Databashantering

Jonas Björk

jonas@trinix.se

Databashantering

av Jonas Björk

Publicerad September 2003

Denna bok är anpassad för gymnasieskolornas kurs Databashantering med kurskod DTR1211 men kan naturligtvis användas av vem som helst som vill lära sig databashantering.

Boken arbetar utifrån MySQL version 3.23.55 i SuSE 8.2 Professional och är testad under MySQL på Microsoft Windows 2000.

Delar av materialet kommer från andra källor, se appendix B för komplett källförteckning.

<jonas@trinix.se> www.trinix.se

Revisions Historik;

Revision 0.2.1-rejas 2004-02-15 Reviderad av: mr
Lagt till ett kapitel om databasdesign
Revision 0.2.1 2003-09-23 Reviderad av: jb
Lagt till DELETE, ordnat datatyper så de ligger i tabeller, la in JOIN (ej klart)
Revision 0.2 2003-09-22 Reviderad av: jb
Ordnat lite fel i texten.
Revision 0.1 2003-09-16 Reviderad av: jb
Första versionen

Innehållsförteckning

Tabellförteckning

Exempelförteckning

Kapitel 1. Introduktion

Data är uppgifter av olika slag. Ibland skiljer man data från information, som är data som man gett en tolkning. Alltså är *23* ett exempel på data, medan det är information om vi vet att *det är 23 grader varmt ute*.

Med ordet *databas* brukar man mena: en samling data som hör ihop, som modellerar en del av världen och är persistent. Det vill säga data försvinner inte när man avslutar programmet eller stänger av datorn.

Om man skall förstå fördelarna med att använda databasteknik måste man jämföra med alternativet. Alternativet är för det mesta att ha en eller flera vanliga filer med data. Genom att skriva ett program som klarar av att hantera dessa filer kan vi lätt skapa en egen databas. Nackdelen är att en databas med över 100 poster blir lätt ohanterlig och därför använder vi en databashanterare (som i och för sig också är ett program).

Om vi till exempel skulle vilja skapa ett kundregister i programmeringsspråket C skulle det kunna se ut så här:

```
struct kund {
  int nummer;
  char namn[50];
  char adress[50];
  struct kund* nextp;
};
```

Sedan fortsätter vi med ungefär 2000 rader programkod, som sköter dialogen med användaren och som läser och skriver datafilen med kunder.

Det finns många fördelar med att i stället använda en databashanterare. De viktigaste fördelarna är: *enkelt*, *kraftfullt* och *flexibelt*.

Enkelt

Många databashanterare erbjuder ett textbaserat gränssnitt. Starta databashanteraren och skriv:

```
CREATE TABLE kund (nummer INT, namn CHAR(50), adress CHAR(50));
```

Sedan fyller vi tabellen med några poster:

```
INSERT INTO kund(nummer,namn,adress) VALUES (’1’,’jonas’,’helsingborg’);
INSERT INTO kund(nummer,namn,adress) VALUES (’2’,’stefan’,’klippan’);
INSERT INTO kund(nummer,namn,adress) VALUES (’3’,’lennart’,’bjuv’);
```

Nu har vi en tabell som ser ut så här:

```
+--------+---------+-------------+
| nummer | namn    | adress      |
+--------+---------+-------------+
|      1 | jonas   | helsingborg |
|      2 | stefan  | klippan     |
|      3 | lennart | bjuv        |
+--------+---------+-------------+
```

Den får du fram genom att skriva:

```
SELECT * FROM kund;
```

Uppgifterna i tabellen kallar vi för *data*. Tabellens utseende, det vill säga vilka kolumner som finns kallas för *schema*. Schemat bestämmer vilka data som kan lagras i databasen.

Kraftfullt

Att ett system är *kraftfullt* betyder att komplicerade saker kan göras på ett enkelt sätt.

Antag att vi vill ha reda på alla kunder som har namn som börjar med **j** och få dem utskrivna i bokstavsordning efter adressen. Då skriver vi så här:

```
SELECT * FROM kund WHERE namn LIKE ’j%’ ORDER BY address;
```

Vill du veta hur många kunder du har i Helsingborg? Enkelt:

```
SELECT address, COUNT(*) FROM kund WHERE adress=’helsingborg’ GROUP BY address;
```

Flexibelt

Att ett system är *flexibelt* betyder att det är lätt att ändra.

Kommer du på att du vill ha med dina kunders telefonnummer i databasen också? Det är enkelt att ordna:

```
ALTER TABLE kund ADD telefon char(10);
```

Nu har du lagt till en kolumn för telefonnummer i tabellen kund.

Att det går att ändra den logiska strukturen på datat så här, utan att man måste skriva om en massa program kallas *logiskt dataoberoende*.

När du använt kundregistret ett tag och det innehåller många kunder kommer det att ta lång att söka i databasen.

```
create index foo on kund(namn);
```

Nu har vi ändrat den fysiska lagringsstrukturen så att det går snabbare att söka efter ett visst namn. Tabellen ser fortfarande likadan ut, men sökningarna på namn går snabbare. Databashanteraren utnyttjar automatiskt den nya lagringsstrukturen.

Att man kan ändra på den fysiska lagringsstrukturen på datat, utan att man måste skriva om en massa program kallas *fysiskt dataoberoende*.

Andra fördelar

Det finns flera saker som är mycket besvärliga att få att fungera om man skriver ett program själv, men som finns inbyggda i de flesta databashanterare.

Om flera användare samtidigt håller på och ändrar i kundregistret är det lätt hänt att en användare skriver över en ändring en annan användare precis gjort. En databashanterare ser till att det inte blir några skadliga krockar.

Vad händer om strömmen går? Om ditt program läser in datafilen i primärminnet och skriver tillbaka filen när man jobbet klart så kommer du förlora de ändringar du gjort i filen. Ännu värre är att om du precis höll på att spara när strömmen gick, kanske en del av de data som finns på disken är de nya och en del är de gamla. Man vet aldrig vilka som är vilka.

Med en databas hanterare slipper man sådana problem. Den ser till att inga data någonsin försvinner, hur olyckligt ett strömavbrott än skulle komma.

I de flesta databashanterare kan man ge olika användare olika rättigheter i databasen för att skydda data mot obehörig åtkomst. Till exempel kan vi ge en användare rätt att ändra i vissa delar av databasen och söka i andra delar, medan hon inte alls får se andra delar av databasen.

Nackdelar med databaser

Databasteknik passar inte för alla tillämpningar. Exempelvis brukar all den där enkelheten och flexibiliteten som vi nämnt tidigare göra att en databashanterare kräver mycket mer resurser än ett specialskrivet program. Det går åt mer minne och diskutrymme, kanske går det också långsammare att köra.

Användningsområden

Allt. Tro inte att det bara är kundregister, videoregister och cdregister man kan använda databaser till. Databaser används också i CAD-system, telefonväxlar och mycket annat.

Kapitel 2. Datamodeller

Datamodellen man använder bestämmer hur schemat får se ut. Schemat bestämmer i sin tur vilka data som skall lagras i databasen.

Man brukar dela in datamodellerna i tre klasser:

- *Konceptuella* eller *begreppsmässiga* datamodeller. Om man skall skapa en databas som beskriver en del av verkligheten, till exempel ett företag, brukar man börja med att göra en beskrivning av hur den delen av verkligheten ser ut och fungerar. Till det kan man använda en konceptuell datamodell.
- *Implementationsmodeller* är de datamodeller som används i databashanterare. Om man vill skapa en databas måste den beskrivning man gjort med hjälp av en konceptuell datamodell översättas till något som går att använda i en dator -- en beskrivning enligt en implementationsmodell.

  De vanligaste implementationsmodellerna är *relationsmodellen* och olika *objektorienterade modeller*.

- *Fysiska datamodeller* som används för att beskriva hur data lagras fysiskt.

Tre-schema-arkitekturen

Det kan vara praktiskt att betrakta sin databas på tre olika nivåer. Det är hela tiden samma data, men man använder tre olika scheman för att beskriva dem:

- Den *externa* nivån beskriver hur användaren ser databasen. Detta beskrivs av det *externa schemat*.
- Den *logiska* nivån beskriver hela databasen, utrryckt i den implementationsmodell som databashanteraren använder. I en relationsdatabas består den logiska nivån av alla tabellerna i databasen. Detta beskrivs av det *logiska schemat*.
- Den *interna* eller *fysiska* nivån beskriver hur datat är lagrat. Som användare märker man nästan aldrig av denna nivå. Den fysiska nivån beskrivs av det *fysiska schemat*.

Logiskt- och fysiskt dataoberoende

- *Logiskt dataoberoende* innebär att man kan ändra i det logiska schemat, utan att det externa schemat påverkas.
- *Fysiskt dataoberoende* innebär att man kan ändra i det fysiska schemat, utan att det logiska schemat påverkas.

Olika typer av användare

Ibland skiljer man på olika typer av användare, det vill säga personer som arbetar med databasen:

- *Expertanvändaren* använder databasen ofta, kanske dagligen. De är väl insatta i databasens uppbyggnad och databashanterarens funktion och sitter kanske och skriver SQL-frågor.
- *Tillfälliga användare* kan inte lika mycket om databasen, kanske för att de använder den mer sällan. De behöver enklare och mer nybörjarvänliga verktyg än experterna. Man kan inte kräva att biljettförsäljare på SJ skall lära sig SQL.
- *Databasadministratören (DBA)* ansvarar för databasens drift. För stora och viktiga databaser finns det ofta en hel grupp databasadministratörer som utformar och förändrar databasschemat, registerar nya användare och ser till att databasen fungerar bra och effektivt.
- *Andra datorprogram*. Det är inte ovanligt att en del av databasens användare är andra datorprogram som hämtar data från databasen och skickar data till den. En del databaser har inga mänskliga användare alls, utan det är ett eller flera datorprogram som använder sig av dem. Kanske kommunicerar de datorprogrammen sen i sin tur med människor, men inte ens det är säkert.

Om jag samlar mina kakrecept i en databas i min hemdator är det antagligen jag själv som spelar alla dessa roller. I stora system, som en stor biljettbokningsdatabas kan det finnas hundratals eller tusentals personer som jobbar med databasen samtidigt.

Kapitel 3. Databashanterare

En databashanterare är ofta ett stort och komplicerat program, eller ett helt system av program. Förutom själva *kärnan* i databashanteraren, som hanterar den lagrade databasen, finns det ofta olika användargränssnitt. Dessa använder användaren för att söka eller ändra i databasen. Det brukar finnas ett eller flera frågespråk som SQL, men också grafiska verktyg. Olika gränssnitt passar för olika användare och användningsområden.

Det är också vanligt att en databashanterare innehåller verktyg för att skapa applikationer. En applikation är ett program som är avsett för ett specifikt ändamål, till exempel för SJ’s biljettbokning. I det sammanhanget lagras data i en databas som hanteras av en databashanterare och applikationen kommunicerar med databashanteraren. Applikationen, där användaren kan klicka på knappar för att välja resmål är enklare för användaren att hantera än databashanteraren.

När man installerat en databashanterare på sin dator kan den ofta hantera flera olika databaser samtidigt. Varje databas består egentligen av två samlingar data: dels databasens innehåll, dels schemat. Schemat kallas också *datakatalog* eller *meta-data* vilket betyder *data om data*.

Kapitel 4. Begrepp

Databas

Datorer används ofta för att hantera register av olika slag, sådana register kallas databaser. Folkbokföringen och telefonkatalogen är exempel på stora databaser som innehåller mycket information. Idrottsföreningens medlemsregister, din cd- eller dvdsamling är exempel på mindre databaser.

För att kunna hantera sådana register i datorn behöver du en *databashanterare*. Vanliga databashanterare i persondatormiljö är Claris FileMaker, Microsoft Access och Borlands dBase. För större register använder man databasservrar (SQL-server), som till exempel IBM DB2, MySQL, Microsoft SQL och Oracle.

Tabell

Databasen byggs upp av tabeller, dessa innehåller posterna där du lagrar informationen.

Post

Varje objekt i databasen ("rad i tabellen") kallas för *post*, posten innehåller i sin tur fälten.

Fält

Fältet är den minsta beståndsdelen i databasen, den innehåller informationen som du vill lagra. Varje enskilt fält har också *attribut* som beskriver vilken typ av data som får lagras i fältet.

Fråga

Frågor du ställer till databaser gör du i ett språk som kallas *Structured Query Language* (SQL). Frågan kan hämta information, lagra information eller manipulera information i databasen beroende på hur frågan ställs.

Nycklar

Varje post i tabellen skall ha ett fält som innehåller ett unikt värde så att man kan särskilja olika poster från varandra. En tabell som innehåller kunder kan ha ett fält för kundnummer, medan en tabell för beställningar kan innehålla ett fält för beställningsnummer.

Det fält som man använder för att särskilja de olika posterna från varandra i en tabell kallas för primärnyckel. Man använder primärnyckeln för att hitta en viss post, utan en primärnyckel blir det mycket svårt att uppdatera eller ta bort specifika poster i en tabell.

Metadata

Metadata är data om data, det vill säga data som beskriver en annan data. Med beskrivning avses datats egenskaper (datatyp, storlek och så vidare), datastruktur och regler eller begränsningar man ställer upp. Metadata kallas också databasschema, systemkatalog eller *data dictionary*. Nyttan med ett separat schema för beskrivningen ger program-data oberoende en stor fördel i dagens programutveckling.

Kapitel 5. Relationsdatabaser

Relationsmodellen är en av flera datamodeller, det vill säga sätt att organisera data. Relationsmodellen är den helt dominerande datamodell i dagens databashanterare och går, enkelt uttryckt, ut på att man lagrar data i tabeller.

Relationer

Relationsmodellen går ut på att data lagras som *relationer*. En relation är samma sak som en *tabell* med *poster* (rader) och namngivna *fält* (kolumner). Egentligen heter posterna *tupler* och fälten *attribut*, men vi behöver inte göra det svårare än så här.

Titta på tabell 5-1, som innehåller data om medlemmarna i en förening.

Tabell 5-1. Medlem

Medlemsnummer	Namn	Telefonnummer
1	Jonas	112233
2	Stefan	123344
3	Lennart	213232
4	Linus	200305

Varje post innehåller data om en medlem i föreningen och varje fält anger en viss egenskap som medlemen har.

Posterna i en tabell utgör en *mängd*. Det betyder att posterna inte har någon speciell ordning, utan kan skrivas i vilken ordning som helst. Det kan inte heller förekomma *dubletter*, det vill säga poster med samma data som en annan post i alla fält.

Ibland definerar man en eller flera *nycklar*. En nyckel är ett fält, eller en kombination av flera fält, vars värden är unika. Om man har en nyckel i en relation så kan det inte finnas flera poster med samma värde i det fältet. Nyckeln används därför för att skapa *unikhet* hos posterna. I tabell 5-1 skulle vi kunna sätta fältet **Medlemsnummer** som nyckel eftersom att vi vet att det bara finns en medlem med ett specifikt medlemsnummer. Fälten **Namn** och **Telefonnummer** kan vi inte använda som nycklar, eftersom vi vet att namn på personer inte är unika och flera medlemmar kan bo på samma adress och ha samma telefonnummer.

Relationen innehåller bara *enkla* och *atomära* värden. Att värdena är enkla betyder att vi inte kan ha

mer än ett enda värde per fält. Om en medlem i tabell 5-1 har två olika telefonnummer kan vi inte ange båda i samma post. Vi måste använda två poster för att kunna lagra båda telefonnumren (eller skriva om vårt schema). Att värdena är atomära betyder att vi bara arbetar med hela datat och inte delar av det. Att relationen har enkla och atomära värden kallas *första normalformen*.

Det finns också något som kallas *nullvärden*. Om en person inte har någon telefon, eller om vi inte vet telefonnumret, kan vi ange värdet **null** i det fältet. Null är inte samma sak som noll (0). Om till exempel en temperaturangivelse satts till null betyder det att det inte finns någon temperatur och det är ju inte samma sak som en temperatur på noll (0) grader.

Schema och innehåll

I databassammanhang brukar man skilja på *schemat*, som beskriver vad som kan finnas i databasen, och det *innehåll* som finns i databasen.

Därför talar man om *relationens schema* och *relationens innehåll*. I schemat ingår bland annat vilka fält som relationen har, deras *domäner* (vilka värden de kan innehålla), och vilka nycklar som finns.

Olika sorters nycklar

Vi börjar med att kalla ett fält, eller en kombination av fält, vars värden garanterat är unika för en *supernyckel*. En supernyckel kan innehålla onödigt många fält. I relationen Medlem (tabell 5-1) finns det fyra supernycklar:

- Medlemsnummer
- Medlemsnummer + Namn
- Medlemsnummer + Telefonnummer
- Medlemsnummer + Namn + Telefonnummer

En *kandidatnyckel* är en minimal supernyckel, det vill säga en supernyckel där man *inte kan ta bort några fält* om den fortfarande skall vara garanterat unik. I relationen Medlem finns bara en kandidatnyckel, nämligen fältet Medlemsnummer (men en kandidatnyckel kan vara sammansatt av flera fält, om alla behövs för att den skall bli unik).

Det finns alltid minst en supernyckel och minst en kandidatnyckel i varje relation. Samtliga fält tillsammans utgör alltid en supernyckel, för det kan inte finnas två poster med samma värden i alla fält. Alltså är kombinationen av alla fälten garanterat unik - och därför en supernyckel.

Kandidatnycklarna heter kandidatnycklar eftersom att det är bland dessa kandidater som vi väljer en *primärnyckel*. Vi väljer alltid en primärnyckel i varje relation och det är primärnyckeln som oftast används för att identifiera poster i tabellen.

De övriga kandidatnycklarna, som inte valdes som primärnyckel, kallas *alternativnycklar* eller *sekundärnycklar*.

Kopplingar mellan relationer

Låt oss nu utöka vår databas, som än så länge bara innehåller relationen Medlem (tabell 5-1), med ytterligare två relationer. Först skapar vi relationen **Sektion** (tabell 5-2) som innehåller data om olika sektioner inom föreningen.

Tabell 5-2. Sektion

Sektionskod	Namn	Ledare
A	Programmering	3
B	Webbdesign	2
C	Nätverk	4

Vi antar att **Sektionskod** är primärnyckel, med **Namn** som alternativnyckel.

Fältet **Ledare** är en *främmande nyckel*, även kallad *referensattribut*, till relationen **Medlem**. En främmande nyckel refererar alltid till primärnyckeln i en annan (eller samma) relation. Vi ser till exempel att ledaren för Programmeringsektionen är medlem nummer 3. Alltså går vi till relationen **Medlem**, letar rätt på medlem nummer 3 och ser att det är **Lennart**.

Om det står att medlem nummer 3 leder en sektion, så måste det också finnas en medlem nummer 3 i medlemstabellen. Detta villkor kallas *referensintegritet*.

Nu skapar vi relationen **Deltar** (tabell 5-3), som anger vilka medlemmar som deltar i vilka sektioner.

Tabell 5-3. Deltar

Medlem	Sektion
1	A
1	B

Medlem	Sektion
1	C
2	B
3	A
3	C
4	A

Medlem är främmande nyckel till relationen **Medlem** och **Sektion** är främmande nyckel till relationen **Sektion**. **Medlem** och **Sektion** utgör tillsammans den enda kandidatnyckeln och blir därför automatiskt primärnyckel.

Kapitel 6. Normalisering

Att normalisera en databas betyder att man konstruerar databasens schema efter vissa regler. Normaliserar gör man alltså medan kan konstruerar databasen och inte i efterhand. Normaliseringen har egentligen ett övergripande mål, att eliminera redundant data. Med redundant menas sådant som är dubbellagrat, det vill säga finns på fler än ett ställe.

Man kan normalisera sin databas i flera steg. Dessa steg har namn. Det första steget kallas *första normalformen* som förkortas *1NF*, efter det så kommer andra och tredje normalformen. Det finns fler normalformer men dessa är de viktigaste.

Första normalformen, 1NF

En tabell är normaliserad enligt första normalformen om:

1. Det inte finns några fält som är likadana.
2. Alla fält i varje post bara har ett värde.
3. Alla poster i samma fält har samma datatyp.

Vi tittar på tabellen från kapitel 1:

```
+--------+---------+-------------+
| nummer | namn    | adress      |
+--------+---------+-------------+
|      1 | jonas   | helsingborg |
|      2 | stefan  | klippan     |
|      3 | lennart | bjuv        |
+--------+---------+-------------+
```

Alla poster (rader) är olika så det är okej enligt 1NF. Alla fält i varje post har bara ett värde, också okej enligt 1NF. Alla poster i samma fält (kolumn) har samma datatyp (int, text, text=), också okej enligt 1NF. Vår tabell är alltså i 1NF.

Vi säger att Stefan bor på två ställen. Han kanske bor i klippan och har en sportstuga i Vemdalen. Ett sätt att göra det är att ändra tabellen så att den ser ut så här:

```
+--------+---------+-------------------+
| nummer | namn    | adress            |
+--------+---------+-------------------+
|      1 | jonas   | helsingborg       |
```

```
|      2 | stefan  | klippan, vemdalen |
|      3 | lennart | bjuv              |
+--------+---------+-------------------+
```

Att göra så gör att databasen inte är i första normalformen, vi bryter ju mot punkt 2 ovan, och det är inte bra. Det är i de flesta fall inte ens möjligt. För att databasen skall vara i 1NF kan man ändra den så att den ser ut så här:

```
+--------+---------+-------------+
| nummer | namn    | adress      |
+--------+---------+-------------+
|      1 | jonas   | helsingborg |
|      2 | stefan  | klippan     |
|      2 | stefan  | vemdalen    |
|      3 | lennart | bjuv        |
+--------+---------+-------------+
```

Nu bor stefan på två ställen och vår tabell är fortfarande i 1NF. Skulle det, vilket det förmodligen gör, bo fler än en Jonas i Helsningborg, så är det bra att vi har det första fältet, nummer. Annars skulle vi inte kunna lägga till en till Jonas i Helsingborg utan att bryta mot punkt 1 ovan.

Andra normalformen, 2NF

För att en tabell skall var i 2NF krävs att den skall vara i 1NF och att alla fält, som inte är nycklar, skall vara direkt relaterade till nyckeln. Nyckeln är det fält som gör varje post unik, i vårt fall "nummer", som är en identifikation för en person som vi lagrar data om.

Vad innebär då 2NF? För att illustrera det skall vi lägga till information om vilka bilar de olika personerna i vår databas har. Vi vill veta vilket bilmärke de har och vilken årsmodell deras bil har. Vi bygger ut tabellen så här:

```
+--------+---------+-------------+---------+---------+
| nummer | namn    | adress      | bil     | arsmod  |
+--------+---------+-------------+---------+---------+
|      1 | jonas   | helsingborg | saab    | 1979    |
|      2 | stefan  | klippan     | porsche | 2003    |
|      3 | lennart | bjuv        | ferrari | 2004    |
+--------+---------+-------------+---------+---------+
```

Denna tabell är forfarande i 1NF men inte i 2NF. Anledningen är att bilens marke och framför allt bilens årsmodell, inte är direkt relaterade till nyckeln som ju identifierar en person. Man kan lösa detta genom att flytta fälten "bil" och "arsmod" till en egen tabell med en egen nyckel. Denna tabell kan man sedan relatera till vår persontabell på följande sätt:

```
[person]
```

```
+--------+---------+-------------+-----+
| nummer | namn    | adress      | bil |
+--------+---------+-------------+-----+
|      1 | jonas   | helsingborg | 1   |
|      2 | stefan  | klippan     | 2   |
|      3 | lennart | bjuv        | 3   |
+--------+---------+-------------+-----+

[bil]
+--------+---------+---------+---------+
| nummer | bil     | modell  | arsmod  |
+--------+---------+---------+---------+
|      1 | saab    | 99      | 1979    |
|      2 | porsche | 911     | 2003    |
|      3 | ferrari | 365GT   | 1969    |
+--------+---------+---------+---------+
```

Nu är tabellen i 2NF och vi har dessutom lagt till mer information om bilarna. Du kanske inte tycker att fältet "bil" i den första tabellen är riktigt enlig 2NF eftersom den inte har med person att göra. Men den har med personen att göra, det är ju personens bil. Däremot så är ju bilens årsmodell ett attribut som hör mer hemma hos bilen än personen, så det hör inte hemma i persontabellen och får inte göra det heller enligt 2NF.

Tredje normalformen, 3NF

Tredje normalformen når man om en tabell är i 2NF och det inte finns några transitiva beroenden. Det är ett fint sätt att säga att inga fält, som inte är nycklar, skall kunna härledas av varandra. Vi säger att vi skall utöka informationen i vår persondatabas med adresser till personerna. Den kan då se up så här:

```
[person]
+--------+---------+---------------+-------------+-------------+-----+
| nummer | namn    | postadress    | postnummer  | postort     | bil |
+--------+---------+---------------+-------------+-------------+-----+
|      1 | jonas   | tabellgatan 2 | 112 12      | helsingborg | 1   |
|      2 | stefan  | postvägen 16  | 322 13      | klippan     | 2   |
|      3 | lennart | fältstigen    | 561 25      | bjuv        | 3   |
+--------+---------+---------------+-------------+-------------+-----+

[bil]
+--------+---------+---------+---------+
| nummer | bil     | modell  | arsmod  |
+--------+---------+---------+---------+
|      1 | saab    | 99      | 1979    |
|      2 | porsche | 911     | 2003    |
```

```
|      3 | ferrari | 365GT   | 1969    |
+--------+---------+---------+---------+
```

Vi är fortfarande i 2NF, men eftersom postorten är helt beroende av postnummret så är vi inte i tredje normalformen (3NF). För att tabellen skall vara i 3NF måste vi flytta postorten till en egen tabell. Resultatet som är i 3NF kan se ut så här:

```
[person]
+--------+---------+---------------+-------------+-----+
| nummer | namn    | postadress    | postnummer  | bil |
+--------+---------+---------------+-------------+-----+
|      1 | jonas   | tabellgatan 2 | 112 12      | 1   |
|      2 | stefan  | postvägen 16  | 322 13      | 2   |
|      3 | lennart | fältstigen    | 561 25      | 3   |
+--------+---------+---------------+-------------+-----+

[bil]
+--------+---------+---------+---------+
| nummer | bil     | modell  | arsmod  |
+--------+---------+---------+---------+
|      1 | saab    | 99      | 1979    |
|      2 | porsche | 911     | 2003    |
|      3 | ferrari | 365GT   | 1969    |
+--------+---------+---------+---------+

[postort]
+-------------+-------------+
| postnummer  | postort     |
+-------------+-------------+
| 112 12      | helsingborg |
| 322 13      | klippan     |
| 561 25      | bjuv        |
+-------------+-------------+
```

Ibland kan det vara lämligt att bryta mot 3NF till exempel för att göra databasdesignen tydligare. Men i de allra flesta fall skall man hålla sin databas enligt tredje normalformen.

Kapitel 7. Reserverade ord

Det är viktigt att känna till de reserverade orden som används av SQL eftersom att de inte får användas för att namnge tabeller eller poster.

Lista över reserverade ord

add, all, alter, analyse, and, as, asc, asensitive, auto_increment

bdb, before, berkeleydb, between, bigint, binary, blob, both, btree, by

call, cascade, case, change, char, character, check, collate, column, columns, connection, constraint, create, cross, current_date, current_time, current_timestamp, cursor

database, databases, day_hour, day_minute, day_second, dec, decimal, declare, default, delayed, delete, desc, describe, distinct, distinctrow, div, double, drop

else, elseif, enclosed, errors, escaped, exists, explain

false, fields, float, for, force, foreign, from, fulltext

grant, group

hash, having, high_priority, hour_minute, hour_second

if, ignore, in, index, infile, inner, innodb, inout, insensitive, insert, int, integer, interval, into, io_thread, is, iterate

join

key, keys, kill

leading, leave, left, like, limit, lines, load, localtime, localtimestamp, lock, long, longblob, longtext, loop, low_priority

master_server_id, match, mediumblob, mediumint, mediumtext, middleint, minute_second, mod, mrg_myisam

natural, not, no_write_to_binlog, null, numeric

on, optimize, option, optinally, or, order, out, outer, outfile

precision, primary, privileges, procedure, purge

read, real, references, regexp, rename, repeat, replace, require, restrict, return, returns, revoke, right, rlike, rtree

select, sensitive, separator, set, show, smallint, some, soname, spatial, specific, sql_big_result, sql_calc_found_rows, sql_small_result, ssl, starting, straight_join, striped

table, tables, terminated, then, tinyblob, tinyint, tinytext, to, trailing, true, types

union, unique, unlock, unsigned, until, update, usage, use, user_resources, using

values, varbinary, varchar, varcharacter, varying

warnings, when, where, while, with, write

xor

year_month

zerofill

Satsbyggnad

När du jobbar mot en SQL-server skriver du kommandon, dessa kommandon måste följa en strikt syntax annars förstår inte SQL-servern vad du vill. Normalt skriver du kommandot du vill att SQL-servern skall utföra och avslutar med ett semikolon (;) följt av enter.

Syntaxen för select ser ut så här:

```
SELECT [STRAIGHT_JOIN] [SQL_SMALL_RESULT] [SQL_BIG_RESULT] [SQL_BUFFER_RESULT]
[SQL_CACHE | SQL_NO_CACHE] [SQL_CALC_FOUND_ROWS] [HIGH_PRIORITY]
[DISTINCT | DISTINCTROW | ALL] utrycket, ... [INTO (OUTFILE | DUMPFILE)
  'filnamn' export_val] [FROM tabell [WHERE definition] [GROUP BY (unsigned
    int | fältnamn | formel) [ASC | DESC], ... [WITH ROLLUP]] [HAVING
```

```
definition] [ORDER BY (unsigned int | fältnamn | formel) [ASC | DESC],
...] [LIMIT [offset,] rader | rader OFFSET offset] [PROCEDURE
procedure_namn(argument lista)] [FOR UPDATE | LOCK IN SHARE MODE]]
```

Inte så svårt, eller hur?

Du kan omöjligt lära dig alla kommandonas syntax utantill, men du skall förstå vad syntaxen är och vart du hittar den. På MySQL's hemsida finns referenser till alla SQL-kommandonas syntax. Leta upp syntaxen för insert och update och titta igenom dem.

Nu använder vi vår syntax för att bygga ett kommando:

```
select * from folkbokforing where ort='helsingborg' and alder > 20 limit 10;
```

Vad tror du ovanstående select gör?

Inbyggda funktioner

I SQL finns en massa olika funktioner som du kan använda, vi går igenom en del av dessa. Denna förteckning är inte alls komplett, men tar upp de vanligaste funktionerna, som du faktiskt kan ha nytta av.

Datum och tidsfunktioner

now() returnerar systemets aktuella datum och tid.

```
select now();
```

Vill du bara ha systemets datum:

```
select curdate();
```

Eller bara systemets tid:

```
select curtime();
```

Vill du veta vilken veckodag ett speciellt datum var?

```
select dayname('1999-12-31');
```

Hur många timmar, minuter och sekunder är 130389 sekunder?

```
select sec_to_time('130389');
```

Matematiska funktioner

pi() returnerar värdet av pi.

```
select pi();
```

round(n, d) avrundar n, och ger det d antal decimaler.

```
select round(12.12945, 2);
```

Eller varför inte avrunda pi till två decimaler?

```
select round(pi(), 2);
```

Med bin(n) kan du konvertera decimala tal till binära:

```
select bin(214);
```

Med hex(n) kan du konvertera decimala tal till hexadecimala:

```
select hex(214);
```

Strängfunktioner

ascii(c) returnerar asciivärdet för tecknet c. char(n) returnerar tecknet när du ger den ett asciivärde.

```
select ascii('z');
select char(122);
```

concat(s1, s2, ...) sätter ihop flera strängar till en.

```
select concat('hello ', 'world');
```

lcase(s) converterar strängen s till gemener.

```
select lcase('HELLO WORLD');
```

length(s) returnerar storleken på strängen s.

```
select length('HELLO WORLD');
```

like används tillsammans med where för att ange jokertecken. I exemplet nedanför kommer vi få tillbaka alla poster i tabellen tabell där fältet adress börjar med bokstaven h.

```
select * from tabell where adress like 'h%';
```

lpad(s,n,c) används för att klippa en sträng *s* till *n* antal tecken, om strängen är kortare än *n* så fylls den upp med tecknet *c* från vänster.

```
select lpad('data', 10, '-');
```

ltrim(s) tar bort blanktecken till vänster om strängen *s*.

```
select ltrim('          hello');
```

reverse(s) kastar om strängen *s* så att den läses baklänges.

```
select reverse('dallas');
```

rpad(s,n,c) fungerar precis som lpad med den skillnad att den arbetar från höger.

```
select rpad('data', 10, '-');
```

rtrim(s) fungerar som ltrim, men arbetar från höger.

```
select rtrim('hello         ');
```

ucase(s) konverterar strängen *s* till versaler.

```
select ucase('hello world');
```

övriga funktioner

database() returnerar namnet på den databas som du jobbar med för tillfället.

```
select database();
```

encrypt(s, salt) krypterar strängen *s* med valfri *salt*. Om du använder samma sträng, men byter ut salt får du olika krypterade strängar. Säg att vi vill spara ett lösenord för användare i en databas. Vi använder deras användarnamn som sträng och deras lösenord som salt. I databasen lagrar vi användarnamnet i klartext och den krypterade strängen vi får från encrypt. För att verifiera sig måste användaren ange rätt användarnamn och lösenord och vår applikation använder dessa för att skapa en ny krypterad sträng, om vi får samma krypterade sträng har användaren angett rätt lösenord.

Fördel? Lösenordet lagras inte i klartext i databasen och det går inte att få fram det med hjälp av användarnamnet och den lagrade krypterade strängen.

```
select encrypt('data','world');
```

md5(s) returnerar en md5summa för strängen *s*. md5 är ett sätt att beräkna kontrollsummor på data. Med hjälp av md5-funktionen kan vi kontrollera integritet i databasen, att inte data som lagras förändrats på något otillåtet sätt.

```
select md5('data');
```

password(s) returnerar en krypterad sträng av strängen *s*. Kan användas på samma sätt som encrypt.

```
select password('data');
```

version() returnerar vilken version av MySQL du använder.

```
select version();
```

count(x) returnerar hur många poster det finns i tabellen i fält *x*.

```
select count(*) from elev;
select count(*) as elever from elev;
```

count(distinct x) returnerar antalet unika poster det finns i tabellen i fält *x*.

```
select count(distinct namn) from elev;
```

avg(x) returnerar medelvärdet från fältet *x* i tabellen.

```
select avg(alder) from elev;
```

min(x) returnerar det minsta värdet från fältet *x* i tabellen.

```
select min(alder) from elev;
```

max(x) returnerar det högsta värdet från fältet *x* i tabellen.

```
select max(alder) from elev;
```

sum(x) returnerar summan av alla värden i fältet *x* i tabellen.

```
select sum(alder) from elev;
```

Kapitel 8. Använda databasservern

I detta kapitel går vi igenom de vanligaste kommandona i SQL. Jag använder MySQL och kommandona kan skilja sig lite åt om du använder en annan SQL-server.

MySQL utvecklas av MySQL AB som är ett svenskt företag och de erbjuder MySQL fritt under LGPL-licensen. Klart värt att titta på om du vill lära dig SQL. MySQL finns tillgänglig till nästan alla plattformer, varav Linux, FreeBSD och Microsoft Windows är några.

Databashanteraren för MySQL heter **mysql** och du ansluter till din MySQL-server genom att skriva:

```
mysql -u jonas -p
```

Där *jonas* är användarnamnet du har i databasservern och *-p* betyder att du vill ange ett lösenord.

Nu kan du visa alla databaser som finns i din server genom att skriva:

```
SHOW DATABASES;
+------------+
| Database   |
+------------+
| mysql      |
| test       |
+------------+
```

Skapa databasen (CREATE DATABASE)

För att kunna hämta data måste vi ha data att hämta, så vi skapar en databas i vår MySQL-server:

```
CREATE DATABASE medlemsreg;
Query OK, 1 row affected (0.00 sec)
```

Du skall få svaret *Query OK*, annars har något gått fel. Se efter i felmeddelandet vad som kan vara fel, ofta får man fel för att man saknar rättigheter att skapa databaser eller att man stavar kommandon fel.

Innan vi fortsätter med att arbeta mot vår nya databas måste vi berätta för databashanteraren att vi vill använda databasen medlemsreg som vi precis skapade. För att arbeta mot din nya databas skriver du:

```
USE medlemsreg;
Database changed
```

Skapa en tabell (CREATE TABLE)

Innan vi börjar lagra data i vår databas skall vi skapa ett databasschema. Det gör vi med kommandot CREATE TABLE. När vi skapar ett databasschema anger vi vilka fält vi vill ha och vilken typ av data fälten skall innehålla. Nu kan vi skapa en tabell i vår nya databas.

```
CREATE TABLE medlem (id INT AUTO_INCREMENT PRIMARY KEY, fnamn CHAR(50), enamn CHAR(50), a
Query OK, 0 rows affected (0.05 sec)
```

Här skapar vi tabellen *medlem* som kommer att ha fälten id, fnamn, enamn, address, postnr, ort och telefon. Datatyperna vi anger är *INT* som är ett heltal och *CHAR* som är ett tecken. Till char anger vi också hur många tecken som skall kunna lagras, det gör vi inom paranteserna.

Till fältet *id* sätter vi också egenskaperna *AUTO_INCREMENT* och *PRIMARY KEY*. De egenskaperna gör att fältet automatiskt räknas upp för varje post som skapas och att fältet är primärnyckel.

Fler datatyper hittar du i appendix A i slutet av denna bok.

Lagra data (INSERT)

Med kommandot **INSERT** kan vi lagra data i vår tabell. Vi lägger in lite medlemmar i vårt register.

```
INSERT INTO medlem(id, fnamn, enamn, adress, postnr, ort, telefon) VALUES
(”,’Kalle’,’Svensson’,’Storgatan 42’,’12345’,’Storstad’,’011-121212’);
Query OK, 1 row affected (0.07 sec)
```

Här berättar vi för SQL-servern att vi vill lägga till data i tabellen medlem och i fälten id, fnamn, enamn, adress, postnr, ort och telefon. VALUES anger vilka värden vi vill lagra i dessa fält. Att vi lägger in tomma data (”) i fältet id beror på att den räknas upp automatiskt av databashanteraren (fältet har egenskapen AUTO_INCREMENT).

Vi kan välja att inte ange fälten som datat skall lagras i och skriva så här:

```
INSERT INTO medlem VALUES (”,’Kalle’,’Svensson’,’Storgatan 42’,’12345’,’Storstad’,’011-12
Query OK, 1 row affected (0.00 sec)
```

Nackdelen med att inte ange fältnamn är att om vi ändrar databasschemat får vi problem med frågorna. Data kommer att lagras i fel fält.

Ibland vill vi bara lagra data i ett par av fälten, då skriver vi så här:

```
INSERT INTO medlem(fnamn, enamn, ort) VALUES (’Kurt’,’Bengtsson’,’Örkelljunga’);
Query OK, 1 row affected (0.00 sec)
```

Skapa nu ett par fiktiva medlemar i din databas.

```
INSERT INTO medlem(id, fnamn, enamn, adress, postnr, ort, telefon) VALUES
(”,’Johan’,’Andersson’,’Lillåstigen 3’,’23423’,’Lillby’,’012-123344’);
Query OK, 1 row affected (0.00 sec)
INSERT INTO medlem(id, fnamn, enamn, adress, postnr, ort, telefon) VALUES
(”,’Jill’,’Jonsson’,’Sommarvägen 72’,’54333’,’österberg’,’043-156789’);
Query OK, 1 row affected (0.00 sec)
```

Nu kan du titta på dina data genom att skriva:

```
SELECT * FROM medlem;
+----+-------+-----------+----------------+--------+-------------+------------+
| id | fnamn | enamn     | address        | postnr | ort         | telefon    |
+----+-------+-----------+----------------+--------+-------------+------------+
|  1 | Kalle | Svensson  | Storgatan 42   |  12345 | Storstad    | 011-121212 |
```

```
|  2 | Kurt  | Bengtsson | NULL           |   NULL | Örkelljunga | NULL       |
|  3 | Johan | Andersson | Lillåstigen 3  |  23423 | Lillby      | 012-123344 |
|  4 | Jill  | Jonsson   | Sommarvägen 72 |  54333 | Österberg   | 043-156789 |
+----+-------+-----------+----------------+--------+-------------+------------+
4 rows in set (0.46 sec)
```

Vi använder en stjärna (*) som fältnamn i selectsatsen. Stjärnan betyder *alla fält*. Som du ser ovanför har raden med id 2 tre fält som har värdet NULL. NULL innebär att det inte finns någon data lagrad i fältet.

Tabellen avgifter

Nu har vi ett adressregister över medlemmar i en förening. Kanske är det en bra ide att ha ett register över vilka som betalat medlemsavgiften också? För detta får vi skapa en ny tabell. Skapa tabellen **avgifter**:

```
CREATE TABLE avgifter (medlem INT, avgift INT, datum DATE);
Query OK, 0 rows affected (0.06 sec)
```

Tabellen avgifter innehåller tre fält: medlem som är av datatypen INT, avgift som är av datatypen INT och fältet datum som är av datatypen DATE. Tanken är att fältet medlem skall kopplas mot fältet id i tabellen medlem.

Sedan lägger vi in medlemsavgifterna för medlemmarna, Kalle har betalat in 200 kronor och hans medlemsskap upphör den 29 maj 2003, Kurt’s medlemsskap upphör den 19 juni 2003, Johan’s medlemsskap upphör den 2 augusti 2003 och Jill’s medlemsskap upphör den 2 december 2003.

```
INSERT INTO avgifter(medlem, avgift, datum) VALUES (’1’,’200’,’2003-05-29’);
Query OK, 1 row affected (0.00 sec)
INSERT INTO avgifter(medlem, avgift, datum) VALUES (’2’,’200’,’2003-06-19’);
Query OK, 1 row affected (0.00 sec)
INSERT INTO avgifter(medlem, avgift, datum) VALUES (’3’,’200’,’2003-08-02’);
Query OK, 1 row affected (0.00 sec)
INSERT INTO avgifter(medlem, avgift, datum) VALUES (’4’,’200’,’2003-12-02’);
Query OK, 1 row affected (0.00 sec)
```

Ställa frågor (SELECT)

När vi använder select för att hämta data måste vi ange minst två saker. Först vad man vill välja (vilket fält) och sedan varifrån vi vill välja det (vilken tabell).

Nu kan vi se alla avgifter genom att skriva:

```
SELECT datum FROM avgifter;
```

Selectsatsen ovanför hämtar fältet **datum** från tabellen **avgifter**. Fältnamnet anges direkt efter SELECT och namnet på tabellen som vi vill hämta data från anges efter FROM.

Det är kanske inte så användbart, men trots allt -- det är data. Lite mer användbart för oss är denna fråga:

```
SELECT fnamn,enamn,datum FROM medlem,avgift WHERE id=medlem;
+-------+-----------+------------+
| fnamn | enamn     | datum      |
+-------+-----------+------------+
| Kalle | Svensson  | 2003-05-29 |
| Kurt  | Bengtsson | 2003-06-19 |
| Johan | Andersson | 2003-08-02 |
| Jill  | Jonsson   | 2003-12-02 |
+-------+-----------+------------+
4 rows in set (0.00 sec)
```

Lite mer användbart. Vad vi gör i frågan är att vi säger att vi vill ha data från fälten fnamn, enamn och datum från tabellerna medlem och avgift, men bara där fältet id är det samma som fältet medlem.

Skall man vara riktigt korrekt bör man skriva frågan så här:

```
SELECT medlem.fnamn,medlem.enamn,avgifter.datum FROM medlem,avgifter WHERE
medlem.id=avgifter.medlem;
```

Här anger vi vilken tabell fältet tillhör också.

Vilka har inte betalat sin medlemsavgift? Det får vi reda på genom att skriva:

```
SELECT fnamn,enamn,datum FROM medlem,avgifter WHERE id=medlem and datum < ’2003-08-11’;
```

I ovanstående fråga hämtar vi data från två tabeller, *medlem* och *avgifter*.

Svaret vi får fram är:

```
+-------+----------+------------+
| fnamn | enamn    | datum      |
+-------+----------+------------+
| Kalle | Svensson | 2003-05-29 |
+-------+----------+------------+
```

Ställa frågor med JOIN

join

Ändra data (UPDATE)

Kalle betalar sin medlemsavgift och vi måste ändra datumet i databasen. Det gör vi genom att skriva:

```
UPDATE avgifter SET datum=’2004-08-30’ WHERE medlem=’1’;
```

Vill du uppdatera flera fält i en post skriver du:

```
UPDATE tabell SET fält1=’data’, fält2=’merdata’ WHERE medlem=’1’;
```

Ta bort poster (DELETE)

Varning

Kommandot DELETE är ett destruktivt kommando! Om du tar bort en post med delete kommer den försvinna för alltid!

Om du vill ta bort poster från din databas använder du kommandot **DELETE**. Säg att medlem 2 (Johan Andersson) inte hade betalat sin medlemsavgift. Vi måste ta bort denna uppgift från databasen och skriver:

```
DELETE FROM avgifter WHERE id='2';
```

Summa av fält (SUM)

En sista fråga då, hur mycket medlemsavgifter har vi fått in?

```
SELECT SUM(avgift) FROM avgifter;
```

Databasen berättar för oss att vi fått in 600 kronor i medlemsavgifter.

Kapitel 9. Jobba med databaser

Visa databaser (SHOW DATABASES)

Kommandot **SHOW DATABASES** visar alla databaser som finns i databasservern.

En nyinstallerad MySQL-server har två databaser installerade: mysql och test.

```
SHOW DATABASES;
+------------+
| Database   |
+------------+
| mysql      |
| test       |
+------------+
```

Skapa databasen (CREATE DATABASE)

Kommandot **CREATE DATABASE** skapar en databas i databasservern. Med hjälp av **IF NOT EXISTS** kontrollerar du att databasen inte redan finns. Du använder naturligtvis bara en av dessa CREATE DATABASE för att skapa en databas.

```
CREATE DATABASE mindatabas;
Query OK, 1 row affected (0.24 sec)

CREATE DATABASE IF NOT EXISTS mindatabas;
Query OK, 0 row affected (0.00 sec)
```

Ta bort databasen (DROP DATABASE)

> **Varning**
>
> Kommandot DROP DATABASE är ett destruktivt kommando! Om du tar bort en databas med DROP kommer den försvinna för alltid! Detta är särskilt viktigt att tänka på när det gäller databasen *mysql* som innehåller information om användare och deras rättigheter i databasen! **Ta inte bort databasen mysql !**

Ibland vill vi ta bort databaser från servern, detta gör vi med kommandot **DROP DATABASE**. Se exemplen nedanför. Du använder naturligtvis en av dessa för att ta bort en databas, inte båda samtidigt.

```
DROP DATABASE mindatabas;
Query OK, 0 rows affected (0.00 sec)
```

```
DROP DATABASE IF EXISTS mindatabas;
Query OK, 0 rows affected (0.00 sec)
```

Kapitel 10. Jobba med tabeller

Skapa tabeller (CREATE TABLE)

Med kommandot **CREATE TABLE** skapar vi tabeller i databasen.

```
CREATE [TEMPORARY] TABLE [IF NOT EXISTS] <tablename>
[ (<create_statement>, ...) ]
[table_options] [select_statement]
```

Det är vid create_statement du anger kolumnnamn och datatyper, primärnyckel, index och restriktioner.

AUTO_INCREMENT använder vi för att få fält att automatiskt räkna upp värden. Ett fält som är satt som AUTO_INCREMENT måste också vara nyckelfält och du kan bara ha ett fält som är AUTO_INCREMENT. När du lagrar data i en kolumn som är AUTO_INCREMENT tar databasen det senaste värdet som lagrats i kolumnen och adderar det värdet med ett (1). Som standard börjar databasen att räkna från siffran ett (1), men du kan naturligtvis ändra detta när du skapar tabellen.

```
CREATE TABLE exempel (ID INT AUTO_INCREMENT PRIMARY KEY, namn CHAR(10));
Query OK, 0 rows affected (0.19 sec)
```

Ovanstående fråga skapar tabellen *exempel*. Tabellen består av två fält: *ID* som är av datatypen int (heltal) och har egenskaperna AUTO_INCREMENT och primärnyckel och fältet *namn* som är av datatypen char (tecken) och vi säger åt databasen att vi som mest kommer att lagra tio (10) tecken i fältet.

```
CREATE TABLE exempel2 (ID INT AUTO_INCREMENT PRIMARY KEY, namn CHAR(10)) AUTO_INCREMENT=2
Query OK, 0 rows affected (0.19 sec)
```

I ovanstående fråga gör vi samma sak som innan, med skillnaden att vi säger att AUTO_INCREMENT skall börja räkna från 20.

> **Tips:** När man använder AUTO_INCREMENT gör man det så gott som alltid på fält av heltalstyp (int).

Ändra tabeller (ALTER TABLE)

Kapitel 11. Använda databaser i programmering (formulär)

Detta kapitel visar hur du med enkelhet kan använda dina databaskunskaper i programmeringsspråket PHP. PHP använder du för att skapa dynamiska webbsidor för Internet och kan man få mer dynamik i webbsidorna än att använda en databashanterare som datalagrare?

Du behöver inte kunna skriva webbsidor sedan tidigare, även om det underlättar om du kan lite XHTML.

> **Tips:** Faktum är att kombinationen Apache (webbserver), MySQL (databashanterare) och PHP (skriptspråk) är mer använd än Microsoft IIS/ASP/SQL på Internet. Källa: *www.netcraft.com* (http://www.netcraft.com) .

SQL-funktioner i PHP

De vanligaste funktionerna som du kommer att behöva för att kommunicera med din MySQL-server från PHP.

mysql_connect

mysql_connect() - skapar förbindelsen med MySQL-servern

resurs **mysql_connect**([sträng server [, sträng anvÃ¤ndarnamn [, sträng lösenord [, bool nylänk [, int klient_flaggor]]]]])

mysql_connect() skapar en förbindelse med MySQL-servern så att vi kan arbeta med den från vårt PHP-skript. Om vi inte anger några värden är **mysql_connect** inställd på att använda följande värden: *server* = localhost:3306, *användarnamn* = den användare som kör skriptet (oftast webbserverns användare; www eller apache) och *lösenord* = tomt lösenord. localhost är den lokala datorn (den som skriptet körs på) och 3306 är standardporten för MySQL-servern.

Mysql_connect returnerar en MYSQL-länk om den lyckas ansluta. Misslyckas den returneras FALSE.

Förbindelsen med MYSQL-servern dör så snart PHP-skriptet körts, eller när du anropar mysql_close()-funktionen.

I exempel 11-1 skapar vi en anslutning (`$conn`) till MySQL-servern på *localhost*, vi loggar in på den som användaren *kalle* med läsenordet *hemligt*. Om anslutningen misslyckas avslutar vi skriptet med *or die...*, där vi anropar funktionen **mysql_error()** som skriver ut varfär anslutningen misslyckades. Om anslutningen lyckades skriver vi ut (**print**) lite text och avslutar färbindelsen med **mysql_close()**-funktionen.

I exempel 11-2 skapar samma anslutning som i exempel 11-1 med en enda skillnad: vi har lagrat servernamnet, användarnamnet och lösenordet i variabler. Variabeln `$host` anger vilken MySQL-server vi vill ansluta till, `$user` anger vilken användare vi vill ansluta som och `$pass` anger vilket lösenord vi vill använda.

> **Not:** Även om vi inte behöver avsluta förbindelsen till MySQL-servern (förbindelsen avslutas automatiskt när skriptet körts) bör vi göra det. Om inget annat ser det snyggare ut i koden.

Exempel 11-1. mysql_connect()

```
<?php
  $conn = mysql_connect( "localhost", "kalle", "hemligt") or die
( "Kunde inte ansluta till databasen: " . mysql_error() );
print ("Anslutningen till databasen lyckades");

mysql_close( $conn );
?>
```

Exempel 11-2. mysql_connect() med variabler

```
<?php
$host = "localhost";
$user = "kalle";
$pass = "hemligt";

$conn = mysql_connect( $host, $user, $pass ) or die
( "Kunde inte ansluta till databasen: " . mysql_error() );
```

```
print ("Anslutningen till databasen lyckades");

mysql_close( $conn );
?>
```

mysql_close

Med funktionen mysql_close stänger du förbindelsen med MYSQL-servern.

`mysql_close ( [ länkid ] )`

Funktionen returnerar `TRUE` om den lyckades stänga förbindelsen och `FALSE` om den misslyckades. Se exempel 11-1 och exempel 11-2 för exempel.

Se exempel 11-1 för exempel.

mysql_query

resurs **mysql_query** (fråga [, resursid])

mysql_query använder du för att ställa frågor till databasen från PHP. Om du inte anger ett *resursid* kommer PHP att använda den senast öppnade länken till en databas. Svaret från databasfrågan buffras i minnet och du kan använda till exempel avsnittet *mysql_fetch_row* för att hämta datat.

> **Not:** Du skall inte använda semikolon (;) i SQL-frågorna du ställer med **mysql_query** !

mysql_query returnerar `TRUE` om frågan lyckas, och `FALSE` om den misslyckas med frågan. Orsaker till misslyckande kan vara: du anger felaktigt namn på tabeller och poster, du saknar behörighet till databasen eller att det saknas en förbindelse till databasen.

Om frågan lyckas kan du använda **mysql_num_rows()** för att få reda på hur många poster den returnerade (detta gäller enbart select-frågor). **mysql_affected_rows()** returnerar på samma sätt antalet poster som användes av en delete-, insert-, replace- eller updatefråga.

Om du ställer frågor av typen SELECT, SHOW, DESCRIBE eller EXPLAIN kommer **mysql_query()** returnera en länk som du kan använda i **mysql_fetch_array()**. När du är klar med behandlingen av datat kan du frigöra resurserna med hjälp av **mysql_free_result()**. Detta sker automatiskt när skriptet avslutats.

Exempel 11-3. Ställa frågor med mysql_query

```
<?php

$query = mysql_query("SELECT * FROM tabell");
?>
```

mysql_fetch_row

array **mysql_fetch_row** (resurs)

Med mysql_fetch_row hämtar vi en post från svaret vi fått från mysql_query. mysql_fetch_row returnerar en array, där varje fält i posten med start på noll (0).

Använder vi **mysql_fetch_row** en gång till kommer vi få nästa post, en gång till ger oss nästa igen och så vidare. Om det inte finns några fler poster får vi tillbaka `FALSE`. Se exempel 11-4 .

> **Not:** Fältnamnen är skiftlägeskänsliga.

Exempel 11-4. mysql_fetch_row

```
$query = mysql_query("SELECT * FROM tabell");

while($tmp = mysql_fetch_row($query))
{
echo $tmp[0];
}
```

mysql_fetch_array

array mysql_fetch_array (resource result [, int resultat_typ])

mysql_fetch_array hämtar SQL-frågan och lagrar den som en tabell, en numerisk tabell eller båda.

Returnerar en tabell baserad på en post, eller `FALSE` om det inte finns några poster kvar.

mysql_fetch_array() är en utökad version av **mysql_fetch_row()**. Skillnaden mellan dem är att med **mysql_fetch_array()** kan du komma åt dina fält med deras namn och inte bara numeriska värden.

Om två, eller fler, fält har samma namn kommer den sista att gälla. För att komma åt fältet innan måste du använda det numeriska värdet, eller skapa ett alias för fältet.

Resultattypen kan vara `MYSQL_ASSOC`, `MYSQL_NUM` eller `MYSQL_BOTH` och anger vilken typ av array vi vill ha som svar. `MYSQL_ASSOC` ger oss en array där vi kan använda `$var['namn']`, där `$var` är namnet på den array vi har och `['namn']` anger vilket fält vi vill använda.

`MYSQL_NUM` numrerar fälten i arrayen så att du kommer åt dem genom `$var[0]`, `$var[1]` ... Denna metod kräver att du känner till din databas väl, eftersom att du måste veta ordningen på fälten. Fälten börjar räknas från noll (0), till exempel `$var[0]`.

> **Not:** Fältnamnen är skiflägeskänsliga !

Exempel 11-5. mysql_fetch_array med MYSQL_NUM

```
$query = mysql_query( "SELECT id, namn FROM tabell" );

while ($row = mysql_fetch_array( $query, MYSQL_NUM ) )
{
print( "ID : " . $row[0] . " Namn: " . $row[1] );
}
```

Exempel 11-6. mysql_fetch_array med MYSQL_ASSOC

```
$query = mysql_query( "SELECT id, namn FROM tabell" );
```

```
while( $row = mysql_fetch_array( $query, MYSQL_ASSOC ) )
{
print( "ID: " . $row['id'] . " Namn: " . $row['namn'] );
}
```

mysql_select_db

mysql_select_db - Väljer vilken databas du vill arbeta mot.

bool **mysql_select_db** (sträng databasnamn [, resurs länkid])

mysql_select_db returnerar `TRUE` om den lyckas och `FALSE` om den misslyckas med att välja databasen.

Med **mysql_select_db** väljer du vilken databas du vill arbeta med på den aktiva MySQL-servern. Om du inte anger `länkid` kommer den att använda den databas du senast öppnade med **mysql_connect**, om du inte har någon aktiv databaslänk kommer den att försöka att öppna en genom att anropa **mysql_connect** utan argument.

Alla följande **mysql_query** i koden kommer att arbeta mot den databas du väljer med **mysql_select_db**.

Exempel 11-7. mysql_select_db

```
<?php

$lank = mysql_connect( "localhost", "användare", "lösenord" ) or
die( "Inte ansluten till servern: " . mysql_error() );

mysql_select_db( "mindatabas" ) or
die( "Kunde inte välja databasen : " . mysql_error() );

?>
```

mysql_affected_rows

Returnerar antalet poster i databasen som berördes av frågan

int mysql_affected_rows ([resource link_identifier])

mysql_affected_rows() returnerar antalet poster som berördes av den senaste **INSERT**-, **UPDATE**- eller **DELETE**-frågan. Du använder **link_identifier** för att specificera vilken MySQL-koppling du vill arbeta med. Om ingen koppling anges, kommer den som senast skapades med **mysql_connect()** att användas.

> **Varning**
>
> Om den senaste frågan var en **DELETE** utan **WHERE** har alla posterna raderats. Trots detta kommer **mysql_affected_rows()** returnera noll (0).

mysql_affected_rows() fungerar inte med **SELECT** eftersom att en **SELECT**-fråga inte ändrar posterna. För att veta hur många poster en **SELECT**-fråga returnerar använder du **mysql_num_rows()**.

mysql_num_rows

Returnerar hur många poster en fråga ger.

int **mysql_num_rows** (resource result)

mysql_num_rows() returnerar antalet poster en fråga ger som svar. Funktionen fungerar enbart för **SELECT**-frågor, om du vill veta hur många poster en **INSERT**, **UPDATE** eller **DELETE**-fråga returnerar använder du **mysql_affected_rows()**.

mysql_errno

...

mysql_error

...

mysql_free_result

...

mysql_insert_id

...

Övningar

Här kommer lite övningar på kapitlet.

Appendix A. Datatyper

Heltal

Alla heltalstyper kan du använda med zerofill och unsigned. Zerofill fyller talet med nollor till vänster om talet, om du tillexempel skapar en int(4) zerofill och lagrar siffran 12 i det fältet kommer det att bli 0012. Unsigned innebär att du inte kan använda negativa heltal och på så sätt kan du få mycket högre positiva heltal. Om du använder zerofill kommer fältet automatiskt bli unsigned.

bigint

Bigint är den största heltalstypen. Du kan använda talen -9223372036854775808 till 9223372036854775807 signed eller 0 till 18446744073709551615 unsigned.

Du kan ange hur många siffror du vill tillåta genom att skriva bigint(*p*), där p anger hur många siffror (max 20 stycken).

int

Int ger dig talen -2147483648 till 2147483647 signed och 0 till 4294967295 unsigned.

Du kan ange hur många siffror du vill tillåta genom att skriva int(*p*), där p anger antal siffor (max tio stycken).

mediumint

Mediumint ger dig talen -8388608 till 8388607 signed och 0 till 16777215 unsigned.

mediumint(*p*) anger hur många siffror du vill tillåta (max åtta stycken).

smallint

Smallint ger dig talen -32768 till 32767 signed och 0 till 65535 unsigned.

smallint(*p*) anger hur många siffror du vill tillåta (max fem stycken).

tinyint

Tinyint är det minsta heltalet och ger dig talen -128 till 127 signed och 0-255 unsigned.

tinyint(*p*) anger hur många siffror du vill tillåta (max tre stycken).

Flyttal

Precis som heltaltyperna kan flyttalstyperna också ha zerofill och unsigned tilläggen.

float

Floatdatatypen är den minsta av de tre datatyperna för att hantera flyttal. Talen kan vara mellan $-3.402823466 * 10^{38}$ till $-1.175494351 * 10^{-38}$ för negativa tal och mellan $1.175494351 * 10^{-38}$ till $3.402823466 * 10^{38}$ för positiva tal.

float(*p*, *m*), där m anger hur många siffror som skall användas för decimalerna och p anger precisionen av talet, hur många siffror som skall användas totalt.

decimal

Decimal erbjuder samma talområde som double men lagrar talen som en sträng, ungefär som char. Talområdet är $-1.7976931348623157 * 10^{308}$ till $-2.2250738585072014 * 10^{-308}$ för de negativa talen och $2.2250738585072014 * 10^{-308}$ till $1.7976931348623157 * 10^{308}$ för de positiva talen.

decimal(*p*, *m*), där m anger hur många siffror som skall användas för decimalerna och p anger precisionen av talet, hur många siffror som skall användas totalt.

double

Double har samma talområde som decimal men lagrar talen i binärt format. Talområdet är $-1.7976931348623157 * 10^{308}$ till $-2.2250738585072014 * 10^{-308}$ för de negativa talen och $2.2250738585072014 * 10^{-308}$ till $1.7976931348623157 * 10^{308}$ för de positiva talen.

double(*p*, *m*), där m anger hur många siffror som skall användas för decimalerna och p anger precisionen av talet, hur många siffror som skall användas totalt.

Datumformat

Datumtyperna har hand om alla tidsformat i MySQL. Den största skillnaden mellan datumtyperna och de övriga numeriska datatyperna i MySQL är hur den hanterar nollvärden. Datumtyperna returnerar hela datumet även när du har ett nollvärde, datet returnerar 0000-00-00 och time returnerar 00:00:00 om du har ett nollvärde.

date

Date representerar ett helt kalenderdatum i formatet åååå-MM-DD. Giltiga värden för date är 1000-01-01 till 9999-12-31. Ogiltiga värden resulterar i värdet 0000-00-00.

datetime

Datetime är likvärdig med date, med den skillnaden att den också lagrar klockslaget. Formatet är åååå-MM-DD TT:MM:SS. Giltiga värden är 1000-01-01 00:00:00 till 9999-12-31 23:59:59 och ogiltiga värden resulterar i värdet 0000-00-00 00:00:00.

timestamp

Timestamp är en sträng som representerar Unix timestamp, som i sin tur kommer från tidsformatet som kallas Epoch. Epoch utgår från att tidräkningen började den 1 januari 1970. Giltiga värden för timestamp är 1970-01-01 00:00:00 till 2038-01-19 03:14:07 och ett ogiltigt värde resulterar i 0000-00-00 00:00:00 .

timestamp(*p*), där p anger hur många siffror som skall användas för tiden (maximalt 14, som också är standardinställningen). Om du anger att udda tal (till exempel 3) kommer du att få tal+1. Se tabell A-1 för mer information.

Tabell A-1. datatypen timestamp

Värde	Format
2	ÅÅ
4	ÅÅMM
6	ÅÅMMDD
8	ÅÅÅÅMMDD
10	ÅÅMMDDTTMM
12	ÅÅMMDDTTMMSS
14	ÅÅÅÅMMDDTTMMSS

time

Time är en generell representation av tidvärden i timmar, minuter och sekunder. Time visar värdet i 24-timmarsformat och klarar av att representera tidvärden i både dåtid och framtid. Giltiga värden är -839:59:59 till 838:59:59.

year

Year används för att representera kalenderår med två eller fyra siffror. I fyrasiffrors-formatet kan du använda värdena 1901 till 2155 eller 0000. I tvåsiffrorsformatet kan du använda talen 1970 till 2069, representerade av talen 70 till 69. Ogiltiga värden blir 0000 och i tvåsiffrorsformatet är en enkel nolla (0) ett ogiltigt värde. år 2000 måste vara 00.

Strängar

Strängtyperna används för textdata. Om en sträng som lagras är längre än vad du angett som giltigt kommer den att klippas i slutet av strängen.

blob

tinyblob, blob, mediumblob, longblob

Blob är strängtypen för binär data. Blob är case-sensitive, vilket innebär att om du lagrat **min text** och söker efter **MIN TEXT** kommer du inte hitta förekomsten.

Du har lite olika blobtyper att välja på, skillnaden mellan dem är hur många tecken (*bytes*) de kan lagra. Se tabell A-2 för mer information.

Tabell A-2. datatypen blob

Datatyp	Storlek
tinyblob	255
blob	65 535
mediumblob	16 777 215
longblob	4 294 967 295

char

Char används för att lagra textsträngar. char(*p*), där p anger hur många tecken det skall finnas plats för (max 255). Char reserverar alltid *p* antal tecken i databasen, även om du bara använder hälften av dem.

text

tinytext, text, mediumtext, longtext

Texttyperna används som blob, med skillnaden att de används för textdata och inte binärdata. Text är inte heller case-sensitive, vilket innebär att om du lagrat **min text** och söker efter **MIN TEXT** så hittar du förekomsten.

Det finns flera olika texttyper att välja på, skillnaden mellan dem är hur många tecken de kan lagra. Se tabell A-3 för mer information.

Tabell A-3. datatypen text

Datatyp	Storlek
tinytext	255
text	65 535
mediumtext	16 777 215
longtext	4 294 967 295

varchar

Varchar används som chartypen men hanterar data lite annorlunda. Medan char fyller upp utrymmet med blanktecken, så klipper varchar dem. Detta leder till att du använder mindre lagringsplats för dina strängar som är kortare än det reserverade området.

varchar(*p*) reserverar p antal tecken för varje varchar som lagras (max 255).

Appendix B. Referenser

Databaser (http://www.ida.liu.se/~tompa/databaser/databaser.html), Thomas Padron-McCarthy (tpm@ida.liu.se)

SQL på 10 minuter, PC Boken (http://www.pcboken.com/)

MySQL Bible, Wiley Publishing

www.php.net (http://www.php.net/)

www.mysql.com (http://www.mysql.com/)