SISU

PUBLIKATION 96:05

RAPPORT – APRIL 1996

Objektorienterade databashanterare

– en introduktion

Stig Berild

SVENSKA INSTITUTET FÖR SYSTEMUTVECKLING

SISU

Objektorienterade databashanterare – en introduktion

1. Inledning 1

2. Bakgrund 2

2.1 Historik, allmänt 2
2.2 Behov, drivkrafter 3

3. Relationsmodeller – datamodeller – objektmodeller 5

3.1 Relationsmodell 5
3.2 Semantisk datamodell contra relationsmodell 5
3.3 Objektmodellen i dbmstappning 7

4. Objektdatabaser, en introduktion 9

4.1 Rdbms, som jämförelse 9
4.2 Odbms, grundidé 12
4.3 Objektidentifierare 14
4.4 Samspel odbms – tillämpning 15
4.4.1 Objektreferenser i virtuellminnet 15
4.4.2 Objektreferenser i databasen 19
4.4.3 Inläsning från databasen 20
4.5 Klient/server-miljö 23
4.5.1 Klient/server-miljö för rdbms 23
4.5.2 Klient/server-miljö för odbms 24
4.6 Odbms – konventionellt språk? 27
4.7 Definition av en tillämpning 27
4.7.1 Översikt 27
4.7.2 Anpassning av språk till odbms-samverkan 28

5. Konventionell dbms-funktionalitet i odbms-tappning 32

5.1 Inledning 32
5.2 Transaktionshantering 33
5.3 Versionshantering 34
5.4 Skydd 35
5.5 Frågespråk 35
5.6 Schemahantering 37
5.7 Distribution 38
5.8 Händelsestyrda operationer 38
5.9 Backup, recovery 39

6. Slutsatser 40

1. Inledning

Objektorienterade databashanterare (odbms) innehåller egenskaper som utgår från den bakomliggande objektmodellen, det objektorienterade programmeringsspråk och de tillämpningsområden den primärt avses stödja. Därutöver återfinns normalt en uppsättning mer konventionella databashanteringsfunktioner som backup, recovery, säkerhet, låsning, fleranvändarfunktionalitet, ... Vissa av dessa funktioner har i odbms-produkter givits en speciell "touch", återigen baserat på de krav de tänkta tillämpningsområdena ställer. Rapporten tar primärt upp den funktionalitet som kan bedömas vara odbms-unik och som i någon form återfinns i de flesta av de produkter som för närvarande erbjuds kommersiellt.

Avsnitt 2 ger ett historiskt perspektiv på odbms. Den som vill förstå odbms måste ha kännedom om de grundläggande egenskaperna i objektmodellen. Avsnitt 3 diskuterar kort objektmodellen och ställer den i perspektiv mot relationsmodellen och konventionella datamodeller. En utförligare beskrivning av objektmodellen finns i SISU rapport 13 "Objektorientering – de vanligaste begreppen". I avsnitt 4 diskuteras översiktligt typiska odbms-egenskaper och de avgörande skillnaderna mot relationsdatabashanterare (rdbms). På sätt och vis är detta rapportens centrala avsnitt. Vissa konventionella databasfunktioner har realiserats på ett delvis nytt sätt inom odbms för att svara upp mot teknik och tänkta användnings-områden. Några av dessa berörs i avsnitt 5. Rapporten avslutas med några slutsatser i avsnitt 6.

Objektorienteringens olika infallsvinklar har ofta hängivna förespråkare. Odbms-området utgör inget undantag därvidlag. Kritiska bedömningar saknas dock inte. Osäkerhet kring idé och teknik saknas inte heller. Föreliggande rapport syftar till att ge en, om möjligt, neutral introduktion till odbms för den som vill veta "vad det är för nå't".

I en separat SISU-publikation "ODBMS – state-of-the-art" presenteras några produkter samt diskuteras tillämpningsområden, marknad och trender. Den som vill veta mer om de så kallade hybrid-ansatserna, d v s produkter baserade på relationsmodellen men utökade med viss objektorienterade påbyggnader, hänvisas till kommande SISU-publikationen "Hybrid-databaser – vad är det?".

2. Bakgrund

2.1 Historik, allmänt

Objektorienterade databashanterare (odbms) har uppstått i skärningslinjen mellan programmeringsspråk och databashantering.

Programmeringsspråk

Programmeringsspråken kompletterades under 60/70-talen med begreppet abstrakt datatyp. Det kom sedermera att vidareutvecklas och resultera i objektklass-begreppet, framförallt genom influens av SIMULA-språkets objektmodell. Syftet var att tillhandahålla modelleringsbegrepp som gjorde det möjligt att avgränsa och sammanföra funktioner/operationer med de data de opererar på samt att åstadkomma en samverkansform mellan modellens olika komponenter. Modellen skulle vara stringent och appellera till en intuitivt naturlig abstraktion och struktur av ett dynamisk miljö. De främsta företrädarna för den objektorienterade språkfalangen (oo-språk) är i dagsläget C++ och Smalltalk. Andra språk finns, t ex Ada och Eiffel.

Oo-språken har sitt ursprung i forskningsprojekt, idéskisser, prototyper eller som stöd något specifikt behov. Under tidiga faser av nya trender tenderar utveckling av olika ansatser ske parallellt och utan koordination. Oo-språksområdet utgör inget undantag därvidlag. Resultatet har blivit att varje språk realiserat sin egen variant av objektmodell, dock ska sägas, med stora likheter. Flera oo-språk har en god stabilitet och är föremål för officiell standardisering.

Databashantering

Databasområdet såg under 70-talet framväxten av databas-principer baserade på hierarkiska och nätverksmodeller. 80-talet innebar ett genombrott för produkter baserade på relationsmodellen och dess språkgränssnitt SQL. Dessa relationsdatabashanterare (rdbms) har under 90-talet befäst och stärkt sin ställning. Dominansen är för närvarande stark. Den trenden väntas kvarstå under överskådlig tid.

Även produkter baserade på någon mer semantisk datamodell (konceptuell modell) har tagits fram men inte nått någon större marknad trots betydligt starkare semantisk uttryckskraft. Orsakerna är flera. Dels växte de fram vid en tid då relationsmodellen redan fått starkt fäste såväl inom forskning som produktutveckling. Dels visade de upp en splittrad bild – många olika modellvarianter. Databasområdet har också haft många andra angelägna problem att tackla som fångat energi och intresse. Dit hör klient/serverarkitekturer, datadistribution, transaktionshantering, fleranvändarmekanismer, backup, recovery, behörighet, m m. Trösterikt nog har de semantiska datamodellerna dock utgjort en primär idékälla vid utveckling av senare tiders odbms.

Skärningspunkten

Under en exekvering av ett oo-språksprogram kommer och går objekt i enlighet med programmets logik. Vid avslutning av exekveringen (sessionen) frigörs exekveringsminnet. Samtliga eventuellt kvarvarande objekt försvinner. Detta är i många sammanhang fullständigt acceptabelt. Man kallar vanligtvis dessa objekt för *transienta* eller *tillfälliga* (eng. transient).

I andra sammanhang, inte minst inom nya tillämpningsområden, tillkommer behov av att ha kvar objekten även sedan sessionen avslutats. Objekts livscykel behöver ju inte

nödvändigtvis sammanfalla med livstiden för den process som genererat det. Dessa mer långlivade objekt kallas *beständiga* (eng. persistent).

En lösning för att härbärgera dess objekt är att t ex använda sig av ett ordinärt rdbms. Nackdelen är dels att man då måste översätta objekten till/från någon motsvarande struktur enligt relationsmodellen, dels att man måste använda sig av SQL, ett fristående språk som inte är integrerat med oo-språket.

Självklart vore det en fördel om objekten kunde lagras undan på ett sätt som svarade mot dess minnesdisposition under sessionen så att det bara var att hämta in objektet "rakt av" i en senare session och "köra igång". Samtidigt blir det nu plötsligt angeläget att hantera objekten på ett korrekt och konsistent sätt under dess livstid. Behovet av beständiga objekt på språksidan sammanfaller naturligt med de faciliteter databas-området av tradition tillhandahållit. Sammansmältning behövs men är i realiteten varken enkel, eller i vissa stycken ens rimlig. Eftersom rdbms inte utgjorde någon idealisk part skapades en ny typ av databasfunktionalitet. Startskottet för odbms hade gått.

Produkter utvecklas

1986 och 1987 introducerades GemStone, Gbase och Vbase. Därefter har produkter kommit och gått. I dagsläget finns ca 6-8 stycken kommersiella odbms.

De flesta odbms var till en början renodlat inriktade på att hantera objekt, skapade i en C++-tillämpning. OO-språksinfluensen var till en början betydligt starkare än databas-influensen. Uttryck som "object-oriented database programming languages" och det kortare "persistent programming languages" kom till användning. Efter hand ökade dock intresset för att komplettera C++-modellen med impulser från semantiska datamodeller. Den starka bindningen till C++ började luckras upp. Varför inte stödja flera oo-språk? Varför inte tillåta frågespråk mot databasen? Vanliga dbms-funktioner inkluderades.

Leverantörerna valde att realisera funktionaliteten på olika sätt. Leverantörerna var dessutom små och sårbara, produkterna instabila och under ständig utveckling. Tilltänkta kunder kände oro, agerade försiktigt. Branschen levde farligt. Följdriktigt etablerade de dominerande odbms-leverantörerna 1991 en gemensam organisation med syfte att ensa produktegenskaper och återställa förtroende. Gruppen kom att kallas Object Database Management Group, i vardagligt tal kortfattat ODMG. 1993 presenterade ODMG en gemensam gränssnittsspecifikation under beteckningen ODMG-93. Samtliga medlemmar ställde sig bakom och lovade att realisera specifikationen i sina produkter. Uppdateringar i form av Release 1.1 (1994) och Release 1.2 (slutet av 1995) existerar.

Rdbms-leverantörerna har samtidigt hållit sitt vakade öga på utvecklingen. Var detta starten för nästa generations dbms eller var det en dagslända? Var den långsiktiga marknaden i farozonen? Relationsmodellens svagheter har alltid inneburit ett långsiktigt hot för att nya, bättre modeller skulle kunna få fotfäste. Parallellt med introduktion av nya faciliteter i rdbms-produkterna har därför kunnat skönjas ett klart ökat intresse för en SQL-påbyggnad med nya, mer objekt-orienterade egenskaper. Produkter, med objekt-orienterade anpassningar av SQL, har också tagits fram. De brukar gå under beteckningen "hybrider". Vi lämnar hybriderna åt sitt öde i denna rapport eftersom de presenteras i en fristående rapport.

2.2 Behov, drivkrafter

Det är knappast riskfyllt att konstatera att samhällsutveckling, globaliserade och snabbt varierande marknader, ökande kundkrav, m m i allt snabbare takt ställer nya krav på verksamheter, krav som bland annat innebär ökad och mer avancerad informations-

hantering. Samhället tar de första stegen in i informationssamhället, baserat på mer eller mindre avancerade informationstjänster.

Existerande tillämpningsområden måste svara upp mot nya, mer diversifierade behov. Nya tillämpningsområden ställer i vissa avseenden helt nya krav. Tillämpningarna blir alltmer komplexa, har mer avancerade gränssnitt, innehåller i ökad utsträckning multimedia, bygger på samverkan, o s v.

I ett databasperspektiv kan bl a följande tendenser skönjas
- komplexare datastrukturer
- mer föränderliga datastrukturer
- mer mångskiftande användning
- ökad decentralisering, distribution, replikering
- klient/server arkitekturer
- från passivt "lager" till aktiv, levande och intelligent aktör.

Databaser har historiskt sett främst stött de så kallade administrativa tillämpningsområdena. En klar tendens mot ökat behov av databasstöd inom mer tekniskt orienterade tillämpningar har på senare tid kunnat konstateras. Dit hör tele/data/mobil-kommunikation, datorstödd konstruktion (CAD/CAM), styrning/övervakning, produktionsplanering, vissa aspekter av geografiska informationssystem (GIS), ...

Givetvis ställer denna inriktning hårdare och delvis helt nya krav på tillämpningar och därmed även indirekt på systemutvecklingsmetoder.

Här kommer objektorientering in i bilden som en förmodad befriare. I den bilden återfinns bland annat de objektorienterade databashanterarna (odbms).

Rdbms dominerar dbms-marknaden. De baserar sig på relationsmodellen. Den definierades av Codd 1970 och har sedermera preciserats, förfinats och standardiserats i olika versioner under beteckningen SQL (SQL-1987, SQL-1989, SQL-1992).

Odbms baserar sig på en objektmodell. Många varianter av objektmodeller finns inom objektorienteringens olika teknikområden. Tyvärr saknas här ännu en modell-standard. (Vissa ansträngningar för att råda bot på detta pågår.) Dock är det mesta i de olika odbms-modellerna överensstämmande.

Vissa anser att odbms i sinom tid kommer att ta över databasmarknaden från rdbms. Andra anser att den bara kommer att fylla en nischroll. Ytterligare andra anser att de båda falangerna så småningom kommer att integreras. För vidare diskussion kring detta tema hänvisas till de tidigare nämnda state-of-the-art- och hybrid-rapporterna.

Vilka är då de mest påtagliga egenskaperna i de odbms-baserade objektmodellerna? Vilka är de avgörande skillnaderna mellan relationsmodellen och dessa modeller?

3. Relationsmodeller – datamodeller – objektmodeller

3.1 Relationsmodell

För de kommande resonemangen kring odbms-egenskaper och dess förhållande till rdbms-egenskaper, behövs en mycket kort sammanfattning av relationsmodellens begrepp.

Relationsmodellen tillämpar en tabellorienterad syn på data. Tanken är att företeelser i verkligheten ska kunna beskrivas med hjälp av attribut som sammanförs i tabeller. I modellabstraktionen svarar företeelsetypen mot en tabell (relation) och attributtypen mot kolumn i tabellen. Attributvärden är alltid lexikala, symbolbaserade. En given grunduppsättning värdetyper (datatyper) finns att tillgå. Om man så önskar, kan man definiera olika typer av restriktioner på tillåtna värden för någon värdetyp, för något behov. Dessa restriktioner kallas domäner. För varje attributtyp (kolumn) anges alltså tillåten värdetyp eller domän. En viss förekomst av en företeelsetyp beskrivs med hjälp av sina attributvärden i en rad i dess tabell. Normalt gäller att ett visst attribut eller kombination av attribut är unikt för en viss företeelse, bara finns i en enda rad i tabellen. En sådan kombination kallas tabellens primärnyckel. Ett primärnyckelvärde kan alltså användas för att identifiera viss företeelse (rad) i en tabell.

Normalt beskrivs många olika typer av företeelser i en och samma databas. Databasen består av många tabeller. Företeelser har ofta relateringar till varandra av intresse att hålla reda på i databasen. En sådan relatering (samband) modelleras genom att komplettera den tabell som har relateringen med den kombination av attributtyper som utgör primärnyckeln hos den tabell man vill relatera till. En pekare till en viss rad (företeelse) i en annan tabell är alltså den kombination av attributvärden som utgör dess unika nyckel.

Utöver relationsmodellen, representerad i form av en uppsättning DDL-instruktioner i SQL, omfattar SQL också instruktioner för att operera på data organiserade enligt relationsmodellen (DML i SQL).

3.2 Semantisk datamodell contra relationsmodell

Under beteckningen semantiska datamodeller placerar vi olika varianter av entity/relationship-modeller, konceptuella modeller, associativa modeller, binära modeller. De har traditionsenligt använts under olika systemutvecklingsfaser för att dokumentera en bild eller modell av någon verklighet, med det bakomliggande syftet att kunna hantera information om denna verklighet. Relationsmodellen och de semantiska modellerna har i detta avseende ungefär samma syfte. En skillnad som brukar föras fram är att relationsmodellen är mer implementeringsnära, d v s hanteras primärt av dem som har att implementera ett system medan de semantiska modellerna kommer till huvudsaklig användning i de tidigare faserna av en systemutveckling för att fånga informationsbehov ur ett användarperspektiv, som ett tilltalande språk mellan kravställare och systemutvecklare. Det finns dock inget som hindrar att semantiska datamodeller används som modell för databashanterare. Att så inte skett i någon större utsträckning står, som tidigare nämnts, att finna i en ansenlig flora modellvarianter och att "timingen" inte legat rätt. Produkter finns, t ex SIM utvecklad inom Unisys.

Modeller som syftar till att beskriva en uppfattning av en verklighet brukar klassas som statiska såtillvida att de alltid står för en tillståndsbild (status) av denna verklighet. Det

som händer, händer i verkligheten. Både relationsmodellen och semantiska datamodeller används för att formulera statiska modeller.

Sett ur ett statiskt modellperspektiv har semantiska datamodeller i allmänhet betydligt mer semantisk uttryckskraft än de objektmodeller som ligger till grund för odbms. Vad som primärt saknas är möjligheten att beskriva objektbeteende. Mer om beteende nedan.

För det fortsatta resonemanget i denna rapport är det av intresse att notera några typiska skillnader och likheter mellan semantiska datamodeller (SM) och relationsmodellen (RM):

Tabell i RM motsvaras i grovt sett av objekttyp eller entitetstyp i SM. Av olika skäl driver egenskaperna i RM fram behovet av en så kallad normalisering av tabell-uppbyggnaden, där 3:e normalformen, åtminstone i teorin, är den eftersträvade. Detta gör bland annat att flervärdiga attribut och samband måste realiseras i separata tabeller. Normalisering tillåter heller inte attributtyper, som har en egen intern struktur. Resultatet blir en fragmenterad objekttyps-beskrivning, d v s att en ur verksam-hetsperspektiv naturlig abstraktion förverkligas i form av ett antal separata tabeller.
I SM är objekttypsdefinitionen en samlad konceptuell beskrivning.

SM skiljer normalt på attribut (egenskapsvärden) och samband (relatering till annat objekt). RM gör ingen klar sådan skillnad. Sambandstyper är i SM något som på lika villkor berör de båda relaterade objekttyperna. Med andra ord ses det antingen som en från de båda objekttyperna fristående specifikation av sambandet dem emellan eller som en dubbelriktad relatering, specificerad hos båda objekttyperna. RM tillämpar en enkelriktad relatering.

SM uppmuntrar till distinktion mellan objektrepresentation och objektreferens. Referens till ett eller en mängd objekt kan ske med angivande av olika kombinationer villkor på attribut och samband. Representationen är en sammanhållande intern identifikator (OID = Object Identifier) för ett objekt. OID utgör sammanbindningspunkten för objektets attribut och samband med omgivande objekt. Identifieraren saknar semantisk valör. Den kan alltså riskfritt användas för objektet under hela dess livslängd. OIDs är ingenting nytt. Det har diskuterats i olika sammanhang under ett tjugotal år, men under olika benämningar (surrogate, internal, ...).

Operationer över samband realiseras i RM i form av joins över främmande och primära nycklar, något som av tradition anses vara en tung operation. Med hjälp av OID kan samband istället hanteras som en explicit logisk länk eller, beroende på OIDs upp-byggnad, till och med som en logisk/fysisk adress. I de flesta fall erhålls på så vis en mycket snabb referens till omgivande objekt. Obs, att det i princip inte finns något som hindrar identifikation av RM-rader (objekt) med hjälp av en egengenererad OID. (Vissa rdbms använder OID som en rent intern identifikator). Dock saknas operationellt stöd för detta begrepp. En annan skillnad gäller immuniteten mot ändrade förutsättningar. Unikhetskrav, existensvillkor m m för attribut kan komma att ändras under objektets livstid. Vissa av dessa attribut kan ingå i nycklar. Ändrad betydelse eller valör får givetvis konsekvenser både i den egna tabellen och för de främmande nycklar som pekar mot tabellen. OIDs däremot påverkas inte alls.

I RM finns en given uppsättning datatyper. Dessa används för att karakterisera enkla värden. SM tillåter normalt även fördefinierade, sammansatta konstruktioner som listor, mängder, "bags" (mängder där dubletter tillåts), arrayer, m m. Vissa SM erbjuder även möjlighet att skapa egendefinierade datatyper.

En kraftfull modelleringsfacilitet i SM är beskrivning av generaliseringsstrukturer. Förutom att vara en användbar abstraktion i många sammanhang kan de utgöra underlag för arvsmekanismer. Varianter i form av enkla (från ett superobjekt enbart)

respektive multipla (från ett antal relevanta superobjekt) arv förekommer. Någon motsvarighet finns inte i RM. SM tillhandahåller ibland även modelleringsbegrepp för att explicit beskriva komponentsamband (part-of), något som vissa tillämpningsområden har stor nytta av.

3.3 Objektmodellen i dbmstappning

Objektmodellen finns beskriven i SISU-rapport 13, "Objektorientering – de vanligaste begreppen". Här konstaterar vi endast att den huvudsakliga skillnaden mellan en semantisk datamodell och en objektmodell är att den senare innehåller beskrivning av objektbeteende. Detta konstateras i litteraturen som ett framsteg, en brist i "gamla" modeller, som nu åtgärdats. Riktiga objekt beter sig! Denna franka inställning beror sannolikt på att objektmodellen har sin upprinnelse från programmeringsområdet. Program beskriver ju både data och funktion. Om man nu ser till att föra samman data som upplevs logiskt höra ihop med de funktioner som opererar på dessa data har man konstruerat ett objekt. Ser man dessutom till att bara de funktioner som placerats tillsammans med sina data får operera på dem, har man skapat en liten lokal miljö, en inkapsling, kallat objekt. Denna lokala miljö har full kontroll över sina förehavanden och bestämmer själv vilken kontakt den vill ha med, vilken service den vill ge, omgivningen. Kompletterar vi därtill med möjlighet att skapa generaliseringsstrukturer av objekt får vi en både kraftfull och elegant modelleringsmiljö. Dessutom är modellen exekverbar eftersom den samtidigt är ett program. Vill ett objekt åt något annat objekts funktion får det helt enkelt skicka en begäran i form av ett meddelande. De objekt-orienterade språken representerar samma grundläggande modellsyn , men i något olika varianter.

Skapas objekten på ett förnuftigt sätt och sätts de att samverka i form av vettiga meddelanden, får vi en samverkande objektmiljö som genom sin uppbyggnad är överblickbar, flexibel och utbyggbar. De mest komplexa resultat anses kunna komma ut av dessa överblickbara mjukvarubyggstenar. I realiteten blir nog resultatet snarare avhängigt graden av problemförståelse och förmågan att konstruera och strukturera en lösning med hjälp av en produktiv utvecklingsmetod än valet av verktyg, d v s programmeringsspråk. Därmed inte sagt att språkets modell och syntax inte kan bidra till ett lyckat resultat.

Nåväl, objektmodeller känns tilltalande och förtjänstfullt enkla, åtminstone i teorin. Att därifrån med automatik konstatera, att dessa och därpå baserade språk och databaser är lämpliga och tillämpbara i alla modellsammanhang, är att ta risker.

Grundidén med ett odbms är att ta hand om de objekt som genererats under en oo-språks-session och som man önskar ha kvar till ett senare tillfälle. För att klara det måste odbms tillämpa samma objektmodell som det språk man avser att stödja. De flesta odbms svarade till en början mot C++s objektmodell. Anledningen är knappast modellens egna förtjänster. Snarare var och är det marknadsmässigt betingat. C++ är det största oo-språket. Viss frigörelse från C++-bindningen har på senare tid kunnat skönjas. Mer om det senare. Den nära oo-språk-kopplingen gör att de förtjänster och brister som kan hänföras till C++-modellen i stort också gäller för odbms med vissa undantag.

Några typiska C++-modell-egenskaper och begrepp:

> Class är motsvarigheten till en objekttyp. En klass kan bestå av ett antal data elements (jmf SMs attributtyp), vardera beskrivet med sitt namn och data type (jmf SMs datatyp eller domän). En klass kan också innehålla ett antal member functions eller methods. Det är här beteendet beskrivs. I fortsättningen använder vi de svenska orden "operation" och "metod" som synonyma begrepp till method.

Därutöver kan generaliseringsstrukturer i form av super/subklassförhållanden specificeras. Det finns ett antal fördefinierade data types och structure types. Med hjälp av dessa kan nya aggregerade types deklareras. Med hjälp av pointers (pekare) kan man referera till andra objekt.

En del av begreppen i den semantiska datamodellen saknas i C++-modellen. Dit hör framförallt "sambandstyp". Begreppet har med konstlade medel infogats i databasmiljön eftersom de naturligt nog upptäckts vara ett viktigt beskrivningsbegrepp. Mer om det senare.

Språkfristående utvidgningar av objektmodeller för odbms har gjorts, bland annat i ODMG-93-standarden. Den standarden definierar också fler typer av domäner/attribut, t ex set, list, bag, array. Även viss sortering kan begäras för flervärda attribut. Normaliseringsprincipen motverkar motsvarande konstruktioner i ett rdbms.

SM erbjuder normalt en rikhaltig flora uttryck för att specificera olika typer av konsistensvillkor. Motsvarigheter saknas i språknära objektmodeller.

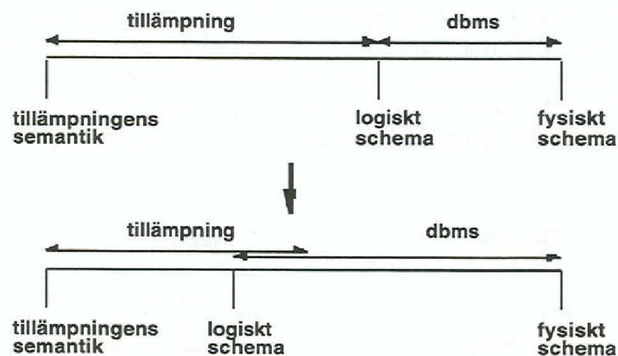
Personer med datamodellerings- eller AI-bakgrund kan av detta få uppfattningen att objekt har beteende på samma sätt som de har attribut, d v s att varje objekt har sin egen beteendebeskrivning. Så är dock inte fallet. Attributen instansieras (placeras ut) per objekt men beteendet stannar som en beteendebeskrivning under klassen att vid anrop appliceras på avsett objekt tillhörigt klassen. Beteendet är momentant och seriellt, d v s det exekveras färdigt för ett objekt innan det appliceras på nästa. Objekten lever alltså inte i någon dimension, t ex i en tidsdimension som ofta är fallet bl a i simulerings-miljöer. Konsekvensen av detta är naturligt nog att ett odbms inte heller behöver lagra något beteende för respektive objekt. I själva verket lagras i allmänhet överhuvudtaget inte ens koden för beteendedefinitionen i databasen, bara dataelement- och pointer-värden, d v s endast tillståndsuppgifter för objektet. Beteendet återfinns i tillämpningens runtime-modul. Undantag finns dock.

Den saktmodige frågar sig kanske vad skillnaden i så fall är mot konventionella databaser. Skillnaderna är inte så många som företrädarna gärna vill göra sken av, i alla fall är det knepigt att finna dem ur ett modelleringsperspektiv. Däremot erhålls ett antal intressanta egenskaper i den unika oo-språk – odbms-kopplingen. Över till nästa avsnitt.

4. Objektdatabaser, en introduktion

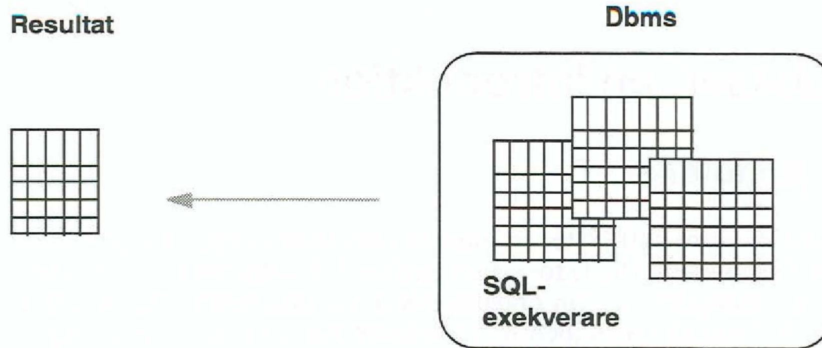
4.1 Rdbms, som jämförelse

Traditionellt har databashanterare haft syftet att hantera intressanta uppgifter om någon verklighet (Universe of Discourse – UoD) för något ändamål. I allmänhet har det varit fråga om ganska stora datamängder. Nästan undantagslöst har innehållet i databaserna givit service till många, för många specialbehov. Databasen har varit en generell lagerhållare av information. Databasens innehåll har mer styrts av lämplig avgränsning av den verklighet man vill hantera information om, än av hur informationen hanteras. Därav den traditionsenliga distinktionen mellan data och funktion. Trenden i den traditionella ansatsen är i dagsläget att minska det så kallade semantiska gapet mellan data och funktion, d v s att tillhandahålla en semantiskt rikare begreppsapparat samtidigt som man strävar efter att placera mer intelligens i databasen. På så vis avlastas tillämpningen sådant som naturligt hör till data snarare än funktioner på data (härledningar, kontroller, vissa typer av händelser, m m). Representanter för denna trend är dels konventionella rdbms som stödjer SQL-92 och därtill kanske egenutvecklade påbyggnader, dels de s k hybriderna, d v s dbms baserade på SQL-gränssnitt men kompletterade med ett antal objektorienterade begrepp och mekanismer. Även de få produkter som baseras på någon semantisk datamodell kan sägas tillhöra denna senare kategori.



Figur 1

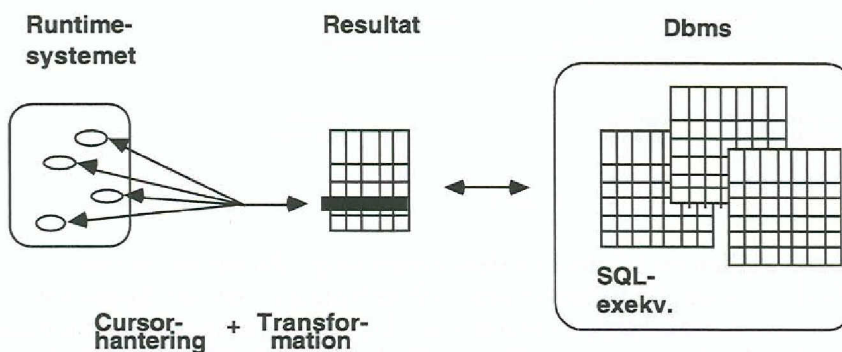
Hur fungerar då samspelet mellan ett rdbms och dess omgivning? Ett rdbms har ett generellt gränssnitt – SQL – , som alla tillämpningar och användare, oavsett databehov, måste använda sig av för att operera på/med databasdata. Ur rdbms-perspektivet anländer en SQL-instruktion, den bearbetas och eventuellt svar sänds till frågeställaren. SQL är mängdorienterat. Resultatet levereras i form av en tabell, d v s samma modellformat som databasens eget. Någon transformation till annat modellformat behövs alltså inte.



Figur 2

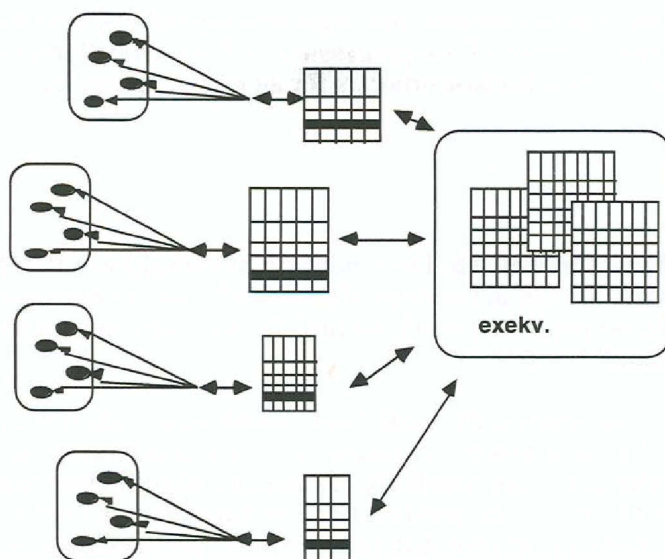
I alla fall behövs inte transformationen om mottagaren är en människa som kan tolka en tabell. Är mottagaren en tillämpning uppstår däremot en semantisk "impedance mismatch" – datamodellen för runtime-systemet överensstämmer inte med datamodellen för rdbms. Detta är priset man får betala för att ge en tillämpningsfristående, språkoberoende, generell databas-service.

En transformation mellan de båda modellerna måste här utföras. Det åstadkoms normalt genom att bädda in SQL-satser i programmeringsspråket. Resultatet av en utsökning hamnar i en buffert under kontroll av dbms. Med hjälp av en pekare (cursor) som sätts att successivt peka på en resultatrad i bufferten i taget, hämtar rdbms en rad åt gången in till tillämpningen. Språkvariabler sätts att representera varje kolumn (kolumnvärde) i en rad. När en rad bearbetats, sätts pekaren att peka på nästa, o s v. Uppdateringar sker på motsatta sättet.



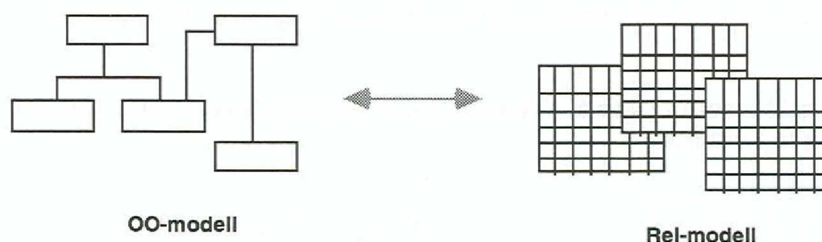
Figur 3

Transformation tar resurser i anspråk. Varje begäran mellan tillämpning och dbms innebär ett interprocess-anrop, något som tar avsevärd exekveringstid i anspråk. Å andra sidan kan rdbms ge service åt många och varierande behov (tillämpningar, användare). SQL kan bäddas in i vilket språk som helst.



Figur 4

Problemet blir inte enklare om språket är objektorienterat. Båda är visserligen modellbaserade, men modellerna är olika.



Figur 5

Antag att ett visst objekt får ett meddelande. Dess statiska beskrivning (dess attribut och pekare) visar sig ligga i databasen och begärs därför in. Befinner sig rdbms-modellen i 3:e normalform är det sannolikt att en objektbeskrivning ligger i form av ett antal rader utspridda över flera tabeller. Det måste finnas en beskrivning någonstans (transformationsschema) över vad som i en rdbms svarar mot ett objekt. Har objektet en aktuell superstruktur av objekt vars värden kan komma att ärvas in i och refereras från den metod som ska exekveras, måste antingen även dessa läsas in, alternativt förberedas för inläsning. Väl funna måste värdena arrangeras ihop i enlighet med det krav språkets runtimesystem ställer. Dessa uppgifter (layoutschema) måste också finnas tillgängliga tillsammans med eventuella omkodningsbehov. Dessutom måste den redigerade objekt-beskrivningen placeras in på den plats där runtime-systemet förväntar sig att finna den.

Tänk vad bra om denna transformation inte behövdes. Varför inte lagra objektet ute i databasen i den layout runtime-systemet etablerade när det skapades? Då skulle ingen transformering mellan beståndsdelar och format behövas. Det är bara det att relations-modellen inte omfattar objektbegreppet och därmed inte en modelluppbyggnad som svarar mot detta behov. Man tillämpar en annan modelleringsfilosofi. En databas baserad på en mer ER-liknande modell skulle däremot mycket väl kunna svara upp mot en gruppering som naturligt samlar alla egenskaper för ett objekt.

Ett kvarstående bekymmer är dock att anpassning till språkets layout m m krävs. Ett annat bekymmer är att ett SM-baserat dbms följer en annan, rikare semantik än C++-modellen eller motsvarande. Dessutom kanske man har valt en lagringsteknik som

tillåter åtkomst av enskilda beskrivningselement, inte bara hela objektbeskrivningen. För att åstadkomma "seamlessness" mot en C++-process måste C++-modellen exakt kunna härbärgeras i databasen. Odbms har konstruerats för att ge denna service.

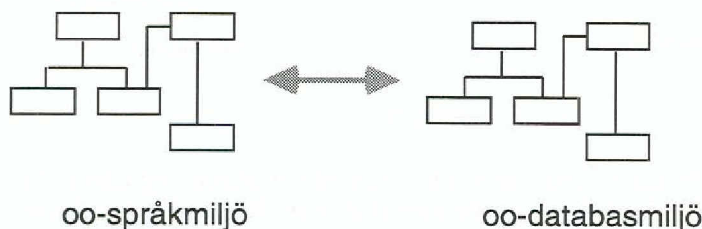
4.2 Odbms, grundidé

Rdbms och odbms baseras på olika modellsyn. De vänder sig huvudsakligen till olika tillämpningsområden. Rdbms ger allmän dataservice för många behov. Odbms ger specifik service för integrerade oo-program (som i sin tur kan svara mot många behov, kan åstadkomma de mest skiftande saker).

De objektorienterade språken representerar en modell som kombinerar data och funktion i ett objektklass-begrepp. Vi har tidigare konstaterat att de flesta odbms primärt stödjer C++. Flera odbms stödjer dessutom Smalltalk. Ytterligare något odbms klarar andra språk. Vi har också konstaterat att i normalfallet endast datadelen av objektet lagras i databasen. (Undantag finns.) Eftersom data placeras invid de funktioner som tillåts operera på det i exekveringsmiljön, finns inget behov av att lagra objekten i databasen på ett fragmenterat och tillämpningsneutralt sätt. Målsättningen är snarare att lagra data i överensstämmelse med primärminneslayouten för objektets datadel eller så att en sådan layout enkelt kan byggas upp när leverans till runtimemiljön ska ske.

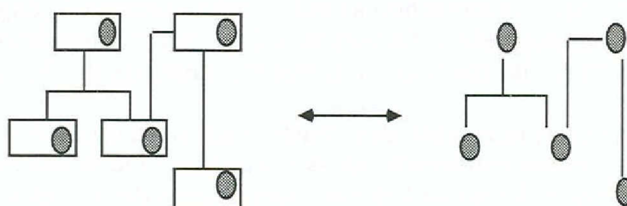
Schemat för en tillämpning är i princip klassdefinitionerna, skrivna i aktuellt språk (oftast C++). Epitetet "Object-oriented database programming language" är knappast missvisande.

Ofta visas i litteraturen bilder som i figur 6. Därav kan man lätt få intrycket att objekten i sin helhet ligger beständigt odbms.



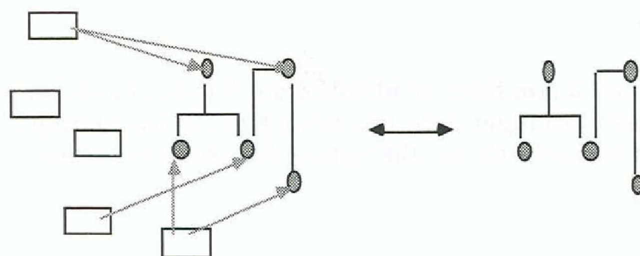
Figur 6

Som redan konstaterats är det i de flesta fall bara datadelen av objektet som lagras i databasen. Till datadelen hör även eventuella pekare (samband) till andra objekt.



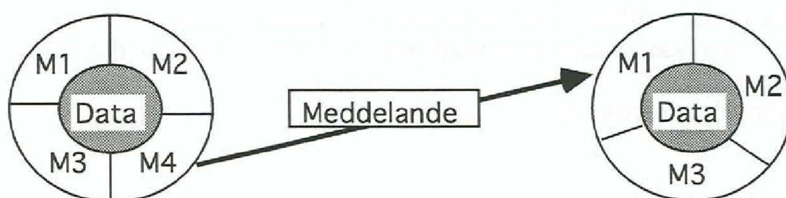
Figur 7 a

Figur 7 a är höggradigt förenklad. I realiteten finns en exekveringsmodul för varje objektklass, inte varje objekt (figur 7 b).



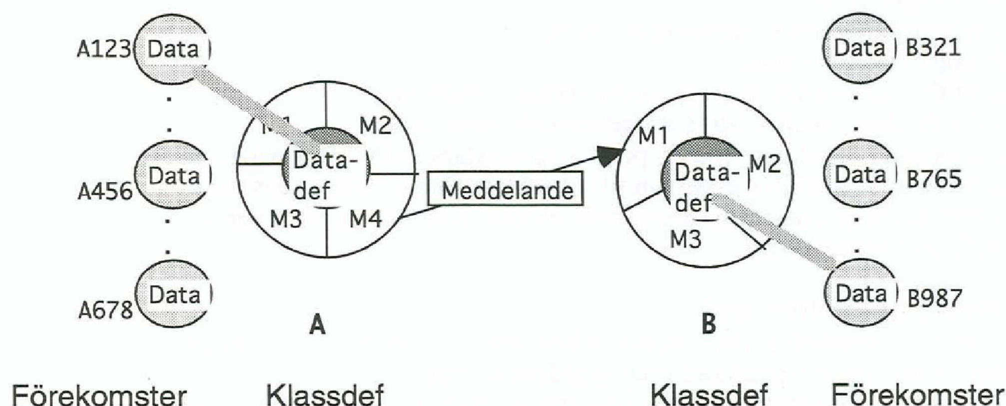
Figur 7 b

En annan figur som brukar användas visas i figur 8 a. Data (attribut) ligger i kärnan med metoderna (M1...) runt om symboliserande inkapsling. Metoderna som är objektets ansikte utåt opererar sedan internt på "sina" data.



Figur 8 a

En mer rättvisande bild visar figur 8 b. Runtimesystemet exekverar M4 för objekt A123 tillhörigt klassen A. M4 begär att få M1 utfört för objekt B987 tillhörigt klassen B genom att skicka ett meddelande till M1. Meddelandet innehåller B987 som parameter (förutom ev andra som kan vara relevanta). Runtimesystemet söker upp B987's datadel och startar M1.



Förekomster Klassdef Klassdef Förekomster

Figur 8 b

Mellan sessioner finns de beständiga objektens datadel (förekomsterna) i databasen. Under en session finns, förutom de beständiga i databasen, ett antal under operation i runtimesystemet. Andra nya objekt i runtimesystemet har ännu inte hunnit placeras in i databasen, o s v.

Objekt med olika klasstillhörighet kommer och går allteftersom exekveringen framskrider. Innan vi går på hur detta samspel i praktiken går till är det viktigt att känna till hur objekten i ett odbms identifieras.

4.3 Objektidentifierare

Varje objekt representeras genom någon form av identifierare (Object Identifier eller kortare OID). Ett antal alternativa lösningar finns och tillämpas. De har samtliga både för- och nackdelar. Vad som är idealiskt för en tillämpning kanske är rena katastrofen för en annan.

Varianter:

- Fysisk adress
  - Antingen i form av en virtuellminnesadress (om detta utgör databasen) eller, mer vanligt, en sekundärminnesadress.
  - Snabbt.
  - Inflexibelt, komplicerat vid behov av objektflyttningar, t ex i samband med reorganisation av databasen p g a fragmentering, outnyttjade luckor, etc. Fragmentering hinner sällan bli något problem i samband med prestandatester men kan orsaka avsevärd prestandadegradering vid normal drift. Bör noga bedömas.
  - Felkänsligt, försvårar recovery.

- Logisk adress
  - En kombination av fysisk och logisk adress. Den fysiska pekar på en sida eller segment medan den logiska anger aktuell "slot" inom sidan. Flyttning av objekt kan utföras "osynligt" inom sida.
  - Måste hantera overflows m m.
  - Ganska snabbt.
  - Något mer flexibelt.
  - Komplicerat vid behov av objektflyttningar.

- Internt genererad identifierare (Surrogate)
  - Stabil över objektets liv.
  - Kan vara räknargenererade, tidstämpel, m m.
  - Ger totalt lagringsoberoende.
  - Kräver avbildningsmekanism mot adress (prestandaförlust)
  - Mappas ofta till den fysiska adressen genom ett hash-index.
  - Ett annat alternativ är att utnyttja en primärminnestabell där OID pekar på en ingång som sedan innehåller adressen till objektets första minnesposition. Används främst för primärminnesbaserade objekt. Kan även kombineras med virtuellminneshantering.

- Typad, internt genererad identifierare
  - Typ kan t ex vara objekttyp eller logisk subdatabas i en distribuerad miljö. Generering inom typ och prefixad med typidentifierare.
  - Ger viss semantik-information direkt i databasen men kan försvåra t ex byte av typ om sådan är tillåten. Typen har man oftast ändå som en del av det aktuella sammanhanget.
  - Kan bl a vara av värde vid interna omflyttningar av exv alla objekt av viss typ mellan noder i ett distribuerat nät eller för optimering av operationer som berör viss logisk databas.
  - Ej användbart om samma objekt kan tillhöra flera typer.

Förutom flexibilitet vid lagring i distribuerade databaser ger interna identifierare ökad frikoppling mellan klient och server vilket underlättar introduktion av mer mekanismer i servern (frågeexekvering, kontroller, ...), när så önskas. Anpassning till nya förut-sättningar, prestandatrimning m m genom omallokering av objekt i databasen kan enkelt utföras om objektet har en intern identifierare men är en komplicerad operation för adressbaserade objekt. De bygger även på en högre grad av språkberoende. Å andra

sidan vinner man normalt prestandafördelar under exekvering. I fortsättningen kallar vi adressbaserade identifierare för A-OID och internt genererade för I-OID.

Ytterligare aspekter att ta hänsyn till:

- Ska I-OID återanvändas? I så fall måste de administreras. Tungt. Alternativet är att använda så många bitar att det aldrig behövs.
- Vissa modeller tillåter bara en unik I-OID för ett objekt medan andra tillåter olika I-OID för objektet, beroende på vilket sammanhang det befinner sig i.
- Distribuerade miljöer kan ge problemen en extra dimension för samtliga varianter ovan. Sammalunda om/när två eller flera existerande databaser ska slås samman eller spridas ut.

4.4 Samspel odbms – tillämpning

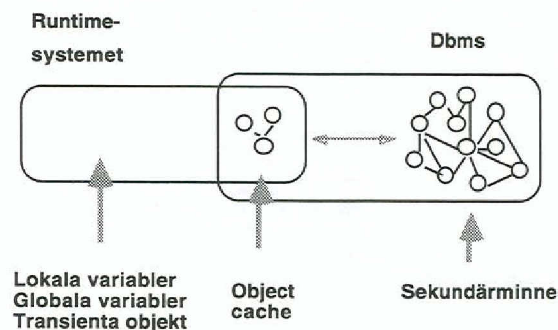
Odms är komplicerade företeelser. Varje odbms har dessutom i många stycken valt att realisera de typiska oo-funktionerna efter egna idéer, inte minst samspelet mellan tillämpning och dbms. Det ligger utanför denna rapport att gå in på alla detaljer och alla varianter. De följande avsnitten beskriver endast generella principer.

4.4.1 Objektreferenser i virtuellminnet

Den del av runtimesystemet som hanterar datadelen av beständiga objekt administreras av språkmoduler och odbms-moduler tillsammans i ett avancerat samspel. Odbms tillämpar något olika principer för detta samspel. Vad som sägs nedan ska endast tolkas som ett idéresonemang.

En C++-programmerare vill i realiteten helst slippa bekymra sig om ett objekts aktuella lagringsplats. Det ska finnas tillgängligt för runtime-systemet när operationer på objektet ska genomföras.

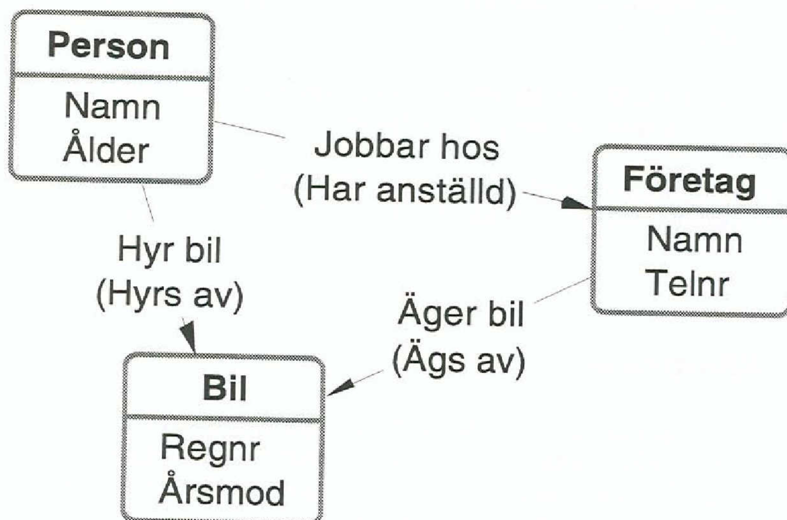
Det är odbms' ansvar att se till att datadelen av ett objekt finns till hands i en object cache när en operation (bland dem som definierats för den objektklass objektet tillhör) önskar operera på objektets data. Object cache är en del av virtuellminnet som både språkets och odbms' moduler kan referera till. Det är ju också en förutsättning för att odbms ska kunna "duka" för exekveringen. Förutom de beständiga objekten i cachen, tar även andra tillfälliga objekt och variabler plats i primärminnet (virtuellminnet).



Figur 9

Vid exekvering refererar ett objekt till andra objekt genom pekare. Denna pekare är antingen en direkt eller indirekt virtuellminnesadress. Båda varianterna finns företrädda i de kommersiella produkterna.

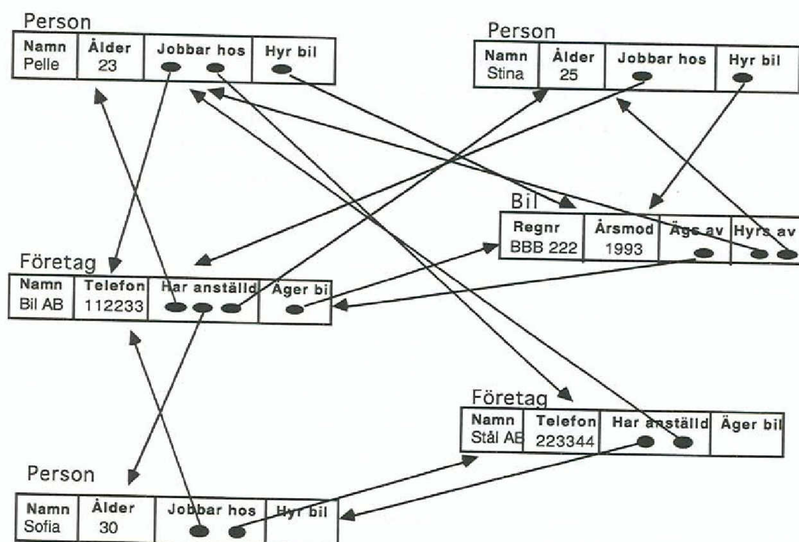
Antag objektmodellen i figur 10 och några objekt inladdade i object cache.



Figur 10

4.4.1.1 Direktreferenser

Figur 11 visar hur det kan se ut med direktreferenser. För överskådlighetens skull visar vi pekarna i form av svarta punkter och pilar. Som synes gäller här dubbelriktade samband. De flesta odbms kan hantera detta. Vissa erbjuder val mellan enkel- och dubbelriktning.



Figur 11

Ersätter vi pilarna mot dess mer korrekta motsvarighet, nämligen adresser, blir samma sak, något omstuvat, enligt figur 12. Till vänster står adressen för respektive objekt. Kursiverade tal är adresspekare.

Person

102

Namn	Ålder	Jobbar hos	Hyr bil
Pelle	23	*516 675*	*290*

Bil

290

Regnr	Årsmod	Ägs av	Hyrs av
BBB 222	1993	*516*	*102 799*

Person

417

Namn	Ålder	Jobbar hos	Hyr bil
Sofia	30	*516 675*	

Företag

516

Namn	Telefon	Har anställd	Äger bil
Bil AB	112233	*102 417 799*	*290*

Företag

675

Namn	Telefon	Har anställd	Äger bil
Stål AB	223344	*417 102*	

Person

799

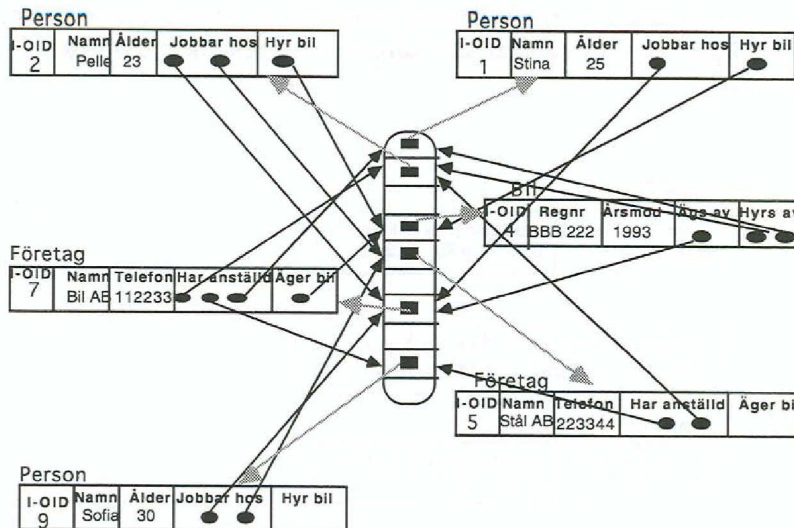
Namn	Ålder	Jobbar hos	Hyr bil
Stina	25	*516*	*290*

Figur 12

Direktreferenser används i normalfallet i de fall odbms använder sig av A-OIDs, se avsnitt 4.3. Inget hindrar i princip att direktreferenser används även i de fall I-OIDs används. Dock ligger då nära till hands att istället utnyttja indirekta referenser.

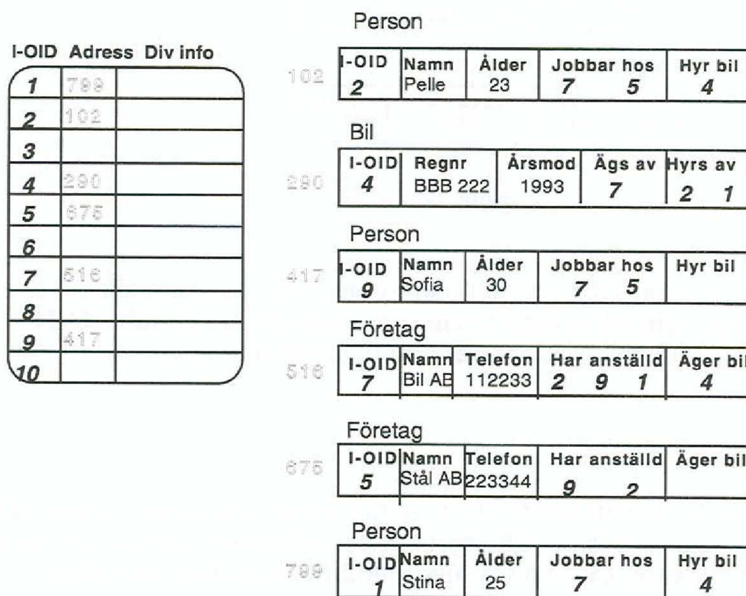
4.4.1.2 Indirekta referenser

Indirekt adressering åstadkoms via en tabell där varje rad är en "object descriptor". Den består av objektets I-OID, pekare (adress) till objektets aktuella position i minnet samt diverse andra uppgifter av intresse, exv om det blivit modifierat, är låst, hur många programvariabler (andra objekt) som pekar på det. Figur 13 visar hur det blir med samma exempel som ovan. En referens sker i två steg. Det första består av hashning på objektets I-OID mot tabellen. Se de svarta pilarna. Där återfinns minnesadressen (svart rektangel) som i nästa steg används för att hitta objektet (grå pil).



Figur 13

Den mer korrekta motsvarigheten, där pilarna bytts mot adresser, visas i figur 14.



Figur 14

Som synes kan mycket väl referens till ett visst objekt ske från flera håll. Dessa refererar alla till samma position i hashtabellen. Fördelar med detta är främst flexibiliteten. Objektet kan enkelt flyttas utan annan effekt än att descriptorns adressuppgift behöver ändras, något som bl a kan behövas av prestandaskäl, garbage collection, eller dyl. Även den naturliga placeringen av övrig information är en tillgång. Som kommer att framgå i avsnitt 4.4.3, kan enklare teknik för läsning/skrivning till/från databasen tillämpas.

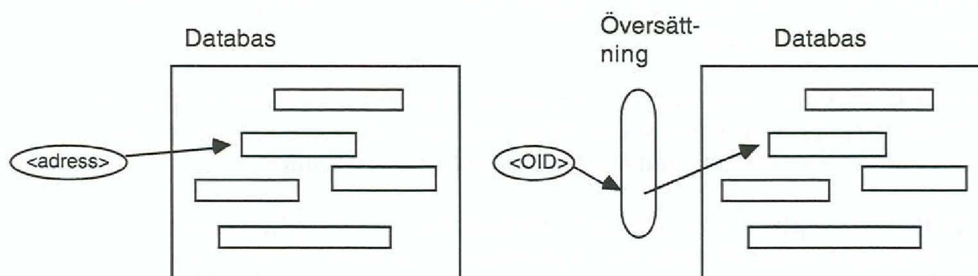
Betalningen sker med hjälp av sämre prestanda p g a den indirekta adresseringen.

Antag att objektet med I-OID 7 behöver flyttas till adressen 918 p g a att fler anställts och därmed dess minnesbehov ökat. Används indirekt referens ändras adressen för I-OID 7 i tabellen till 918 och så är saken klar. Används direktreferens måste samtliga

refererade objekt sökas upp (4 stycken) och deras respektive adresspekare till det omplacerade objektet ändras till 918.

4.4.2 Objektreferenser i databasen

Vad som sagts i förra avsnittet är i princip applicerbart även för åtkomster i databasen dock med stora variationer i de interna detaljerna. Även i databasen ger A-OID givetvis snabbare access genom att adressen direkt kan användas (figur 15 a). Används I-OID måste de översättas till adress. Figur 15 b.

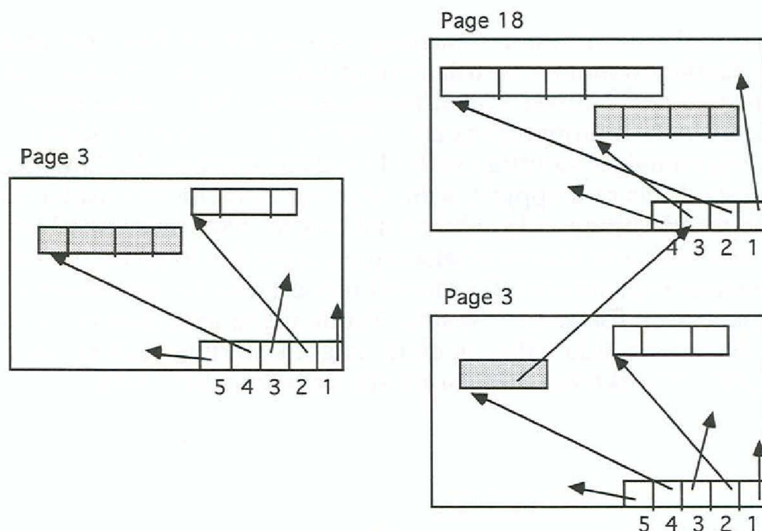


Figur 15 a *Figur 15 b*

En adress kan referera till exakt minnesposition alternativt bestå av en sid-adress plus slot-nummer inom sida (logisk adress). Se figur 16 a.

I distribuerade system kan det finnas anledning att komplettera med adress till aktuell server. Andra indelningar tillämpas också. Se vidare avsnitt 4.3.

A-OID ger även här upphov till inflexibilitet. Omorganisation kan visserligen utföras inom sida om logisk adress används, men i stora system är det sannolikt inte ovanligt att objekt behöver flyttas till nya sidor som en följd av minnesfragmentering, nya distributionsprinciper, prestandautjämningar. En variant som används i vissa produkter för att till del motverka nackdelarna är att använda vidarepekare från den gamla platsen till den nya. Se figur 16 b.



Figur 16 a *Figur 16 b*

Den interna funktionaliteten är av mindre intresse för användaren så länge som gränssnittet är specificerat och funktionsrepertoaren given. Av betydligt större intresse är princip-erna för överföring av data mellan databas och tillämpningens runtime-system, inte minst eftersom detta är en av de egenskaper som på ett avgörande sätt skiljer rdbms och odbms åt.

4.4.3 Inläsning från databasen

4.4.3.1 Inledning

Hur flyttas då ett objekt mellan databas och virtuellminne, d v s hur arbetar ett odbms för att "duka" objekt för en tillämpning?

Två grundprinciper kan utskiljas, beroende på om direkt eller indirekt referens tillämpas i virtuellminnet. Inom varje princip finns sedan olika varianter som inte vidare berörs i denna rapport. För exakta redogörelser hänvisas till respektive leverantör.

Oavsett princip, gäller det att placera in objektet i virtuellminnet i enlighet med oo-språkets krav. I en homogen miljö är detta ingen problem. Minnesbilden kan tas ur virtuellminnet exakt så som det ser ut och lagras i databasen. Där ligger det förberett och klart att föras tillbaka in i virtuellminnet vid anrop. Men de flesta odbms kan operera i klient/server-miljö, se avsnitt 4.5, och behöver där, bl a av marknadsmässiga skäl, stödja ett antal plattformar (hårdvara-operativsystem) för både klient och server. Dessutom måste ett antal olika oo-språks-kompilatorer stödjas. De varianter som kan finnas i byte-representation, inkodningsformat, m m måste osynligt kunna överbryggas.

Trenden har på senare tid varit mot språkoberoende, vilket ställer ytterligare krav på anpassbarhet. Flera odbms klarar integrering både med C++ och Smalltalk-tillämpningar, vissa även med Eiffel och C. Antingen har man löst det genom att kräva att objekt skapade i visst språk bara får opereras vidare med detta språk eller, mer avancerat, genom att tillåta full transparens.

4.4.3.2 Objekt som har A-OID

Denna princip är snarlik normal virtuellminneshantering, som grovt sett fungerar enligt följande. De process- och datakomponenter en tillämpning behöver kunna hantera och därmed referera till ges normalt en 32-bitars virtuellminnesadress. Den fysiska överföringen mellan virtuellminne och tillämpningens arbetsminne sker i form av sidor. Arbetsminnet rymmer ett visst antal av samtliga sidor. I en översättningstabell, med en ingång per virtuellminnessida, finns bl a uppgift, som visar om virtuellminnessidan för närvarande utnyttjar någon primärminnessida och i så fall vilken. Sker referens till en adress, som härrör till en sida som inte finns i arbetsminnet, läses den in på första lediga arbetsminnessida samt uppdateras positionen för virtuellminnessidan i översättningstabellen med arbetsminnesadressen för just inläst sida. Skulle ingen ledig plats finnas, måste först en lämplig primärminnessida väljas ut enligt något kriterium (exv en som inte refererats på länge) samt sidan skrivas tillbaka till sin virtuella adress (om innehållet ändrats).

4.4.3.3 Objekt som har I-OID

När ett nytt objekt läses in från databasen omvandlas samtliga pekare till andra objekt till en position (adress) i en adressomvandlingstabell (hashtabell). För refererade objekt som redan finns inlästa anges deras tabellposition, för övriga skapas först en "object descriptor" som placeras in i tabellen. Det refererade objektets I-OID hos det refererande objektet ersätts med adress till object descriptor i tabellen. Därmed har man förberett för inläsning om och när objektet refereras vid en senare tidpunkt.

Sker så småningom en referens till ett objekt, som inte finns i virtuellminnet, läses det in och placeras på lämplig plats. Dess virtuellminnesadress förs in i objektets object descriptor. Figur 14 visar ett exempel.

Vissa produkter för över endast det refererade objektet (object server) medan andra väljer att föra över hela sidor (page server).

Utnyttjas clustering kan det senare alternativet vara fördelaktigt, samtidigt som man använder sig av operativsystemets normala mekanismer.

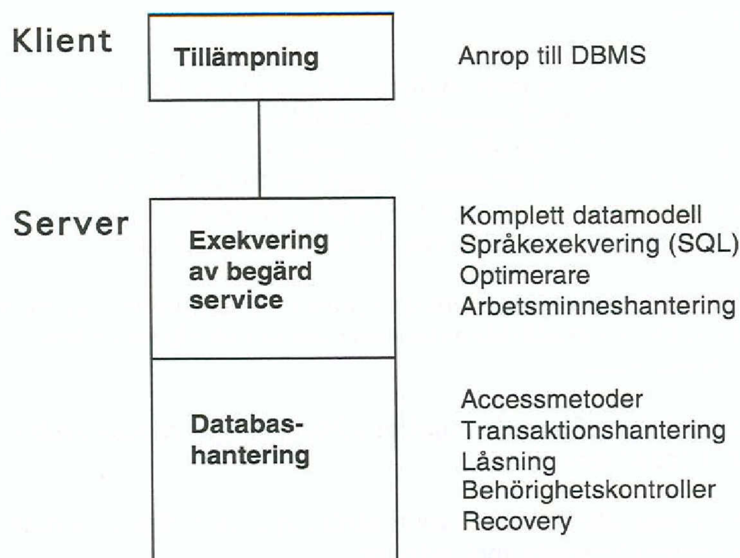
Vid tillämpningar med mycket varierande behov av objektåtkomst eller åtkomstbehov till spridda objekt, d v s där clustering inte ger påtagliga fördelar, är objektöverföring effektiv. Dess naturliga koppling till den indirekta referensansatsen bidrar till flexibilitet i primärminneshanteringen, något som indirekt kan bidra till färre databas-accesser. Normalt tillåter de produkter som tillämpar objektöverföring också mer bearbetning på servern, exv fråge- och metodexekvering. På så vis ombesörjer man att endast relevanta objekt förs över, dessutom om möjligt tillsammans, för att minska I/O.

4.5 Klient/server-miljö

Många databastillämpningar arbetar i klient/server (C/S)-miljö. I det generella fallet kan det vara fråga om ett stort antal klienter mot en eller flera databaser, var och en kanske distribuerad. Rdbms har av tradition haft en sådan inriktning. Givetvis strävar även odbms att vara C/S-anpassade. Beroende på deras olika "ideologiska" bakgrund har rdbms och odbms dock valt olika principiella lösningar.

4.5.1 Klient/server-miljö för rdbms

Anropen mellan klient och server sker normalt i form av Remote Procedure Calls (RPC). Antalet RPC bör minimeras eftersom de tar ansenlig tid (10 ms?). Den funktionella uppdelningen mellan klient och server bygger på en kraftig server-orientering. Se figur 20.

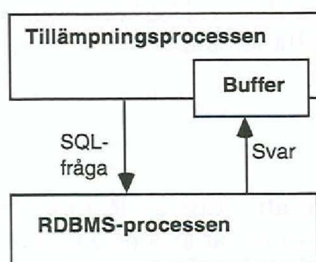


Figur 20

Exekvering hos server gör att endast relevanta data förs över till klienten. Detta minimerar datatransport. En viktigare fördel med server-orientering är att severn fungerar i rollen som central logisk sammanbindningspunkt för datahanteringen, något som är naturligt i konventionell databashantering där man skiljer på funktion och data. I den rollen ingår bl a att kontrollera databasens integritet, dess överensstämmelse med specifikation i schemat, kontroll av säkerhet, samtidighet, ...

Å andra sidan kan servern bli överbelastad.

Data i ett rdbms förs till en generell buffer hos klienten. Därifrån kan data hämtas i sin helhet, exv för presentation, men oftast för vidare bearbetning i en tillämpning. Radvis överföring till tillämpningen med hjälp av en cursor är vanligt. Se figur 21.



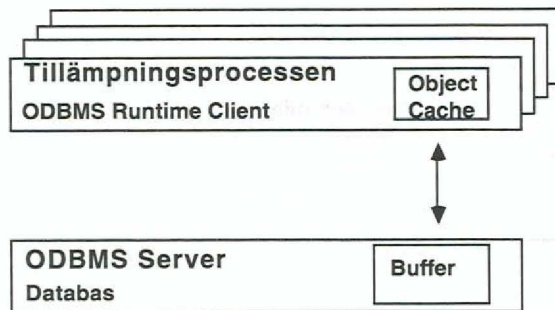
Figur 21

4.5.2 Klient/server-miljö för odbms

Som tidigare diskuterats förs data i ett odbms direkt in i tillämpningens adressutrymme (object cache) i form av hela objekt eller hela sidor för runtimesystemet att operera på. Anpassningsjobbet utförs av en odbms-funktion som ligger hos klienten (Odbms klient-runtime).

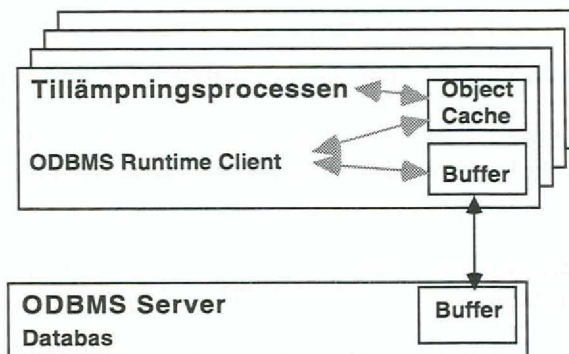
Klient-runtime samarbetar med programmeringsspråkets runtime modul och exekverar normalt inom detta språks exekveringsmiljö. De båda runtime-modulerna är länkade i samma process. Servern hanterar objekten i databasen och ser till att plocka fram resp. ta emot objekt/sidor för kommunikation till/från klient-runtime. I allmänhet hanterar

servern aktuella objekt/sidor i en egen buffer för att underlätta kontroll och styrning samt för att undvika onödig I/O på serversidan. Bedömningen är att det är större sannolikhet för ett använt objekt att bli efterfrågat igen än för övriga objekt.



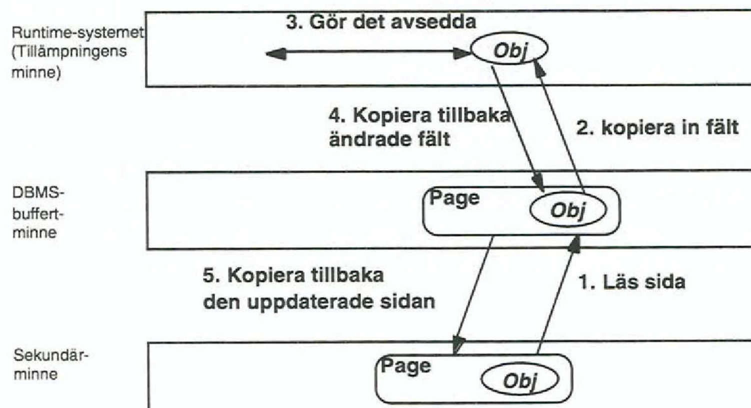
Figur 22

När en transaktion är klar töms Object Cache eftersom pekarstrukturerna därefter inte längre med säkerhet håller korrekta referenser. Uppdaterade objekt skrivs till servern. Skulle vid nästa transaktion hos klienten något tidigare använt objekt efterfrågas igen (och det inte under tiden ändrats av någon annan klient) vore det praktiskt om objektet inte behövde återinläsas från servern. Eftersom odbms normalt också tillämpar tidsödande RPC anrop mellan klient och server, finns här mycket att vinna. Vissa odbms opererar av den anledningen med en buffer även på klient-sidan enligt figur 23.



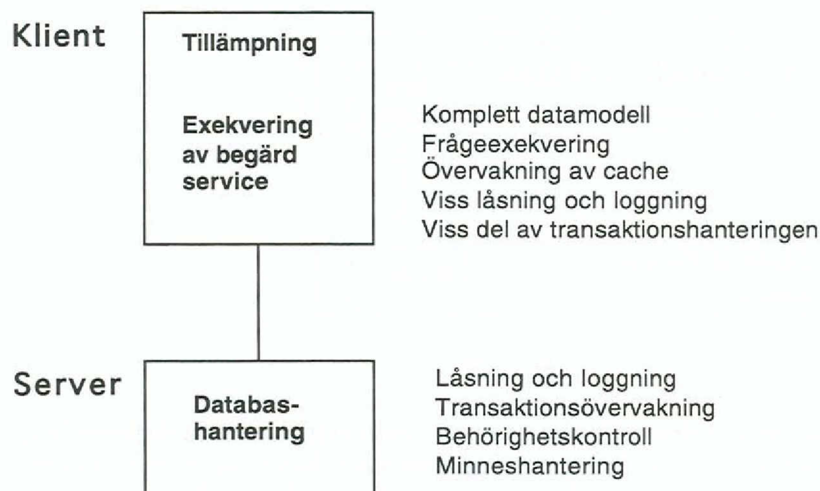
Figur 23

Samspelet sker enligt figur 24.



Figur 24

Av tradition ligger en hel del funktionalitet hos klienten. Odbms är ju en expansion av oo-språkstillämpningar till beständig lagring av objekt, senare även i C/S-miljö. Exakt vad som ligger var skiljer mellan produkterna. Figur 25 ger ett typexempel.



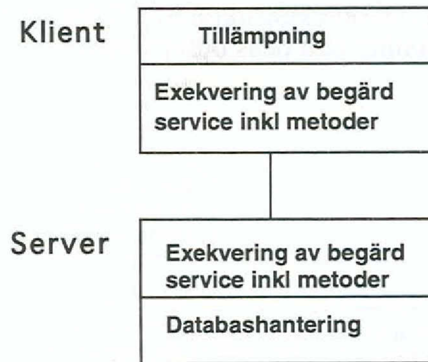
Figur 25

Fördelen är mycket effektiv exekvering så snart objektet ligger i klienten. Många, kraftfulla klienter tar över en hel del jobb som servern har att sköta i rdbms-miljö. Bland bekymmer kan nämnas:

- Integrity constraints kollas normalt på klient-sidan. Tveksam lösning.
- Eftersom objekten hanteras under kontroll av klienten och inte servern finns risk att bristande överensstämmelse mellan klienternas datastrukturer får konsekvenser när det gäller databas-konsistens.
- Frågespråk exekveras normalt på klient-sidan. Kan orsaka ansenlig transport av grunddata istället för enbart resultat.
- Även utan frågespråk måste alla på något sätt berörda objekt över till klient-sidan.
- Kod lagras oftast inte i databasen – kan därför inte delas, samordnas.

I framtiden kan odbms förväntas erbjuda exekvering av delar av tillämpningen på servern, exv för frågespråks- och metod-exekvering. Dessa kommer då att kallas **aktiva objektdatabaser**. Resultatet blir en alltmer ökad frikoppling från språkmiljön – på sätt och viss ett återtåg till konventionell databashantering.

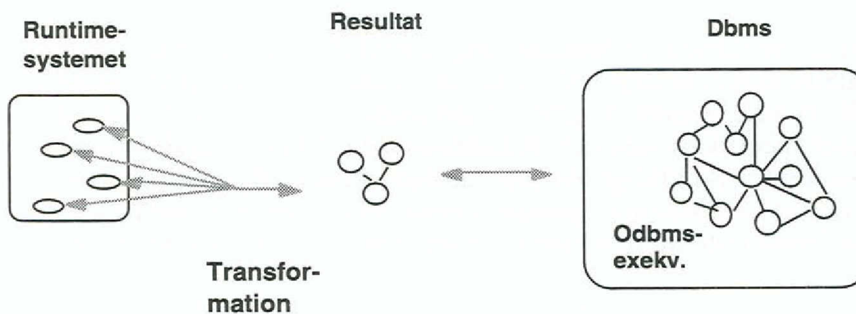
Ett par leverantörer erbjuder denna facilitet redan idag på så sätt att man dynamiskt kan välja var metod önskas exekveras. Genom att frikoppla metoder från tillämpningen och istället relatera dem separat till objektmodellen, kan metoderna beskrivas i olika språk. Exekveringsställe kan väljas dynamiskt enligt eget lämplighetskriterium. En förutsättning är att servern opererar som en objekt-server, d v s kan hantera objekt – inte bara sidor.



Figur 26

4.6 Odbms – konventionellt språk?

Vi har konstaterat en mycket nära koppling mellan odbms och oo-språk. Det förekommer också att oo-språk används mot rdbms. Går det då att använda ett konventionellt språk mot en odbms? Javisst, men en transformation konventionellt språk-odbms behövs. Odbms kan här baseras på valfri objektmodell eftersom ingen modellmässig språkbindning finns. I princip blir odbms en SM-dbms.



Figur 27

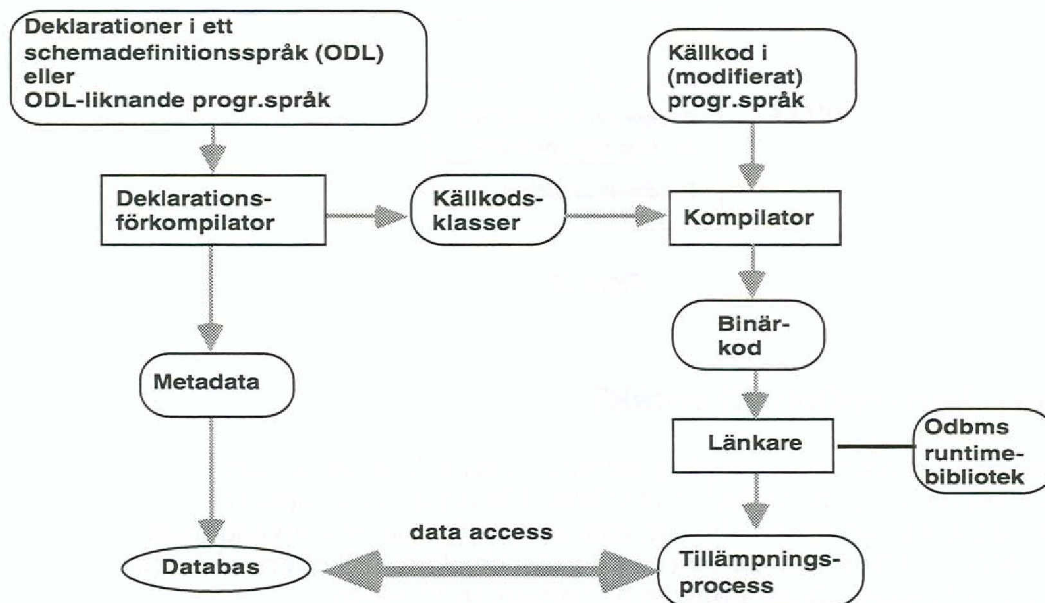
4.7 Definition av en tillämpning

4.7.1 Översikt

Ett odbms är definitionsmässigt lierat med ett eller flera oo-språk. Om inte, har man ju samma problematik som de konventionella lösningarna, d v s ett separat språk för att definiera och hantera objekten i databasen och ett annat för att beskriva operationerna. De dbms som är språkoberoende men som tillåter metoddefinitioner som en del av objektmodellen är att hänföra till hybriderna.

Alltså, för odbms är databas-schemat detsamma som klassdefinitionerna i tillämpningen. Av avgörande vikt är förstås att se till att samtliga klienter-tillämpningar utnyttjar samma version av tillämpningen. Annars blir databasens innehåll snabbt inkonsistent. Exekveringsavbrott blir följden om klient A i databasen placerat objekt av objekttyp O1 enligt sin uppfattning och klient B sedan försöker hämta in dessa enligt sin, något avvikande definition av objekttypen. Ett absolut krav är förstås att klient och

server (odbms) har samma uppfattning om objektstrukturen. ODMG-93 redovisar följande princip för att integrera tillämpning och dess odbms-stöd.



Figur 28

Datastrukturen definieras i ett Object Definition Language (ODL). ODL är att jämföra med DDL i konventionella sammanhang. ODL kan formuleras i ett OO-språks syntax, enligt ODMG-93s ODL-syntax, genom ett grafiskt användargränssnitt, eller liknande. I alla händelser bearbetas specifikationen och ut kommer metadata som underlag till generering av databas-schemat. Samtidigt blir klassdefinitionerna, tillsammans med källkoden, inklusive eventuella OML-utvidgningar, input till det använda språkets kompilator. Binärkoden länkas sedan ihop med runtime-biblioteket för klient-delen av odbms.

4.7.2 Anpassning av språk till odbms-samverkan

4.7.2.1 Generellt

Diskussionen förs utifrån C++-perspektivet. Liknande resonemang är tillämpbart även gentemot andra språk.

Det finns vissa saker som måste kunna hanteras i en C++-tillämpning men som inte ingår i språkets egen repertoar. Dit hör att kunna definiera vilka objekt som ska lagras beständigt och att kunna skilja dem från de "vanliga" transienta objekt. Inte heller finns sambandsbegreppen i C++-modellen, bara pekare. Två grundläggande lösningsprinciper finns företrädda bland C++-baserade produkter. Den ena löser det utan, den andra med modifiering av C++-syntaxen.

Ingen C++-syntaxmodifiering

Det ideala är om ordinarie C++ kan användas också vid samverkan med ett odbms ("seamless integration"). Genom att etablera odbms-klienten som ett klassbibliotek och låta C++-programmet anropa metoder i klassbiblioteket för de olika operationerna mot odbms, exv för att länka till visst dbms, lagra objekt, committa transaktion, o s v åstadkoms ett språkneutralt gränssnitt mot odbms. Klassbiblioteket kan användas av

många C++-kompilatorer. Detta låter gott och väl men osynligheten är inte total. De klasser som ska lagras i databasen måste vara subklasser till en klass som ombesörjer kontakten med odbms, d v s som innehåller de metoder som behövs för ändamålet. Superklassen brukar benämnas "persistent class". Dessa odbms använder oftast också templated types för inter-objektreferenser istället för att använda C++-pekare. En pekare behöver alltså deklareras olika om den ska peka på transienta resp permanenta objekt. Om en given C++-tillämpning plötsligt ska sättas att arbeta med permanenta objekt fordras med andra ord en hel del ändringar (dels deklaration av pointers, dels subklassning under persistent class). Utgörs en del av tillämpningen av ett inköpt klassbibliotek måste kanske ändringar göras i detta. Om det endast kommer i binärform .....? Alltså långt ifrån "seamless".

C++-syntaxmodifiering

Den andra ansatsen bygger på ett antal ändringar i syntaxen. Att tolka dessa och för att generera rätt mekanismer krävs en förkompilator som översätter till ren C++-syntax. Syntaxpåbyggnaden råder bot på ett antal brister i C++, exv hantering av sambands-typer. Dessa kan hanteras och kontrolleras med användning av vanliga C++-pointers. Dessutom hanteras beständighet utan att någon separat "persistent class" behöver finnas. Istället utvidgas klass-syntaxen för att inkludera både begäran om persistens och klass-extensioner.

Det kan ske genom att objektklassen explicit deklareras som typen persistent med hjälp av prefix.

Alternativt kan man explicit begära att ett visst objekt ska lagras i databasen (liksom radering, utsökning, ...), exv genom overloading av new-instruktionen, d v s att där uttrycka om beständighet önskas eller ej, samt om viss clusteringsstrategi ska tillämpas m m. Med detta alternativ kan vissa objekt av en klass begäras beständiga medan andra bara ges ett tillfälligt liv. Man tillämpar en princip som i odbms-sammanhang brukar gå under formuleringen "orthogonality of persistence and type".

En följd av new-alternativet blir behov av en konvention för hur man ska hantera de objekt som ett visst beständigt objekt har samband med. I normalfallet blir även de beständiga även om inte detta explicit uttryckts. De blir det så länge som de är refererbara från explicit beständiga objekt. Dock gäller nåbarhetsprincipen endast de objekt som "hängt med" på grund av referens från ett persistent objekt. De som explicit har begärts lagrade kvarstår tills de explicit raderas.

Nackdelen är just behovet av förkompilering med åtföljande risk för bristande överensstämmelse i genererad C++-kod mellan olika förkompilatorer.

I Smalltalk-baserade produkter blir ett objekt beständigt när det sätts att referera till ett annat beständigt objekt. I konsekvensens namn antas objekt vara raderingsbara när de inte längre har någon sådan referens. Generell garbage collection-funktion tar då hand om jobbet. I de lägen man vill behålla isolerade objekt (vilket mycket väl kan vara relevant) måste dessa objekt sättas att relatera till något artificiellt, beständigt objekt.

4.7.2.2 Hantering av relationships

Ofta realiseras flervärdiga samband som någon form av länkad struktur. De kan därmed, till skillnad från rdbms, vara ordnade (manuellt eller automatiskt), d v s i form av lista eller array.

Ofta tillämpas dubbelriktade samband, vilket dels är modellmässigt snyggt, dels gör det möjligt att alltid finna de objekt som påverkas i samband med ändring av något objekt (radering, exv). Det är inget fel att hantera samband med hjälp av dubbla pekare så

länge som det är en intern angelägenhet och inget användaren behöver se eller ta hänsyn till. Dock måste de deklareras vid båda objektklasserna istället för att hanteras som en objektsammanbindande, egen företeelse. Vissa odbms tillåter valfrihet mellan enkla och dubbelriktade samband.

4.7.2.3 Prestandatrimning

Ibland hörs svepande schabloner som att hårdvaruutvecklingen gör prestandatrimning ointressant. Den hittillsvarande historieutvecklingen talar ett helt annat språk. Aptiten växer i minst lika snabb takt. Däremot blir det allt viktigare att undvika tekniklösningar som försvårar dynamisk anpassning till nya förutsättningar, som låser till teknikval, som riskerar osynliga degenereringseffekter, o s v.

För odbms gäller det framförallt att undvika dyra diskaccesser och nättransporter. Teknikutvecklingen kommer snarast att accentuera detta förhållande än mer i framtiden. Flera typer av prestandatrimning på programmerarnivå är tänkbara. Produkterna erbjuder i olika grad denna typ av service. Nedan följer några exempel.

Composite objects

Vissa ODBMS poängterar hanteringen av composite objects, d v s "part of"-sambandstypen, som väsensskilt övriga sambandstyper. Ett composite object kan ju ses som en enhet där resp komponent inte har någon självständig existens. Integrity constraints kan appliceras på hela objektmassan. Exv innebär en radering av ett superobjekt definitionsmässigt också radering av subobjekten. Andra operationer, som kopiering av superobjektet, innebär kopiering av samtliga objekt. På samma sätt kan låsning och versionsgenerering enkelt appliceras på hela objektstrukturen. Kan även indikera fördelaktig fysisk clustering.

Container objects

Under ett container object grupperas de objekt som på något sätt logiskt hör ihop. På samma sätt som vid composite objects kan olika operationer appliceras på container objects. De har en egen semantisk innebörd som deklareras i schemat. I vissa produkter hanteras container objects helt enkelt som vanliga objekt, förutom att vara en gruppering av andra objekt. Därigenom kan de bl a ha egenskaper och samband. Vid inläsning av ett container object gäller normalt att alla contained objects läses in samtidigt. Här kan avsevärda prestandavinster åstadkommas. Samtidigt finns risken för prestandaförluster om grupperingen gjorts dåligt.

Clustering

Clustering-specifikation erbjuds av flera odbms. Det står för en begäran om att vissa objekt fysiskt ska placeras nära varandra, helst på samma sida eller segment. Syftet är rent prestandabetingat. Åtgärden kan vara extremt prestandabefrämjande om sannolikheten för att objekten ska skapas, bearbetas, raderas, ges ny version, etc samtidigt är stort. Dock gäller att noga tänka sig för när specifikationen formuleras samt att löpande hålla uppsikt över dem eftersom det kan finnas risk för att de på sikt inte längre speglar de reella förutsättningarna. Degenereringsrisken bör alltid hållas under uppsikt. Clustering kan exv begäras

- över viss sambandstyp, d v s alla objekt som är relaterade från visst objekt över viss sambandstyp
- per objekttyp, d v s alla objekt av viss typ
- genom explicit begäran vid lagring.

Objektkopior

I distribuerade databaser kan det vara mycket accesstidsbesparande att lagra objekt vid flera noder. Speciellt i situationer där utsökningsfrekvensen vida överstiger uppdateringsfrekvensen. Att arbeta med kopior kan även ge ökad tillgänglighet för objektet. Går en nod ner kan sökningen ske vid en annan. Vid synnerligen tidskritiska tillämpningar kan det till och med vara motiverat att placera kopior i samma databas, exv baserade på olika sökstrategier eller under olika clusterings. Priset betalas genom mer avancerad kopiehantering (konsistenta uppdateringar, bedömning av vilket objekt att hämta, o s v). En väl genomtänkt strategi för grad av mångfaldigande och av vilka objekt(typer) behövs. Observera att kopiehantering bör vara helt intern och utanför tillämpningens kontroll. Tillämpningen ska överhuvudtaget inte behöva vara medveten om att kopiehantering finns "under ytan".

Indexering

Rdbms har av tradition använt olika typer av index för att snabba upp sökningar. Odbms kan använda samma grundteknik för att snabba upp värderelaterade utsökningar.

Lagringsutrymme

Vid stora databaser är det av vikt att effektivt utnyttja det fysiska minnesutrymmet. Indirekt påverkar givetvis minnesbehovet även tidsrelaterad prestanda. Minnesfragmentering vid användning av A-OIDs, många indexes, frekvent användning av kopior, brist på stöd för uppdelning och expansion av databasen över flera fysiska enheter, ofullständig radering av inaktuella objekt, m m är faktorer att hålla under uppsikt. Stora OIDs, exv 70-90 bitar, kan uppta en avsevärd del av databaser med många komplexa samband och där objektens attribut är få och/eller svarar mot korta, enkla datatyper.

5. Konventionell dbms-funktionalitet i odbms-tappning

5.1 Inledning

Ovan har vi berört några typiska odbms-egenskaper:

- En mer eller mindre stark integrering med OO-språk, en integrering som möjliggör smidig datahantering mellan OO-språk och databas. Ingen "impedance mismatch".
- En mycket språkinfluerad objektmodell, i första hand C++.
- Inga begränsningar till vissa datatyper som annars är vanligt hos rdbms.

Men det behövs mer.

Rdbms har vanligtvis en rik, kompletterande funktionalitet för uppbyggnad, administration och övervakning av databasen. Kommersiell drift ställer därvidlag allt större krav. Dit hör

- Fleranvändarhantering (Concurrency control)
- Transaktionshantering
- Schemahantering
- Loggning, Backup, Recovery (Återhämtning)
- Frågespråk
- Behörighetskontroll
- Villkors/konsistenskontroller (Constraints)
- Distribution
- Klient/Server-miljö.

Självfallet berörs odbms i allt väsentligt av liknande krav. Objektmodellen och dess realisering i en odbms innebär dock delvis nya förutsättningar som kräver sin speciella lösning. Ofta har odbms-leverantörerna valt olika lösningsstrategier.

De tillämpningsområden som C++ vänder sig till ställer även en hel del nya krav, som inte ansetts vara av väsentlig vikt inom konventionell administrativ databashantering. Dit hör framförallt hantering av

- långa transaktioner
- flera versioner av objekt.

Viss konventionell rdbms-funktionalitet låter sig inte helt enkelt införas i odbms-miljö. Den intima kopplingen till ett oo-språk innebär nya förutsättningar och ställer nya villkor. Dit hör

- icke-proceduriellt frågespråk
- explicit schemahantering.

Värt att notera är att vy-hantering normalt inte finns i odbms.

Klient/Server-miljö har vi redan berört. Några av de övriga funktionerna kommenteras kortfattat nedan. För produktspecifika lösningar hänvisas till leverantörernas manualer.

5.2 Transaktionshantering

Transaktioner används för att gruppera ett antal operationer som logiskt hör ihop. Det bakomliggande motivet är att se till att antingen allt eller inget görs. Samtidigt behöver man kunna jobba opåverkat med de data som berörs. Här kommer låsningsmekanismer in i bilden. Allt i syfte att tillse att databasen hela tiden är konsistent. Av tradition är dessa transaktioner korta, exv i form av en kombination av utsökningar och uppdateringar i en affärsprocess.

Odbms används bland annat inom tillämpningsområden med inslag av konstruktion, exv inom CAD/CAM. Här är normalsituationen att en person eller en grupp av personer vill kunna jobba ostört med en konstruktion av något slag under längre tid. Först när man anser sig åstadkommit något som är stabilt och värt att släppa till omgivningen, vill man göra det. Den instabila fasen kan sträcka sig över dagar, veckor.... Vi har en s k lång transaktion.

Kanske har andra behov att jobba mot delvis samma konstruktionsunderlag för något annat syfte eller för att kreera en alternativ design. De accepterar givetvis inte att behöva vänta på det andra arbetet. De vill omgående kunna jobba på sin version. Med andra ord uppstår behov av versionshantering. Se nedan.

Vissa odbms tillåter nästlade transaktioner, andra inte.

I de fall låsning används, utför vissa produkter låsning på fysiska enheter som sida eller segment, vilket är ett naturligt val i de fall dessa skickas mellan databas och tillämpning. Även låsning per konfiguration förekommer. Möjligheten till clusteringsstrategier ökar dessutom sannolikheten att ett flertal objekt inom en sida som samtidigt bedöms vara av intresse alla låses genom en operation. Objektbegreppet verkar annars vid första påseende vara en naturligare entitet för låsning. Vissa produkter tillåter val mellan låsning av objektklasser, enskilda objekt och till och med versioner av objekt. De två sistnämnda nivåerna ger en finfördelad låsning där risken för att man låser mer än vad som behövs, minimeras. "Betalningen" sker med overhead.

Det finns här ingen bästa låsningsprincip. Tillämpningens egenskaper avgör vilken princip som i den enskilda fallet är att föredra. Produkterna har också valt olika lösningar. Därutöver finns variationer baserade på optimistisk och pessimistisk ansats.

Som vid vanlig transaktionshantering tillämpas olika typer av låsningar (exclusive, shared locks, ...). I vissa odbms sker låsningen dolt som en följd av transaktionsdefinitionen. I andra anges önskad låsning explicit. Transaktioner över distribuerade databaser måste kunna hanteras. I odbms läggs normalt transaktionskoordinatorn i klient istället för i servern, så som är fallet i rdbms. Däremot måste samspel till för korrekt hantering av de underliggande låsningsbehoven. Detta samspel kan vara förenligt med en hel del overhead i de fall objekt cachas hos klienten "så länge som möjligt", d v s med förhoppningen att samma klient ska komma att vara nästa efterfrågare av objektet. Typisk sekvens av operationer:

> En klient A startar en ny transaktion i vilket bl a objekt X ska användas. Klienten kontrollerar om objekt X finns i cachen. Om så, kontrolleras att ingen annan process' transaktion utnyttjar den. Om så, vänta. Finns inte objektet hos klienten begärs det in från servern. Servern, som har reda på vilka objekt som finns hos vilken klient, kontrollerar om X finns ute hos någon av dem. Om inte, skickas X över till den efterfrågande klient A vars transaktion kan fortsätta. I annat fall skickas en begäran till klient B (som har X) att släppa det. Använder klient B objekt X på ett sätt som inte kan förenas med den tänkta användningen i A meddelar B servern detta och klient A får vänta tills B är färdig. I sinom tid är B färdig, tar bort X från sin cache och meddelar servern att det nu är fritt fram för annan användning. Om istället B inte använder X när servern frågar, utförs sista steget omedelbart. Obs, att flera klienter

kan behöva till-frågas, exv om de alla har objekt X för läsning och A behöver det för uppdatering.

För att undvika låsningar och deadlocks p g a lästa objekt tillåter vissa odbms, när så begärs, att läsning utförs mot en konsistent tidigare version av databasen medan uppdateringar utförs på vanligt sätt mot en senare version av databasen. Andra odbms har funktionalitet för grupparbete såtillvida att uppdateringar utförs utan låsning från flera klienter samtidigt enligt "sist till kvarn ..."-princip. För sann grupparbetsmiljö krävs sedan mekanismer för replikering av uppdateringar ut till samtliga klienter som arbetar med det uppdaterade objektet.

5.3 Versionshantering

Behovet av flera versioner av element i databasen har redan konstaterats. I allmänhet tas ny version av ett helt objekt eller grupp av objekt. Objekten länkas i tidsordning (dubbellänkning för fri navigering mellan versionerna). Normalt kan man, vid behov, etablera flera versionsgrenar, exv om flera jobbar på sin version av något som i slutänden ska sammanföras till en sammanjämkad helhet. Att hantera versioner på mindre granularitet än objekt, exv attribut eller samband kan vara vettigt vid stora objekt. Detta stöds dock inte av något odbms.

Att inte versioner normalt hanteras i ett rdbms kan dels bero på bristande behov inom de normala tillämpningsområdena men också på att det inte finns något lika påtagligt att versionshantera som ett objekt. En tuple är sällan en homogen företeelse, i alla fall inte i en databas enligt 3:e normalformen.

Företeelsen "version" är mångfacetterad och vilar i odbms-sammanhang knappast på någon stabil teoretisk grund. Standard saknas. Varje odbms har också valt en egen lösning. Frågan är om den givits en övertänkt generell utformning eller blivit "tillsnickrad" för någon kunds behov?

Att ta ställning till, bl a:

- Är "objektversion" något som existerar under arbete med visst objekt, d v s innan det är "färdigt"? Är det något som helt enkelt finns i olika alternativ, exv olika kundanpassade versioner av en bilmodell? Är det något som uppstår med automatik p g a att man utför viss operation, exv initierar en lång transaktion (avsnitt 5.2)? Hur ställer sig versioner av objekt till olika versioner av ett schema under vilket objektets klass finns definierat? Börjar man om per schemaversion eller skapas automatiskt nya versioner av alla objekt eller finns inget beroendeförhållande eller ....?

- När/hur skapas ny version? Finns möjlighet att påverka?

- Ska samband med andra objekt enbart gälla viss version av objektet eller versionsneutralt eller både-och enligt användarens önskemål. Om önskemålen ändras, finns i så fall stöd för nödvändiga anpassningar?

- Finns preciserad åtskillnad mellan begreppen version och konfiguration? En konfiguration är en uppsättning objekt som för något behov ska "leva" tillsammans, exv ett sammansatt dokument. Konfiguration kan av den anledningen vara ett lämpligt element att versionshantera. Samtidigt kan det i andra sammanhang finnas behov av att kunna kombinera olika versioner av olika objekt till en viss konfiguration.

- Hur ska ett antal utgrenade, parallella versioner sammanföras till ett "releasat" objekt?

- Hur mycket hanteringsstöd för versionsstyrning, presentation och kontroll ges i produkten?
- Att skapa en version kan rent tekniskt antingen innebära att en full kopia eller att endast differenserna mot grundversionen registreras. Vad gäller för produkt X? Får det konsekvenser exv i en distribuerad miljö?
- Vissa odbms tillåter att objektversion som har senare versioner får uppdateras, andra låser dessa objekt för vidare operationer annat än kopiering.
- Vissa odbms ger möjlighet att ange default-versioner för de situationer där specifik version inte preciseras.

Som lätt inses är versionshantering är en mekanism enbart för dem som förstår dess innebörd och som noga bedömt dess effekter för den egna tillämpningen.

5.4 Skydd

Skydd i form av säkerhet mot felaktiga handgrepp (safeness) och åtkomst (security, authorization) är centrala faciliteter i ett dbms. För odbms blir lösningarna ganska komplexa med tanke på den svårreglerade flyttningen av objekt mellan processinternt minne och sekundärminne, generaliseringshierarkier, klient-baserade operationer, etc. Ofta handskas man dessutom med hela, fysiska sidor (snarare än med objekt) och kopplar behörighet till dessa. Detta ger bl a som konsekvens att attribut inom objekt som ska ges olika behörighet måste läggas på olika sidor. Det ligger utanför rapportens ram att gå in på variationer och detaljer kring skydd.

5.5 Frågespråk

Frågespråkshantering ligger egentligen inte i linje med objektmodellens idé, därmed inte heller som en odbms-facilitet. En fråga är ju något som oftast riktar sig mot en mängd objekt, och deras attribut. Ofta berörs även samband till andra objekt, allt under en mer eller mindre komplex villkorsbeskrivning. Frågan appliceras på objektmassan med rättighet att, utanför objekts egen kontroll, titta in under "skalet" på dess attribut och samband. Dels strider det mot inkapslingsprincipen, dels kräver det initialt ett mängdbegrepp att utgå ifrån.

Odbms-företrädarna har givetvis motargument mot detta synsätt. Mängder kan exv erhållas genom att alltid relatera de objekttyper, som kan komma att bli startobjekt för frågor, på ett universum-objekt. Via detta kan alla förekomster av en objekttyp återfinnas via ett samband. Därefter vidtar logiskt ett förfarande som innebär att hanteringsmekanismen skickar meddelanden till respektive objekt för att hämta attribut och samband, hanterar dessa enligt frågespecifikationens villkor, går vidare till kvarvarande objekt enligt sambanden, o s v. Att så inte sker i praktiken av prestandaskäl inses lätt. Normalt hanterar odbms med automatik en extension av en objekttyp. Extensionen består av referens till samtliga objekt tillhöriga objekttypen. Dessutom tillåter man sig helt enkelt att inspektera objektens innehåll i databasen.

Vissa anser att frågespråk inte behövs. Man har ju tillgång till C++-språkets fulla uttryckskraft. Antagligen är detta tillfyllest för programmerare, men är dessa målkategorin? Andra anser en utvidgning av C++ som en smaklig lösning. En förkompilering behövs dock. Mer användartillvänd uttryckskraft erhålls men fortfarande med för stor C++-barlast.

De flesta odbms har, bl a av marknadsmässiga hänsyn, inkluderat någon form av separat frågespråk. Användning av frågespråk minskar risken för många varierande formuleringar av samma frågeställning. Ökad konsistens uppnås. Ett frågespråk kan dessutom lättare optimeras. Frågespråken innehåller normalt konstruktioner för navigering genom objektstrukturen, något som ligger nära till hands eftersom objektmodellens hanterar explicita samband. Kritiker brukar föra fram att man därmed tagit ett steg tillbaka till nätverksdatabasernas hur-orienterade operationer från SQLs vad-orientering. I realiteten är skillnaden mycket liten. Det är knappast någon som vågar ställa en fråga i SQL utan att ha kunskap om datastrukturen. Har man väl denna kunskap kan det knappast vara något fel att utnyttja den för frågeformulering. Snarast ger en navigeringsspecifikation en mer påtaglig förståelse i en entydig och kompakt form. Utgående från schemat i figur 10 kan företaget BIL AB snabbt ta fram vilka som hyr företagets bilar genom följande navigering med hjälp av punkt-notation:

"Bil AB".Äger_bil.Hyrs_av

Om vi utgår ifrån att de båda sambandstyperna är M:M blir en SQL-fråga mot en motsvarande relationsmodell enligt 3:e normalformen ingen lätt formulering, än mindre att förstå för en utomstående.

En strävan mot anpassning till utvidgad SQL-syntax tycks numer råda. Eftersom objektmodellen "täcker" relationsmodellen "med råge" bör det resulterande språket kunna bli ett superset av SQL, något som databasmarknaden knappast skulle ogilla. Dock saknas ännu stabilitet och samsyn. Problem att tackla är bl a

- Ska uppdateringar tillåtas eller enbart utsökningar (Select-sats)?
- Ska operationer (metoder) kunna refereras och exekveras som en del av frågeexekveringen?
- Ska man kunna "titta på" operationsbeskrivningar på samma sätt som attribut?
- Ska exekveringen ligga hos klient eller server? Båda alternativen används i produkter. Server-exekvering belastar servern. Konsistensskäl kan kräva att uppdaterade objekt som ännu ligger kvar hos klienter först förs till servern innan exekvering startas. Å andra sidan behöver endast relevanta objekt kommuniceras. Vid klient-exekvering, ska objekt som uppdaterats inom ännu inte avslutad transaktion anses vara med i frågeunderlaget? Om inte, måste ju eventuellt "gamla" objekt läsas in varvid vi får två olika kopior av samma objekt att hålla reda på? A-OIDs bygger på klient-exekvering medan I-OIDs underlättar serverbaserad exekvering.
- Ska tillfälliga objekt också omfattas av en fråga?

De facto-standarden ODMG-93 (se separat rapport) innehåller en abstrakt frågespråksspecifikation, OQL. En fråga i OQL skickas som textsträng till databasen för exekvering. Resultatet tas emot antingen som en mängd eller genom vanlig cursor-hantering. I princip har man här samma situation som SQL i programmeringsspråk, d v s den mappning man argumenterat inte behövs vid användning av odbms. I framtiden kan en mer språkintegrerad ansats förväntas. Frågespråkssidan är den svagast utvecklade funktionaliteten hos odbms. Fortsatt arbete behövs innan ett stabilt frågespråk existerar. Alternativt tar man till sig SQL3, när väl detta standardiserats.

En kritik som riktats från rdbms-håll och som har viss bäring på frågespråk är att odbms saknar vy-hantering. Detta är en modellmässig begränsning snarare än en ren produktbrist. En annan kritik är att frågor bara kan riktas mot en fördefinierad struktur. SQL

tillåter ju i princip joins mellan vilka kolumner som helst, bara de är av samma datatyp. Om denna frihet svarar mot legitima behov råder det dock olika uppfattningar om.

5.6 Schemahantering

Tillämpningar utvecklas mer eller mindre kontinuerligt. Följden blir inte sällan ändringar i databasschemat. Till viss del överensstämmer problematiken med motsvarande för rdbms. Objektmodellen genererar även nya problem.

Bland typiska uppdateringar kan nämnas

- Attribut
  » add, drop, rename, change default, change data type
- Metoder
  » add, drop, rename, change internal code
- Generaliseringsstrukturen
  » add A eller drop A från sub/superclass for B
- Class
  » add, drop, rename, partition

Både ändringar i metoder och i generaliseringsstrukturen innebär nya utmaningar jämfört med rdbms.

En ytterligare dimension skapar den nära kopplingen till ett oo-språk. Programmet är ju i princip schemat. Överensstämmelse måste gälla mellan klassdefinitionerna i tillämpningen och schemat i odbms. Schemaändringar kräver normalt omkompilering. Om tillämpningen dessutom är utspridd i en klient/server-miljö med många klienter ökar risken dramatiskt för att konsistensen inte är hundraprocentig överallt. Detta blir konsekvensen eftersom operationen inte följer objektet utan objektklassen, d v s definieras i C++runtime-modulen.

Schemaevolution påverkar också databasens innehåll. Ett sätt att hantera detta är att låta en ändring få till följd att alla befintliga objekt i databasen ändras för att svara upp mot ändringen.

Ett annat alternativ är att vänta med uppdateringen tills objektet refereras. Dock måste objektet då ha en indikator som talar om vilken version av schemat det svarar mot samt måste alla versioner av schemat och deras samband finnas kvar så länge det kan finnas objekt som svarar mot något av dem. Ofta är ändringen av en art som inte enkelt låter sig beskrivas genom en enkel mappning. Det kan röra sig om nya constraints, nya typer av samband som kanske till del ska appliceras på redan existerande objekt i databasen, o s v. Programmering blir i allmänhet nödvändig.

Ytterligare ett alternativ är att för alltid koppla ett visst objekt med den schemaversion under vilken det skapades. Inte minst kan behovet gälla olika versioner av metoder. En ordersumma uträknad 1988 ska exv alltid räknas ut med hjälp av då gällande momsregler även om dessa ändrats i senare versioner av schemat.

Databasanpassning till schema är i och för sig ett generellt problem och inget specifikt för odbms. Objektmodellens rikare semantik och dess dynamiska ingredienser (metoder) samt nya odbms-tillämpningsområdens specifika krav accentuerar vikten av fungerande lösningar.

5.7 Distribution

Distribuerade databaser blir alltmer vanligt. Orsakerna är många. Dit hör bl a mycket stora datavolymer, säkerhetsaspekter, prestandakrav, replikering av data, ökad lokal flexibilitet, ...

Samma behov finns för odb. Beroende på vald tekniklösning, inte minst principerna för identifiering av objekt, ställer sig distribution olika komplicerat. Kopplat till distribution av data är även distribution av schema. Odbms-produkterna visar upp en bred flora av lösningsalternativ. Det är alldeles tydligt att distribution är en av de minst stabila mekanismerna i existerande produkter. En presumtiv odbms-användare bör noga studera produktegenskaper i perspektivet mot de egna behoven. En noggrann redogörelse för de olika varianterna ligger utanför denna rapport. Dessutom sker här snabba förändringar. Vi inskränker oss till några noteringar:

- Med vilken granularitet kan distribution begäras? Per objekt, objektklass, konfiguration, ...?
- Fungerar distribution enligt master-slave eller peer-peer ansats?
- Var ligger uppgifterna om vad som ligger var? I respektive schema eller fristående?
- Ligger fulla scheman vid varje nod eller enbart behövligt? Hur uppdateras scheman? Var, vem och hur kontrollerar erforderlig schema-konsistens?
- Vid klientbaserad transaktionsövervakning, har klienten tillgång till distributionsinformation, exv för att skicka frågor och sammanställa frågesvar?
- Hur styr man önskad lagringsplats?
- Kan objekt flyttas? Hur begärs detta?
- Sker transaktionssamverkan mellan noder med hjälp av two-phase commit? Detta är en angelägen princip som innebär att en transaktion med stor säkerhet lämnar en distribuerad databas i ett konsistent läge vid commit. En server har rollen som koordinator. Vid transaktionsslut sänder koordinatorn en begäran till övriga servrar att göra sig beredda för avslutning av transaktionen (första fasen). Om samtliga svarar positivt, d v s har kunnat markera sina loggar för avslutning, skickar koordinatorn ett commit-meddelande till servrarna (andra fasen). Dessa tömmer loggen alternativt använder den för att uppdatera databasen (beroende på vald princip) och skickar konfirmation till koordinatorn. När alla konfirmerat kan koordinatorn avsluta sitt jobb. Skulle första fasen visa att samtliga inte svarar positivt skickas istället ett abort-meddelande till dem varvid de återställer databasen i det skick den befann sig före transaktionsstart med hjälp av loggen alt. tömmer den (beroende på vald princip).
- Vilken grad av plattformsoberoende erbjuds?

5.8 Händelsestyrda operationer

Vissa odbms tillåter specifikation av operationer som ska utföras i samband med att en viss händelse inträffar. Alternativt kan klienter begära att få meddelande när specificerad händelse ägt rum. Händelsen kan t ex beröra ett visst objekt, objektklass eller objekt inom viss klass. Exempel på händelser kan vara uppdatering, radering, låsning. Vanligtvis ligger funktionen hos servern där den fulla överblicken finns. Serverbasering gör det också enklare att ändra/lägga till händelsespecifikationer

eftersom klienterna inte påverkas, uppdateras. Denna typ av specifikation kan även användas för replikerade objekt. Uppdateras visst objekt skickas motsvarande uppdateringsmeddelande till de klienter/servrar som hanterar dess duplikat.

Funktionen är inte specifikt oo-relaterad men förmodligen extra användbar inom de tillämpningsområden odbms vänder sig till.

5.9 Backup, recovery

Odbms är synnerligen komplicerade mekanismer. De innehåller numer de flesta av de funktioner som normalt förknippas med ett dbms, oavsett typ. Till viss del har de expanderat dessa funktioner med ytterligare faciliteter. Därutöver företräder de en avancerad objektmodell med tillhörande funktionsstöd. Språkintegreringen innebär ytterligare en komplexitetsfaktor, speciellt i klient/server-miljö. Rimligtvis borde följden av detta bli ett betydligt mer komplicerat recovery-arbete. Speciellt gäller detta situationer (förhoppningsvis mycket sällsynta), som innebär "rädda/återskapa vad som räddas kan". Samtliga produkter säger sig stödja backup och recovery. En presumtiv odbms-användare bör noga studera hur den tilltänkta produkten åstadkommer denna service.

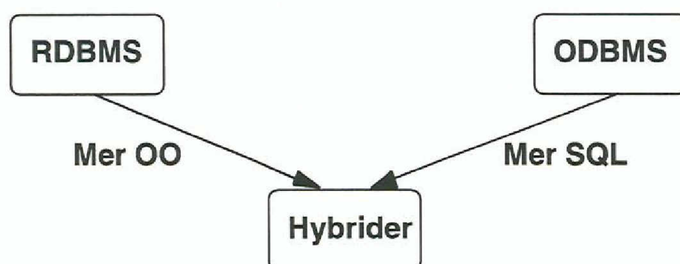
6. Slutsatser

Efter de första årens turbulens, hype och osäkerhet kring odbms börjar nu en viss samstämmighet att uppträda i artiklar och på konferenser. Odbms och rdbms baseras på olika datamodeller, har olika ursprung (språk resp datamodeller) och syftar till att tillgodose olika tillämpningsbehov. Så kommer sannolikt att vara fallet framöver också. ODMG definierar ett odbms som "a DBMS that integrates database capabilities with object programming language capabilities". Odbms-leverantörernas försök att tränga in på rdbms-reviret (administrativa tillämpningar) kommer att successivt mötas med olika typer av anpassningar och tillägg i rdbms-systemen. Av den anledningen kommer inte förhoppningen att infrias, speciellt inte som de unika odbms-faciliteterna inom det administrativa tillämpningsområdet inte ger något nämnvärt mervärde. I vissa fall är det snarare tvärtom. Om dagens system skulle vara baserade på en mer semantiskt rik datamodell (extended ER-modell) än relationsmodellen och inklusive ett därtill anpassat DDL/DML, skulle sannolikt aldrig odbms-tanken överhuvudtaget kommit upp för konventionella databastillämpningar. Snarare är det så att relationsmodellens svagheter inbjudit till konkurrensen.

OO-språksbaserade tillämpningar kommer att bli fler och kommer att driva in i nya tillämpningsfält. Tillämpningar kommer att bli alltmer komplexa till struktur och funktionalitet. Man kan förmoda att ett stort antal av dessa tillämpningar kommer att ha behov av stöd för någon typ av beständig lagring av språkobjekten. Odbms kommer i dessa lägen att fylla ett angeläget behov. Om odbms kommer att leva kvar som en självständig produktkategori eller att vara inbakade i generella produkter, operativ-system, m m återstår dock att se. Det finns till och med de som anser att C++ (den objektmodell som de flesta odbms baseras på) har en kortare kvarvarande livslängd än SQL.

Odbms är synnerligen komplexa produkter med en mångfacetterad funktionalitet. Vissa funktioner är stabila, andra under intensiv utveckling. Stabilitet är knappast odbms-områdets mest framträdande kännetecken. Komplexa produkter behöver höggradigt kompetenta och stabila leverantörer. Att utnyttja produkterna för industriella tillämpningar ställer därtill mycket höga krav på utvecklare och underhållare.

Låt oss sluta kalla odbms för nästa generations dbms. Snarare är det en alternativ dbms-teknik med utmärkta egenskaper inom vissa tillämpningsområden. Den aktuella trenden mot ökat språkoberoende kan öppna nya marknadssegment men riskerar samtidigt göra odbms-produkterna mer profillösa eller utsatta för suget in i den så kallade hybrid-kategorin. Kanske kommer spelplanen på sikt att utvecklas i enlighet med figur 29.



Figur 29