

2018

ACADEMY

Håkan Johansson

[INTRODUKTION TILL RELATIONSDATABASER]

Allt du behöver för att förstå vad en relationsdatabas är och komma igång med programmering i SQL.

Innehållsförteckning

<i>Avsnitt</i>	<i>Sida</i>
Kursens syfte	2
Databasburken	2
Vad är data i en databas?	3
Hur organiseras och lagras data i en databas?	3
Relational Database Management System.....	5
SQL är inte lägeskänsligt.....	5
Formatering av SQL kod	6
Anslut till SQL Server och skapa en ny databas.....	6
Databasdesign.....	7
Skapa en tabell i databasen.....	10
Mer om SQL Datatyper	11
Null-värden.....	12
Kommentarer i SQL	13
Insert - Lägga till poster i en tabell.....	14
Select - Lista poster i en tabell.....	15
Jämförelseoperatorer	17
Tabell-alias	17
Fält-alias	18
Hakparenteser []	19
Mer om null-värden	20
Strängjämförelser med jokertecken	21
Update - Uppdatera (ändra) poster i en tabell.....	22
Delete - Radera poster i en tabell.....	22
Primärnycklar (Eng. Primary key).....	23
Unika fält.....	24
Radera tabeller.....	25
Boolsk algebra	26
Order by – sortering.....	29
Group by, aggregeringsfunktioner och having	30
Främmande nyckel (Eng. Foreign key)	32
Olika typer av JOINS	36
Join	36
Left join	37
Right join.....	37
Full join	38
Cross join.....	38
Olika typer av tabellrelationer	39
Tabellrelationen <i>En-till-en</i>	39
Tabellrelationen <i>En-till-många</i>	40
Tabellrelationen <i>Många-till-många</i>	40
Tabellrelationen <i>Självrefererande</i>	42
Scheman	43
Vyer (Views)	44
Lagrade procedurer (Stored Procedures)	45
Lagrade funktioner (Stored Functions).....	48
Triggers.....	51

Introduktion till relationsdatabaser med Microsoft SQL Server

Kursens syfte

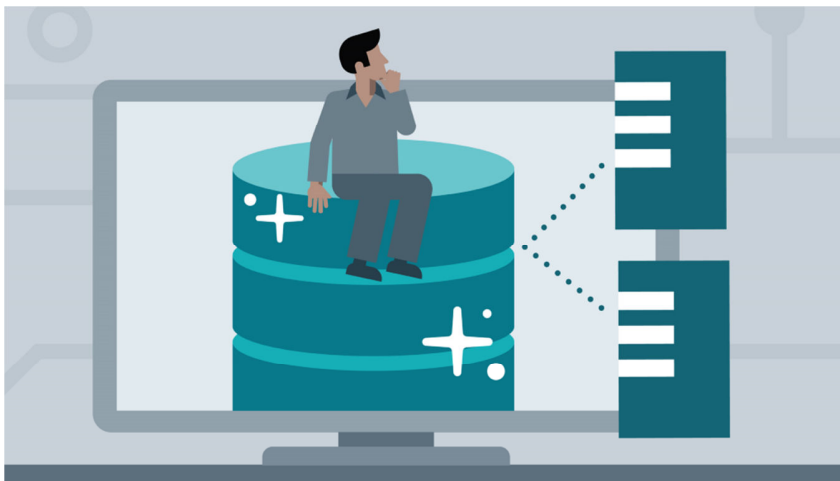
Kursens syfte är att förmedla en tydlig och användbar bild av vad en *relationsdatabas* är, men också att introducera grundläggande kommandon i frågespråket *SQL*.

Den här introduktionen handlar enbart om *relationsdatabaser*. För att förenkla kommer den följande texten huvudsakligen att använda ordet "databas" istället för det längre ordet "relationsdatabas".

Det finns andra typer av databaser, så kallade *NoSQL* databaser. Det är inte entydigt vad *NoSQL* står för, men några förslag är *non SQL*, *non relational SQL*, och *not only SQL*.

Databasburken

Den traditionella bilden av en databas är en *databasburk*.



Databasburken används för att visa hur en databas relaterar till andra objekt i en IT miljö, men säger naturligtvis ingenting om vad en databas är. Det enda vi kan sluta oss till om databasburken är att den innehåller eller *lagrar data*, exempelvis personuppgifter, ekonomisk information, väderstatistik, order- och lageruppgifter, etc. En databas är i regel en mycket välordnad och strukturerad samling av olika sorters data.

Vad är data i en databas?

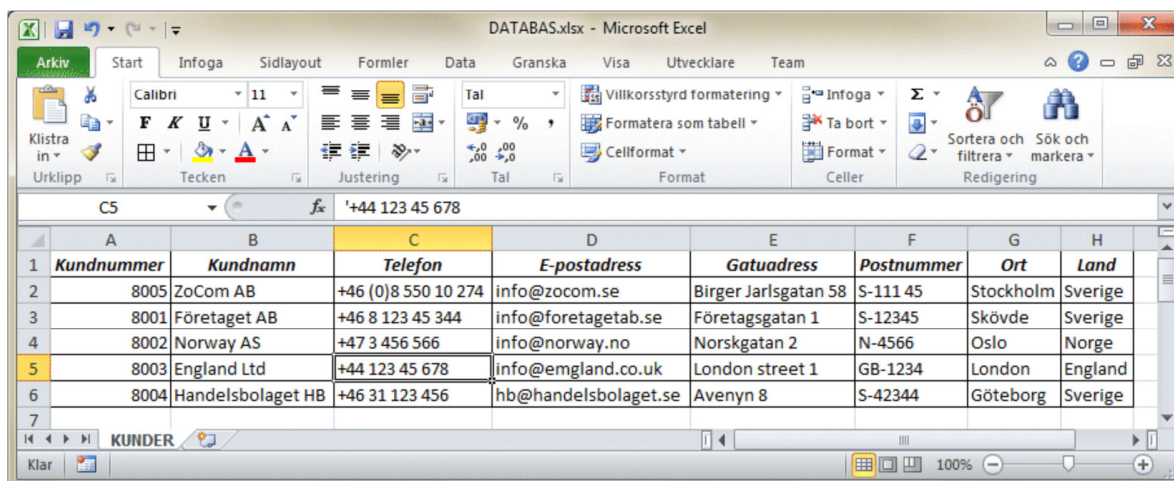
Data i en databas är enkelt uttryckt alla typer av information som databasen *kan* lagra, så som exempelvis *text*, *heltal*, *decimaltal*, *datum* och *tid*, och så vidare.

Även om det är fullt möjligt att lagra *binär data*, det vill säga video, bilder, ljud, etc. i en databas, så brukar man undvika det eftersom det påverkar databasens prestanda negativt. Istället lagrar man textreferenser ([URL:er](#) och sökvägar) i databasen till filer med binär data som är sparade på en filserver.

Hur organiseras och lagras data i en databas?

Alla data i en databas lagras i *namngivna tabeller*. Exempel på tabellnamn skulle kunna vara **Kunder**, **Leverantörer**, **Kundorder**, **Lagersaldon**, **Fakturor**, och så vidare.

Ett kalkylblad med namnet **Kunder** i Excel ger en *tydlig och användbar* bild av vad en tabell i en databas är.



	A	B	C	D	E	F	G	H
	Kundnummer	Kundnamn	Telefon	E-postadress	Gatuadress	Postnummer	Ort	Land
2	8005	ZoCom AB	+46 (0)8 550 10 274	info@zocom.se	Birger Jarlsgatan 58	S-111 45	Stockholm	Sverige
3	8001	Företaget AB	+46 8 123 45 344	info@foretagetab.se	Företagsgatan 1	S-12345	Skövde	Sverige
4	8002	Norway AS	+47 3 456 566	info@norway.no	Norskgatan 2	N-4566	Oslo	Norge
5	8003	England Ltd	+44 123 45 678	info@emgland.co.uk	London street 1	GB-1234	London	England
6	8004	Handelsbolaget HB	+46 31 123 456	hb@handelsbolaget.se	Avenyn 8	S-42344	Göteborg	Sverige

I en databas så skulle kalkylbladet **KUNDER** kallas *tabell* (eng. *Table*).

I en tabell kallas motsvarigheten till kalkylbladets kolumner för *fält* (eng. *Fields/Columns*), och motsvarigheten till kalkylbladets rader kallas *poster* (eng. *Records/Rows*).

Precis som kalkylblad lagras i en fil, i exemplet ovan i filen **DATABAS.xlsx**, så lagras även databasens tabeller i en (eller flera) filer. Motsvarigheten till filen **DATABAS.xlsx** skulle i relationsdatabasen Microsoft SQL Server heta **DATABAS.mdf** där filändelsen **.mdf** står för *Main/Master Data File*.

Till skillnad mot kalkylblad i Excel så hanterar vi *inte* databasens filer manuellt. Det vill säga, när vi vill redigera data i en tabell i en databas så gör vi det *inte* genom att öppna, redigera och spara filer. Istället gör vi det med hjälp av så kallade *SQL kommandon* eller *SQL queries*, som i sin tur automatiskt sparar tabelländringarna i databasfilen efterhand som vi exekverar eller kör dessa SQL kommandon eller queries.

SQL en förkortning för *Structured Query Language*. Ibland säger man även T-SQL där T står för *Transact*.

SQL är ett *deklarativt frågespråk*. Programmeringsspråk så som exempelvis C#, Java, och JavaScript är *imperativa kommandospråk*.

Med hjälp av SQL kommandon kan vi på ett deklarativt eller beskrivande sätt tala om vilka data som ska *visas*, *läggas till*, *ändras* eller *raderas*, och eftersom det är deklarativt vare sig kan vi eller behöver vi beskriva hur detta ska ske i teknisk mening. I detta avseende skiljer sig SQL väsentligt från programmeringsspråk så som C# och Java där man i detalj måste beskriva *hur* data ska visas och manipuleras.

Det finns bara *fyra SQL kommandon* för att hantera *all* tabelldata. Dessa är:

select - Visar/listar data från en eller flera tabeller i databasen.

insert - Läger till en eller flera nya poster (rader) i en tabell.

update - Uppdaterar/ändrar en eller flera befintliga poster i en tabell.

delete - Raderar en eller flera poster i en tabell.

Dessa fyra nyckelord i SQL-språket kallas tillsammans *Data Manipulation Language (DML)*.

Ytterligare en grupp av nyckelord i SQL är *Data Definition Language (DDL)* och används för att *skapa*, *ändra* och *radera* tabeller i databasen. DDL består av tre nyckelord, ***create***, ***alter*** och ***drop***.

Tabeller är ett av flera exempel på *databasobjekt* som kan skapas, ändras och raderas i en databas med hjälp av DDL. Exempel på andra databasobjekt är *vyer*, *lagrade procedurer*, *lagrade funktioner* och *triggers* som vi ska bekanta oss med senare i denna introduktion till SQL.

Introduktion till relationsdatabaser - Quiz 1

Relational Database Management System

SQL Server, DB2, Oracle, MySQL, och så vidare, är samtliga exempel på populära ”Relational Database Management Systems” (RDBMS).

I denna introduktion använder vi Microsoft SQL Server, men samtliga relationsdatabaser fungerar i grunden, med mindre skillnader, på samma sätt.

En av de mest utmärkande egenskaperna för en relationsdatabas är just att *tabeller relaterar till varandra och kan kopplas samman (relateras)* när vi skriver *select kommandon* eller så kallade *select queries* för att lista data.

(Man skulle kunna tro att ordet ”realional” kommer av att tabellerna i en databas i regel alltid relaterar till varandra, men i själva verket är ordet ”realional” en matematisk term som beskriver förhållandet mellan fält och poster i en tabell.)

I exemplet nedan relaterar tabellerna **Kunder** och **Kundorder** till varandra eftersom båda tabellerna innehåller ett fält med namnet **Kundnummer**, vilket gör att tabellerna kan *kopplas samman*. Det vill säga, med hjälp av fältet **Kundnummer** i tabellerna **Kunder** och **Kundorder** kan vi se vilka kunder som hör ihop med vilka order, och tvärtom.

Kundnummer	Kundnamn	Telefon	E-postadress	Gatuadress	Postnummer	Ort	Land
8005	ZoCom AB	+46 (0)8 550 10 274	info@zocom.se	Birger Jarlsgatan 58	S-111 45	Stockholm	Sverige
8001	Företaget AB	+46 8 123 45 344	info@foretagetab.se	Företagsgatan 1	S-12345	Skövde	Sverige
8002	Norway AS	+47 3 456 566	info@norway.no	Norskgatan 2	N-4566	Oslo	Norge
8003	England Ltd	+44 123 45 678	info@emgland.co.uk	London street 1	GB-1234	London	England
8004	Handelsbolaget HB	+46 31 123 456	hb@handelsbolaget.se	Avenyn 8	S-42344	Göteborg	Sverige

Ordernummer	Kundnummer	Orderdatum	Betalningsvillkor
167986	8005	2014-11-01	30 dagar
167987	8002	2014-11-15	10 dagar 2 %, 30 dagar
167988	8004	2014-10-10	10 dagar
167989	8005	2014-10-10	30 dagar
167990	8003	2014-10-02	30 dagar

SQL är inte lägeskänsligt

Till skillnad mot många andra programmeringsspråk så är SQL-kod som vi skriver inte lägeskänslig (case sensitive). Det vill säga, SQL gör ingen skillnad på stora och små bokstäver. Det spelar alltså inte roll om vi exempelvis skriver “select” (med små bokstäver), “Select” (med inledande stor bokstav), eller “SELECT” (med stora bokstäver).

Detsamma gäller även data som vi lagrar i databasens tabeller. Vid jämförelser av data så betraktar SQL företagsnamnet “Academic Work” (med inledande stora bokstäver) som likvärdigt med “academic work” (små bokstäver) och “ACADEMIC WORK” (stora bokstäver).

Formatering av SQL kod

Det spelar ingen roll hur vi formaterar (layoutar) vår SQL kod så länge syntaxen är korrekt (att orden i SQL-språket sammanfogas på rätt sätt). Vill vi, så kan vi skriva all SQL-kod på en enda lång rad, eller så kan vi använda radbrytningar och indrag (inflyttning av marginaler) efter eget godtycke eller enligt någon etablerad praxis.

Oftast används en etablerad praxis för hur SQL kod ska formateras, men denna varierar mellan olika programmerare och mjukvaruföretag. Som ny på en arbetsplats kan det därför vara fördelaktigt att observera hur SQL-koden formateras på just den platsen, eftersom det underlättar samarbete, förståelse och underhåll av gemensam kod.

Anslut till SQL Server och skapa en ny databas

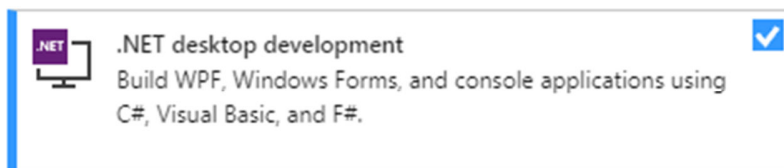
Innan vi kan börja hantera databaser och tabeller i dessa, måste vi först ansluta till en SQL Server. Även om namnet SQL Server antyder att vi ska ansluta till en annan dator, en *server*, så är SQL Server i själva verket en programvara i form av en tjänst/service som antingen körs på den egna lokala datorn (*local*), en annan dator i ett nätverket (*Network*), eller en dator i en molntjänst så som exempelvis *Microsoft Azure*.

När vi installerar Visual Studio så installeras automatiskt en SQL Server på den egna datorn (*local*) som vi kan ansluta till och arbeta mot.

Koduppgift 1

Installera Visual Studio 2017 via länken: <https://www.visualstudio.com/downloads/>

Under installationen, markera *endast* gruppen:



Under rubriken "Summary", markera *endast* "SQL Server Express 2016 LocalDB".

Summary

- ☐ F# desktop language support
- ☐ PreEmptive Protection - Dotfuscator
- ☐ .NET Framework 4.6.2 development tools
- ☐ .NET Framework 4.7 development tools
- ☐ .NET Framework 4.7.1 development tools
- ☐ .NET Core 2.0 development tools
- ☐ .NET Core 1.0 - 1.1 development tools
- ☐ Windows Communication Foundation
- ☒ SQL Server Express 2016 LocalDB

Följande [video](#) visar hur vi ansluter till SQL Server i Visual Studio och skapar en databas med namnet CobolDB. Databasens namn är helt godtyckligt och kunde lika gärna ha hetat exempelvis Academy, Sverige, eller något helt annat. Databasnamnet bör endast bestå av bokstäverna A till Z, a till z. Använd aldrig mellanslag och undvik svenska bokstäver i databasnamnet.

(<https://VWatchie.tinytake.com/sf/MjUyNzczMV83NTk5MDg4>)

Koduppgift 2

Anslut till SQL Server och skapa databasen CobolDB på det sätt som visades i föregående video. För att underlätta support och samarbete, hitta inte på ett eget databasnamn!

Databasdesign

Generellt vill vi alltid att vår databas ska vara *normaliserad*. Det innebär att vi överlag inte vill upprepa samma data, exempelvis ett personnummer, i samma tabell på fler rader (poster) eller upprepa samma data i olika tabeller.

Syftet med en *normaliserad* databas är att det bara ska finnas en enda tabell och en enda post att uppdatera för ett visst fält så som exempelvis ett personnummer.

Om samma data skulle förekomma i flera poster eller i flera tabeller, så skulle det finnas risk för att vi glömmer att uppdatera på samtliga platser när vi uppdaterar på en plats. Vi skulle dessutom få en onödigt stor databas!

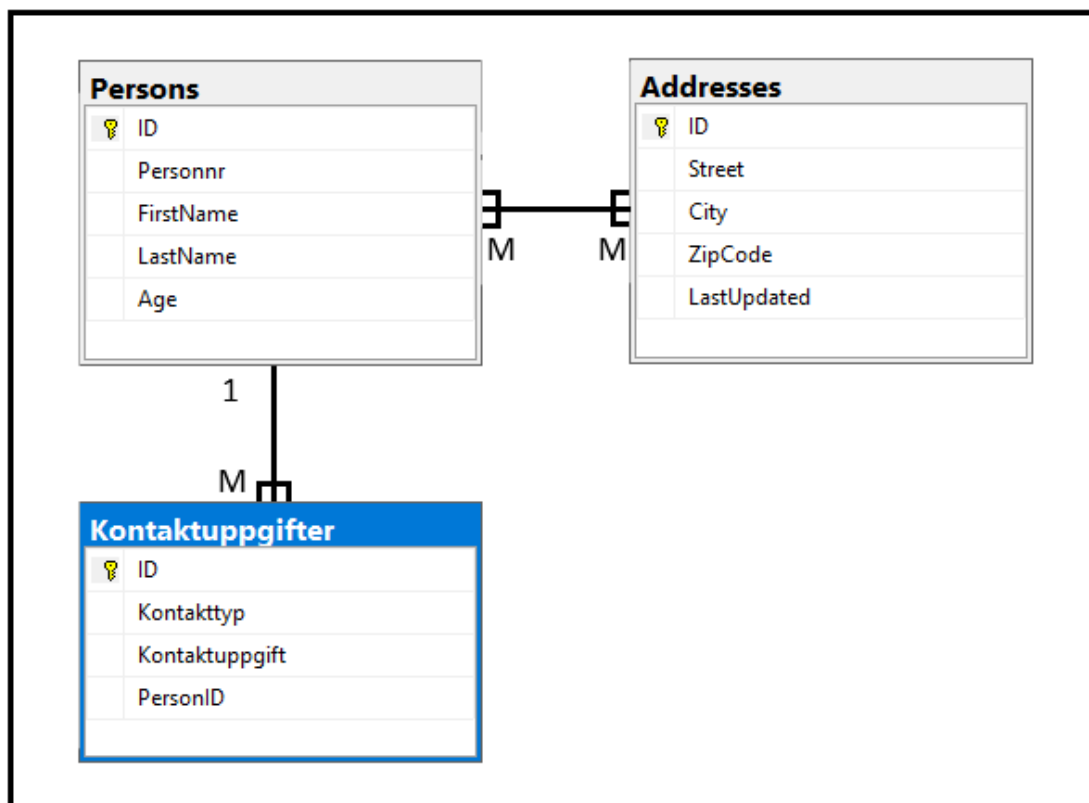
I sin enklaste form designar vi en normaliserad databas i *tre steg*.

Steg 1: Identifiera de verkliga objekt (substantiv) som databasen ska hantera. Varje objekt blir en egen tabell. Exempelvis: ***Personer***, ***Adresser***, och ***Kontaktuppgifter***.

Steg 2: Identifiera varje objekts egenskaper. Varje egenskap blir ett fält i tabellen. Exempelvis fälten *Förnamn* och *Efternamn* i tabellen *Personer*, fälten *Postnummer* och *Ort* i tabellen *Adresser*, och fälten *Kontakttyp* och *Kontaktuppgift* i tabellen *Kontaktuppgifter*.

Steg 3: Bestäm hur tabellerna relaterar till varandra.

Relationerna mellan tabellerna är enklast och tydligast att beskriva grafiskt, exempelvis med hjälp av papper och penna. Exempel:




I diagrammet ovan framgår att:

- “En (1) person kan ha noll, en eller flera (Många) adresser”,
- “En (1) adress kan tillhöra noll, en eller flera (M) personer”,
- “En (1) person kan ha noll, en eller flera (M) kontaktuppgifter”.
- “En (1) kontaktuppgift kan bara tillhöra en (1) person”

I verkligheten kan naturligtvis en kontaktuppgift höra ihop med flera personer, exempelvis ett gemensamt telefonnummer, men i vår databasdesign har vi beslutat oss för att en kontaktuppgift endast får höra ihop med en person.

Nedan två tabeller, **Personer** och **Kontaktuppgifter**, exemplifierar en en-till-många-relation i en normaliserad databas.

Tabell: **Personer**

ID	Personnummer	Förnamn	Efternamn
1	19620601-1234	Anna	Andersson
2	19760809-1234	Bertil	Bertilsson
3	19890205-1234	Cecilia	Nilsson
4	19901205-1234	David	Danielsson
 PERSONER > KONTAKTUPPGIFTER			

Tabell: **Kontaktuppgifter**

ID	Kontakttyp	Kontaktuppgift	PersonsID
1	Mobil	070 123 45 67	3
2	E-post	cecilia.nilsson@gmail.com	3
3	Hem tfn	08 123 45 67	3
4	Mobil	073 987 65 43	2
5	E-post	anna.andersson@gmail.com	1
6	Hem tfn	031 753 76 23	4
7	Mobil	072 222 66 89	4
 PERSONER > KONTAKTUPPGIFTER			

I tabellen **Personer** har varje person ett unikt **ID**. I fältet **PersonsID** i tabellen **Kontaktuppgifter** upprepas detta **ID**-värde för respektive kontaktuppgift.

På detta sätt kan vi utläsa vilka kontaktuppgifter en viss person har, och vilken person som respektive kontaktuppgift hör ihop med. Även tabellen **Kontaktuppgifter** har ett fält med ett unikt **ID**, vilket är praxis, men detta fält har ingen betydelse för själva relationen mellan de båda tabellerna.


Om exempelvis *Cecilia Nilsson* skulle få skyddad identitet och byta både namn och personnummer, så skulle vi endast behöva uppdatera denna information på en enda plats, i tabellen **Personer**, och ingen annan information eller relation skulle påverkas i databasen.

I en *de-normaliserad* databas hade vi samlat samtliga tabellers egenskaper i en enda tabell.

Nedan tabell, **Kontakter**, exemplifierar en *en-till-många-relation* i en de-normaliserad databas.

Tabell: **Kontakter**

ID	Personnummer	Förnamn	Efternamn	Kontakttyp	Kontaktuppgift
1	19620601-1234	Anna	Andersson	E-post	anna.andersson@gmail.com
2	19760809-1234	Bertil	Bertilsson	Mobil	073 987 65 43
3	19890205-1234	Cecilia	Nilsson	Mobil	070 123 45 67
3	19890205-1234	Cecilia	Nilsson	E-post	cecilia.nilsson@gmail.com
3	19890205-1234	Cecilia	Nilsson	Hem tfn	08 123 45 67
4	19901205-1234	David	Danielsson	Hem tfn	031 753 76 23
4	19901205-1234	David	Danielsson	Mobil	072 222 66 89



Om *Cecilia Nilsson* i detta fall skulle få skyddad identitet skulle vi behöva byta namn och personnummer i tre poster. Risken finns att vi glömmer detta eller exempelvis stavar namnet fel eller skriver fel personnummer på någon av posterna.

Databasfrågor, så kallade *select queries*, som ställs mot de-normaliserade tabeller kan ge bättre prestanda i form av snabbare svarstider eftersom query:n involverar färre tabeller som annars måste kopplas samman i query:n. Av detta skäl händer det att databasdesigners avsiktligt de-normaliserar delar av databasens tabeller.

Mer om relationer och hur dessa definieras i SQL-kod beskrivs senare i denna introduktion.

Introduktion till relationsdatabaser - Quiz 2

Skapa en tabell i databasen

I den föregående videon visades hur vi öppnar ett *Query window* i Visual Studio mot en bestämd databas (genom att högerklicka på en databas i fönstret "SQL Server Object Explorer" och välja "New Query..."). Det är i query windows som vi skriver och exekverar SQL kod.

I videon skrev vi DDL-kod (Data Definition Language) för att skapa en databas:

```
create database CobolDB.
```

På ett liknande sätt kan vi skriva och exekvera DDL-kod för att skapa en *tabell* i den aktuella (valda) databasen. *Exempel:*

```
create table Persons
(
    FirstName varchar(32),
    LastName varchar(64),
    Age int
)
```

Ovan kodexempel skapar tabellen ***Persons*** i databasen.

Varje post i tabellen kommer att ha tre fält, ***FirstName***, ***LastName***, och ***Age***.

Fälten ***FirstName*** och ***LastName*** är deklarerade med datatypen *varchar* vilket innebär att fälten endast kan lagra data som är alfanumerisk, det vill säga *text* (i form av bokstäver och siffror). Siffran 32 inom parentes efter datatypen *varchar* anger att fältet ***FirstName*** kan innehålla maximalt 32 tecken. På motsvarande sätt kan fältet ***LastName*** innehålla maximalt 64 tecken.

Observera att datatypen *varchar* (alfanumerisk) aldrig kan användas för att göra beräkningar, trots att vi även kan spara siffror i denna datatyp. För att göra beräkningar måste datatypen vara av taltyp så som exempelvis *int* eller *money* (mer om datatyper nedan).

Datatypen för fältet *Age* är *int* (integer) vilket innebär att det endast kan lagra heltal.

När vi skapar fält i en tabell måste fältens datatyp alltid anges. Det finns med andra ord ingen default (förvald) datatyp.

[Denna video](https://VWatchie.tinytake.com/sf/MjUyNzg4Nl83NTk5NTQ2) (https://VWatchie.tinytake.com/sf/MjUyNzg4Nl83NTk5NTQ2) visar hur tabellen ***Persons*** skapas med hjälp av Visual Studio.

Koduppgift 3

Skapa tabellen ***Persons*** i databasen CobolDB på det sätt som visades i föregående video.

Mer om SQL Datatyper

Som vi redan sett så måste vi ange vilken datatyp varje enskilt fält ska ha när vi skapar en tabell. I tabellen ***Persons*** använde vi datatyperna ***varchar*** och ***int***, men det finns betydligt fler datatyper i SQL. Några av de allra vanligaste datatyperna listas i nedan tabell.

<i>Vanliga datatyper i SQL</i>	<i>Användning</i>
int	Heltal mellan -2.147.483.648 och 2.147.483.647
money	decimaltal mellan -922.337.203.685.477,5808 och 922.337.203.685.477,5807
float	Flyttal
datetime	datum och tidpunkt
varchar	Text
nvarchar	Unicode text (Ex. kyrilliska tecken)

En av de viktigaste anledningarna till att det finns olika datatyper är att vi på så sätt kan låta SQL bevaka att vi inte av misstag lagrar felaktig typ av data i ett visst fält.

Ett par exempel:

Eftersom fältet **Age** i tabellen **Persons** är deklarerad som en **int** (heltal) kommer SQL att generera ett felmeddelande om vi exempelvis försöker lagrar ett förnamn i fältet.

Ytterligare ett exempel är om vi har ett fält av datatypen **DateTime**, vilket innebär att SQL kan säkerställa att vi lagrar datum i ett giltigt format så att vi kan beräkna kalenderdata, exempelvis antalet dagar mellan två datum, och så vidare.

Null-värden

Typiskt för fält i en tabell, *oavsett vilken datatyp de har*, är att de kan lagra så kallade null-värden. Null betyder “*okänt värde*” men tolkas ibland felaktigt som att värde inte existerar, eller är noll (0).

Anta att vi skapar en ny post i tabellen **Persons**. Vi kan då exempelvis ange värdet null för fältet **Age**, vilket alltså indikerar att åldern är okänd, och inte att personen saknar ålder.

När vi skapar en tabell kan vi bestämma om ett fält ska *tillåta null-värden* eller *inte tillåta null-värden* genom att antingen skriva *null* eller *not null* efter deklarationen av fältet.

Exempel:

```
create table Persons
(
    FirstName varchar(32) null,
    LastName varchar(64) not null,
    Age int null
)
```

Praxis är att *tillåta* null-värden så långt som möjligt. Anledningen är att värden, exempelvis ett organisationsnummer, kan vara okända när posten registreras, och om null-värden inte tillåts så kan posten överhuvud taget inte sparas i tabellen.

Null är oftast default (förvalt) när vi skapar tabeller, men SQL Server kan konfigureras (ställas in) så att *not null* är default. För att vara på den säkra sidan bör vi därför alltid uttryckligen (som visas i ovan kodexempel) ange om respektive fält ska vara *null* eller *not null*.

Introduktion till relationsdatabaser - Quiz 3

Kommentarer i SQL

När vi skriver SQL kod kan vi i bland vilja kommentera den kod vi skriver. Kanske vill vi förklara hur vi tänkt, vad kodens syfte är, vem som skrivit den och när, och så vidare.

Det finns två typer av SQL kommentarer, *enradskommentarer* och *flerrads-kommentarer*.

En enradskommentar inleds av två bindestreck efter varandra `--` och följs av själva kommentaren, exempelvis:

```
-- Detta är en enradskommentar i SQL.
SELECT ArtNr, ArtBeskr -- Lista artikelnr och artikelbeskr.
```

I Visual Studio kan vi enradskommentera en eller flera rader samtidigt genom att först markera raderna, och sedan använda snabbkommandot `[Ctrl]+[K]+[C]` (*omment*).

Vi kan även ta bort kommentarstecknen `--` genom att först markera raderna och sedan använda snabbkommandot `[Ctrl]+[K]+[U]` (*ncomment*).

En flerradskommentar inleds av tecknen `/*`, följt av kommentaren, och avslutas med tecknen `*/`, exempelvis:

```
/*  
Syfte: Beräkning av omsättningshastighet:  
Kod: Håkan Johansson  
Version: 1.0  
Datum: 2018-05-09  
*/
```

Koduppgift 4

Skapa en tabell med namnet *Addresses* i databasen *CobolDB* med fälten “*Street*”, “*City*”, “*ZipCode*”, samt “*LastUpdated*”. Använd lämpliga datatyper. Inga fält, utom fältet “*ZipCode*”, får tillåta null-värden. Lägg till en flerradskommentar ovanför tabelldeklarationen och en enradskommentar efter något eller några av fälten.

Tips: Använd datatypen *datetime* för fältet “*LastUpdated*”.

Insert - Lägga till poster i en tabell

För att lägga till *nya poster* (rader) i en tabell används DML kommandot ***insert***. Syntaxen för detta kommando är:

```
insert into  
    Tabellnamn  
values  
    (fältvärde1, fältvärde2,...), -- Ny Post 1  
    (fältvärde1, fältvärde2,...) -- Ny Post 2
```

Exempel:

```
insert into  
    Persons  
values  
    ('Håkan', 'Johansson', 56), -- Ny Post 1  
    ('Petra', 'Pettersson', 28) -- Ny Post 2
```

Om vi inte har behov av att tilldela värden till tabellens samtliga fält, kan vi uttryckligen ange vilka fält som vi vill tilldela värden. Övriga fält får då värdet *null*, vilket naturligtvis förutsätter att dessa fält tillåter null-värden.

Exempel:

```
insert into
  Persons(FirstName, LastName)
values
  ('Håkan', 'Johansson'), -- Ny Post 1
  ('Petra', 'Pettersson'), -- Ny Post 2
```

Följande video demonstrerar hur vi lägger till två nya adressposter till tabellen *Addresses*.

(<https://VWatchie.tinytake.com/sf/MjUzMDU3MV83NjA2Mjk1>)

Koduppgift 5

Lägg till två nya valfria adresser i Stockholm, två nya valfria adresser i Kista, samt två helt valfria adresser (utom Stockholm och Kista) i tabellen *Addresses*. Låt två av dessa sex poster få värdet null i fältet *ZipCode*.

Tips: För att ange ett null-värde skriv just bokstäverna *null*, utan citattecken.

Tips: För att ange datum, skriv datum på formatet '*ÅÅÅÅ-MM-DD*' inklusive citattecknen. Alternativt, anropa SQL funktionen *GETDATE()* som returnerar aktuellt datum och aktuell tidpunkt.

Select - Lista poster i en tabell

DML kommandot *select* används för att lista poster i en tabell, eller flera sammankopplade tabeller. Kommandot *select* är det i särklass mest omfattande kommandot i SQL. Här visar vi dess grundläggande användning.

Till skillnad mot alla andra DML- och DDL-kommandon så är *select* kommandot det enda kommandot som är ”*helt ofarligt*” att *exekvera*, det vill säga inte påverkar någonting i databasen. Det finns alltså inga som helst risker med att skriva *select* kommandon, oavsett hur de ser ut.

De poster som listas när vi skriver ett *select* kommando är endast en spegling av de data som *select* kommandot hämtar från databasens tabeller. Listan med poster som *select* skapar och visar på datorskärmen kallas en *resultatmängd* (Eng. *Result set*) och existerar endast temporärt i datorns RAM-minne (RAM = Random Access Memory). *Select* kommandots grundläggande syntax ser ut enligt följande exempel:

```
select
    Fältnamn1, Fältnamn2, ...
from
    Tabellnamn
```

Nyckelordet *from* anger från vilken (eller vilka) tabell(er) vi vill hämta data.
Nyckelordet *select* anger vilka fält vi vill lista i de tabeller vi anger i from-delen.

För att söka fram vissa bestämda poster i tabellen kan select kommandot kompletteras med en så kallad *where klausul*:

```
select
    Fältnamn1, Fältnamn2, ...
from
    Tabellnamn
where
    villkor
```

Om vi exempelvis endast vill lista personer i tabellen Persons som är äldre än 24 år kan vi skriva villkoret: *where Age > 24*:

```
select
    FirstName, LastName, Age
from
    Persons
where
    Age > 24
```

Istället för att specificera vilka fältnamn vi vill visa (efter nyckelordet *select*), kan vi istället skriva en asterisk *. *En asterisk anger att vi vill lista tabellens (eller tabellernas) samtliga fält.*

Praxis när vi skriver select kommandon är att alltid börja med att skriva *select ** och sedan, när queryn är klar med sina övriga klausuler så som where klausulen, ersätta asterisken med de fält som vi faktiskt vill lista.

[Följande video](https://VWatchie.tinytake.com/sf/MjUzMDY4OF83NjA2NDYw) visar några exempel på användningen av *select*.
(<https://VWatchie.tinytake.com/sf/MjUzMDY4OF83NjA2NDYw>)

Jämförelseoperatorer

SQL innehåller ett flertal jämförelseoperatorer. Dessa är:

= Lika med. Exempel: `City = 'Kista'`

!= Skilt från (inte lika med). Exempel: `City != 'Kista'`

<> Skilt från (inte lika med). Exempel: `City <> 'Kista'` (rekommenderas ej!)

> Större än. Exempel: `Age > 24`

< Mindre än. Exempel: `Age < 18`

>= Större än eller lika med. Exempel: `Age >= 25`

<= Mindre än eller lika med. Exempel: `Age <= 17`

!< Inte mindre än. Exempel: `Age !< 24` (Motsvarar: `Age >= 10`)

!> Inte större än. Exempel: `Age !> 10` (Motsvarar: `Age <= 10`)

Koduppgift 6

Lista fälten `ZipCode` och `City` i tabellen `Addresses` för samtliga adresser i Stockholm.

Koduppgift 7

Lista samtliga fält i tabellen `Addresses` för samtliga adresser vars `City` *inte* är Stockholm.

Introduktion till relationsdatabaser - Quiz 4

Tabell-alias

Eftersom det är så vanligt att queries refererar flera tabeller, vilket vi snart kommer att se exempel på, och det dessutom är vanligt att olika tabeller innehåller samma fältnamn (exempelvis `ID`), så bör vi göra det till en vana att *alltid* kvalificera fältet med fältets tabellnamn när vi listar fältet, även då vi bara använder en enda tabell.

Detta gör vi genom att först skriva tabellnamnet, följt av en *punkt*, följt av fältnamnet, exempelvis: *`Addresses.ZipCode`*

Eftersom tabellnamn ofta består av flera tecken och ibland kan vara ganska långa, i alla fall om de är beskrivande, exempelvis *Addresses*, så kan de bli ganska omständliga att ständigt behöva skriva i en query, framförallt när vi listar många av tabellens fält. För att lindra denna olägenhet så kan vi använda så kallade *tabell-alias*.

Ett tabell-alias i en query ger *tillfälligt* tabellen ett annat oftast kortare namn i just den query vi skriver.

När vi vill använda ett tabell-alias skriver vi nyckelordet *as* efter tabellens namn i query:ns *from-klausul*, följt av *alias-namnet* (många gånger bara en enda bokstav).

Observera att tabell-alias är tillfälliga, inte sparas i databasen och endast gäller i den query där tabell-alias används.

Exempel:

```
select
    A.Street,
    A.ZipCode
from
    Addresses as A -- A är ett tabell-alias
where
    A.City = 'Stockholm'
```

Om vi anger ett tabell-alias *måste* vi använda detta alias. Vi kan alltså inte växla mellan att använda tabellens riktiga namn och dess alias.

Koduppgift 8

Använd tabell-alias i de två föregående koduppgifterna.

Fält-alias

Fält-alias liknar tabell-alias och används för att tillfälligt, då vi skriver en query, ge fälten som vi listar andra namn. Kanske vill vi lista fältnamnen på ett annat språk, eller vill använda mer beskrivande fältnamn.

Observera att fält-alias endast kan användas i query:ns *select-* och *order by-*klausuler (mer om *order by* och sortering senare).

Exempel:

```
select
    A.Street as Gata, -- Gata är ett fält-alias
    A.ZipCod as Postnummer -- Postnummer är ett fält-alias
from
    Addresses as A
where
    A.City = 'Stockholm'
```

Koduppgift 9

Använd lämpliga fält-alias i den föregående koduppgiften.

Tips: För att kunna använda fält-alias i en select query för samtliga fält i en viss tabell så kan vi inte använda asterisk *, utan måste uttryckligen lista varje fält för sig. Med hjälp av fönstret ”SQL Server Object Explorer” i Visual Studio kan vi markera samtliga fält i en viss tabell och sedan släpa ut dem i redigeringsfönstret.

Hakparenteser []

Hakparenteser kan användas i SQL-kod för att omsluta tabell- och fältnamn som annars hade varit ogiltiga. Tabell- och fältnamn som består av flera ord, icke alfanumeriska tecken, och SQLs egna nyckelord så som exempelvis *select* och *create*, är ogiltiga som tabell- och fältnamn.

Anta exempelvis att vi skulle vilja använda nyckelordet *select* som ett tabellnamn. Normalt skulle SQL inte tillåta detta utan generera ett felmeddelande om vi försökte.

Exempel:

```
create table Select -- Incorrect syntax near the keyword 'Select'.
(
    ...
```

Men, genom att skriva nyckelordet inom *hakparenteser* så ”förstår” SQL att vi inte avser att använda nyckelordet i sig, utan som just ett tabellnamn.

Exempel:

```
create table [Select]
(
    ...
```

Samma princip gäller ogiltiga fält- och tabellnamn.

Exempel:

```
create table [Strange Table]
(
    [Insert] varchar(32) null,
    [Update] varchar(64) not null,
    [Delete] int null
)
```

Som framgår av ovan exempel kan vi även använda hakparenteser för att namnge tabeller och fält (samt även tabell-alias och fält-alias) med namn som visserligen inte är nyckelord men som ändå hade varit ogiltiga, exempelvis tabellnamn med mellanslag så som "Strange Table" i kodexemplet ovan.

Koduppgift 10

- a) Skapa tabellen [Strange Table] enligt exemplet ovan.
- b) Infoga (insert) två poster med valfria data i tabellen.
- c) Lista (select) tabellens samtliga poster.

Mer om null-värden

När vi ska jämföra ett fältvärde med *null*, kan vi *inte* använda jämförelseoperatorerna *lika med* (=) och *inte lika med* (!=). Istället måste vi använda operatorerna *is null* och *is not null*.

Anledningen är att null inte betraktas som ett *värde*, och att det därför skulle vara ologiskt att använda jämförelseoperatorer (=, !=, <, >, etc.), som ju opererar på just *värden*.

Exempel:

```
select -- Lista personer med okänd ålder.
    *
from
    Persons
where
    Age is null
```

Koduppgift 11

- a) Skriv ett select query som listar samtliga poster i tabellen **Addresses** där fältet ZipCode är skilt från (inte lika med) null.
- b) Vad blir resultatet av query:en i a) övningen om du använder *lika med operatorn* (=) istället för *is-operatorn*, samt *inte lika med operatorn* (!=) istället för *is not-operatorn*?

Strängjämförelser med jokertecken

Anta att vi vill lista alla poster i tabellen *Addresses* där fältet *Street* innehåller ordet “gata”, exempelvis “Drottninggatan” och “Norgegatan”. Då använder vi jokertecknet % (procent) som betyder “vilka tecken som helst, hur många eller få som helst” i kombination med *like-operatorn* och *not like-operatorn*.

Exempel:

```
select
    *
from
    Addresses
where
    Street like '%gata%'
```

I exemplet ovan innebär de två %-tecknen att ordet **gata** får föregås och efterföljas av vilka och hur många tecken som helst.

I följande exempel listas samtliga poster i tabellen **Persons** där efternamnet får börja med vilka tecken som helst men måste avslutas med tecknen **son**.

```
select
    *
from
    Persons
where
    LastName like '%son'
```

Vid exakta jämförelser *utan jokertecken* kan och bör “*lika med-operatorn*” (=) och “*inte lika med-operatorn*” (!=) används då *like* och *not like* operatorerna ger dålig prestanda, d.v.s. kan leda till att query:n tar lång tid att exekvera.

Exempel:

```
LastName = 'johansson'
```


Koduppgift 12

- a) Skriv en query som listar samtliga poster i tabellen Addresses där fältet “Street” *inte* innehåller ordet “väg”. Lista endast fältet “Street”.
- b) Skriv en query som listar samtliga poster i tabellen Addresses där fältet “Street” inleds med bokstaven “S” (Eller någon annan enstaka bokstav).

Update - Uppdatera (ändra) poster i en tabell

DML kommandot **update** används för att ändra fältvärden i en eller flera poster i en tabell. Syntaxen för update:

```
update
    Tabellnamn
set
    fältnamn1 = nytt fältvärde,
    fältnamn2 = nytt fältvärde,
    ...
where
    villkor...
```

Följande video visar hur vi kan uppdatera samtliga poster i tabellen Addresses så att de adresser som finns på orten Kista ändras till orten Kil.
(<https://VWatchie.tinytake.com/sf/MjUzMDc3NV83NjA2Njk0>)

Koduppgift 13

- a) Uppdatera tabellen Addresses så att samtliga Ortsnamn (City) sparas med versaler (stora bokstäver) i tabellen. *Tips:* Använd SQL funktionen **Upper(City)**.
- b) Skriv en select query som listar fältet City (endast) för samtliga poster i tabellen Addresses. Har samtliga City-namn verkligen sparats med stora bokstäver?

Delete - Radera poster i en tabell

DML kommandot **delete** raderar en eller flera poster i en tabell.

Syntax:

```
delete from
    Tabellnamn
where
    villkor...
```

Följande video visar hur vi kan radera samtliga adresser i Kil.
(<https://VWatchie.tinytake.com/sf/MjUzMTI4NF83NjA4MTc4>)

Observera att ***delete*** kommandot inte kan användas för att radera enstaka kolumner i en tabell utan alltid raderar *hela poster*.

Observera även att ***delete*** kommandot alltid raderar samtliga poster i tabellen om vi inte inkluderar en ***where*** klausul som bestämmer vilka rader som ska raderas.

Koduppgift 14

Radera samtliga poster i tabellen Addresses vars gatunamn (Street) har fler än 16 tecken. Tips: Använd den inbyggda SQL funktionen ***LEN(Street)***.

Introduktion till relationsdatabaser - Quiz 5

Primärnycklar (Eng. Primary key)

De allra flesta tabeller i en databas definierar ett fält som unikt identifierar varje enskild post i tabellen. Detta fält kallas *primary key* (Sv. primärnyckel) En *primärnyckel* fungerar alltså som ett slags *personnummer* för varje enskild post i tabellen.

Skulle vi av misstag försöka skapa en ny post i en tabell med samma primärnyckelvärde ("personnummer") som en befintlig post så skulle SQL förkasta den nya posten och generera felmeddelandet **Violation of PRIMARY KEY**.

I tabellen ***Kunder*** är fältet ***Kundnummer*** en typisk primärnyckel, och i tabellen ***Kundorder*** är fältet ***Ordernummer*** en typisk primärnyckel.

En tabell kan också definiera en så kallad *sammansatt primärnyckel*, d.v.s. en primärnyckel som består av flera fält i tabellen och som tillsammans unikt identifierar varje enskild post. Exempelvis skulle en tabell med adresser kunna ha en primärnyckel bestående av en kombination av ***gata***, ***gatunummer*** och ***ort***.

Primärnyckeln har två huvudfunktioner. Den första säkerställer, vilket redan nämnts, att varje enskild post i tabellen kan identifieras unikt. Den andra funktionen är att kunna relatera tabellen till andra tabeller i databasen på ett säkert sätt, så som visas i exemplet med tabellerna "Kunder" och "Kundorder" ovan.

Primärnycklar som baseras på verkliga data, så som exempelvis ett personnummer eller ett kundnummer, kallas "*Natural primary keys*" och är mycket vanliga i tabeller i lite äldre databaser.

En modernare typ av primärnyckel kallas "Surrogate Primary Key". En surrogate primary key är ett heltalsfält (int) i tabellen vanligen kallat **ID**. För varje ny post som läggs till i tabellen automatgenererar SQL ett nytt och för tabellen unikt heltal för detta fält. Detta heltalsvärde kan inte uppdateras (ändras) och förblir alltså oförändrat så länge tabellen existerar.

För att automatgenerera heltal används nyckelordet **identity** som anges när fältet deklarerar i samband med att vi skapar tabellen. Se exempel nedan.

Nummerserien som SQL genererar för **ID**-fältet för varje ny post i tabellen börjar i regel på 1 och ökar i regel med 1. **ID**-fältet för den första posten får alltså i regel värdet 1, den andra posten värdet 2, och så vidare. Ett **ID**-värde som redan använts *återanvänds aldrig*. Det vill säga, om vi raderar en post i tabellen så kommer den raderade postens **ID**-värde aldrig att återanvändas.

Fördelen med surrogate primary keys och anledningen till att de föredras i modernare databasdesign är att till synes oföränderliga värden, så som exempelvis personnummer, undantagsvis ändå kan behöva uppdateras. Exempelvis uppdateras personnummer i samband med att en person får skyddad identitet.

Med en natural primary key hade personnumret behövt ändras i samtliga tabeller där det förekommer för att inte bryta relationerna mellan tabellerna. I en normaliserad databas med en surrogate key uppdateras personnumret endast i en enda tabell i en enda post.

Syntax för att skapa en Surrogate Primary Key:

```
create table Persons
(
    ID int primary key identity not null,
    ...
)
```

Nyckelorden **primary key** talar om att fältet måste vara unikt för varje post i tabellen. Nyckelordet **identity** talar om att SQL automatiskt ska öka på fältets värde med 1 för varje ny post som skapas i tabellen.

Unika fält

När vi arbetar med Surrogate Primary Keys behöver vi ofta ange att vissa andra fält eller kombinationer av fält i en tabell ska vara unika för varje post, så exempelvis fältet **personnummer** i en kontakttabell, eller fälten **gata**, **gatunummer** och **ort** i en adresstabell. Det vill säga de fält som annars hade använt som "Natural Primary Keys".

För att ange att ett enskilda fält ska vara unikt används nyckelordet *unique*.

Exempel:

```
create table Persons
(
    ID int primary key identity not null,
    Personnr varchar(13) unique not null, -- ÅÅÅÅMMDD-NNNN
    ...
)
```

Följande exempel visar syntaxen för att ange att en kombination av fält ska vara *unika* för varje post i en tabell:

```
create table Addresses
(
    ID int primary key identity not null,
    Street varchar (32) not null,
    City varchar (32) not null,
    ZipCode varchar (16) null,
    LastUpdated datetime not null,
    unique (Street, City, ZipCode)
)
```

Radera tabeller

Syntax för att radera en tabell:

```
drop table tabellnamn
```

Koduppgift 15

- a) Radera tabellerna Persons och Addresses i CobolDB databasen.
- b) Skapa sedan dessa två tabeller (Persons och Addresses) på nytt enligt exemplen i stycket **Unika fält** ovan. Addera fälten "FirstName varchar(32)", "LastName varchar(32)" samt "YearOfBirth int" till tabellen "Persons". Tillåt dessa fält att acceptera null-värden.
- c) Lägg till fem poster i tabellen Persons.
- d) Lägg till fem poster i tabellen "Addresses". Tips: Använd gärna samma data som du använde i *Koduppgift 5*.
- e) Säkerställ att tabellerna *Persons* och *Addresses* innehåller de fem posterna i respektive tabell som du lagt till i c) och d) genom att skriva select queries.

Boolsk algebra

Jämförelser som görs med jämförelseoperatorerna lika med "=", skilt från "!=", större än ">", och så vidare resulterar alltid i ett värde som är *sant* eller *falsk*.

Exempel:

Anta att värdet för fältet Age är 25, då kommer *jämförelseuttrycket* **Age > 18** att resultera i *sant*, eftersom 25 är större än 18, medan jämförelseuttrycket **Age < 18** kommer att resultera i *falskt*, eftersom 25 inte är mindre än 18.

I where-klausuler är det vanligt att kombinera flera jämförelseuttryck med hjälp av de logiska operatorerna **and** och **or**.

Två jämförelseuttryck som kombineras med **and** måste både resultera i *sant* för att hela uttrycket ska resultera i *sant*.

Exempel:

where

```
Age >= 25 and City = 'Stockholm'
```

I ovan exempel kommer endast de poster att visas i resultatmängden där samtliga poster har gemensamt att fältet Ages värde är större än eller lika med 25, samtidigt som fältet Citys värde är lika med 'Stockholm'.

När två jämförelseuttryck kombineras med **or** räcker det med att det ena av de två jämförelseuttrycken resulterar i *sant* för att hela uttrycket ska resultera i *sant*.

Exempel:

where

```
Age >= 25 or City = 'Stockholm'
```

I ovan exempel kommer resultatmängden att innehålla alla poster där Age är större än eller lika med 25 oavsett vad fältet City innerhåller för värde, samt alla poster där City är lika med 'Stockholm' oavsett vilket värde fältet Age innerhåller.

Följande tabell visar resultatet av samtliga kombinationer av sant och falsk som kombineras med *and* och *or*.

<i>Jmf.uttryck 1</i>	<i>Operator</i>	<i>Jmf.uttryck 2</i>	<i>Resultat</i>
falskt	and	falskt	falskt
falskt	and	sant	falskt
sant	and	falskt	falskt
sant	and	sant	sant
falskt	or	falskt	falskt
falskt	or	sant	sant
sant	or	falskt	sant
sant	or	sant	sant

*Jämförelseuttryck som kombineras med **and** evalueras alltid med högre prioritet än jämförelseuttryck som kombineras med **or**. Detta kan få oväntade konsekvenser om vi inte beaktar det.*

Anta exempelvis att vill få tag i alla *bananer* och *apelsiner* i en Fruits-tabell som kostar mindre än 30 kronor kilot. Det innebär att vår query måste innehålla tre jämförelser:

```
PricePerKg < 30.00,  
FruitType = 'Banana', samt  
FruitType = 'Orange'.
```

För att få det önskade resultatet måste vi kombinera dessa jämförelseuttryck med hjälp av *and* och *or*.

Exempel:

where

```
PricePerKg < 30.00 and  
FruitType = 'Banana' or  
FruitType = 'Orange'
```

Det önskade resultatet från vår query uteblir dock eftersom **and** evalueras *före* **or**. Resultatmängden kommer att visa bananer med ett pris lägre än 30.00, men samtliga apelsiner oavsett pris, eftersom uttrycket...

```
PricePerKg < 30.00 and  
FruitType = 'Banana'
```

...evalueras med högre prioritet än (före) uttrycket...

```
or FruitType = 'Orange'
```

För att få den önskade resultatmängden måste vi använda *parenteser*.

Jämförelseuttryck som omsluts av parenteser evalueras med högre prioritet än

Jämförelseuttryck som inte omsluts av parenteser.

where

```
Price < 30.00 and  
(FruitType = 'Banana' or  
FruitType = 'Orange')
```

Koduppgift 16

a) Skapa en tabell **Fruits** med fälten **FruitType**, **FruitName** och **PricePerKg** med lämpliga datatyper. Skapa även ett primärnyckelfält med namnet **ID** och definiera vilken kombination av fält som ska vara unika, samt vilket eller vilka fält som *ska*, respektive *inte ska*, tillåta *null*.

b) Till tabellen **Fruits**, lägg till...

- ♦ Tre poster med **FruitType** *Banana* och **FruitName** *Chiquita*, *Dole* och *Del monte*.

- ♦ Tre poster med **FruitType** *Orange* med **FruitName** *Valencia*, *Mandarin* och *Blood*.

- ♦ Två poster med **FruitType** *Pear* med **FruitName** *Anjou* och *Bartlett*.

Låt frukternas priser variera mellan 22 och 32 kronor, men låt minst en banansort och en päronsor sort kosta mer än 25 kronor.

c) Skriv en query som endast listar samtliga frukter med ett pris lägre än 30 kronor.

d) Skriv en query som endast listar samtliga apelsiner som kostar mer än 25 kronor.

e) Skriv en query som endast listar päron och bananer som kostar mer än 25 kronor.

Order by – sortering

Med hjälp av nyckelorden **order by** i en select query kan vi sortera en resultatmängd i stigande eller fallande ordning på ett eller flera fält.

Order by-klausulen anges alltid *sist* i query:n, och fastställer vilka fält som resultatmängden ska sorteras på. Om vi anger flera fält så separeras fälten med ett *kommatecken*.

Om flera fält anges så sorteras resultatmängden i första hand på det först angivna fältet, i andra hand på det andra angivna fältet, och så vidare.

Varje angivet fält i **order by**-klausulen kan antingen sorteras i stigande ordning (*ascending*) A till Ö, 1 till 9, eller i fallande ordning (*descending*) Ö till A, 9 till 1, genom att skriva nyckelorden **asc** (förvalt värde) eller **desc** efter fältnamnet.

Exempel:

```
select
    P.FirstName,
    P.LastName,
    P.YearOfBirth
from
    Persons as P
order by
    P.LastName asc, -- asc är förvalt och behöver inte anges!
    P.YearOfBirth desc
```

*Oerhört viktigt att känna till vad gäller sorteringsordning i SQL, är att sorteringsordningen aldrig är garanterad så vida vi inte anger en **Order by**-klausul.*

Vi skulle lätt kunna tro att om vi inte anger en sorteringsordning med hjälp av **order by** så skulle resultatmängden alltid visa posterna i samma ordning som de lagts till i tabellen, men detta är som sagt *inte* fallet.

Koduppgift 17

Exekvera query:n i föregående exempel (inte koduppgift). Experimentera med att sortera resultatmängden på olika fält i tabellen, samt i stigande och fallande ordning.

Koduppgift 18

Skriv en query som sorterar samtliga poster i tabellen Fruits. Sortera i första hand på fältet "FruitType" i *stigande* ordning, och sortera i andra hand på fältet PricePerKg i *fallande* ordning.

Group by, aggregeringsfunktioner och having

Anta att vi har en tabell med namnet Personal som består av all personal på ett företag, och att tabellen bland annat innehåller fält för varje anställds titel och månadslön.

Tabell: *Personal*

ID	Sektion	Titel	Namn	Lön	ChefsID
1023	1	Godsmottagare	Anna Olofsson	22450.00	1011
1020	2	Godsmottagare	Bertil Sjögren	21900.00	1011
1008	2	Innesäljare	Cecilia Danielsson	26000.00	1006
1025	1	Innesäljare	David Eriksson	25400.00	1006

I en select query kan vi med hjälp av nyckelorden *group by* skapa en resultatmängd som består av grupper av personal, exempelvis en grupp för varje unik titel.

Därefter kan vi använda någon av SQLs så kallade *aggregeringsfunktioner*, exempelvis *AVG(Lön)* för att beräkna medellönen inom respektive grupp. Vi kan också räkna antalet anställda inom respektive grupp med hjälp av aggregeringsfunktionen *COUNT(*)*, eller summera lönerna inom respektive grupp med hjälp av aggregeringsfunktionen *SUM(Lön)*.

Exempel:

Följande select query grupperar tabellen *Personal* på fältet *Titel* och beräknar respektive titel-grupps medellön med hjälp av aggregeringsfunktionen *AVG()*.

```
select
    Titel,
    AVG(Lön) -- Aggregerat fält. Medelvärde av Lön.
from
    Personal
group by -- Gruppering.
    Titel
```

Ovan exempel grupperar resultatmängden på fältet **Titel**, men vi kan gruppera resultatmängden på fler än ett fält, exempelvis fälten **Sektion** och **Titel**. Detta innebär att vi får en grupp för varje unik kombination av **Sektion** och **Titel**.

```
select
    Sektion,
    Titel,
    AVG(Lön) -- Aggregerat fält. Medelvärde av Lön.
from
    Personal
group by -- Gruppering.
    Sektion,
    Titel
```

När vi använder **group by** så måste samtliga icke aggregerade fält som vi listar i query:ns **select**-klausul även listas i query:ns **group by**-klausul. Däremot kan **select**-klausulen lista *färre* icke aggregerade fält än **group by**-klausulen.

Vi kan även villkora vilka grupper som ska visas i resultatmängden med en **having**-klausul. **Having**-klausulen anges efter **group by**-klausulen och fungerar på samma sätt som en **where**-klausul men kan till skillnad mot **where**-klausulen även inkludera aggregeringsuttryck. Följande query visar endast de grupper där medellönen understiger 15000.00.

```
select
    Titel,
    AVG(Lön) -- Aggregerat fält. Medelvärde av Lön.
from
    Personal
group by -- Gruppering.
    Titel
having -- Villkor för aggregerade fält
    AVG(Lön) < 15000.00
```

Koduppgift 19

- a) Exekvera SQL koden i filen Personal.sql
- b) Skriv en select query som visar den högsta respektive den lägsta lönen för varje **titel**-grupp samt antal anställa i respektive grupp. Tips: Använd aggregeringsfunktionerna **Max(Lön)**, **Min(Lön)** och **Count(*)**.
- c) Inkludera endast de grupper i b) uppgiften som har fler än en medlem i varje grupp. Tips: Använd en **having**-klausul.
- d) Sortera resultatmängden i c) uppgiften så att den grupp som har den högsta max-lönen visas överst i resultatmängden. Tips: Använd en **order by**-klausul sist i query:n. Lämpliga fält-alias kan användas i **order by**-klausulen vilket kan göra den tydligare att förstå. Fält-alias kan inte användas i **having**-klausulen.

Introduktion till relationsdatabaser - Quiz 6

Främmande nyckel (Eng. Foreign key)

En främmande nyckel, eller *foreign key*, är ett fält i en tabell vars värden matchar värden i ett *primary key*-fält i en annan relaterad tabell.

Exempel:

Fältet **Kundnummer** i tabellen **Kundorder** är en typisk främmande nyckel eftersom fältet innehåller värden som matchar värden i primärnyckelfältet **Kundnummer** i tabellen **Kunder**.

Med hjälp av främmande nycklar och primärnycklar kan vi i *teknisk mening* koppla samman tabellerna **Kunder** och **Kundorder** så att SQL "förstår" att tabellerna är relaterade till varandra. (främmande nycklar kan i likhet med primärnycklar vara sammansatta av flera fält).

Relationen mellan tabellerna **Kunder** och **Kundorder** är en typisk så kallad *en-till-många-relation*. Det vill säga, *en kund kan ha många order, men en order kan bara tillhöra en bestämd kund* (inte många olika kunder).

Det är *inte obligatoriskt* att skapa främmandencklar som relaterar till primärnycklar i andra tabeller för att kunna koppla samman tabellerna i *select* queries, vilket görs med hjälp av nyckelordet **join** som vi strax kommer att bekanta oss med.

Anledningen till att vi ändå vill skapa relationer mellan tabeller med hjälp av primär- och främmandencklar är att SQL då kan säkerställa att posterna i de två relaterade tabellerna alltid relaterar till varandra på ett logiskt och korrekt sätt.

Om vi skapar en relation mellan tabellerna **Kunder** och **Kundorder** med hjälp av primär- och främmandenycklar så innebär det att SQL inte kommer att tillåta oss att lägga till poster i tabellen **Kundorder** om det kundnummer vi anger inte redan existerar i tabellen **Kunder**, och att SQL inte kommer att tillåta oss att radera poster i tabellen **Kunder** om det existerar poster i tabellen **Kundorder** som hör ihop med kunden.

Följande exempel visar syntaxen för att skapa en relation mellan tabellerna **Kunder** och **Kundorder** med hjälp av primär- och främmandenycklar:

```
create table Kunder
(
    -- PRIMARY KEY
    Kundnummer int primary key not null,

    -- Övriga fält...
)
create table Kundorder
(
    -- PRIMARY KEY
    Ordernr int primary key not null,

    -- FOREIGN KEY
    Kundnummer int references Kunder(Kundnummer) not null,

    -- Övriga fält...
)
```

Av exemplet ovan framgår att främmandenyckelfältet **Kundnummer** i tabellen **Kundorder** refererar till primärnyckelfältet **Kundnummer** i tabellen **Kunder**.

Koduppgift 20

a) Skapa en tabell i databasen CobolDB med namnet **Kontaktuppgifter** som innehåller fälten:

ID (surrogate primary key),

Kontakttyp (max 8 tecken),

Kontaktuppgift (max 32 tecken), samt

PersonsID (Främmandenyckel till **ID** fältet i tabellen **Persons**).

Inga fält förutom fältet **PersonsID** ska acceptera **null**-värden. Samtliga värden i fältet **Kontaktuppgift** ska definieras som unika.

b) Lägg till tre till fem poster i tabellen *Kontaktuppgifter*. Låt en person ha minst två kontaktuppgifter. Exempel:

```
insert into
    Kontaktuppgifter(Kontakttyp, Kontaktuppgift, PersonsID)
values
    ('E-mail', 'hakan@kvarnskogen.se', 1),
    ('Telefon', '070 464 74 31', 1)
```

c) Försök att lägga till en post i tabellen *kontaktuppgifter* med ett *PersonsID*-värde som inte finns i tabellen *Persons*. Vad får vi för felmeddelande? Varför?

d) Försök att lägga till en post i tabellen *Kontaktuppgifter* med ett giltigt *PersonsID* men med en *Kontaktuppgift* som redan finns i tabellen *Kontaktuppgifter*. Vad får vi för felmeddelande? Varför?

e) Försök att radera en post i tabellen *Persons* som har minst en kontaktuppgift. Vad får du för felmeddelande? Varför?

Tabellrelationer och join

Anta att tabellen *Persons* innehåller följande poster:

ID	Personnr	FirstName	LastName	YearOfBirth
7	19620601-1234	Håkan	Johansson	1962
8	19760809-1234	Marilyn	Johansson	1976
9	20091205-1234	Kenneth	Johansson	2009

Anta att tabellen *Kontaktuppgifter* innehåller följande poster:

ID	Kontakttyp	Kontaktuppgift	PersonsID
11	Telefon	070 464 74 31	7
12	E-post	hakan@kvarnskogen.st	7
13	Telefon	073 839 44 13	8
14	Telefon	08 123 45 67	Null

Som framgår av tabellerna har “Håkan” med *ID* 7 två kontaktuppgifter i tabellen *Kontaktuppgifter* vilket framgår av fältet *PersonsID*. “Marilyn” har enligt samma mönster en (1) kontaktuppgift, medan “Kenneth” helt saknar kontaktuppgifter.

Observera att kontaktuppgiften med *ID* 14 saknar relation till tabellen *Persons* (*PersonsID* is null) och alltså är ett “ledigt telefonnummer” som skulle kunna tilldelas en person genom att uppdatera värdet i fältet *PersonsID* för *ID* 14.

För att sammanfoga (joina) tabellerna *Persons* och *Kontaktuppgifter* i en *select* query använder vi nyckelorden **join on**.

Följande exempel join:ar tabellerna *Persons* och *Kontaktuppgifter* så att vi kan lista personer och deras kontaktuppgifter i en och samma resultatmängd.

```
select
    *
from
    Persons join Kontaktuppgifter on
    Persons.ID = Kontaktuppgifter.PersonsID
```

join kopplar samman tabellerna *Persons* och *Kontaktuppgifter*. I exemplet säger vi att de poster i tabellerna *Persons* och *Kontaktuppgifter* som har samma värde i fälten *Persons.ID* respektive *Kontaktuppgifter.PersonID* hör ihop och ska kopplas samman.

Koduppgift 21

- a) Radera samtliga poster i tabellen *Kontaktuppgifter*.
- b) Radera samtliga poster i tabellen *Persons*. Kan vi radera dessa poster innan vi raderar posterna i tabellen *Kontaktuppgifter*?
- c) Infoga de poster som visas under rubriken “*Tabellrelationer*” ovan i respektive tabell *Persons* och *Kontaktuppgifter*.
- d) Exekvera ovan **select** query. Varför visas inte “Kenneth” med ID 9, och varför visas inte kontaktuppgiften med ID 14? Svaret ges nedan!

Kännetecknade för **join** är att **join** endast inkluderar de poster i resultatmängden där värdena i de fält som kopplar samman tabellerna matchar varandra (*Persons.ID = Kontaktuppgifter.PersonsID*).

Det vill säga, om det i tabellen *Persons* finns en person som inte har någon kontaktuppgift (“Kenneth”) så kommer den personen att exkluderas från resultatmängden, och omvänt gäller att om det i tabellen *Kontaktuppgifter* finns en kontaktuppgift som inte har någon innehavare (*Kontaktuppgifter.PersonsID is null*) så kommer den kontaktuppgiften att exkluderas från resultatmängden.

Koduppgift 22

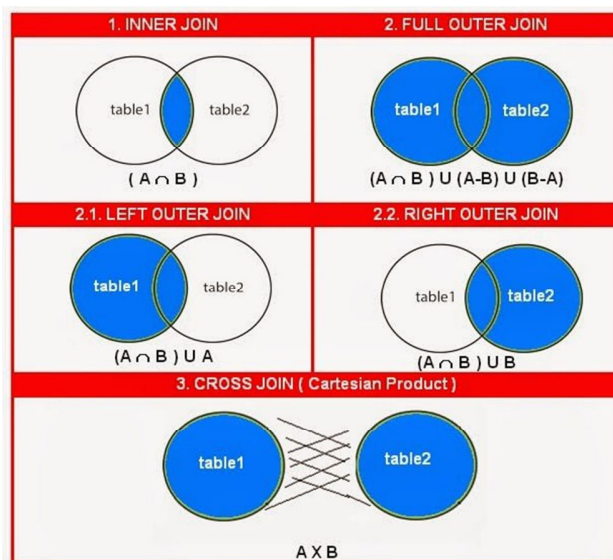
Skriv en **select** query med hjälp av en **join** som listar samtliga fält från tabellen *Persons* och samtliga fält utom fältet *PersonsID* från tabellen *Kontaktuppgifter*.

Tips: Vi kan lista samtliga fält från en viss tabell genom att skriva *tabellnamn.**

Olika typer av JOINS

Det finns sammanlagt fem olika typer av JOINS.

1. **join** (kan även skrivas *inner join*)
2. **left join** (kan även skrivas: *left outer join*)
3. **right join** (kan även skrivas *right outer join*)
4. **full join** (kan även skrivas *full outer join*)
5. **cross join** (kan även skrivas ,)



För att på enklast tänkbara sätt illustrera skillnaden mellan dessa olika joins ska vi fortsätta att använda tabellerna "Persons" och "Kontaktuppgifter". Tabellen ***Persons*** utgör vår *huvudtabell* och tabellen ***Kontaktuppgifter*** utgör vår *detaljtabell*.

Det finns med andra ord en *en-til-många-relation* mellan tabellerna ***Persons*** och ***Kontaktuppgifter***. Det vill säga en person kan inneha noll, en eller *många* kontaktuppgifter, medan en kontaktuppgift endast kan innehas av *en* bestämd person, eller ingen person.

Join

Den join vi använde i koduppgift 21 och 22 kallas för en "*inner join*" och vi skulle uttryckligen kunnat skriva just ***inner join*** i stället för just bara ***join***. Det finns dock inga som helst praktiska skillnader mellan att skriva ***join*** eller ***inner join***.

Left join

De två tabeller vi kopplar samman med en **join** kallas för vänster **left** respektive höger **right** tabell, där den vänstra tabellen avser den tabell vi anger till vänster om nyckelordet **join** (direkt efter nyckelordet **from**) och den högra tabellen avser den tabell vi anger till höger om nyckelordet **join**. I koduppgift 21 och 22 räknas alltså vår huvudtabell **Persons** som *vänster* tabell medan vår detaljtabell **Kontaktuppgifter** räknas som *höger* tabell.

En **left join** inkluderar samtliga poster från den vänstra tabellen (**Persons**) i resultatmängden, men endast de poster i den högra tabellen (**Kontaktuppgifter**) som matchar poster i den vänstra tabellen. Det vill säga, med hjälp av en **left join** kan vi lista samtliga personer oavsett om de innehar en kontaktuppgift eller inte.

Koduppgift 23

- a) Skriv en **select** query som listar samtliga fält från tabellen **Persons** och samtliga fält utom fältet **PersonsID** från tabellen **Kontaktuppgifter** med hjälp av en **left join**.
- b) Skriv en **select** query som endast listar de personer i tabellen **Persons** som helt saknar kontaktuppgifter. Lista endastfälten **FirstName** och **LastName** i resultatmängden. Tips: Lägg till en **where**-klausul som gör en null jämförelse.

Right join

En **right join** fungerar på precis samma sätt som en **left join**, men inkluderar istället *samtliga* poster från den *högra* tabellen (**Kontaktuppgifter**) i resultatmängden, men endast de poster i den vänstra tabellen (**Persons**) som matchar poster i den högra tabellen. Det vill säga, med hjälp av en **right join** kan vi lista *samtliga* kontaktuppgifter oavsett om kontaktuppgiften har en innehavare eller inte.

Koduppgift 24

- a) Skriv en **select** query som listar samtliga fält från tabellen **Persons** och samtliga fält från tabellen **Kontaktuppgifter** med hjälp av en **right join**.
- b) Lägg till en **where**-klausul i query:n som exkluderar alla personer som innehar en kontaktuppgift. Lista slutligen endastfälten **Kontaktuppgifter.ID** samt **Kontaktuppgifter.Kontaktuppgift**. TIPS: Gör en *null jämförelse* i where-klausulen.

Full join

En **full join** fungerar som en kombination av en **left join** och en **right join**. Det vill säga, samtliga poster från båda tabellerna inkluderas i resultatmängden och de poster som matchar varandra kopplas samman.

Koduppgift 25

- a) Skriv först en query som listar samtliga fält från tabellen Persons och tabellen Kontaktuppgifter med hjälp av en **full join**.
- b) Lägg därefter till en where klausul som exkluderar samtliga personer respektive samtliga kontaktuppgifter som *inte* har någon matchning i respektive tabell.
- c) Skriv och exekvera en **update query** som tilldelar den lediga kontaktuppgiften till den person som saknar kontaktuppgift. Exekvera åter igen query:n i koduppgift 25 a) för att säkerställa att uppdateringen fungerat.

Cross join

Join-typen **cross join** finns med i moderna versioner av SQL av historiska skäl och behöver inte längre användas, men kan vara bra att känna till.

En **cross join** skiljer sig från övriga joins genom att den inte definierar vilka fält som skall användas för att koppla samman två tabeller. D.v.s. en **cross join** har ingen *on*-del. En **cross join** är därför i sig själv inte användbar om vi inte kombinerar den med en *where*-klausul.

En **cross join** kopplar samman samtliga poster i den vänstra tabellen med samtliga poster i den högra tabellen. Det innebär att antalet poster i resultatmängden kommer att bestå av antalet poster i den vänstra tabellen, *gånge* antalet poster i den högra tabellen. Om den vänstra tabellen består av exempelvis 10 poster och den högra tabellen av 5 poster så kommer resultatmängden att bestå av 10 x 5 poster = 50 poster. Detta är som redan nämnts i sig själv inte användbart och skälet till varför en **cross join** alltid måste kombineras med en *where*-klausul.

Nyckelorden **cross join** kan ersättas av ett kommatecken ", " mellan de två tabellerna.

I nedan exempel visas en cross join som ger samma resultatmängd som en inner join.

```
select
    *
from
    Persons, Kontaktuppgifter -- cross join
where
    Persons.ID = Kontaktuppgifter.PersonID
```

Olika typer av tabellrelationer

Den vanligaste typen av relation mellan två tabeller är en *en-till-många-relation*, men det finns sammanlagt fyra typer av tabellrelationer. Dessa är:

1. **En-till-en**
2. **En-till-många**
3. **Många-till-många**
4. **Självrefererande**

Tabellrelationen *En-till-en*

Relationen “*en-till-en*” innebär att vi delar upp en tabells fält i två (eller flera) tabeller. Ett exempel skulle kunna vara en tabell med *produkter* som vi delar upp i två tabeller. Det ena tabellen skulle då kunna innehålla fält för produkternas ekonomiska fält (produkter_ek) medan den andra tabellen skulle kunna innehålla fält för artikelns övriga egenskaper (produkter_övr). Båda tabellerna skulle då ha samma primärnyckelvärde (ID) för respektive produkt, och kopplas samman på detta fält.

Exempel:

Tabell: *Produkter_Övr*

ID	Produktnr	Produktbeskr.	Lagersaldo
1	83740	Snus	150
2	64066	Laptop	15

Tabell: *Produkter_Ek*

ID	Inköpspris	Försäljningspris
1	18.50	37.00
2	2500.00	5990.00

För att få ut en resultatmängd med samtliga fält från respektive tabell kan vi skriva följande query:

```

select
    *
from
    Produkter_övr
join
    Produkter_Ek on
    Produkter_övr.ID = Produkter_Ek.ID

```

Relationen *en-till-en* är ovanlig. En teknisk anledning till att skapa tabeller med en *en-till-en-relation* skulle kunna vara om tabellen har så många fält att de inte ryms i enda tabell. Det maximala antalet fält som ryms i en tabell i SQL Server är 1024 fält.

Tabellrelationen *En-till-många*

Tabellrelationen *en-till-många* är den i särklass vanligaste tabellrelationen och vi har redan sett exempel på den med hjälp av tabellerna ***Persons*** och ***Kontaktuppgifter***.

Tabellrelationen *Många-till-många*

Tabellrelationen *många-till-många* mellan två tabeller är en vanligt förekommande tabellrelation och fordrar alltid en *sammanlänkande tredje tabell*.

Denna sammanlänkande tredje tabell består huvudsakligen av två främmandenyckelfält, där respektive främmandenyckelfält innehåller primärnyckelvärdet från de två tabeller som länkas samman i en *många-till-många* relation.

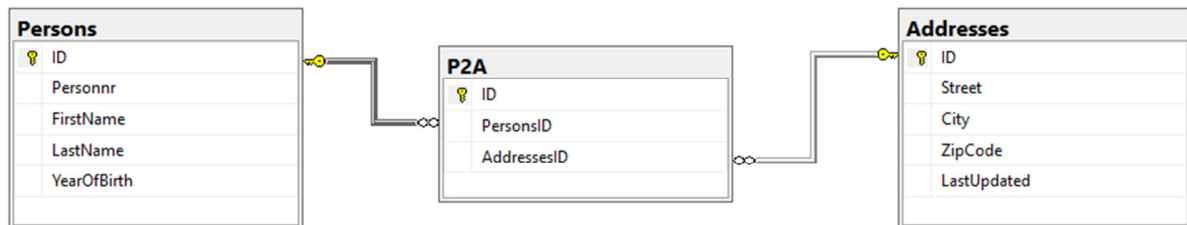
I kod implementeras alltså en *många-till-många* relation som två stycken *en-till-många* relationer.

För att lättare förstå tabellrelationen *många-till-många* ska vi skapa en *många-till-många* relation mellan tabellerna ***Persons*** och ***Addresses*** med hjälp av en tredje tabell som vi kallar ***P2A (Persons To Addresses)***.

Tabellen ***P2A*** består huvudsakligen av de två fälten ***PersonsID*** och ***AddressesID*** som båda är främmandenycklar till primärnyckelfälten ***ID*** i tabellerna ***Persons*** respektive ***Addresses***.

Många-till-många relationen i detta exempel betyder alltså att en person kan inneha många adresser, och att en adress kan innehas av många personer.

I diagramform ser relationerna mellan de tre tabellerna **Persons**, **P2A**, samt **Addresses** ut på följande sätt:



Koduppgift 26

a) Skapa tabellen **P2A** enligt följande exempel:

```
create table P2A
(
    ID int primary key identity not null,
    PersonsID int references Persons(ID) not null,
    AddressesID int references Addresses(ID) not null,
    unique(PersonsID, AddressesID)
)
```

Observera att primärnyckelfältet **ID** i tabellen **P2A** inte har någon betydelse för många-till-många relationen mellan tabellerna **Persons** och **Addresses**, utan endast är en så kallad *best practice*. Primärnyckelfältet **ID** i tabellen **P2A** finns i detta sammanhang bara till för den händelse att vi i framtiden skulle behöva koppla tabellen **P2A** till ytterligare någon tabell.

b) Skapa sex nya poster i tabellen **P2A** via en **insert** query, för att på så sätt skapa relationer mellan personer och adresser.

Para ihop *Håkan* med tre valfria adresser.

Para ihop *Marilyn* med två adresser, varav den ena ska vara gemensam med *Håkan* och den andra inte.

Para inte ihop *Kenneth* med någon av adresserna.

Tips: Innan du skriver **insert** query:n för **P2A**, lista först samtliga poster i tabellen **Persons** och samtliga poster i tabellen **Addresses** (**select * from Persons select * from Addresses**) så att du kan se vilket **ID** som respektive person och adress har.

Koduppgift 27

- a) Skriv en *select* query med en *join* mellan tabellerna *Persons* och *P2A*. Lista samtliga fält från de båda tabellerna.
- b) Skriv en *select* query med en *join* mellan tabellerna *Addresses* och *P2A*. Lista samtliga fält från de båda tabellerna.
- c) Kopiera query:n i koduppgift 27 a) och lägg till en *join* till tabellen *Addresses*. Lista samtliga fält, utom fälten i tabellen *P2A*.
- d) Kopiera query:n i koduppgift 27 c) och lägg till en *join* till tabellen *Kontaktuppgifter*. Lista samtliga fält utom fälten i tabellen *P2A*.
- e) Lägg till en *where*-klausul till koduppgift 27 c) som visar vilka adresser som innehas av en bestämd person (använd fältet *FirstName*) i tabellen *Persons*. Sortera slutligen resultatmängden i fallande ordning på fältet *ZipCode*.

Tabellrelationen *Självrefererande*

Tabellrelationen *självrefererande* är liksom tabellrelationen *en-till-en* inte särskilt vanligt förekommande men ändå användbar att känna till.

För att ge exempel på denna relation använder vi tabellen *Personal*. Tabellen *Personal* listar alla anställda på företaget och vem som är chef för varje anställd.

För att en tabell skall kunna vara självrefererande, d.v.s. kunna kopplas till sig själv med en *join*, så *måste* vi använda *tabell-alias*.

Koduppgift 28

Skriv en *select* query som listar alla poster och alla fält i tabellen *Personal*. Sortera sedan resultatmängden på fältet *ID*.

Koduppgift 29

Kopiera query:n i koduppgift 28 och lägga till en *join* till tabellen **Personal** (samma tabell som i *from-delen*). Ge tabellerna ett tabell-alias, exempelvis **P** för **Personal** respektive **C** för **Chef**. *Join*:a tabellerna på (*on*) fälten **P.ChefsID** respektive **C.ID**.

Lista samtliga fält i tabellen **P** och samtliga fält i tabellen **C**. Separera listningen av fälten från respektive tabell med ett fält som utgörs av en *strängkonstant*, exempelvis: `'***' as [/]`, (Obs! Mellanslagstecken mellan hakparenteserna)

Sortera resultatmängden i stigande ordning i första hand på fältet **C.ID** och i andra hand på fältet **P.Titel**.

Koduppgift 30

Eftersom vi använder en vanlig (*inner*) *join* i koduppgift 29 så listas *inte* den personal som saknar chef. Kopiera query:n i koduppgift 29 och ändra query:n så att även den personal som saknar chef listas i resultatmängden. Tips: Byt ut *join* mot *en annan typ av join*.

Introduktion till relationsdatabaser - Quiz 7

Scheman

När vi skapar databasobjekt i databasen så som tabeller, så placeras de förvalt och automatiskt i ett så kallat schema. Förvalt så använder SQL schemat med namnet *dbo* (*Database Owner*).

Scheman används för att gruppera databasobjekten i logiska grupper. Vid behov kan vi skapa egna scheman och placera våra databasobjekt i dem när vi skapar databasobjekten.

Vi skulle exempelvis kunna skapa ett schema med namnet *EkAvd* och placera tabeller med namn så som exempelvis *Konton*, *Fakturor*, o.s.v. i detta schema.

Syntaxen för att skapa ett schema i SQL är:

```
create schema schemanamn
```

Syntaxen för att placera ett databasobjekt, exempelvis en tabell, i ett bestämt schema är:

```
create table schemanamn.tabellnamn
```


Med hjälp av scheman kan vi skapa tabeller med exakt samma namn i samma databas så länge de placeras i unika scheman. När vi sedan skriver queries med dessa tabeller måste vi dock komma ihåg att alltid skriva schemanamnet följt av en punkt framför tabellnamnet. Exempelvis:

```
select
    *
from
    HR.Persons -- Tabellen Persons skapad i schemat HR.
order by
    HR.Persons.LastName -- Schema.Tabell.Fält.
```

Vyer (Views)

I SQL kan vi namnge och lagra *select* queries och sedan använda dessa namngivna *select* queries som om de vore tabeller i databasen. En namngiven *select* query kallas vy eller på engelska, *view*.

Med hjälp av vyer kan komplexa queries brytas ner i mindre delar vilket kan underlätta skapandet av den slutgiltiga queryn.

Följande exempel skapar en vy med namnet **Sektion1** baserad på tabellen **Personal**. Denna vy listar endast anställda som tillhör sektionen 1.

```
create view Sektion1 as
select
    *
from
    Personal as P
where
    P.Sektion = 1
```

När vi har sparat vyn i databasen kan vi använda vyn precis som om den vore en tabell och skriva andra *select* queries som använder vyn. Exempelvis:

```
select
    Count(*) as [Antal personer],
    Max(Lön) as [Högsta lön],
    Min(Lön) as [Lägst lön]
from
    Sektion1 -- Använd vyn som en tabell.
```

Observera att en vy alltid hämtar aktuella data från de tabeller som vyn använder och alltså inte kopierar data till någon särskild "vy-tabell" när vyn skapas.

Om vi så önskar eller har behov av det kan vi skapa alias för de fält som vyn listar när vi skapar vyn, antingen på vanligt sätt som beskrivs under rubriken *fält-alias*, eller med hjälp av syntaxen i nedan exempel:

```
create view Sektion2 (Title, [Name], Salary) as
select
    Titel,
    Namn,
    Lön
from
    Personal as P
where
    P.Sektion = 2
```

Koduppgift 31

a) Skapa en vy med namnet **Adressbok** baserad på queryn i koduppgift 27 d). Lista endast fälten: *Personnr, FirstName, LastName, Street, City, ZipCode, Kontakttyp, och Kontaktuppgift*.

b) Skriv en query som använder vyn Adressbok och grupperar resultatmängden på fälten: *FirstName, LastName*, och *Kontaktuppgift*, samt sorterar resultatmängden i stigande ordning på i 1:a hand *LastName* och i 2:a hand på *FirstName*.

Koduppgift 32

a) Kopiera och modifiera vyn Adressbok (koduppgift 31 a)) så att samtliga listade fält använder svenska fältnamn via fält-alias på det sätt som visas i föregående exempel.

Tips: För att ändra en befintlig vy i databasen, byt ut "create view" mot "alter view", och läs in vyn igen (markera vyn och exekvera).

b) Kopiera och modifiera query:n i koduppgift 31 b) så att den fungerar med den modifierade vyn.

Lagrade procedurer (Stored Procedures)

Liksom alla andra programmeringsspråk innehåller SQL syntax för att skapa namngivna block av kodrader som kan köras (exekveras) genom att anropa kodblockets namn. I SQL kallas dessa kodblock *lagrade procedurer*, eller på engelska *stored procedures*.

Den viktigaste fördelen med lagrade procedurer är att vi inte behöver upprepa samma kodrader många gånger för att göra samma sak många gånger.

SQL innehåller ett stort antal fördefinierade procedurer, så som exempelvis *sp_help*, *sp_rename*, *sp_helpindex*, *sp_helpdb*, och så vidare.

För att anropa en procedur skriver vi nyckelordet *execute* följt av den lagrade procedures namn, exempelvis: *execute sp_helpdb*.

Koduppgift 33

Exekvera den lagrade proceduren *sp_helplanguage*.

Vi kan även definiera och lagra egna procedurer i databasen. Syntaxen för att skriva en lagrad procedur ser ut enligt följande:

```
create procedure ProcedureName as
begin
    -- Kod som proceduren ska exekvera.
end
```

Följande exempel visar hur vi kan skapa och köra en procedur som visar all personal i tabellen *Personal* som har titeln *Lagerarbetare*.

```
create procedure LagArb as
begin
    select
        Namn,
        Lön
    from
        Personal
    where
        Titel = 'Lagerarbetare'
end
go

execute LagArb
```

Koduppgift 34

a) Skapa en lagrad procedur med namnet **HighestWages** visar anställda i tabellen **Personal** som har en lön som överstiger 30000.00 kronor sorterad på fältet **Lön** i fallande ordning. *Tips:* Skriv först queryn som den lagrade proceduren ska innehålla.

b) Exekvera den lagrade proceduren **HighestWages**.

För att få ökad flexibilitet kan lagrade procedurer arbeta med så kallade *in-parametrar*. Detta innebär att en procedur kan *ta emot* data-värden när proceduren exekveras. Dessa data-värden kan sedan *användas i proceduren*.

SQL kan även arbeta med så kallade *ut-parametrar* vilket vi dock inte kommer att gå igenom i denna introduktion.

Följande exempel visar en lagrad procedur som tar emot ett data-värde via en in-parameter med namnet **@titel** och datatypen **varchar(32)**, och sedan använder detta värde i en query i proceduren. Fördelen med detta är att vi kan bestämma vilken titel personalen ska ha i resultatmängden när vi exekverar proceduren. Observera att parameternamn (och variabelnamn) i SQL alltid inleds med ett *alfatecken* ("snabel-a") **@**.

```
create procedure PersonalMedTitel @titel varchar(32) as
begin
    select
        Namn,
        Lön
    from
        Personal
    where
        Titel = @titel
end
go

execute PersonalMedTitel 'Innesäljare'
```

Koduppgift 35

- a) Använd in-parametrar för att skapa en lagrad procedur med namnet *AverageSalary* som listar medellönen för alla anställda i tabellen *Personal* som har en gemensam titel och en gemensam sektion. Observera att parametrarna ska ha samma datatyper som fälten *Titel* och *Sektion* har i tabellen *Personal*. Separera den lagrade procedures in-parametrar med kommatecken.
- b) Exekvera den lagrade proceduren *AverageSalary* med parametervärdena 'Lagerarbetare' och 1.
- c) Visa medellönen för all personal med titeln 'Chaufför' som tillhör sektion 3.

Koduppgift 36

Skapa och exekvera en lagrad procedur med namnet *AddFruit* som lägger till en ny frukt i tabellen *Fruits* och som därefter listar samtliga frukter i den tillagda fruktens frukttyp. Anroparen av proceduren ska ange den tillagda fruktens namn, typ, och pris per kilogram.

Koduppgift 37

Koduppgift 37

Kopiera och ändra proceduren i koduppgift 35 a) så att den istället för medellön visar antalet anställda med en gemensam titel och sektion.

Tips: För att uppdatera en befintlig lagrad procedur i databasen efter att den redigerats, ersätt *create procedure* med *alter procedure*, och markera sedan som vanligt koden för proceduren och exekvera (Ctrl+ Shift +E).

Introduktion till relationsdatabaser - Quiz 8

Lagrade funktioner (Stored Functions)

Lagrade funktioner påminner i sin syntax om lagrade procedurer. Till skillnad mot lagrade procedurer, som ofta returnerar en resultatmängd, så används lagrade funktioner enbart för att returnera *ett* specifikt, ofta beräknat eller aggregerat värde.

Lagrade funktioner kan anropas och användas överallt där vi normalt använder ett fältnamn, exempelvis i *select*-klausulen i en *select* query.

Lagrade funktioner har följande syntax:

```
create function FunktionsNamn ([ev. in-parametrar...]) returns datatype
begin
    return
    -- returvärde
End
```

Syntaktiskt så används lagrade funktioner som om de vore fält. De enda skillnaderna är att funktionens schema följt av en punkt alltid måste anges framför funktionsnamnet och att funktionsnamnet alltid följs av parenteser och *eventuella* parametervärden:

```
select
    dbo.FunktionsNamn(Ev. parametervärden)
```

När vi skapar en funktion använder vi nyckelordet `function` istället för `procedure`. Vidare är det *obligatoriskt* att eventuella in-parametrar anges *inom parenteser*, och datatypen för funktionens returvärde måste deklarerars (`returns datatype`). Allra sist i funktionen använder vi nyckelordet `return` följt av det returvärde som funktionen ska returnera.

Följande exempel visar hur vi skapar en lagrad funktion som beräknar cirkelarean baserad på en viss radie:

```
create function CircleArea(@radius float) returns float
begin
    return
        @radius * @radius * PI()
end
```

Observera att `PI()` är inbyggd lagrad funktion i SQL Server!

En speciell egenhet med just lagrade funktioner är att vi alltid måste ange det schema som funktionen tillhör när vi anropar funktionen men inte när vi skapar den. Om vi inte uttryckligen anger ett schema när funktionen skapas så placeras den förvalt (default) i schemat *dbo*.

Följande query anropar den lagrade funktionen *CircleArea* för att visa arean på en cirkel med radien 1.5.

```
select
    dbo.CircleArea(1.5)
```

Följande exempel visar en lagrade funktioner som returnerar medelpriset per kg för en viss frukttyp i tabellen **Fruits**:

```
create function AveragePrice (@FruitType varchar(16)) returns money
begin
    declare
        @returnvalue money -- Local variable declaration.

    select
        @returnvalue = AVG(F.PricePerKg) -- Assign return value.
    from
        Fruits as F
    where
        F.FruitType = @FruitType

    return
        @returnvalue -- Return function result.
end
go
```

Även om det inte är obligatoriskt så är det praxis att deklarerar en lokal variabel som tilldelas det värde som funktionen slutligen kommer att returnera, i vårt exempel:

```
declare
    @returnvalue money
```

Den lokala variabeln tilldelas ett värde i funktionens **select** query:

```
select
    @returnvalue = AVG(F.PricePerKg)
```

Funktionsresultatet returneras slutligen med hjälp av nyckelordet **return**.

```
return
    @returnvalue
```

Följande query anropar den lagrade funktionen **AveragePrice** för att visa medelpriset per Kg för frukttypen apelsiner (Orange).

```
select
    dbo.AveragePrice('Orange')
```

Koduppgift 38

a) Skapa en lagrad funktion med namnet *AvgWage* som tar en in-parameter av datatypen *varchar(32)* och som returnerar ett värde av datatypen *money*. Funktionen ska returnera *medellönen* för samtliga personer i tabellen *Personal* som via in-parametern har en gemensam *titel*.

b) Använd tabellen *Personal* och den lagrade funktionen *AvgWage* för att skriva en *select* query som listar *Namn*, *Lön*, samt *Medellön* (använd fält-alias!) för den titelgrupp som respektive personen i tabellen *Personal* tillhör. Sortera resultatmängden på fältet *Namn*. När du kör *select* queryn ska resultatmängden se ut enligt följande:

<i>Namn</i>	<i>Lön</i>	<i>Medellön</i>
Christer Olofsson	13450	13450
Christer Sjögren	10000	9700
Claes Danielsson	15800	15450
Håkan Johansson	10000	10000
Håkan Lundberg	21000	21000
Ingela Buskas	14500	14500
Jakob Werkelin	22400	23075
Jennie Larsson	13900	15450
Jerry Söderberg	6000	11850
Johan Alm	12000	10162,5
Och så vidare...		

Tips: Ange fältet *Titel* som parameter till funktionen *AvgWage*.

Triggers

Triggers är *en speciell typ av lagrad procedur* som exekverar automatiskt när något av DML-kommandona *insert*, *update* och/eller *delete* exekveras.

Syftet med triggers är att automatiskt kunna upprätthålla affärslogik i databasen, allt från enkla regler så som att namn måste registreras med stor inledande bokstav, till att uppdatera data i en mängd olika tabeller som affärsmässigt påverkas av den tabell som förändrats. Om exempelvis lagersaldot för en produkt uppdateras så att det ökar eller minskar, så kan en trigger i sin tur uppdatera tabeller som reflekterar ekonomiska värden och tabeller som genererar inköpssignaler, och så vidare.

Nästa koduppgift demonstrerar hur en tabell med namnet *HistoricalPrices* automatiskt håller reda på historiska priser för produkter när produkternas aktuella priser uppdateras i tabellen *Products*.

Koduppgift 39

a) Skapa tabellen *Products* enligt nedan:

```
create table Products
(
    ID int identity primary key not null,
    ProductName varchar(32) not null,
    Price money not null
)
```

b) Skapa tabellen *HistoricalPrices* enligt nedan:

```
create table HistoricalPrices
(
    ID int identity primary key not null,
    ProductsID int references Products(ID) not null,
    ChangeDate datetime not null,
    NewPrice money not null,
    OldPrice money not null
)
```

c) Skapa triggern *Products_update* enligt nedan:

```
create trigger Products_update on Products for update as
begin
    if update(Price)
        insert into
            HistoricalPrices
        select
            D.ID,
            GETDATE(),
            I.Price,
            D.Price
        from
            inserted as I
        join
            deleted as D on
            I.ID = D.ID
end
```

d) Lägg till tre nya poster i tabellen *Products* med hjälp av en *insert* query.

e) Ändra priset på *en* av produkterna i tabellen **Products** med hjälp av en *update* query och kontrollera sedan vad som hänt i tabellen **HistoricalPrices** med en *select* query.

f) Höj priset med 10 % på samtliga produkter i tabellen **Products** med hjälp av en *update* query och kontrollera sedan vad som hänt i tabellen **HistoricalPrices** med en *select* query som sorterar resultatmängden i första hand på fältet **ProductsID** och i andra hand på fältet **ChangeDate**.

Vi ska nu titta närmare på triggern **Products_update**'s olika delar.

create trigger Products_update ...

Denna del av triggern talar om att vi vill skapa en trigger med namnet **Products_update**. Praxis är att döpa triggern efter den tabell och det DML kommando (*insert*, *update*, *delete*) som den "triggar på". Denna trigger ska alltså exekvera automatiskt när vi gör en *update* på tabellen **Products**.

... **on** Products ...

Talar om att triggern ska vara knuten till tabellen Products.

... **for update** ...

Talar om att triggern enbart ska exekvera när en *update* görs på tabellen. Om vi istället skulle vilja att triggern exekverar när vi gör en *insert* eller *delete* på tabellen kan vi byta ut nyckelordet *update* mot *insert* eller *delete*.

Vi kan även skapa flera triggers mot samma tabell där en trigger ansvarar för *update*, en annan trigger för *insert*, och en tredje trigger för *delete*.

En och samma trigger kan *dessutom* knytas till flera DML kommandon genom att ange flera DML kommandon separerade med ett kommatecken, exempelvis:

for update, **insert**

Villkoret ...

if update(Price)

... returnerar *sant* om värdet i just fältet **Price** förändrats när *update* query:n mot tabellen **Products** exekveras, vilket i så fall indikerar att *insert* query:n i triggern ska exekvera. Om värdet i fältet **Price** inte uppdaterats returnerar villkoret istället *falskt* vilket indikerar att *insert* query:n i triggern *inte* ska exekvera. Fördelen är att triggern inte behöver exekvera sin *insert* query om det exempelvis endast är fältet

ProductName som förändrats.

Insert query:n i triggern **join:ar** tabellerna **inserted** och **deleted**. Dessa två tabeller skapas och underhålls automatiskt av SQL och existerar endast tillfälligt medan triggern exekverar.

Tabellen **inserted** innehåller den nya datan för de poster som uppdaterats. Det vill säga den data som de uppdaterade posterna har i tabellen **Products** efter uppdateringen.

Tabellen **deleted** innehåller den gamla datan för de poster som uppdaterats. Det vill säga den data som de uppdaterade posterna hade i tabellen **Products** före uppdateringen.

Genom att använda tabellerna **inserted** och **deleted** kan vi infoga nya poster i tabellen **HistoricalPrices** med både nya och gamla data.

Triggers som knyts till DML kommandot **insert** har bara tillgång till triggertabellen **inserted**, medan triggers som knyts till DML kommandot **delete** bara har tillgång till triggertabellen **deleted**. Triggers som knyts till DML kommandot **update** har tillgång till båda triggertabellerna **inserted** och **deleted**, eftersom en **update** både raderar och infogar nya data.

Introduktion till relationsdatabaser - Quiz 9