Databaser och SQL - en kort introduktion

Databaser är inte precis något som i sig är svårbegripligt. Det är bara en massa data samlade på ett ställe i strukturerad form. Problemen består i att det just är en *massa* data, och att det skall gå att få ut *information* ur databasen via olika *urval* av data, som man gör genom att ställa *frågor* till databashanteraren. Det gäller alltså att få databashanteraren att hantera ofta enorma datamängder på ett effektivt sätt. Hashtabeller, snabba sorteringsalgoritmer och en intelligent ordnad datalagring är viktiga frågor när man konstruerar databaser. Ett annat problem är att en och samma databas oftast används av många personer på olika håll och i olika syften. Man vill kunna uppdatera databasen under drift, och ofta från flera håll samtidigt, vilket ger en hel del ganska svårlösta synkroniseringsproblem. Dessutom måste man hålla reda på vem som får göra vad med informationen i databasen, så det finns många aspekter på datasäkerhet inblandade.

Inget av dessa intrikata problem kommer att behandlas här. Vi nöjer oss med att skrapa på ytan av ämnet och endast beskriva en mycket enkel databas ur användarens synvinkel. Vi kommer också att begränsa oss till databaser med en enkel och rättfram struktur. Att designa en databas för att vara formellt optimal och effektiv för ett visst behov av att lagra och söka information kallas databasmodellering, och är ett av huvudmomenten i kurser i databasteknik. För att använda och även designa databaser för weblösningar så klarar man sig dock ofta bra med en intuitiv förståelse utan några djupare teoretiska kunskaper, och det är detta vi försöker förmedla här.

Relationsdatabaser

Termen “relationsdatabaser” har fått en ganska oförtjänt mystisk klang. Grundidén är egentligen mycket enkel. Fördelen med relationsmodellen är att den är väl formaliserad och därför trevlig att arbeta med. Den har börjat visa sig något otillräcklig för de allra modernaste tillämpningarna av databaser, men för de flesta normala behov är det en mycket bra modell.

Relationer

Alla data i en relationdatabas lagras i *relationer* (eng. *relations*), eller, som de oftare kallas, *tabeller* (eng. *tables*). Det finns ingen datastruktur i en relationsdatabas som inte är en relation. Anledningen till att relationer kallas tabeller är att det bekvämaste sättet att presentera dem är just i form av en tabell.

Det är som vanligt enklast att förklara med ett exempel. Nedanstående tre tabeller eller relationer utgör en enkel, liten relationsdatabas:

staff

username	firstname	lastname
stegu	Stefan	Gustavson
marka	Martin	Karlsson
wente	Wendy	Testaburger
erica	Eric	Cartman
kenmc	Kenny	McCormick
stama	Stan	Marsh
kylbr	Kyle	Broflowski

groups

groupcode	groupname
media	Medieteknik
hyper	Hyperteknik
gizmo	Prylteknik
ultra	Ultrateknik

affiliation

username	groupcode
stegu	media
marka	media
stama	ultra
kylbr	gizmo
erica	gizmo
wente	ultra
wente	media
stegu	hyper
erica	hyper

Vi behöver definiera några grundläggande begrepp innan vi går vidare.

Relationsnamn. Varje relation har ett *namn*, vilket oftast skrivs som en titel till tabellen. I vårt exempel har vi relationerna **`staff`**, **`groups`** och **`affiliation`** (affiliation betyder "medlemskap" på engelska).

Fältnamn. Varje relation innehåller ett antal *fält* (eng. *fields*), som helt enkelt är kolumnerna i tabellen. Fält har också *namn*. I exemplet har relationen **`staff`** de tre fälten **`username`**, **`firstname`** och **`lastname`**.

Post. Varje rad i en relation kallas en *post* (eng. *record*).

Nyckel. Varje post i en relation bör ges en unik *nyckel* (eng. *key*), vilket är ett fält som för varje post garanteras ha ett unikt värde. I vårt exempel är det lämpligt att använda fältet **`username`** som nyckel för relationen **`staff`**. Fältet **`groupcode`** är något artificiellt skapat särskilt för att vara en unik nyckel för relationen **`groups`**. Om man inte hittar någon naturlig nyckel eller vill skapa sig någon vettig sådan kan man även använda ett godtyckligt fält med till exempel ett unikt id-nummer, men någon unik nyckel skall det finnas. Nyckeln behöver dock inte vara ett enda fält. Om kombinationen av två eller flera fält tillsammans har ett unikt innehåll för varje post så kan man använda denna kombination av fält som nyckel. Detta kallas då för en *sammansatt nyckel* (eng. *composite key*). Relationen **`affiliation`** är ett exempel på en sådan relation. En gruppkod och ett användarnamn kan förekomma på flera rader i tabellen, men kombinationen av gruppkod och användarnamn är unik för varje rad.

Externa nycklar. Detta begrepp är själva kärnan i relationsdatabaser. Ett fält i en relation är en *extern nyckel* (eng. *foreign key*) om det innehåller samma slags data som *nyckelfältet i en annan relation*. Båda fälten i relationen **`affiliation`** är externa nycklar. Observera att det inte är frågan om någon fysisk länk eller pekare till den andra relationen. Det är upp till databashanteraren att göra kopplingen till den andra relationen och kontrollera att fältet inte innehåller andra data

än en nyckel från den andra relationen. Från användaren sett är det bara ett vanligt fält, om än med litet speciellt innehåll.

Uppdelning i relationer

En stor del av teorin kring relationsdatabaser går ut på att organisera data på ett optimalt sätt inom och mellan sina relationer. Mycket kortfattat kan man säga att det går ut på att identifiera *kopplingar mellan poster* som 1-till-1, 1-till-n eller n-till-m. En typisk 1-till-1-koppling är den unika kopplingen mellan personer och användarnamn i databasen ovan. Ett sådant fält kan utgöra en naturlig nyckel för en relation. Kopplingar 1-till-n är till exempel personers namn. En person har bara ett namn, men två eller flera personer kan heta likadant. Ett annat exempel är telefonnummer. En person kan ha flera telefonnummer (även noll stycken), men varje nummer leder bara till en person. Kopplingar 1-till-n kan antingen lagras som ett fält i samma relation eller i en separat relation via en extern nyckel, beroende på hur data ser ut och hur man vill kunna söka i sina data. Ett exempel på en typisk n-till-m-koppling är grupptillhörigheten (`affiliation`) i databasen ovan. Varje grupp kan ha ett godtyckligt antal medlemmar, inklusive noll, och varje person kan vara medlem i flera grupper, en grupp eller ingen grupp. En sådan koppling n-till-m lagras enklast och snyggast i en separat relation, med hjälp av externa nycklar för de tabeller som kopplas samman.

Alla data i en relationsdatabas lagras i relationer som poster uppdelade i fält. Som tidigare nämnts finns det inga data i en ren relationsdatabas som explicit lagras på något annat sätt. Databashanteraren kan i och för sig göra vad som helst bakom kulisserna, och ofta ser själva implementationen på maskinnivån mycket annorlunda ut, men för användaren ser det ut som om allt verkligen ligger i rader i tabeller, eller, om man skall vara petig med termerna, i poster i relationer. Posterna i en relation är inte sinsemellan ordnade, eller snarare är ordningen oväsentlig. Fälten i en relation har dock en ordning, vilken är samma ordning som de definierades i.

Utifrån denna modell kan man skapa sig en mycket strikt och formaliserad algebra som definierar alla urval och uppdateringar av data som mängdoperationer inom och mellan relationer. Vi hoppar raskt över dessa vackra teorier och går genast in på den praktiska beskrivningen av dessa operationer i ett frågespråk, närmare bestämt SQL.

SQL

SQL står för “Structured Query Language” och är en internationell standard som definierats för att olika databashanterare från olika tillverkare skall kunna tala samma språk utåt mot användaren. De flesta kommersiella databaser i dag stödjer SQL som frågespråk, även om de ibland dessutom har ett eget gränssnitt. Språket har utökats något i de flesta databashanterare för att stödja olika specialfunktioner, men standard-SQL fungerar oftast rakt av utan ändringar. Språket är mycket litet och enkelt, eftersom det egentligen inte behöver klara av särskilt mycket. Man lär sig grunderna i SQL på en kvart ungefär.

Nedanstående beskrivning förtiger mycket av de mer avancerade möjligheterna i SQL. Den tar helt enkelt bara upp det som behövs för att överhuvudtaget komma igång med SQL. De grundläggande kommandona i SQL beskrivs, men vi nämner inte alla varianter av kommandona och avstår från att beskriva många av de mer avancerade sökmöjligheterna. Det är dock en god grund att stå på, och det som beskrivs här räcker faktiskt ganska bra för många mindre behov, som till exempel enklare databaser på WWW.

Om någon vill läsa mer om SQL än vad som nämns här finns det gott om böcker i ämnet, alltifrån handfasta handledningar på praktisk nivå till mycket teoretiska och grundliga beskrivningar av databaser i allmänhet, men med konkreta exempel i SQL.

I styckena nedan behandlas de sex grundläggande satserna i SQL, nämligen:

- CREATE TABLE
- DROP TABLE
- INSERT INTO
- DELETE FROM
- UPDATE
- SELECT

CREATE TABLE: Skapa en relation

En relation skapas med satsen **`CREATE TABLE`**. Tabellen **`staff`** i exemplet ovan skapas till exempel med satsen:

```
CREATE TABLE staff (
  username char(8) primary key,
  firstname char(20) not null,
  lastname char(20)
)
```

Först namnges själva tabellen, och sedan namnges fälten som skall ingå inom parentesen. Man måste också tala om vilken datatyp som skall lagras i respektive fält. Datatypen **`char(n)`** anger att fältet skall innehålla en sträng om maximalt n tecken. Andra datatyper är **`int`** (heltal) och **`real`** (flyttal). De flesta databaser har även separata datatyper för datum (**`date`**) och klockslag (**`time`**). Anledningen är att man ofta lagrar just datum och klockslag i databaser och vill lagra dem på ett konsekvent sätt.

Redan när man skapar tabellen måste man ange alla fält med namn och typ. Hur man talar om vilket fält som utgör nyckelfält skiljer sig något mellan olika databashanterare. I exemplet används orden **`primary key`** för att ange att fältet **`username`** utgör nyckel i relationen. I många databashanterare anger man dock nycklarna separat i efterhand med kommandot **`CREATE KEY`**, även den primära nyckeln. I de flesta moderna databaser måste det inte alltid tvunget finnas en primär nyckel, men det är oftast väldigt praktiskt att ha en, även om det så är ett godtyckligt ID-nummer som bara är unikt men inte betyder något speciellt. Orden **`not null`** anger att fältet **`firstname`** måste innehålla data och inte får lämnas tomt. (Att just detta fält inte får vara tomt är inte uppenbart motiverat - det är gripet ur luften som ett exempel.)

DROP TABLE: Ta bort en relation

Om man vill ta bort en relation använder man satsen **`DROP TABLE`**. Tabellen vi just skapade tas bort med:

```
DROP TABLE staff
```

Obserera att det kan vara en väldigt dramatisk operation att ta bort en tabell. Den tas bort även om den innehåler miljontals poster, utan varningar. SQL-tolken förutsätter att man vet vad man gör och ställer inga motfrågor i stil med “Är du säker?”. Att skapa och ta bort tabeller är något man gör när man *skapar* en databas, det sker inte under själva *användningen* av den.

INSERT INTO: Lagra data i en relation

När man skapat en relation är den tom. För att kunna använda den till något nyttigt måste man fylla den med data. En post skapas och fylls med data med satsen **`INSERT INTO`**. Den översta raden i tabellen **`staff`** skapas till exempel med:

```
INSERT INTO staff VALUES ('stegu', 'Stefan', 'Gustavson')
```

Om man av någon anledning inte vill fylla i alla fält, eller om man inte vill fylla på dem i den ordning de definierades, kan man specificera vilka fält man vill fylla i enligt följande:

```
INSERT INTO staff (firstname, username) VALUES ('Stefan', 'stegu')
```

DELETE FROM: Ta bort en post

En post tas bort med satsen **DELETE FROM**. Posten vi just lade till tas bort med:

```
DELETE FROM staff WHERE username = 'stegu'
```

Med **WHERE** väljer man ut vilken eller vilka rader man vill ta bort. Detta förklaras närmare nedan i stycket om satsen **SELECT**.

UPDATE: Ändra data i en post

Innehållet i fälten i en post som redan är skapad ändras med satsen **UPDATE**. Om Stefan Gustavson till exempel äntligen skulle komma sig för att byta namn till det hans fru heter skulle fältet **lastname** i hans post i relationen **staff** behöva ändras. Detta skulle kunna göras så här:

```
UPDATE staff SET lastname = 'Qvarford' WHERE username = 'stegu'
```

Med **WHERE** väljer man ut vilken eller vilka rader man vill ändra. Detta förklaras närmare nedan i stycket om satsen **SELECT**.

SELECT: Plocka ut information ur databasen

Urval och strukturering av data görs med satsen **SELECT**. Denna sats är den ojämförligt vanligaste i de flesta användningar av SQL, och också den mest kapabla och mest komplicerade.

Antag att vi har en databas med data i tre relationer enligt det ovan presenterade exemplet: **staff**, **groups** och **affiliation**. Om vi vill se vad som finns lagrat om alla användare i relationen **staff** skriver vi:

```
SELECT * FROM staff
```

Svaret från databashanteraren kommer i form av (vad annars) en relation:

```
+----------+-----------+-------------+
| username | firstname | lastname    |
+----------+-----------+-------------+
| stegu    | Stefan    | Gustavson   |
| marka    | Martin    | Karlsson    |
| wente    | Wendy     | Testaburger |
| erica    | Eric      | Cartman     |
| kenmc    | Kenny     | McCormick   |
| stama    | Stan      | Marsh       |
| kylbr    | Kyle      | Broflowski  |
+----------+-----------+-------------+
```

Posterna är inte sorterade i någon särskild ordning om man inte ber om det. Man ber om det genom att skriva ORDER BY sist i frågan:

```
SELECT * FROM staff ORDER BY lastname
```

```
+----------+-----------+-------------+
| username | firstname | lastname    |
+----------+-----------+-------------+
| kylbr    | Kyle      | Broflowski  |
| erica    | Eric      | Cartman     |
| stegu    | Stefan    | Gustavson   |
| marka    | Martin    | Karlsson    |
```

```
| stama    | Stan      | Marsh       |
| kenmc    | Kenny     | McCormick   |
| wente    | Wendy     | Testaburger |
+----------+-----------+-------------+
```

För textfält (**`char(n)`**) sorteras posterna i bokstavsordning, för numeriska fält (**`int`** eller **`real`**) i storleksordning. Man kan också specificera baklänges ordning med nyckelordet **`DESC`** (“descending”):

```
+----------+-----------+-------------+
| username | firstname | lastname    |
+----------+-----------+-------------+
| wente    | Wendy     | Testaburger |
| kenmc    | Kenny     | McCormick   |
| stama    | Stan      | Marsh       |
| marka    | Martin    | Karlsson    |
| stegu    | Stefan    | Gustavson   |
| erica    | Eric      | Cartman     |
| kylbr    | Kyle      | Broflowski  |
+----------+-----------+-------------+
```

Om vi bara vill veta för- och efternamn för användaren med användarnamnet “erica” skriver vi:

```
SELECT firstname, lastname FROM staff WHERE username='erica'

+----------+-----------+----------+
| username | firstname | lastname |
+----------+-----------+----------+
| erica    | Eric      | Cartman  |
+----------+-----------+----------+
```

Mellan **`SELECT`** och **`FROM`** anger man vilka fält man vill se. Vill man se alla fält skriver man **`*`** som i det förra exemplet. Efter **`WHERE`** anger man ett villkor som skall vara uppfyllt för de poster man vill se. I vårt exempel är villkoret att username skall vara lika med “erica”.

För att se om det finns någon som har ett förnamn som börjar på “St” i tabellen kan vi skriva:

```
SELECT firstname, lastname FROM staff WHERE firstname LIKE 'St%'

+-----------+-----------+
| firstname | lastname  |
+-----------+-----------+
| Stefan    | Gustavson |
| Stan      | Marsh     |
+-----------+-----------+
```

Här söker man på ett annat fält än nyckeln, och med ett villkor som är avsett att matcha flera poster. Det returneras också flera poster. Hade det inte funnits någon med ett matchande förnamn i relationen **`staff`** hade relationen som returnerats varit tom. Tomma relationer är fullt tillåtna i relationsdatabaser.

Operatorn **`LIKE`** är en strängmatchningsoperator. I strängen som man jämför med matchar "**`%`**" noll eller flera tecken vilka som helst, och "_" exakt ett tecken vilket som helst. De två tecknen "**`\%`**" matchar ett procenttecken, tecknen "_" matchar ett understrykningstecken, och "\\" matchar en backslash. Alla andra tecken matchar sig själva. Det är skillnad på små och stora bokstäver i matchningen.

De operatorer man kan använda efter **`WHERE`** i standard-SQL är:

`=, >, <, >=, <=, <>, LIKE`

(De flesta dialekter av SQL utökar denna lista med några fler.)

Dessutom kan man ange flera villkor med **AND** eller **OR** emellan. Parenteser används för att ange prioritetsordningen om man har både **AND** och **OR** i en och samma villkorssats efter **WHERE**.

Det vi hittills sett är mycket grundläggande sökningar inom en och samma tabell. Styrkan med relationsdatabaser kommer fram när vi börjar blanda in fler än en tabell i vår **SELECT**. Vi har flera tabeller i exemplet, så låt oss använda dem i kombination till något vettigt. Antag att vi vill ha en alfabetiskt ordnad lista över alla medlemmar i gruppen “Medieteknik”. För enkelhets skull antar vi till att börja med att vi redan vet att Medieteknik har gruppkoden “media”.

Detta är en fråga med två delfrågor:

1. Vilka användarnamn är medlemmar i gruppen "media"?
2. Vad heter dessa personer i verkliga livet?

Svaret på fråga 1 finns i tabellen **affiliation**, och svaret på fråga 2 finns i tabellen **staff**. Vi behöver nu inte ta detta i två steg och kolla något mellanresultat, utan vi kan formulera frågan som en så kallad “join query” i en enda sats enligt följande:

```
SELECT staff.firstname, staff.lastname
 FROM staff, affiliation
 WHERE affiliation.groupcode='media'
 AND staff.username=affiliation.username

+-----------+-------------+
| firstname | lastname    |
+-----------+-------------+
| Stefan    | Gustavson   |
| Martin    | Karlsson    |
| Wendy     | Testaburger |
+-----------+-------------+
```

För att ange vilket av de två möjliga fälten med namnet **username** man menar måste man ange relationens namn, en punkt, och därefter fältnamnet. För tydlighets skull kan man ange relationens namn för alla fältnamn, så som vi har gjort i exemplet ovan.

Observera att en fråga i SQL inte nödvändigtvis måste skrivas på en enda rad. Radbrytningar har faktiskt samma roll som mellanslag i språket. En fråga avgränsas på annat sätt, oftast genom att man bygger upp frågan som en sträng och sedan skickar hela frågan på en gång. Annars får man ange någon annan slutmarkör, som till exempel ett semikolon efter frågan. En sådan eventuell slutmarkör är inte en del av SQL-standarden och varierar mellan olika databashanterare.

Man kan också blanda in alla tre relationerna i exemplet i en och samma fråga. Om man av någon anledning vill ta reda på vilka som är medlemmar i en grupp vars namn börjar på “U” kan man skriva:

```
SELECT staff.firstname, staff.lastname, groups.groupname
 FROM staff, affiliation, groups
 WHERE groups.groupname LIKE 'U%'
 AND groups.groupcode = affiliation.groupcode
 AND affiliation.username = staff.username

+-----------+-------------+-------------+
| firstname | lastname    | groupname   |
+-----------+-------------+-------------+
| Stan      | Marsh       | Ultrateknik |
| Wendy     | Testaburger | Ultrateknik |
+-----------+-------------+-------------+
```

Ytterligare en aspekt av sådana här "join queries" skall nämnas, nämligen det faktum att en och samma relation kan förekomma flera gånger i frågan under olika namn. Om vi till exempel av någon här icke närmare definierad anledning vill ha en lista över alla personer som är medlemmar i samma grupp som Stefan Gustavson behöver vi besvara tre delfrågor med hjälp av information ur databasen, närmare bestämt:

1. Vilket "username" har Stefan Gustavson?
2. Vilka grupper, om några, är han medlem i?
3. Vad är användarnamnet för övriga personer som deltar i dessa projekt?
4. Vad heter dessa personer i verkligheten?

Detta är en fråga i fyra steg, och den kan naturligtvis skrivas som fyra separata frågor, där vi tar mellanresultaten från föregående delfråga som ledning när vi ställer nästa fråga. Om vi vill vara effektiva och göra allt automatiskt i en enda fråga kan vi emellertid baka ihop frågorna till ett "join query".

Problemet är att delfrågorna 1) och 4) samt även frågorna 2) och 3) i listan ovan hämtar data ur samma tabell, men för olika ändamål. Vi behöver använda vardera av tabellerna **`staff`** och **`affiliation`** två gånger i frågan, och till olika saker, och då behöver vi ge dem två olika namn eller *alias*. Ett aliasnamn tilldelas under **`FROM`** i **`SELECT`**-satsen. Detta illustreras bäst med ett exempel. Frågan vi beskrev ovan kan ställas så här i SQL:

```
SELECT staff2.firstname, staff2.lastname, affiliation2.groupcode
FROM staff=staff1, staff=staff2,
affiliation=affiliation1, affiliation=affiliation2
WHERE staff1.firstname ='Stefan' AND staff1.lastname='Gustavson'
AND affiliation1.username = staff1.username
AND affiliation2.groupcode = affiliation1.groupcode
AND staff2.username = affiliation2.username
```

```
+-----------+-------------+-----------+
| firstname | lastname    | groupcode |
+-----------+-------------+-----------+
| Stefan    | Gustavson   | media     |
| Martin    | Karlsson    | media     |
| Wendy     | Testaburger | media     |
| Stefan    | Gustavson   | hyper     |
| Eric      | Cartman     | hyper     |
+-----------+-------------+-----------+
```

Notera att Stefan kommer med två gånger i listan. Detta är faktiskt helt i sin ordning, eftersom han är med i två grupper och frågan ser ut som den gör. Om vi däremot hade uteslutit fältet **`groupcode`** från svaret skulle det ha sett litet förvirrande ut. Vi kan be att få slippa dubbla kopior av Stefan via kommandot **`SELECT DISTINCT`**:

```
SELECT DISTINCT staff2.firstname, staff2.lastname
FROM ... (i övrigt exakt samma som ovan)
```

```
+-----------+-------------+
| firstname | lastname    |
+-----------+-------------+
| Stefan    | Gustavson   |
| Martin    | Karlsson    |
| Wendy     | Testaburger |
| Eric      | Cartman     |
+-----------+-------------+
```

Om vi inte alls vill ha med Stefan själv i listan, och dessutom visa vad gruppkoderna betyder i klartext, så kan vi skriva om frågan med ytterligare ett villkor på slutet och blanda in vår tredje relation också:

```
SELECT staff2.firstname, staff2.lastname, groups.groupname
FROM staff=staff1, staff=staff2, affiliation=affiliation1, affilia-
tion=affiliation2, groups
WHERE staff1.firstname ='Stefan' AND staff1.lastname='Gustavson'
AND affiliation1.username = staff1.username
AND affiliation2.groupcode = affiliation1.groupcode
AND staff2.username = affiliation2.username
AND staff2.username <> staff1.username
AND groups.groupcode = affiliation2.groupcode
```

```
+-----------+-------------+-------------+
| firstname | lastname    | groupname   |
+-----------+-------------+-------------+
| Martin    | Karlsson    | Medieteknik |
| Wendy     | Testaburger | Medieteknik |
| Eric      | Cartman     | Hyperteknik |
+-----------+-------------+-------------+
```

Det kan tyckas patetiskt att man skall behöva skriva sådana här komplicerade frågor i SQL för att ta reda på vad som ändå är ganska enkel information. Det finns alternativa och kanske litet mer lättbegripliga sätt att skriva frågan ovan med så kallade “nested queries”, där man tydligare använder sig av ett eller flera mellanresultat. Frågan är faktiskt också litet konstruerat tillkrånglad. Det är dock inte helt ovanligt att man behöver konstruera ganska långa SQL-satser när man ger sig in i mer avancerad databasprogrammering. SQL är ett enkelt språk i det avseendet att det är lätt för *datorer* att tolka, men det är inte alltid så lätt för människor att skriva och läsa SQL. Styrkan hos SQL ligger inte i att språket är särskilt lättarbetat, utan i att det är extremt regelbundet, litet och effektivt, och att det är en standard som i stort sett alla databaser använder.

Avslutning

Det skulle föra litet för långt att här försöka gå in på exakt hur en “join query” som beskrevs ovan egentligen går till. Det är sådant man behöver ta ett bättre grepp om teorin bakom relationsdatabaser för att begripa sig på ordentligt. Intresserade läsare hänvisas till någon av de många böckerna om databaser och till separata kurser i ämnet.

Författaren till detta dokument och ovanstående exempel inser att någonstans kring förklaringen av alias och när vi började blanda in tre eller fler tabeller i sökningen går man lätt vilse om man är hänvisad till bara en intuitiv förståelse av mekanismerna bakom det satsen **`SELECT`** gör. Man kan dock trösta sig med att man faktiskt kan göra rätt mycket nyttigt med SQL utan att egentligen veta vad man håller på med. I många databasprogram finns det praktiska interaktiva verktyg för att bygga upp sökfrågor, och man kan i många fall slippa undan utan att skriva en enda rad SQL på egen hand. Det är dock alltid en fördel om man förstår vad man gör på ett mer övergripande plan. SQL går att lära sig snabbt från böcker och exempel. Om man däremot inte vet hur en databas är uppbyggd och ur den fungerar i stort kan man inte ens förstå att ett visst problem går utmärkt att lösa med en databas.

Det finns ganska många mer avancerade funktioner i SQL som inte nämnts ovan, men vilka finesserna är och hur de ser ut varierar en hel del mellan olika databashanterare, och de behövs i många fall inte för enklare tillämpningar. Därför slutar detta dokument här.

Stefan Gustavson (stegu@itn.liu.se) 2006-11-05