SQL

SQL (Structured Query Language) är ett standardiserat frågespråk som är nära knuten till relationsmodellen och som är baserad på de relationsoperatorerna vi tittade på i lektion 2. Det är ett komplett frågespråk och stöds av de flesta relationsdatabasleverantörer. Vi kan använda det här frågespråket för att:

- Skapa databaser, tabeller och vyer - Data Definition Language (DDL)
- Infoga, uppdatera och ta bort data från tabeller - Data Manipulation Language (DML)
- Gör enkla och komplexa frågor för att hämta data från tabeller (DML)

SQL innehåller inte kontrollstrukturer som vi känner till från mer traditionella datorspråk. SQL-frågor består av vanliga engelska ord som: CREATE TABLE, INSERT, SELECT etc. Eftersom SQL är så utbrett är det bra att känna till hur det används för att ställa frågor till en databas.

Data Definition Language

Med DDL kan vi skapa, modifiera och ta bort databaser, domäner, tabeller, vyer och index med mera. Nedan visas syntax för några av de SQL-kommandon som ingår i DDL:

```
CREATE/ALTER/DROP TABLE <tabellnamn>
CREATE/DROP VIEW <vynamn>
CREATE/DROP INDEX <indexnamn>
```

Skapa tabeller

Här kommer vi att titta på hur vi skapar tabeller i SQL med kommandot CREATE TABLE. Därefter kommer vi att titta på begrepp som är viktiga när vi skapar tabeller: datatyper, dataintegritet, entitetsintegritet, referensintegritet och nödvändiga data. Vi kommer att skapa följande tabeller (som även kommer att användas i inlämningsuppgiften):

```
postort (postnr, postort)
student (portalid, födelsedatum, förnamn, efternamn, adress, postnr*)
kurs (kurskod, namn, hp)
betyg (portalid*, kurskod*, år, betyg)
```

Tabellen postort skapas så här med SQL:

```
CREATE TABLE postort (
  postnr  CHAR(5),
  postort  VARCHAR(20) NOT NULL,
CONSTRAINT postort_pk PRIMARY KEY(postnr));
```

Här används postort_pk som namn på entitetsintegritetsregeln som uttrycket PRIMARY KEY ger. Detta ser automatiskt till att data måste anges (NOT NULL) och att kolumnen indexeras. En primärnyckel kan aldrig vara utan värde (NULL). Kolumnen postort är också ett nödvändigt attribut som inte får vara NULL.

Datatypen CHAR skapar en kolumn som rymmer en textsträng med ett fast antal tecken, i detta fall exakt 5 tecken. Om datatypen är definierad som en fast längd och vi lägger in en sträng med färre tecken än denna längd, då kommer extra mellanslag att läggas till i strängen för att ge önskad storlek.

Datatypen VARCHAR skapar en kolumn som också rymmer en textsträng men med en varierande längd upp till maximalt det antal tecken som anges i parentesen. I vårt fall kan textsträngen bestå av maximalt 20 tecken. Om datatypen är definierad som en varierande längd och vi lägger in en sträng med färre tecken än den maximala längden, då kommer endast de tecken som finns i textsträngen att lagras.

Tabellen kurs skapas på följande sätt:

```
CREATE TABLE kurs (
  kurskod  CHAR(6),
  namn  VARCHAR(45),
  hp  NUMERIC(3,1) NOT NULL DEFAULT 7.5,
CONSTRAINT kurs_pk PRIMARY KEY(kurskod),
CONSTRAINT kurs_un UNIQUE(namn));
```

Uttrycket UNIQUE säger att kolumnen måste innehålla unika värden, det får inte finnas två rader vars värden är lika. I det här fallet innebär det att vi inte kan lagra två kurser som har samma namn. När UNIQUE används är det tillåtet att lagra NULL-värden i kolumnen.

Datatypen NUMERIC (eller DECIMAL) skapar en kolumn som rymmer ett decimaltal. Första siffran i parentesen anger hur många siffror talet totalt kan ha. Den andra siffran anger hur många av dessa siffror decimaldelen ska bestå av. I vårt exempel skapas en kolumn som kan lagra decimaltal bestående av totalt 3 siffror (inklusive decimaldelen) och där en decimal används. Med andra ord kan tal upp till 99,9 lagras i kolumnen.

Uttrycket DEFAULT används för att lägga till ett standardvärde i en kolumn. Det värde som anges efter nyckelordet DEFAULT kommer att läggas till i alla nya rader om inget annat värde har angetts.

Tabellen student har vi skapat så här:

```
CREATE TABLE Student (
  portalid  CHAR(8),
  "födelsedatum"  DATE,
  "förnamn"  VARCHAR(20) NOT NULL,
  efternamn  VARCHAR(30) NOT NULL,
  adress  VARCHAR(30),
  postnr  CHAR(5),
CONSTRAINT student_pk PRIMARY KEY(portalid),
CONSTRAINT student_fk FOREIGN KEY(postnr) REFERENCES postort(postnr));
```

Här används student_kk som namn på referensintegritetsregeln som uttrycket FOREIGN KEY ger. Denna säger att postnr är främmandenyckel i tabellen student och som refererar till tabellen postort och dess primärnyckel postnr. Denna regel ser samtidigt till att borttagning av postnr från Postort inte tillåts om det finns rader i tabellen Student som har detta postnr.

Med andra ord kan ett postnummer inte tas bort förrän alla studenter som har detta postnr tagits bort.

Vi kan göra ett tillägg till regeln enligt följande:

```
CONSTRAINT student_fk FOREIGN KEY(postnr) REFERENCES postort(postnr) ON
DELETE CASCADE
```

Med ON DELETE CASCADE fungerar regeln så här: vi tillåts att ta bort vilket postnr som helst I Postort, även om det finns studenter som har detta postnr. Dessa studenter tas helt enkelt bort (hela raden) från tabellen Student. Det finns som du kanske förstår anledning till att vara väldigt försiktig med användningen av CASCADE.

Kolumnen födelsedatum är skapad med datatypen DATE. Det skapar en kolumn som kan lagra ett datum enligt ett giltigt format, t.ex. 1999-01-08 för 8 januari 1999. Det finns även datatyper för att lagra enbart tid (TIME) och för att lagra både datum och tid (TIMESTAMP). Olika databashanterare kan ha olika namn för dessa datatyper.

Till sist har vi skapat tabellen betyg så här:

```
CREATE TABLE betyg (
  portalid  CHAR(8),
  kurskod  CHAR(6),
  "år"  CHAR(4),
  betyg  INTEGER,
CONSTRAINT betyg_pk PRIMARY KEY(portalid, kurskod),
CONSTRAINT betyg_fk1 FOREIGN KEY(portalid) REFERENCES student(portalid),
CONSTRAINT betyg_fk2 FOREIGN KEY(kurskod) REFERENCES kurs(kurskod));
```

Notera att tabellen har en sammansatt primärnyckel bestående av kolumnerna portalid och kurskod samt att tabellen har två främmandenycklar. Kolumnen betyg använder datatypen INTEGER i vilken vi kan lagra ett heltal (4 bytes stor).

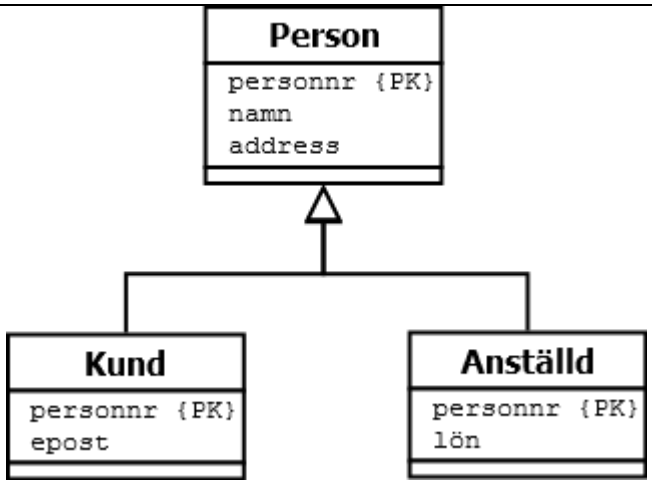
Vi kan ändra och ta bort tabeller samt skapa index i en tabell. Exempel:

```
ALTER TABLE student ADD COLUMN telefon VARCHAR(11);

CREATE INDEX betyg_index2 ON student(telefon);

DROP TABLE student;
```

Ett index är en struktur som ger snabbare tillgång till raderna i en tabell som bygger på värden från en eller flera kolumner. Närvaron av ett index kan avsevärt förbättra prestanda för en fråga.

Arv

Ibland ställs vi inför situationen där arv behöver användas. Vi kan ha ett ER-diagram enligt figur 1 nedan:



Figur 1. ER-diagram som visar ett arv.

Som du säkert kommer ihåg från lektion 4 översätts detta till relationsmodellen enligt följande:

```
Person (personnr, namn, address)
Kund (personnr*, epost)
Anställd (personnr*, lön)
```

Attributet personnr i subklasserna Kund och Anställd blir en främmande nyckel till primärnyckeln personnr i superklassen Person. Implementerat i SQL får vi följande kod:

```
CREATE TABLE person (
  personnr CHAR(11),
  namn VARCHAR(50),
  address VARCHAR(60),
  CONSTRAINT person_pk PRIMARY KEY (personnr)
);
```

```
CREATE TABLE kund (
  personnr CHAR(11),
  epost VARCHAR(50),
  CONSTRAINT kund_pk PRIMARY KEY (personnr),
  CONSTRAINT kund_fk FOREIGN KEY (personnr) REFERENCES person(personnr)
    ON UPDATE CASCADE ON DELETE CASCADE
);
```

```
CREATE TABLE anställd (
  personnr CHAR(11),
  lön INTEGER,
  CONSTRAINT anställd_pk PRIMARY KEY (personnr),
  CONSTRAINT anställd_fk FOREIGN KEY (personnr) REFERENCES person(personnr)
    ON UPDATE CASCADE ON DELETE CASCADE
);
```

För främmande nyckeln i subklasserna vill vi att värdet i tabellen ska uppdateras alternativt tas bort när den refererade tabellen person uppdateras eller tas bort. För att uppnå detta skriver vi ON UPDATE CASCADE följt av ON DELETE CASCADE.

Data Manipulation Language

DML handlar om uppdateringar av tabellerna i databasen och sökning efter information. Vi kommer att titta på hur vi lägger till nya rader i tabellen, hur vi ändrar och tar bort befintliga rader och hur vi utför enkla frågor mot en tabell.

Manipulera rader i en tabell

Det finns två sätt att lägga till nya rader med data i en tabell. Det första lägger till en enda rad i en namngiven tabell:

```
INSERT INTO kurs VALUES('DT062G','Java för C++ programmerare',7.5);
```

Det är viktigt att ordningsföljden på de värden vi lägger in stämmer överens med kolumnernas ordningsföljd i tabellen. Om vi inte vill lägga in data i alla kolumner och/eller vill ange data i en annan ordningsföljd, kan vi specificera vilka kolumner som data ska läggas in för enligt:

```
INSERT INTO kurs (hp, kurskod) VALUES(7.5, 'DT006G');
```

Det andra sättet gör det möjligt att kopiera rader från en eller flera tabeller till en annan tabell:

```
INSERT INTO kurs (kurskod, namn, hp)
SELECT *
FROM gamla_kurser;
```

Om vi inte har något värde för en av kolumnerna, och kolumnstrukturen tillåter det (inte NOT NULL), kan vi skriva NULL som värde:

```
INSERT INTO kurs VALUES('DT007G',NULL, 7.5);
```

Uttrycket UPDATE låter oss att uppdatera värdet i en existerande rad i den namngivna tabellen:

```
UPDATE kurs
SET namn = 'Java I'
WHERE kurskod = 'DT006G';
```

Uttrycket DELETE låter oss att ta bort hela rader från den namngivna tabellen:

```
DELTE
FROM kurs
WHERE kurskod = 'DT006G';
```

Om WHERE-klausulen utesluts kommer alla rader i tabellen att tas bort.

```
DELTE
FROM kurs;
```

Det sista exemplet kommer inte att ta bort tabellen i sig utan enbart raderna med data i den. Om du vill ta bort hela tabellen, data inklusive strukturen, måste vi som bekant använda DROP TABLE i stället.

Urvalsfrågor

Syftet med uttrycket SELECT är att hämta och visa data från en eller flera tabell. En tabell i en databas kommer alltid att innehålla en mängd med rader (mängden kan vara tom). Denna mängd med rader är dynamiskt eftersom användaren av databasen kommer att infoga, uppdatera och ta bort rader när som helst. Därför kommer en SELECT-sats att producera en ny tabell baserad på de rader den ursprungliga tabellen hade i exakt det ögonblick när frågan körs.

Enklare urvalsfrågor i relationsdatabaser bygger på relationsoperatorerna selektion och projektion. Dessa frågor görs i SQL genom en kombination av SELECT-, FROM- och WHERE-klausuler.

```
SELECT <projektion>
FROM <tabellnamn>
WHERE <selektion>
```

Som ett första exempel ska vi se på en fråga som listar alla uppgifter om alla kurser som finns i tabellen kurs:

```
SELECT kurskod, namn, hp
FROM kurs;
```

Det finns ett snabbt sätt att uttrycka "alla uppgifter" genom att använda en asterisk (*) som jokertecken i stället för kolumnnamnen.

```
SELECT *
FROM kurs;
```

Resultatet av båda frågorna visas i tabell 1.

Tabell 1

kurskod	namn	hp
DT013G	Datavetenskaplig introduktionskurs	7.5
DT011G	Operativsystem introduktionskurs	7.5
DT062G	Java för C++ programmerare	7.5
DT022G	Databaser, introduktion	7.5
DT076G	Databaser, implementering och modellering	7.5
DT031G	Applikationsutveckling för Android	7.5
DT006G	Java I	7.5
DT007G	Java II	7.5
MA072G	Tillämpad matematik	7.5
DT099G	Examensarbete	15

SQL är inte känsligt för versaler och gemener. Dock är det är mycket vanligt i litteratur om SQL att reserverade ord skrivs i versaler och användardefinierade ord (tabell- och kolumnnamn) skrivs i gemener.

I många fall vill vi göra en projektion, det vill säga att extrahera information från olika kolumner i tabellen. En sådan fråga kan se ut enligt följande:

```
SELECT kurskod, hp
FROM kurs
```

Resultatet av denna fråga visas i tabell 2.

Tabell 2

hp	kurskod
7.5	DT013G
7.5	DT011G
7.5	DT062G
7.5	DT022G
7.5	DT076G
7.5	DT031G
7.5	DT006G
7.5	DT007G
7.5	MA072G
15	DT099G

SELECT eliminerar inte dubbletter när den utför en projektion över en kolumn eller flera kolumner. För att eliminera dubbletter använder vi det reserverade ordet DISTINCT.

Låt oss ta en titt på innehållet i tabellen betyg genom att skriva följande fråga:

```
SELECT *
FROM betyg;
```

Resultatet visas här nedan i tabell 3.

Tabell 3

portalid	kurskod	år	betyg
bean1100	DT022G	2012	1
bean1100	DT011G	2011	2
bean1100	DT076G	2012	4
sasv1010	DT022G	2012	4
sasv1010	DT011G	2012	2
kama1203	DT022G	2012	

Som vi ser förekommer en del av studenterna på mer än en rad. Anledning är att dessa studenter har läst mer än en kurs. Om vi nu är intresserade av att få reda på de studenter som har läst kurser (och endast en gång lista studenter som läst flera kurser) kan vi utnyttja DISTINCT så här:

```
SELECT DISTINCT portalid
FROM betyg;
```

Resultatet visas i tabell 4.

Tabell 4

portalid
bean1100
kama1203
sasv1010

Hade vi inte använt DISTINCT skulle resultatet av frågan sett ut så här:

Tabell 5

portalid
bean1100
bean1100
bean1100
kama1203
sasv1010
sasv1010

I SQL finner vi också beräknade kolumner. För att använda en beräknad kolumn anger du ett SQL-uttryck i SELECT-listan. Ett SQL-uttryck kan innebära addition, subtraktion, multiplikation eller division. Resultatet lagras aldrig i ursprungstabellen utan finns endast med i resultattabellen.

Låt oss säga att vi vill beräkna hur många veckors heltidstudie varje kurs innebär. För att få fram detta måste vi dividera antalet högskolepoäng (hp) med 1,5. Följande SQL-fråga gör detta:

```
SELECT kurskod, namn, hp, hp / 1.5 AS veckor
FROM kurs;
```

Här har vi gett den beräknande kolumnen ett namn genom att använda ett alias (AS). Resultatet av denna fråga visas i tabell 6.

Tabell 6

kurskod	namn	hp	veckor
DT013G	Datavetenskaplig introduktionskurs	7.5	5.0
DT011G	Operativsystem introduktionskurs	7.5	5.0
DT062G	Java för C++ programmerare	7.5	5.0
DT022G	Databaser, introduktion	7.5	5.0
DT076G	Databaser, implementering och modellering	7.5	5.0
DT031G	Applikationsutveckling för Android	7.5	5.0
DT006G	Java I	7.5	5.0
DT007G	Java II	7.5	5.0
MA072G	Tillämpad matematik	7.5	5.0
DT099G	Examensarbete	15	10.0

Vi behöver ofta begränsar raderna som hämtas. Detta uppnås med en WHERE-klausul, som består av nyckelordet WHERE följt av ett sökvillkor som anger de rader som ska hämtas. Några av de jämförelseoperatorer som vi kan använda är: =, <, <=,>,> = och <>.

Om vi önskar att lista alla kurser som har mer än 10 högskolepoäng skriver vi följande fråga:

```
SELECT kurskod, namn, hp
FROM kurs
WHERE hp > 10;
```

Resultatet visas i tabell 7.

Tabell 7

kurskod	namn	hp
DT099G	Examensarbete	15

Vi kan också använda de logiska operatorerna AND, OR och NOT i SQL-frågor. OCH har företräde framför OR. Vi ska nu skapa en fråga som kommer att hitta alla uppgifter om kurserna Java I och Java II:

```
SELECT kurskod, namn, hp
FROM kurs
WHERE namn = 'Java I'
  OR namn = 'Java II';
```

Resultatet av denna fråga ser vi i tabell 8.

Tabell 8

kurskod	namn	hp
DT006G	Java I	7.5
DT007G	Java II	7.5

Vi kan också använda operatorn BETWEEN som begränsar raderna till de som är inom en viss kolumnvärde. Gränser ingår i intervallet. Vi vill nu skapa en fråga som kommer att lista namn och adress på alla studenter som är födda någon gång under år 1985:

```
SELECT "förnamn", efternamn, adress
FROM student
WHERE "födelsedatum" >= '1985-01-01'
  AND "födelsedatum" <= '1985-12-31';
```

Eller:

```
SELECT "förnamn", efternamn, adress
FROM student
WHERE "födelsedatum" BETWEEN '1985-01-01' AND '1985-12-31';
```

Resultatet av båda dessa frågor visas i tabell 9.

Tabell 9

förnamn	efternamn	adress
Sara	Svensson	Blomstigen 2

Mängdoperatorn IN kan användas i WHERE-klausuler för att kontrollera att ett kolumnvärde finns bland en mängd av värden. Vi vill hitta portalid och postnummer till alla studenter som heter Svensson eller Martinsson.

```
SELECT portalid, postnr
FROM student
WHERE efternamn IN ('Svensson', 'Martinsson');
```

Eller:

```
SELECT portalid, postnr
FROM student
WHERE efternamn = 'Svensson'
  OR efternamn = 'Martinsson';
```

Resultatet av dessa två frågor kan ses i tabell 10.

Tabell 10

portalid	postnr
sasv1010	83432
kama1203	89597

Vi kan använda jokertecken för att söka efter speciella mönster i en textsträng. Vi måste då använda operatoren LIKE istället för =. _ i strängen betyder ett godtyckligt tecken, medan % i strängen betyder godtyckligt antal tecken. Vi vill skapa en fråga som hittar förnamn och portalid till alla studenter som har ett förnamn som slutar på bokstaven L:

```
SELECT "förnamn", portalid
FROM student
WHERE "förnamn" LIKE '%l';
```

I tabell 11 ser vi resultatet av denna fråga.

Tabell 11

förnamn	portalid
Bertil	bean1100
Karl	kama1203

Vi använder IS och IS NOT som ett test mot NULL-värden. Observera att NULL varken är större än, mindre än eller lika med något annat värde. NULL är inte heller samma som siffran 0. SQL stöder tre olika logiska värden: sant, falskt, okänt. Alla logiska uttryck som innehåller NULL har värdet okänt. Om vi vill hitta portalid och kurskod för alla studenter som inte har fått betyg i kursen, ser frågan ut så här:

```
SELECT portalid, kurskod
FROM betyg
WHERE betyg IS NULL;
```

Resultatet av frågan ser vi i tabell 12.

Tabell 12

portalid	kurskod
kama1203	DT022G

Om vi i stället vill se alla studenter som fått betyg i en kurs ser frågan ut så här:

```
SELECT portalid, kurskod
FROM betyg
WHERE betyg IS NOT NULL;
```

Resultatet ser vi i tabell 13.

Tabell 13

portalid	kurskod
bean1100	DT022G
bean1100	DT011G
bean1100	DT076G
sasv1010	DT022G
sasv1010	DT011G

Vi kan sortera resultatet av en fråga med hjälp av ORDER BY-klausul i en SELECT-sats. ORDER BY ska alltid vara den sista klausulen i SELECT-uttrycket. Som standard är sorteringsordningen stigande, men vi kan ange att resultatet ska sorteras i sjunkande ordning genom att lägga till DESC.

Låt oss säga att vi vill hitta portalid namn för alla studenter som har ett postnummer som börjar med 83, sorterade i stigande ordning efter efternamn och förnamn.

```
SELECT portalid, efternamn, "förnamn"
FROM student
WHERE postnr LIKE '83___'
ORDER BY efternamn, "förnamn";
```

Resultatet av denna fråga ser vi i tabell 14.

Tabell 14

portalid	efternamn	förnamn
anan0912	Andersson	Anna
bean1100	Andersson	Bertil
sasv1010	Svensson	Sara

Här sorteras resultatet i första hand efter efternamn. Vid lika efternamn sorteras resultatet efter förnamn. Nu önskar vi i stället att lista alla uppgifter om kurserna sorterat efter antalet högskolepoäng. Kursen med högst hp vill vi lista först. Vid lika antal hp ska resultatet sorteras efter kursnamnet.

```
SELECT *
FROM kurs
ORDER BY hp DESC, namn;
```

Resultatet av denna fråga ser vi tabell 15.

Tabell 15

kurskod	namn	hp
DT099G	Examensarbete	15
DT031G	Applikationsutveckling för Android	7.5
DT076G	Databaser, implementering och modellering	7.5
DT022G	Databaser, introduktion	7.5
DT013G	Datavetenskaplig introduktionskurs	7.5
DT006G	Java I	7.5
DT007G	Java II	7.5
DT062G	Java för C++ programmerare	7.5
DT011G	Operativsystem introduktionskurs	7.5
MA072G	Tillämpad matematik	7.5