efterfølgende afsnit). DB2 og Oracle har delt standardens tre OID former mellem sig. DB2 understøtter brugerdefinerede, og Oracle understøtter systemgenererede og afledte. Om dette er tilfældigt, eller om der er tale om marketingstrategier fra en eller begge parter er ikke til at sige.

7.4 Subtyper, tabelhierarkier og referencer

7.4.1 Eksempler

Disse emner er samlet i ét afsnit, fordi de hører tæt sammen. Syntaksmæssigt adskiller de tre systemer sig en del fra hinanden, men grundlæggende understøtter de næsten de samme funktioner, på nær Informix, som tilsyneladende ikke understøtter identitetsbaserede referencer.

SQL99

```
CREATE TYPE KundeType UNDER KontaktType AS
(  saelgernr   REF(KontaktType) SCOPE kontakt,
   aktiv       BOOLEAN
) NOT FINAL;

CREATE TABLE kunde OF KundeType UNDER kontakt;
```

Oracle

```
CREATE OR REPLACE TYPE KundeType UNDER KontaktType AS OBJECT
(  saelgernr   REF KontaktType SCOPE IS kontakt,
   aktiv       BOOLEAN
);

CREATE TABLE kunde OF KundeType UNDER kontakt;
```

Det er muligt at angive en `SCOPE IS <tabelnavn>` restriktion på referencedefinitionen. Derved begrænses referencen til at pege på rækker i én tabel. Dette vil give en væsentligt hastighedsforbedring, hvis der eksisterer mange tabeller af typen KontaktType.

Informix

```
CREATE ROW TYPE KundeType
(  saelgernr   <Informix understøtter ikke REFs>,
   aktiv       BOOLEAN
)
UNDER KontaktType;

CREATE TABLE kunde OF TYPE KundeType UNDER kontakt;
```

Det har ikke været muligt, at finde beskrivelser af identitetsbaserede referencer i dokumentationen for Informix. Det er muligt at de eksisterer, men i så fald er det gemt godt. Dette er det typiske eksempel på en manglende funktionalitet som ikke omtales i dokumentationen. Følgende argumenter taler tilsammen for, at Informix ikke understøtter identitetsbaserede referencer, selvom ingen af dem afgører spørgsmålet definitivt.

1. Det er ikke muligt at angive, hvilken type OID en given rækketype skal anvende. Dette kan dog også betyde, at de er systemgenererede.
2. Dokumentationen indeholder ingen forekomster af reserverede ord som `REF`, `DEREF`, `OID` eller `SCOPE`.
3. Typedefinerede tabeller kan ikke indeholde `ROWID`s (automatisk generede nøgleværdier).

> "Important: Typed tables do not support rowids. Therefore you cannot specify the WITH ROWID or ADD ROWID clauses when you create tables in a table hierarchy."
>
> [Informix, 1999]

Hvis Informix ikke understøtter identitetsbaserede referencer, må dette siges at være en markant mangel i forhold til standarden. Det kan diskuteres hvor anvendelige identitetsbaserede referencer er. Det er ikke sikkert, at det er en stor mangel, set fra et udvikler synspunkt, men i en sammenligning med SQL99-standarden, må det siges at være en reelt anklagepunkt.

DB2

```
CREATE TYPE KundeType UNDER KontaktType AS
(  saelgernr   REF(KontaktType) SCOPE kontakt,
   aktiv       SMALLINT
)
MODE DB2SQL;

CREATE TABLE kunde OF KundeType UNDER kontakt
   INHERIT SELECT PRIVILEGES;
```

Angivelsen `INHERIT SELECT PRIVILEGES` skal altid angives. Dette for at illustrere at forespørgselsrettigheder altid nedarves fra en supertabel. Dette skyldes formentligt at en forespørgsel på tabellen kontakt, altid også bør kunne returnere rækker fra tabellen kunde.

7.4.2 Vurdering

Hvis Informix ikke understøtter identitetsbaserede referencer, må den siges at være meget handicappet i kapløbet med de to øvrige systemer. Dette er en central del af SQL99s objektmodel.

7.5 Operationer

7.5.1 Eksempler

Brugerdefinerede operationer er generelt understøttet af alle tre systemer. Dette er ikke overraskende. På netop dette punkt har de fleste kommercielle databasesystemer været forud for standarden. Lagrede procedurer (*Stored procedures*) og udløsere

(*triggers*) skrevet i funktionelt komplette sprog har været tilgængelige i adskillige år. Den eneste tilretning, som derfor var nødvendig, var tilpasningen til de brugerdefinerede typer, såsom understøttelse af selvreferencen.

SQL99

```
-- funktionshoved erklæret sammen med typen
… INSTANCE METHOD postadresse() RETURNS VARCHAR
LANGUAGE SQL
DETERMINISTIC
CONTAINS SQL …

-- funktionskroppen erklæret separat
INSTANCE METHOD postadresse() RETURNS VARCHAR
RETURN SELF.navn || ”, ” || SELF.adresse;
```

Oracle

```
-- funktionshoved erklæret sammen med typen
… MEMBER FUNCTION postadresse RETURN VARCHAR …

-- funktionskroppen erklæret separat
CREATE OR REPLACE TYPE BODY KontaktType AS
  MEMBER FUNCTION postadresse RETURN VARCHAR IS
  BEGIN
    RETURN( navn || ', ' || adresse );
  END postadresse;
END;
```

Eksemplet viser kun definitionen af én operation, men havde typen haft flere, ville alle disse være blevet defineret i samme `CREATE TYPE BODY…END` konstruktion. Syntaksen passer igen ikke helt med standarden, men de vigtigste egenskaber er understøttet. Dette gælder f.eks. selv-referencen (`SELF`), overstyring af operationer fra eventuelle supertyper, samt eksterne operationer skrevet i andre programmeringssprog (som f.eks. C og Java).

Som noget enestående blandt de tre systemer, understøtter Oracle `MAP` og `ORDER` operationer. Disse sikrer, at serveren er i stand til at sammenligne og sortere værdier af typen korrekt. Hverken Informix eller DB2 har tilsvarende funktionalitet.

Informix

```
CREATE FUNCTION postadresse(KontaktType t)
   RETURNING VARCHAR;
   RETURN t.navn || ”, ” || t.adresse
END FUNCTION;
```

Informix understøtter kun separate operationer. De er således ikke knyttet til nogen type og har derfor heller ikke nogen selv-reference.

Både overstyring af operationer, samt eksterne operationer skrevet i C understøttes. Det er således muligt at definere operationer med samme navn men med forskellige argumenter. Angives en subtype som argument til en operation, vælges den udgave af operationen, som har den nærmeste supertype angivet som argumenttype.

DB2

```
-- funktionshoved erklæret sammen med typen
… METHOD postadresse() RETURNS VARCHAR
LANGUAGE SQL
DETERMINISTIC
CONTAINS SQL
NO EXTERNAL ACTION …

-- funktionskroppen erklæret separat
CREATE METHOD postadresse() FOR KontaktType
RETURN SELF..navn || ”, ” || SELF..adresse;
```

DB2 ligger igen tættest på standarden rent syntaksmæssigt. Dog benytter man dobbelt punktum for at tilgå attributterne i en værdi, som f.eks. `SELF`. Dette var foreslået i SQL3, men blev senere fjernet. SQL99 benytter enkelt punktum, hvilket nok må siges at være den gængse standard blandt programmeringssprog.

En begrænsning i forhold til SQL99 er, at DB2 ikke tillader tilknytning af operationer til distinkte typer. Dette er understøttet i SQL99, men denne foreskriver også at kun sammenligningsoperationerne nedarves. Som tidligere nævnt nedarves al funktionalitet fra den prædefinerede type til den distinkte.

Overstyring af operationer er understøttet. Eksterne operationer kan enten være skrevet i C eller Java eller i andre sprog, som kan tilgås via *Object Linking and Embedding* (OLE)[25].

Et af problemerne med eksterne operationer er, at de fremtvinger et valg mellem hastighed og sikkerhed. Startes den eksterne operation i samme systemproces som databasesystemet, risikerer hele systemet at gå ned. Hvis operationen foretager en ulovlig systemhandling og operativsystemet standser den, vil dette også standse databasesystemet. Alternativet er at lave en separat proces til operationen. Hvis den så går ned, tager den ikke databasesystemet med. Prisen for dette er at starte en systemproces, hvilket tager lang tid. DB2 giver systemadministratoren mulighed for at vælge hvilken metode, som skal benyttes, på hver enkelt ekstern operation. Dette gøres med den valgfri parameter `FENCED` eller `NOT FENCED` (indhegnet eller ikke indhegnet). Hvis den ikke er indhegnet, foretages kaldet i databasesystemets systemproces. `FENCED` er default.

7.5.2 Vurdering

Informix mangler den tætte knytning mellem type og operationer. Dette giver et indtryk af et mindre objektorienteret system end de to øvrige, men det giver også konkrete problemer på grund af indkapslingen (se afsnit 7.9.1 ”Indkapsling”).

[25] OLE er en proprietær standard for kald af operationer udviklet af Microsoft. Det er en del af COM (*Common Object Model*). OLE-operationer er kun understøttet på Microsofts egne Windows-platforme.

7.6 Indsættelse af data

7.6.1 Eksempler

Begge eksempler på indsættelse af data fra forrige afsnit er medtaget her. Først indsættes en værdi af typen KontaktType i tabellen kontakt. Derefter indsættes en værdi af typen KundeType i tabellen kunde. Det centrale i eksemplerne er hvordan en værdi af en brugerdefineret type defineres, samt hvorledes en identitetsbaseret reference angives. Det er forholdsvis få linier i eksemplerne for de tre systemer, som varierer i forhold til SQL99-eksemplet. De steder, hvor deres respektive syntakser adskiller sig fra standarden, er fremhævet med fed skrift.

SQL99

```
INSERT INTO kontakt
          (navn,
           adresse,
           telefonnr,email,firma)
   VALUES (”Hans Hansen”,
           ROW(”Østergade”,”1”,”Østerby”,2222),
           11223344,”hans@hansen.dk”,FALSE);

INSERT INTO kunde
            (navn,
             adresse,
             telefonnr,email,firma,saelgernr,aktiv)
  SELECT ”Jens Jensen”,
         ROW(”Ny Østergade”,”1”,”Ny Østerby”,4444),
         55667788,”jens@jensen.dk”,FALSE,kontaktID,TRUE
  FROM kontakt
  WHERE navn = ”Hans Hansen”;
```

Oracle

```
INSERT INTO kontakt
          (navn,
           adresse,
           telefonnr,email,firma)
   VALUES (”Hans Hansen”,
           AdresseType(”Østergade”,”1”,”Østerby”,2222),
           11223344,”hans@hansen.dk”,FALSE);

INSERT INTO kunde
            (navn,
             adresse,
             telefonnr,email,firma,saelgernr,aktiv)
  SELECT ”Jens Jensen”,
         AdresseType(”Ny Østergade”,”1”,”Ny Østerby”,4444),
         55667788,”jens@jensen.dk”,FALSE,ref(k),TRUE
  FROM kontakt k
  WHERE navn = ”Hans Hansen”;
```

Oracle adskiller sig fra standarden på to punkter. For det første benyttes navnet på den brugerdefinerede type AddresseType som konstruktør for adresse-værdien. I SQL99 benyttes det reserverede ord ROW for alle brugerdefinerede typer. For det andet angiver

man en reference til en række med udtrykket `ref(<tabelnavn>)`. I SQL99 angiver man blot navnet på OIDen.

Med hensyn til kontruktørnavnet virker Oracles udgave mere objektorienteret. Det er almindeligt i objektorienterede programmeringssprog, at konstruktører navngives med typens navn. Dette gælder f.eks. i C++ og Java. Dog er det ikke standard i SQL99, og Oracle er derfor ikke kompatibel med standarden på dette område. Ligeledes virker `ref` operatoren bedre end en eksplicit navngiven OID. Dette svarer lidt til adresseoperatoren i C og C++ (operatoren "`&`"), men igen er det ikke standard.

Informix

```
INSERT INTO kontakt
          (navn,
           adresse,
           telefonnr,email,firma)
   VALUES ("Hans Hansen",
           ROW("Østergade","1","Østerby",2222),
           11223344,"hans@hansen.dk",FALSE);

INSERT INTO kunde
            (navn,
             adresse,
             telefonnr,email,firma,saelgernr,aktiv)
  SELECT "Jens Jensen",
         ROW("Ny Østergade","1","Ny Østerby",4444),
         55667788,"jens@jensen.dk",FALSE,
          <Informix har ikke referencer>,TRUE
  FROM kontakt k
  WHERE navn = "Hans Hansen";
```

Informix understøtter som bekendt ikke referencer, men på de øvrige områder er den kompatibel med standarden.

DB2

```
INSERT INTO kontakt
          (kontaktID,navn,
           adresse,
           telefonnr,email,firma)
   VALUES (KontaktType(GENERATE_UNIQUE()),"Hans Hansen",
           AdresseType("Østergade","1","Østerby",2222),
           11223344,"hans@hansen.dk",FALSE);

INSERT INTO kunde
            (kontaktID,navn,
             adresse,
             telefonnr,email,firma,saelgernr,
             aktiv)
  SELECT KundeType(GENERATE_UNIQUE()),"Jens Jensen",
         AdresseType("Ny Østergade","1","Ny Østerby",4444),
         55667788,"jens@jensen.dk",FALSE,KontaktType(kontaktID),
         TRUE
  FROM kontakt k
  WHERE navn = "Hans Hansen";
```

Som vist i afsnit 7.3 ”Typedefinerede tabeller” benytter DB2 den indbyggede operation `GENERATE_UNIQUE()` til at skabe en unik OID med typen `VARCHAR(13) FOR BIT DATA`. Den returnerede værdi skal typekonverteres til den brugerdefinerede type, som den skal være OID for. Selve kolonnen med OIDen skal inkluderes i listen af kolonner, som skal have indsat værdier.

På samme måde som hos Oracle benyttes typenavnet som konstruktør.

Indsættelsen af OIDen for ”Jens Jensen” kunne også have været gjort med en indlejret forespørgsel på følgende måde:

```
INSERT INTO kunde
            (kontaktID,navn,
             adresse,
             telefonnr,email,firma,
             saelgernr,
             aktiv)
  VALUES(KundeType(GENERATE_UNIQUE()),”Jens Jensen”,
         AdresseType(”Ny Østergade”,”1”,”Ny Østerby”,4444),
         55667788,”jens@jensen.dk”,FALSE,
          KontaktType((SELECT kontaktID
                       FROM kontakt
                       WHERE navn = ”Hans Hansen” ))
        ,TRUE);
```

Denne syntaks er en smule mere intuitiv.

7.6.2 Vurdering

Havde Informix understøttet referencer, havde den formentligt været den mest kompatible med standarden på dette område. Som det er nu, må det nok siges, at DB2 er den, som kommer tættest. Dog mener jeg, at Oracle har en lidt pænere måde at hente OIDen for en værdi (ved hjælp af `REF`).

7.7 Dereferering

7.7.1 Eksempler

Kun Oracle og DB2 er medtaget, eftersom Informix ikke understøtter identitetsbaserede referencer. Som det kan ses af nedenstående, ligger begge systemer meget tæt op ad standarden. Oracle benytter dog punktum som derefereringsoperator i stedet for en højrepil. Derudover indeholder de to systemer ingen væsentlige forskelle.

SQL99

```
SELECT k.navn
FROM kunde k
WHERE k.saelger->navn = ”Hans Hansen” AND k.aktiv;
```

Oracle

```
SELECT k.navn
FROM kunde k
WHERE k.saelger.navn = ”Hans Hansen” AND k.aktiv;
```

DB2

```
SELECT k.navn
FROM kunde k
WHERE k.saelger->navn = ”Hans Hansen” AND k.aktiv=1;
```

7.7.2 Vurdering

Både Oracle og DB2 viser den syntaksmæssige fordel ved dereferering. Dog burde Oracle også have understøttet standardens notation med en højrepil. Det fremmer læsbarheden, at derefereringen er tydeligt markeret. Ovenstående eksempel kunne ellers lige så godt være en attribut ved navn saelger med en struktureret brugerdefineret type, som indeholdt en attribut ved navn navn.

7.8 Polymorfi

7.8.1 Eksempler

I dette afsnit beskrives den understøttede polymorfi i de tre databasesystemer. Eksemplet fra afsnit 6.3.2 ”Forespørgsel med polymorfi” om polymorfi på tabelniveau, kunne løses på to måder. Dette afhang af, om flossede rækker var understøttet eller ej. Ingen af de tre databasesystemer understøtter flossede rækker. Derfor vil den alternative løsning blive valgt. Som omtalt tidligere er standarden meget uklar på dette område. Det vil derfor blive antaget, at flossede rækker ikke er med i SQL99, da en så omfattende nyskabelse formentligt ville have været beskrevet grundigt. De tre systemer vil derfor blive betragtet som værende korrekte i forhold til standarden på dette område.

Polymorfi på attributniveau er mere problematisk. Som tidligere omtalt bryder det meget med databasesystemernes måde at lagre rækker på. Det er derfor heller ikke overraskende, at ingen af de tre understøtter polymorfi på attributniveau.

Alle tre systemer understøtter `ONLY` operatoren og er derfor helt kompatible med standarden.

SQL99, Oracle, Informix, DB2

```
SELECT *
FROM ONLY(kunde)
UNION
SELECT k.*, NULL, NULL
FROM ONLY(kontakt) k;
```

7.8.2 Vurdering

Det er tydeligt, at objektorienterede egenskaber, som kræver større ændringer i databasesystemernes fundament, ikke er blevet implementeret endnu. Oracle bruger som argument mod flossede rækker, at det er for besværligt at få kunder og 3. parts leverandører til at benytte faciliteten. Hvis den ikke er medtaget i SQL99, vil den formentligt heller ikke blive implementeret af de andre systemer. Polymorfi på attributniveau kommer nok også først noget senere. Der er, som beskrevet, nogle helt grundlæggende problemer med lagringen af disse. Det er mit bud, at denne funktionalitet ikke bliver implementeret foreløbigt.

7.9 Andre egenskaber

7.9.1 Indkapsling

I SQL99 styres indkapslingen ved hjælp af passende `EXECUTE` rettigheder på en given types observatør- og mutatoroperationer. Dette gøres en smule anderledes i de tre systemer, men effekten er den samme. I stedet for at oprette to automatiske operationer, kan læse- og skriverettigheder tildeles direkte på attributniveau. Det er desuden muligt i alle tre, at tildele udførelsesrettigheder på alle brugerdefinerede operationer.

Ikke alle rettigheder kan angives på attributniveau. Nedenstående tabel viser mulighederne for de tre systemer. Hvis et plus angives betyder det, at produktet understøtter den givne rettighed på attributniveau. Et minus angiver, at rettigheden ikke kan tildeles på attributniveau men kun på tabelniveau. F.eks. er det muligt i Informix at begrænse en bruger til kun at måtte udvælge bestemte attributter i en given tabel. Derfor står der plus ud for Informix/`SELECT`. I Oracle og DB2 er dette ikke muligt. Her kan man kun begrænse en brugers adgang til tabellen som helhed. Derfor står der minus ud for Oracle/`SELECT` og DB2/`SELECT`.

Produkt / Rettighed	ORACLE	INFORMIX	DB2
`SELECT`	÷	+	÷
`INSERT`	+	÷	÷
`UPDATE`	+	+	+
`REFERENCES`	+	+	+

I samtlige systemer er det muligt at begrænse en bruger eller rolles mulighed for at opdatere værdien i en attribut, samt at referere til en given attribut. Dog er det kun Informix, som giver mulighed for at skjule en attribut, så den ikke kan aflæses. Funktionaliteten af `INSERT`-privilegiet hos Oracle, virker umiddelbart en smule

underligt. Indsættelse af nye data fungerer på tupelniveau. Derfor kan det undre, at indsættelse kan begrænses på kolonneniveau. Hvis data indsættes i en tabel, som er begrænset på en specifik kolonne, kan dette kun lade sig gøre, hvis den er valgfri (`NULL` tilladt), eller der er angivet en `DEFAULT` værdi for den.

Indkapslingen kan ikke siges at være optimal i nogen af systemerne. Det er vigtigt at kunne skjule typernes interne repræsentation. Informix er mest omfattende på dette område.

Umiddelbart ser det ud til, at Informix kan få problemer med brugerdefinerede operationer på grund af den understøttede indkapsling. Der findes ikke nogen tæt knytning mellem operationer og typer i Informix. Dermed kan en operation ikke være sikker på at kunne tilgå de attributter i dens ”tilhørende” type, som den har brug for. Eftersom operationen ikke har nogen selv-reference, kan den kun tilgå attributterne udefra. Derved er den underlagt de samme rettigheder, som den bruger, der udfører operationen. Det har ikke været muligt at finde nogen beskrivelse af dette problem eller en løsning i dokumentationen til Informix [Informix, 1999].

7.9.2 Typekonvertering

SQL99 understøtter både `CAST` og `TREAT` udtryk, som beskrevet i afsnit 5.2.6 ”Typekonvertering”. Alle tre databasesystemer understøtter `CAST` udtryk mellem prædefinerede typer. Hvert system har sine regler for hvilke typer, som kan konverteres til hinanden. Kun Informix giver mulighed for at lave brugerdefineret typekonvertering. Disse kan både være implicitte og eksplicitte.

Oracle og DB2 benytter `TREAT` til typekonverteringer fra supertyper til subtyper, som SQL99 foreskriver. I Informix understøttes denne funktionalitet af `CAST` operationen. Informix skelner dermed ikke mellem `CAST` og `TREAT`, som jeg også tidligere har argumenteret for i afsnit 5.5.5 ”Stærkt typekontrol”.

7.9.3 Skemavedligeholdelse

Når et skema designes for en database, er det næsten helt sikkert, at det på et senere tidspunkt skal ændres. Tilføjelse og fjernelse af tabeller er ofte enkelt at håndtere, men en ændring i typen for en tabel kan være mere besværlig. Til det formål understøtter SQL92 `ALTER TABLE` kommandoen, som gør det enklere at foretage den slags ændringer. Det er derfor vigtigt, at den tilsvarende `ALTER TYPE` kommando i SQL99 understøttes i systemerne. Dette er tilfældet for Oracle og DB2. Informix understøtter ikke ændring af brugerdefinerede typer.

Det fremgår ikke klart af SQL99, hvilken effekt `ALTER TYPE` skal have på de tabeller og attributter, som er baseret på den forrige udgave af typen. I DB2 er det kun muligt at ændre typen, hvis ingen tabeller eller attributter er baseret på den. Dette må siges at gøre kommandoen næsten ubrugelig. Typisk vil en brugerdefineret type blive defineret ved hjælp af et skript, som først fjerner og derefter opretter typen. Arbejdet

med at konstruere en passende `ALTER TYPE` kommando kan næppe stå mål med arbejdet ved at udføre skriptet igen med de passende modifikationer. Oracle skulle i version 9i give mulighed for at ændre en type med tilsvarende ændring af samtlige tabeller og attributter. Dette angives ved hjælp af en valgfri tilføjelse til `ALTER TYPE` kommandoen.

Eksempel:

```
ALTER TYPE AdresseType ADD ATTRIBUTE(Land VARCHAR(40))
  CASCADE INCLUDING TABLE DATA;
```

7.10 Java og SQL

Ikke alle relationsdatabasesystemer går i retning af SQL99. Sybase har valgt at satse på en tættere integration med Java. Det er således muligt at benytte Java klasser som typer på attributter. Ligeledes kan serveren udføre Java bytekode. Derved får databaseudviklere adgang til hele Javas store klassebibliotek. Problemet med denne løsning er det samme som for alle Java-produkter: hastigheden er lav. I et system som kræver hurtige svartider, kan dette ikke anbefales.

F.eks. Oracle og DB2 arbejder også hen imod en tættere integration med Java. Første skridt er det såkaldte **SQLJ**, som er *embedded SQL* i Java. Oracle understøtter endvidere lagrede procedurer skrevet i Java. I takt med at både Java og maskinerne bliver hurtigere, kan det måske være en brugbar løsning på sigt. I så fald kan de objektorienterede egenskaber i SQL99 godt risikere at blive trængt i baggrunden. Til forskel fra SQL99s objektmodel tilbyder Java en standardiseret objektmodel, som er fuldt kongruent med dens eksisterende implementationer. Som vist er objektmodellerne i Oracle, Informix og DB2 ikke kompatible med SQL99 endnu, og det kan betvivles, om de nogensinde bliver det. Jeg tror, at standarden er kommet tre til fire år for sent. Den skulle have været fremme før, databasesystemerne begyndte at implementere objektmodellen. På nuværende tidspunkt er det svært at ændre de proprietære syntakser. Af den grund kan valget af objektmodel måske på sigt falde på Javas, eller en anden vedtaget model, som f.eks. objektmodellen i C++.

7.11 Delkonklusion

De tre systemer er nået forholdsvis langt i forbindelse med indføring af de objektorienterede faciliteter. Oracle og DB2 må siges at være nået længst. Informix mangler identitetsbaserede referencer og en tættere knytning af typer og operationer. Den er kun foran på områderne typekonvertering og indkapsling. De må også siges at være væsentlige, men hvis Informix har det omtalte problem med indkapslingen, så kan denne egenskab næppe være en fordel. Oracle og DB2 deler generelt fordele og ulemper imellem sig. Oracle mangler distinkte typer og brugerdefinerede OIDer. Det har DB2. DB2 mangler til gengæld automatiske systemgenererede OIDer, afledte OIDer, samt `MAP` og `ORDER` operationer. Det har Oracle. Det er derfor svært at vurdere

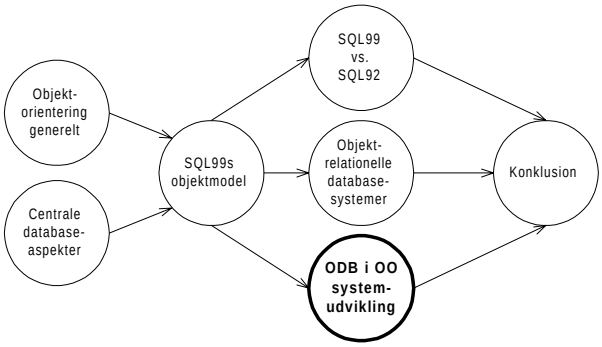
hvilken af de to, som må siges at være den mest objektorienterede. Oracle har formentligt de mest nyttige egenskaber. Her tænker jeg specielt på afledte OIDer og `MAP` og `ORDER` operationerne. Dog ligger DB2 nok tættest på standarden. Selvom det måske ikke er kommet til udtryk i denne kortfattede gennemgang, bærer dokumentationen for DB2 stærkt præg af et meget SQL99 kompatibelt system. Som tidligere nævnt skyldes dette formentligt, at standarden er lagt tæt op ad DB2 og ikke omvendt. SQL99 bærer tydeligt præg af at være et komitéarbejde og IBM har formentligt haft en del at skulle have sagt. Det oprindelige forslag, SQL3, tog nærmest direkte udgangspunkt i Informix/Illustra, men er langsomt blevet drejet over i retning af DB2, og er altså endt der.

Helt generelt kan man sige at de tre systemer ikke er enige om syntaksen. Selvom dette må siges at være sekundært, betyder det alligevel, at applikationer vil have svært ved at flytte fra et databasesystem til et andet. Det bliver interessant at se fremover om de øvrige produkter ændrer syntaks og derved bliver mere SQL99/DB2 kompatible, eller om de holder fast ved deres egen måde. Personligt mener jeg, at hvis ændringen ikke sker meget snart, så sker den ikke. Jo flere kunder som benytter en leverandørs proprietære syntaks, jo sværere bliver det at lave den om.

Tabellen på næste side viser en oversigt over hvilke egenskaber de forskellige systemer understøtter. Plus angiver, at egenskaben er understøttet af det givne produkt. Minus angiver, at den ikke er.

EGENSKABER \ PRODUKTER	ORACLE	INFORMIX	DB2
Distinkte typer	÷	+	+
Strukturerede typer	+	+	+
Unavngivne rækketyper	÷	+	÷
Stærk typekontrol	+	+	+
Subtyper	+	+	+
Abstrakte typer	+	÷	+
Referencer	+	÷	+
OIDer: sys.gen / afledt / brugerdefineret	+ / + / ÷	÷ / ÷ / ÷	÷ / ÷ / +
Operationer	+	+	+
Selv-reference i typens operationer	+	÷	+
Brugerdef. `MAP` og `ORDER` operationer	+	÷	÷
Brugerdefinerede typekonverteringer	÷	+	÷
Typedefinerede tabeller	+	+	+
Tabelhierarkier	+	+	+
Polymorfi på tabelniveau	+	+	+
Polymorfi på attributniveau	÷	÷	÷
Simuleret indkapsling vha. rettigheder	+	÷	+
Uvægtet kvantitativ sammenligning :	14 / 19	10 / 19	13 / 19

8 ODB i OO systemudvikling



I dette afsnit vil jeg beskrive, hvordan jeg ser de objektorienterede databaser og databasesystemers placering i objektorienteret systemudvikling[26]. Jeg vil diskutere anvendeligheden af objektorienterede databaser og databasesystemer. Hvordan kan de nye egenskaber benyttes, og hvilke faldgruber bør man undgå. Derefter vil jeg kort komme ind på objektorienteret design. Hvilke fordele og ulemper kan det give. Afslutningsvis vil jeg kort beskrive, hvorledes multipel klassifikation ville kunne afhjælpe visse designmæssige problemer ved objektorienterede databaser.

Hele dette afsnit er grundlæggende et udtryk for min personlige holdning til emnet objektorienterede databaser i objektorienteret systemudvikling. Det bygger på tanker og erfaringer, jeg har gjort mig, igennem mit studie og mit arbejde som systemudvikler. Det bør derfor ikke betragtes som værende en komplet gennemgang af f.eks. objektorienteret design. Det er blot en gennemgang, af de af mine synspunkter og holdninger, som jeg mener, er relevante for specialet.

Generelt vil objektorienterede databaser i dette afsnit omfatte både objekt-relationelle systemer og ”rene” objektorienterede databaser. Hvis der er specifikt tale om den ene eller anden gruppe, vil dette fremgå af teksten.

8.1 Anvendelighed af ODB

I dette afsnit vil jeg forsøge at belyse de objektorienterede databasers anvendelighed. Til det formål vil jeg komme ind på følgende spørgsmål:

- Hvilken funktion skal de opfylde?
- Er der nogen grundlæggende problemer med at bruge dem?
- I hvilke systemer vil de have størst anvendelighed?

[26] ”ODB” skal i denne sammenhæng forstås som både objektorienterede databaser og databasesystemer.

8.1.1 Mere end blot persistens?

ODMG har den holdning, at objektorienterede databasesystemers primære funktionalitet bør være at give **persistens** til objektorienterede programmeringssprog. Det er med andre ord lagring af værdier, som er deres primære opgave. Det er således et ønske, at værdier og variable i programmeringssproget blot kan angives som værende persistente, og så behøver programmøren ikke at tænke nærmere over det. Vedkommende er sikret, at tilstanden varer ved efter programmet er afsluttet.

Der kan ikke herske nogen tvivl om, at databaser har lagring af data som noget helt centralt, men med introduktionen af relationelle databaser og SQL, blev denne rolle udvidet. Nu bruges databasesystemerne også til at foretage komplekse forespørgsler og analyser på de lagrede data. Selv en simpel *join* foretager en forholdsvis kompleks behandling af data. Hvis databasesystemerne udelukkende tilbød persistens, ville det være nødvendigt at foretage sådanne analyser manuelt. Persistens er en meget central egenskab for databasesystemet, men absolut ikke den eneste.

8.1.2 Dobbelt vedligeholdelse?

En anden måde at modellere databasen på er ved at lave de samme typer i databasen, som benyttes i programmerne. Dette er f.eks. Stonebraker og Brown [1999] fortalere for. De har adskillige eksempler på typer, som de mener bør have 10-14 forskellige operationer hver. Typerne inkluderer geografiske figurer, billeder etc. De har ret i, at den slags ville være rart at have, men problemet er, at det er de samme operationer, som programmeringssprogene også har brug for. Det betyder, at enten skal databasesystemet overtage al behandling af værdierne eller også skal operationerne implementeres to gange. Det er muligt for databasesystemet at kalde eksterne operationer. Dette kunne være en løsning, men det kræver at programmerne udvikles i et sprog, som databasesystemet understøtter. Selvom koden kan genbruges, vil der stadig være risiko for at kun det ene system opdateres. Databasesystemerne kræver ofte, at koden er tilgængelig i et specielt bibliotek, og dette skal opdateres hver gang en ændring foretages.

8.1.3 Hvor og hvornår?

Sådan som jeg ser det, er den store fordel ved objektorienterede databasesystemer, at det er muligt at udvide databasesystemet med nye typer. Så langt er jeg enig med Stonebraker og Brown [1999]. Dog bør man begrænse den understøttede funktionalitet til det absolut mest nødvendige. Det er vigtigt, at skelne mellem programmernes rolle og databasesystemets rolle. Databasesystemet bør kun have funktionalitet nok til at sammenligne og udvælge værdier af den nye type, samt at kontrollere at værdierne er gyldige. Som grundregel ville jeg ikke lægge forretningslogik ned i databasen, med mindre der var en meget god grund.

Eksempel:
Et ordresystem benytter specielle ordrenumre som starter med enten I eller N efterfulgt af 4 cifre (I0001, I1234, N9876 etc.). I og N angiver henholdsvis om ordren er international eller national. Hvis en ordre liste skal sorteres, skal I og N ikke medtages. Kun løbenummeret skal benyttes.
Det vil her være fornuftigt at lave en OrdreNummer type, som kan håndtere sammenligning af to ordrenumre, samt kontrollere at de har den korrekte form. Omvendt ville det ikke være passende, at lægge operationer ned i databasen som foretog den egentlige behandling af ordren, selvom dette var muligt.

En type som OrdreNummer fra eksemplet vil være en fordel for systemudviklingen. Databasesystemet kan give en bedre håndtering af værdierne af typen OrdreNummer end af typen `CHAR(5)`. Dog kræver det, at de programmer, som allerede benytter ordrenumrene i databasen skrives om. Derfor tror jeg, at de nye objektorienterede egenskaber først og fremmest vil vinde indpas i nye systemer. Store eksisterende systemer med hundredvis af programmer, vil næppe kaste sig ud i en større omskrivning.

Præcis i det viste eksempel kunne en større omskrivning dog godt undgås. Dette skyldes, at der er et en-til-en forhold mellem de gamle attributter og de nye. I det tilfælde kan konverteringen håndteres af databasen ved hjælp af brugerdefinerede `CAST` operationer. Hvis ordrenummeret tidligere var gemt som f.eks. `CHAR(5)`, kunne man skabe en implicit konverteringsoperation mellem ordrenummertypen og `CHAR(5)`. Dette bringer os over i et helt separat del af SQL99 standarden, som beskriver overførsel af data mellem databasen og **værtssproget** (del 5: *"Host Language Bindings*"). Dette er udenfor specialets omfang, men det skal lige nævnes, at netop udvekslingen af data altid har været betragtet som et problem for SQL (*impedance mismatch*). Muligheden for at definere egne typer i databasen, burde også kunne hjælpe på dette anklagepunkt.

Den primære grund til, at jeg ikke bryder mig om at lægge for meget forretningslogik ned i databasen, er vedligeholdelsen. For igen at referere til Gamma [1995], bør programmer skrives til et interface, ikke en implementation. Dette er helt grundlæggende for den objektorienterede tankegang. Ved at holde de forskellige **applikationslag** (*tiers*) klart adskilte, bliver programmerne lettere at vedligeholde. Typiske lag er datalagring, forretningslogik og præsentation. Ved at gøre disse uafhængige af hinanden kan de ændres, uden at det påvirker de øvrige lag. Med indførelsen af den objektorienterede model i SQL99, frygter jeg, at den i visse tilfælde vil blive misbrugt, således at disse lag udviskes, og systemerne reelt set bliver mindre objektorienterede. Det faktum, at SQL99 ikke understøtter indkapsling særlig godt, forværrer blot situationen.

8.1.4 Komplekse typer

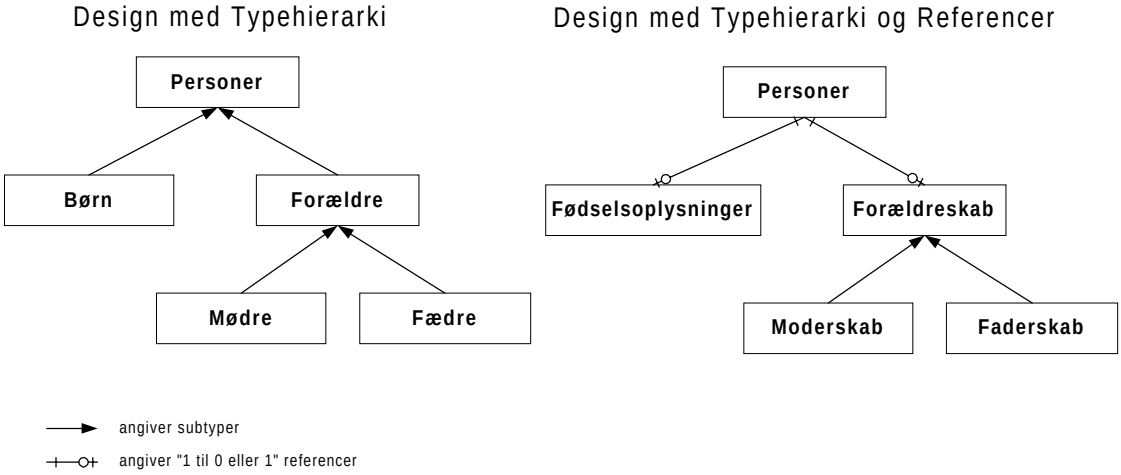
Når man står med et konkret behov for nye typer i databasen, bør man overveje følgende: Har jeg behov for objektorienterede egenskaber som indkapsling, polymorfi, identitetsbaserede referencer og subtyper eller er det bare strukturerede typer som f.eks. arrays jeg mangler? Ofte bliver behovet for sidstnævnte brugt som argument for objektorienterede databaser, men disse kunne lige så godt have været indført uden de objektorienterede egenskaber. De svarer fuldstændigt til de strukturerede typer i f.eks. Pascal (`array` og `record`).

8.2 Objektorienteret design

Grundlæggende set er objektorienteret design og relationelt design forskellige, men i forbindelse med de objekt-relationelle databaser er de objektorienterede egenskaber en overbygning på de relationelle. Det objektorienterede design vil derfor være baseret på det relationelle design (relationer, tupler, etc.) med visse udvidelser (subtyper, identitetsbaserede referencer etc.). Jeg vil dog stadig betegne designet som værende objektorienteret.

Som instruktor på DIKUs databasekursus i efteråret 1998 havde jeg den fornøjelse at rette adskillige rapportopgaver. Blandt andet skulle der laves et relationelt design. Det var interessant at se hvor pæne løsninger, der blev præsenteret, efter kun to måneders kursus, hvoraf kun en del havde omhandlet databasedesign. Ingen af de præsenterede løsninger var ens, men langt de fleste var ganske fornuftige. Samtidig lavede jeg et andendelsprojekt med Lars Hindborg Jensen [Hindborg Jensen & Borch, 1999], som blandt andet omhandlede objektorienteret design. Vi forsøgte at løse samme opgave, som var blevet stillet på kurset, med objektorienteret design. Det viste sig at være meget sværere at lave end det relationelle design. Det overrasker nok ikke specielt meget, eftersom vi begge var forholdsvis trænede i relationelt design og ikke i objektorienteret design. Alligevel kom vi frem til et grundlæggende spørgsmål, som man bør være bevidst om, når man laver objektorienteret design: Hvornår skal man benytte nedarvning (subtyper), og hvornår skal man benytte referencer? Svaret giver sig selv, når der er tale om relationelt design, eftersom kun referencer er til rådighed. Men når subtyper er blevet indført i SQL99, er det vigtigt at man reflekterer over valget, inden man kaster sig ud i objektorienteret design. Af hensyn til pladsen vil jeg ikke gengive hele diskussionen fra vores opgave her, men problemet var grundlæggende typemigrering. Databasen skulle indeholde medicinske oplysninger om forældre og børn i forbindelse med fødsler. Problemet opstod i det øjeblik, at databasen havde eksisteret længe nok til at registrerede børn kunne vende tilbage som forældre. De havde derfor brug for at blive typemigreret således, at de både kunne optræde, som værende børn af deres forældre og forældre for deres egne børn. Løsningen var at benytte referencer til at lagre informationer om fødsler og forældreskab i stedet for at have specielle Mødre, Fædre og Børn subtyper af Personer,

som indeholdt den nødvendige information. Nedenstående design er meget simplificeret i forhold til det oprindelige, men det fremhæver det væsentlige.



Ovenstående problem kunne også delvist løses ved hjælp af multipel nedarvning. Derved kunne der skabes en fælles subtype mellem Børn og Mødre (Børn-Mødre) og en fælles mellem Børn og Fædre (Børn-Fædre). Problemet ville dog stadig være at typemigrere værdier af typen Børn til typen Børn-Mødre eller Børn-Fædre.

Med indførelsen af subtyper i SQL99 vil ovenstående blive aktuelt for alle, som ønsker at benytte egenskaben i deres design. Det farlige er, at problemet med typemigrering ofte først opstår efter lang tid. I ovennævnte eksempel opstår problemet ikke før adskillige år senere. Det er derfor vigtigt, at man overvejer, om entiteter med nedarvede typer kan ændre sig med tiden.

8.3 Multipel klassifikation

Den klassiske objektmodel er ikke den eneste model for objektorientering. **Multipel klassifikation** kan betragtes som værende en mere general udgave af den klassiske objektmodel. I sidstnævnte har værdier præcist én ”mest specifik type”. Det vil sige, at den er begrænset til at være med i ét typehierarki (evt. med flere supertyper ved multipel nedarvning). Multipel klassifikation åbner mulighed for, at en værdi kan have flere typer på én gang [Fowler & Scott, 1997]. Dette er den eneste forskel i forhold til den klassiske objektmodel. Alle øvrige objektorienterede egenskaber beskrevet i afsnit 3.2 ”Den klassiske objektmodel” gælder også for multipel klassifikation (subtyper, indkapsling, operationer, polymorfi etc.).

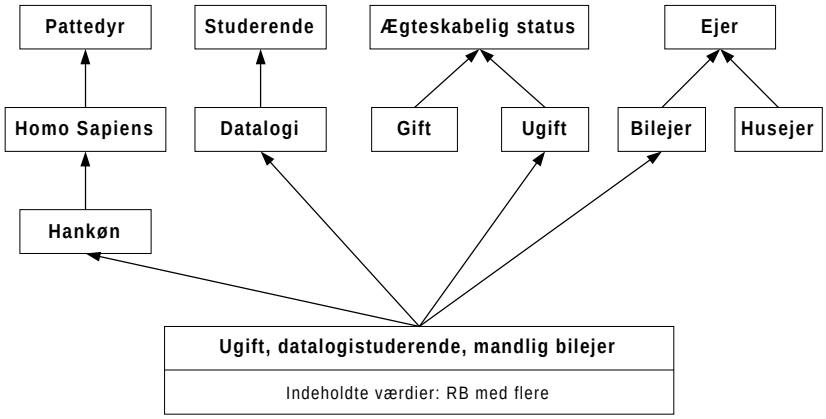
8.3.1 Motivation

Objektorientering startede som en ny måde at modellere virkeligheden på. I vores virkelige verden beskæftiger vi os hele tiden med logisk sammenhængende enheder, som har visse egenskaber, som vi kan benytte os af. Heraf kom den klassiske

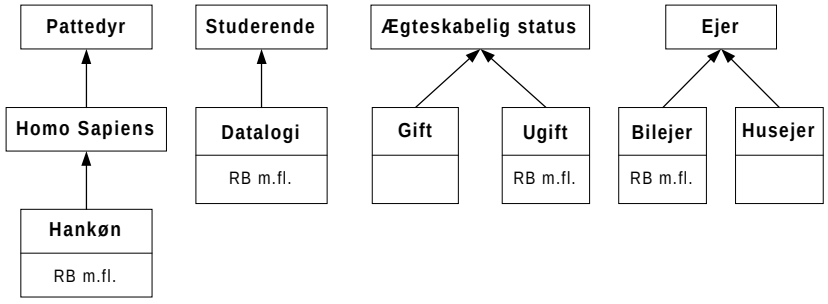
objektmodel. Multipel klassifikation er opstået ud fra den opfattelse, at virkelighedens logiske enheder ofte tilhører flere disjunkte typer på én gang.

Eksempel:
Forfatteren til dette speciale kan opfattes som værende af typen Homo Sapiens. Samtidig tilhører han også typerne Datalogistuderende, Ugift, Bilejer, Pattedyr, Hankøn.
Alle disse egenskaber kan sagtens modelleres med den klassiske objektmodel, men det vil kræve multipel nedarvning. Helt specifikt ville det kræve en særlig subtype, som havde alle de nævnte egenskaber som sine supertyper. Dette er nødvendigt, eftersom værdier kun kan tilhøre netop én type. Ligeledes ville enhver anden person skulle have sin særlige kombinationer af supertyper samlet i en subtype – som de formentligt ville være eneste værdi i.
I forbindelse med databasedesign er dette ikke hensigtsmæssigt. Specielt ikke når entiteten flytter over i en anden kommune og bliver gift. Så skal typehierarkiet udvides med en ny kombination. Løsningen kommer med multipel klassifikation, hvor en værdi kan tilhøre flere forskellige typer på en gang og kan dynamisk tildeles til og fjernes fra typerne. Nedenstående illustration viser forskellen på de to måder at modellere på. Ingen af typehierarkierne er på nogen måde komplette. Kun de relevante typer er medtaget. RB viser forekomsten af repræsentationen af mig i databasen.

Modellering med Den Klassiske Objektmodel



Modellering med Multipel Klassifikation



Ændres f.eks. den ægteskabelige status til Gift, flyttes RB blot fra Ugift til Gift. Købes et hus, tildeles RB typen Husejer. RB er dog stadig medlem af typen Bilejer, selvom disse har en fælles supertype. Afsluttes datalogistudiet, fjernes RB fra typen Datalogistuderende.

8.3.2 Beskrivelse

Indenfor multipel klassifikation identificeres en entitet stadig med et OID, som den gør i den klassiske objektmodel. Forskellen er, at der ikke initielt er tilknyttet nogen funktionalitet til entiteten. Den har så at sige ikke nogen defineret type (andet end at den er en OID). Efter at OIDen er oprettet, kan den tildeles forskellige typer. Disse kan være disjunkte typer, som f.eks. komplekse tal og streng, men dette er ikke noget krav. Man kan f.eks. angive, at OIDen skal tilhøre typen af komplekse tal. Derved får den automatisk alle karakteristika, som tilhører et komplekst tal. Samme OID kan derefter tildeles typen streng. Derved får den alle egenskaber, som er forbundet med at være en streng. OIDen er dog stadig også et komplekst tal og har alle disse karakteristika. Det er derved muligt at klassificere en entitet på flere forskellige måder på én gang (heraf betegnelsen ”multipel klassifikation”).

En kodestump som illustrerer ovenstående er vist nedenfor (skrevet i et fiktivt sprog inspireret af Java):

Eksempel:

```
x = new OID();       // nyt OID oprettes

x.addType(Complex);  // nu er x et komplekst tal
x.setComplex(3,7);

x.addType(String);   // nu er X også en streng
x.setString(”Dette er en streng”);

if (x.getComplex() = Complex(3,7) &&
    x.getString() = ”Dette er en streng”)
   doSomething();    // denne linie vil blive udført

x.dropType(Complex);

if (x.hasType(Complex))
  doSomething();     // denne linie vil blive sprunget over
```

Den væsentligste fordel ved ovenstående er, at typemigrering bliver meget enkel. Man behøver ikke at være så påpasselig med sit typehierarki, fordi ens OIDer kan tilhøre flere typer på én gang. Skulle en medarbejder får tildelt et ekstra arbejdsområde tildeles han blot sin nye stillingstype. Alle referencer til ham forbliver uændrede fordi hans OID er den samme.

8.3.3 Implementation

Multipel klassifikation er ikke særlig omstændigt at implementere i relationelle databasesystemer. For hver type oprettes en tilsvarende tabel. Tabellen indeholder de værdier, som er med i typen. Som tidligere defineret er en type en mængde af værdier (tilknyttet en mængde af operationer). Eftersom en tabel (relation) også er en mængde af værdier (tupler), kan denne nemt modellere en type. Attributterne i tabellen er OIDen på den værdi som er med i typen, samt alle de attributter som er nødvendige for at lagre en værdi af den relevante type. Eksemplet fra forrige afsnit kunne modelleres på følgende måde:

Eksempel:
Følgende to tabeller oprettes. Den ene indeholder alle værdier som er komplekse tal, den anden alle som er strenge. Den samme værdi kan godt optræde i begge tabeller på én gang. Det vil sige, at værdien har begge typer. Dette gælder for X. OIDen for X er indkodet som værdien 0001.

COMPLEX		
OID	`INT`	`INT`
0001	3	7

STRING	
OID	`VARCHAR(1024)`
0001	”Dette er en streng”

Når en værdi tildeles en type, indsættes en række i typens tabel med værdiens OID.

Eksempel:
`”X.addType(Complex)”`
udføres med følgende SQL-kommando:

```
INSERT INTO COMPLEX(OID) VALUES (X.OID);
```

For at undersøge om et OID har en given type, undersøges det om OIDen findes i den relevante tabel.

Eksempel:
`”X.hasType(Complex)”`
udføres med følgende pseudo SQL forespørgsel:

```
EXISTS (SELECT * FROM COMPLEX
        WHERE OID=X.OID);
```

Ligeledes fjernes en OID fra en type ved at slette dens række i typens tabel.

Eksempel:
`"X.dropType(Complex)"`
udføres med følgende SQL-kommando:

```
DELETE FROM COMPLEX
WHERE OID=X.OID;
```

Typehierarkier skal repræsenteres med én tabel per type i hierarkiet. Skal en type migreres op eller ned i hierarkiet tilføjes eller slettes den igen fra subtypens tabel.

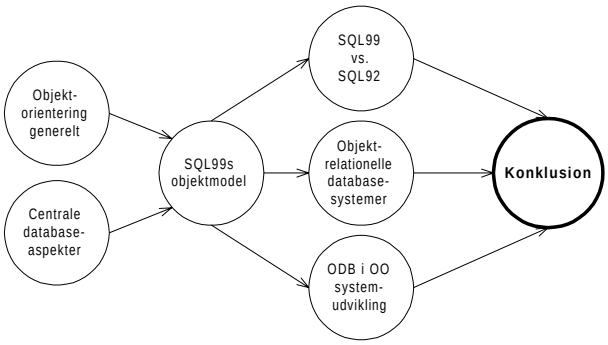
Dette ville gøre systemet langt mere dynamisk. Dog har dynamik sin pris. Den kode, som skal udnytte disse egenskaber, vil unægtelig blive mere kompliceret. Hver gang man får en værdi tilbage fra databasen, kræver det en nærmere undersøgelse at bestemme dens præcise typer. Dog vil det ofte kun være ganske få typer, som er relevante i en given sammenhæng.

Der er i princippet intet til hinder for at gøre ovenstående manuelt, men det ville være en stor fordel, hvis databasesystemet havde direkte understøttelse af egenskaberne.

8.3.4 Vurdering

Det gode ved denne objektmodel er, at den ligger så tæt op af den virkelige verden som muligt. Det gør det langt enklere at modellere den virkelige verden. Når nu SQL99 skulle udvides med objektorienterede egenskaber, havde jeg foretrukket denne model frem for den klassiske objektmodel, primært af hensyn til modelleringsevnen.

9 Konklusion



SQL99 er et meget kontroversielt emne indenfor databaseverdenen. Den forsøger at forene to forskellige paradigmer: det relationelle og det objektorienterede. Det er derfor oplagt, at den modtager skarp kritik fra puritister på begge sider. SQL99 må siges at have en meget pragmatisk tilgang til opgaven at forene de to paradigmer. Alligevel er SQL99s objektmodel meget omfattende. Det er svært at sætte en finger på egenskaber eller mangler, som er kontra-objektorienteret. Der er nogle få anklagepunkter, hvoraf indkapslingen er den væsentligste. Det ville have været meget enkelt at indføre forskellige indkapslingsniveauer, som kunne svare til dem, som findes i objektorienterede programmeringssprog. Den valgte løsning vil være meget besværlig at administrere og vil formentligt føre til at begrebet i de fleste henseender vil blive ignoreret. Den manglende indkapsling på tabelniveau skal nok mere ses som en bevidst objekt-relationel egenskab end en fejl eller mangel ved standarden. Alligevel mener jeg personligt, at den burde have været understøttet alligevel, eventuelt som et valgfrit supplement. Specielt fordi egenskaben alligevel kan simuleres ved hjælp af *views*.

Relationsdatabaserne har haft stor succes igennem de sidste 20 år. Dette skyldes blandt andet, at de er simple og hviler på et solidt teoretisk grundlag (den relationelle model og algebra defineret af Codd, 1972). Introduktionen af de objektorienterede egenskaber bryder delvist med dette fundament. Dertil kommer, at objektorientering generelt ikke har et tilsvarende solidt teoretisk grundlag. Dette kan føre til forståelsesproblemer blandt brugerne og forskellige fortolkninger af standarden blandt leverandørerne. Det klare og enkle billede af relationsdatabaserne kan derved blive forstyrret.

Umiddelbart lader det ikke til, at det bliver væsentligt mere kompliceret at benytte de nye egenskaber, hvis man bruger dem med måde. Dog kan de bruges til at lave meget komplekse strukturer, som kan være svære at overskue, og som let kan føre til fejl. Her tænker jeg specielt på de identitetsbaserede referencer. Hvis de ikke begrænses med virkefelts angivelser, kan de pege på kryds og tværs i databasen. Dette kan også medføre, at databasesystemet ikke kan foretage en fornuftig optimering af brugerens

forespørgsler. Ansvaret for en fornuftig svartid flyttes tilbage på brugeren, som det var tilfældet i f.eks. de hierarkiske databaser. Netop forespørgslernes deklarative natur er en stor styrke ved SQL. Desuden kan opslag kun foretages én vej, hvilket giver samme begrænsninger som i de hierarkiske databaser. Dette må betragtes som værende et skridt tilbage for alle andre end den meget erfarne programmør.

Generelt bør man kun benytte de objektorienterede egenskaber til at udvide mængden af typer, som databasesystemet kan håndtere. De nye typer bør understøtte de samme operationer, som de prædefinerede har. Det vil sige, at det primært drejer sig om sammenligning. Det er nødvendigt for databasesystemet at kunne sortere og udvælge de rette data baseret på brugerens forespørgsel. Jeg mener ikke, at man bør fylde databasen med forretningslogik, med mindre at der er gode grunde til det. Det er vigtigt, at bevare den opdeling af programmerne i adskilte lag, som er begyndt at vinde indpas i moderne systemudvikling, ellers risikerer man at systemerne ender med at blive mindre objektorienterede, end de er i dag.

Betragter man de tre omtalte databasesystemer, så omfatter de alle store dele af SQL99. Dette skyldes, at standarden har været på tegnebordet i længere tid. Det er tydeligt, at den har taget sit udgangspunkt i Informix/Illustra, men har så bevæget sig meget tæt på DB2. Derfor er det måske heller ikke overraskende, at Informix klarer sig dårligst i sammenligningen med SQL99, men de manglende referencer i Informix er et reelt anklagepunkt.

Nogen påstår, at ingen standard på over 1000 sider er blevet fuldt implementeret. Alene part 1 og 2, som dette speciale har omhandlet, er på over 1200 sider tilsammen. Man kunne frygte, at standarden ville bukke under for sin egen størrelse, og at leverandørerne ikke ville ønske at bruge de nødvendige ressourcer på at implementere en så omfattende standard. Dog bringer SQL99 hovedsageligt blot standarden *up-to-date* med markedet. Mange af de nye egenskaber har været implementeret i flere år. Jeg tror, at standarden vil vinde indpas, men grundet det lidt usikre grundlag, den er baseret på, vil systemerne formentligt variere mere, end det var tilfældet med SQL92.

De nye objektorienterede egenskaber vil også blive brugt – om ikke andet, så fordi det er moderne at gøre tingene objektorienteret. Netop det moderne, tror jeg, spiller en større rolle, end man skulle tro. De, som kraftigt taler for de objektorienterede udvidelser, bruger ofte meget tid på at forklare, hvorfor det er nødvendigt. Det tror jeg, at de gør, fordi langt de fleste, som benytter relationsdatabaser i dag, ikke ser nogen mangler. Det er ganske specialiserede eksempler, som nævnes, når fordelene skal fremhæves. Dermed ikke sagt, at de objektorienterede egenskaber ikke har deres berettigelse. Det mener jeg, at de har. Men jeg tror, at grunden til, at leverandørerne har taget konceptet så hurtigt til sig, skyldes, at det er et godt salgsargument. Objektorientering sælger. Man kan spørge sig selv, hvis behovet for det er så stort, hvorfor tog det så over 20 år at introducere det? Jeg tror, at i langt de fleste tilfælde er der tale om, at det er *nice-to-have*. Det har også sin berettigelse, men det store skridt i

evolutionen, mener jeg ikke, at der nødvendigvis er tale om. Skulle en objektmodel indføres, havde jeg foretrukket multipel klassifikation. Det havde bedre kunnet understøtte databaserne på områder, som har stor betydning som f.eks. databasedesign og typemigrering.

Min anbefaling til folk, som skal benytte SQL99, er, at man bør gøre sig klart, præcist hvilke egenskaber man har brug for, og hvordan man vil bruge dem. Den relationelle model er langt hen ad vejen svær at misbruge. Sådan forholder det sig ikke med SQL99s objektmodel. Den kræver mere omtanke, forsigtighed, planlægning og selvdisciplin. Allesammen egenskaber, som efter min erfaring ikke er kendetegnende for flertallet af verdenens systemudviklere (undertegnede inklusiv!) ☺

10 Efterskrift

Opgaveforløbet er gået meget glat. Mit kendskab til SQL99 ved opgavens start var meget begrænset. Jeg havde dog et vist kendskab til SQL3 i en forholdsvis tidlig udgave [Ullman & Widom, 1997]. Mine forventninger til standarden var derfor ikke høje, men jeg er blevet positivt overrasket. Jeg havde ikke forestillet mig, at den ville indeholde et så omfattende objektorienteret paradigme.

Jeg vil takke Troels Andreasen (vejledning), Jyrki Katajainen (vejledning), Martin Jensen (information vedrørende Oracle 9i), Tommy Borch (korrektur) og Lars Hindborg Jensen (korrektur). Derudover vil jeg særligt takke min kæreste Nina for interesse, opmuntring og opvartning i hele perioden, samt vores kommende barn, som gjorde, at jeg fik skrevet specialet færdigt indenfor en fornuftig tidsramme.

RASMUS BORCH, FEBRUAR 2001

11 Litteratur

[Atkinson, 1993]

Atkinson, *The Object-Oriented Database System Manifesto*, Artikel, 1993

Hentet fra: *http://www.acm.org*

Tidligt grundlag for de ”rene” objektorienterede databaser.

[Bird & Wadler, 1988]

Bird & Wadler, *Introduction to Functional Programming*, Prentice Hall, 1988

Beskrivelsen af funktionsprogrammeringssprogenes manglende variable.

[Cattell, 1995]

Cattell (redaktør), *The Object-Database Standard: ODMG-93, Release 1.2*, Morgan Kaufmann, 1995

Ønsket om dynamiske typer.

[Cattell & Barry, 1997]

Cattell & Barry (redaktører), *The Object-Database Standard: ODMG 2.0*, Morgan Kaufmann, 1997

ODMGs forhold til SQL92/99.

[Date, 2000]

Date, *An Introduction to Database Systems, 7th Edition*, Addison-Wesley, 2000.

Omtale af SQL99. Blandt andet ”The Great Blunders”.

[Date & Darwen, 1998]

Date & Darwen, *Foundation for Object/Relational Databases – The Third Manifesto*, Addison-Wesley, 1998

Kun Date og Darwens typebegreb.

[Elmasri, 2000]

Elmasri, *Fundamentals of Database Systems*, Addison-Wesley, 2000

Hovedsageligt kapitel 13 – om SQL3.

[Fowler & Scott, 1997]

Fowler & Scott, *UML Distilled – Applying The Standard Object Modelling Language*, Addison-Wesley, 1997

Beskrivelsen af multipel klassifikation.

[Gamma, 1995]

Gamma, *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995

Gammas typebegreb og vejledning omkring objektorienteret design.

[Gulutzan & Pelzer, 1999]

Gulutzan & Pelzer, *SQL-99 Complete, Really*, R&D Books, 1999

Hovedsageligt kapitel 11 (”*Row Types*”) og 27 (”*User-Defined Types*”). Det gode ved bogen er, at den er meget omfattende og præsenterer standarden i en mere læselig form. Ulemperne er, at eksemplernes syntaks ofte er forkert og forklaringerne er rodede og mangelfulde.

[IBM, 2000]

IBM, *IBM DB2 Universal Database Documentation*, International Business Machines Corporation, 2000

Hentet fra: *http://www-4.ibm.com/software/data/db2/library/publications/*

Beskrivelse af DB2s objekt-relationelle egenskaber.

[Informix, 1999]

Chase, *Extending INFORMIX Universal Server*, Informix Press, 1999

Hentet fra: *http://www.informix.com/answers/english/ssql.htm*

Beskrivelse af Informix’s objekt-relationelle egenskaber.

[ISO, 1999]

ISO/IEC 90751:1999, *Database Language SQL*, ISO, 1999

Udvalgte dele med betydning for SQL99s objektmodel er benyttet fra *"Part 1: SQL/Framework"* og *"Part 2: SQL/Foundation"*.

[Hindborg Jensen & Borch, 1999]

Hindborg Jensen & Borch, *Objektorienterede databaser – en sammenligning*, DIKU, 1999

Sammenligning af ODMG og SQL92, samt Oracle 8 og Jasmine 1.2. Indeholder blandt andet et objektorienteret design præsenteret i objektorienteret form, objekt-relationel form og relationel form.

[Oracle, 1998]

Oracle, *Oracle Document Library*, Oracle Corporation, 1998

Hentet fra: Oracle 8.1.3 CD fra Oracle Denmark.

Beskrivelse af Oracles objekt-relationelle egenskaber.

[Ramakrishnan & Gerhke, 2000]

Ramakrishnan & Gerhke, *Database Management Systems, Second Edition*, McGraw–Hill, 2000

Kapitel 25 – OODB. Beskrivelsen præsenteres som værende baseret på SQL99. Faktum er, at indholdet nærmere omhandler SQL3.

[Stonebraker & Brown, 1999]

Stonebraker & Brown, *Object-Relational DBMSs – Tracking the Next Great Wave, Second Edition*, Morgan Kaufmann, 1999

Beskrivelse af objekt-relationelle egenskaber og argumentation herfor. Syntaks og indhold minder ikke overraskende om Informix/Illustra.

[Ullman & Widom, 1997]

Ullman & Widom, *A First Course in Database Systems*, Prentice-Hall, 1997

Beskrivelse af ODL/OQL og SQL3.

12 Appendix A – Standardens opbygning

Standarden [ISO, 1999] er opdelt i fem dele:

- Part 1: Framework (SQL/Framework)
- Part 2: Foundation (SQL/Foundation)
- Part 3: Call-Level Interface (SQL/CLI)
- Part 4: Persistent Stored Modules (SQL/PSM)
- Part 5: Host Language Bindings (SQL/Bindings)

Kun de to første er blevet brugt til dette speciale. Første del beskriver generelt indholdet i standarden. Anden del går nærmere i detaljer med indholdet og den præcise grammatik for syntaksen.

Som de fleste andre værker af den slags indeholder standarden en lang række henvisninger til andre sektioner. Desuden er den meget præcis med at benytte de samme gloser og navne hele standarden igennem. Henvisningerne gør det til en fordel at læse den i elektronisk form, da man derved hurtigere kan søge alle forekomster af de nøgleord, man er interesseret i.

Eftersom standarden er en kraftig udvidelse i forhold til SQL92, er det ikke alle tilføjelser, som er obligatoriske. SQL99 indeholder en central kerne af funktionalitet, som skal understøttes. Dette kaldes for *Core SQL*. Derudover har standarden en lang række *features*. Disse er hver identificeret med en kode og dækker over valgfri tilføjelse til standarden. Centrale *features* for den objektorienterede del af standarden er blandt andet:

- S023 Basic structured types
- S024 Enhanced structured types
- S041 Basic reference types
- S043 Enhanced reference types
- S051 Create table of type
- S071 SQL paths in function and type name resolution

I forbindelse med grammatikken angives efter hver definition hvilke regler, som gælder alt efter hvilke features, som er understøttet. Disse regler kaldes for *Conformance Rules*. Standarden giver derved mulighed for, at ikke alle databasesystemer behøver at indeholde den fulde funktionalitet.

Hvert system bør understøtte en såkaldt *Object Identifier*. Dette har intet at gøre med de unikke værdier, som identificerer rækker i typedefinerede tabeller. Det er en

strengværdi, som andre systemer kan aflæse og deraf afgøre i hvor høj grad databasesystemet understøtter SQL standarden. Strengen indeholder blandt andet en angivelse af hvilken standard, som er understøttet (1987, 1989, 1992 eller 1999). Hvis ”1999” understøttes, skal databasesystemet være kompatibelt med *Core SQL*. Derudover er det muligt at angive hvilke andre dele af standarden, som også understøttes.

Det bør nævnes, at standarden absolut ikke egner sig til introduktion eller indlæring af SQL99. Har man ikke læst diverse beskrivelser af standardens indhold inden, så vil det nok være en tidskrævende opgave at få et overblik over indeholdet. Hovedsageligt består den af en række syntaksregler på pseudo BNF form med tilhørende kommentarer. Disse er meget kortfattede. Det er tydeligt, at præcision og konsekvens er sat i højsæde, på bekostning af pædagogikken.

13 Appendix B – Ordliste

Dette appendix giver en oversigt over udvalgte fagord, som benyttes i dette speciale.

Term	Betydning i dette speciale	SQL99 Standarden
Attribut	Navn og type tilknyttet en kolonne i en tabel.	Afhænger af konteksten: *Column* (Tabeller), *Field* (Rækketyper), *Attribute* (Brugerdefinerede typer)
Basistabel	Tabel uden supertabeller, også kaldet en rod tabel (*root table*).	*Maximal supertable*
Domæne	Se *Type.*	*Domain*
Entitet	En faktisk forekommende ting, personer, steder eller lignende.	Omtales ikke
Extent	Mængden af instantierede objekter som er skabt fra en given klasse.	*Table*
Klasse	Se *Type.*	*Type*
Objekt, konstant	Se *Værdi.*	*Value*
Objekt, variabelt	Se *Variabel.*	*Column, Field, Attribute*
OID	Objekt Identifikator.	*Self-referencing column*
Operation	En funktion som transformerer værdier.	*Method*
Reference	En variabel indeholdende en OID.	*Reference*
Relation	Variabel indeholdende en mængde af tupler.	*Table*
Relvar	Variabel indeholdende en mængde af tupler (Dates udtryk).	*Table*
Række	Se *Tupel.*	*Row*
Rækketype	Typen på en tupel.	*Row Type*
Tabel	Se *Relation.*	*Table*
Tupel	Kompleks værdi, sammensat af flere værdier. Typerne på de enkelte delværdier er angivet af rækketypen.	*Row*
Type	Mængde af værdier tilknyttet mængde af operationer.	*Type*
Variabel	Pladsholder for værdier identificeret ved adresse.	*Table, Field*
Værdi	Unikt element uafhængigt af tid og rum.	*Value*

14 Appendix C – Kollektionstyper

En af de egenskaber, som ofte omtales i forbindelse med objektorienterede databaser, er kollektionstyper. For mig at se har kollektionstyper ikke noget med objektorienterede egenskaber at gøre – ikke mere end alle andre typer har. Jeg ser dem som værende mere komplekse byggesten til at konstruere strukturerede typer ud fra. Derfor er deres omtale henvist til dette appendix. Jeg vælger at omtale dem kort, da de stadig er en vigtig del af udvidelserne til SQL99s typebegreb.

Kollektionstyper beskrives ganske kort i første del af standarden. Både lister, mængder, multimængder[27] og arrays er omtalt, men kun arrays er defineret mere præcist i standarden. Alle relationsdatabaser har jo understøttelse af mængder i form af relationer. Mange giver også mulighed for at skabe tabeller uden primærnøgler eller unikke indekser, hvilket resulterer i multimængder. Lister er svære at finde i relationsdatabaser. Man kan argumentere for, at resultatet af en forespørgsel med `ORDER BY` er en liste. Dog kræver en liste ikke, at indeholdet er sorteret efter et givet kriterie. Det er også muligt at sammenkæde to lister, uden at rækkefølgen ændres internt i de to. Alt dette taler imod, at en `ORDER BY` skulle generere en liste. En ”ordnet mængde” er nok et bedre udtryk.

Det interessante er imidlertid, når kollektioner understøttes på attributniveau. Dette gælder i SQL99 for array. Det er muligt at definere en attribut som havende typen `CHAR(10) ARRAY[5]`. Dette skaber et array med fem elementer af typen `CHAR(10)`. Der er to begrænsninger på typen af elementerne.

1. Arrays kan kun være én dimensionale (ingen `INT ARRAY[10] ARRAY[10]`).
2. Elementerne i et arrays kan ikke indeholde typen `REF`.

Størrelsen på arrayet er dynamisk. Den angive værdi er den maksimale størrelse. Antallet af elementer i et array på et givent tidspunkt kan bestemmes med udtrykket

```
CARDINALITY(array_attribut)
```

Oracle understøtter arrays, som de mere sigende kalder for `VARRAY` (*Varying Array*). De har desuden også support for indlejrede tabeller, hvilket i praksis giver mulighed for at lave attributter med typerne mængde og multimængde.

Informix har en meget gennemført håndtering af kollektionstyper. Den understøtter `SET`, `MULTISET` og `LIST`. Følgende eksempel er fra dokumentationen:

[27] *bags*, tillader dubletter

```
”CREATE TABLE tab1
(       int1 INTEGER,
        list1 LIST( ROW(a INTEGER, b CHAR(5) ) NOT NULL),
        dec1 DECIMAL(5,2)
 )”
```

[Informix, 1999]

Kollektionstyperne er direkte understøttet som typer. Det ligger meget tæt op ad SQL3, men kom ikke med i SQL99.

DB2 understøtter ingen kollektionstyper.

I et det føromtalte projekt med Lars Hindborg Jensen analyseres kollektionstyper nærmere [Hindborg Jensen & Borch, 1999].