**Databasteknik**

Individuell Inlämning 2

DML and DDL

*Data Manipulation Language* eller *Data Definition Language* är ett generisk namn på de språk som används av *Database Management System* och används för att instruera vad som ska hända med data i databasen. SQL är av båda typerna, där olika delar i SQL faller under DML eller DDL. DML-delen av SQL är SELECT, UPDATE, INSERT och DDL-delen är CREATE TABLE, CREATE INDEX.  
  
Sedan finns det huvudsakligen två typer av DML, procedural och deskriptiva som syftar på att användaren beskriver vad som ska göras (inte hur) och användaren beskriver exakt hur något ska utföras. SQL-DML-del är deskriptiv.

**Relationsalgebra**

Ett sorts teoretisk språk som *SQL* är grundat på. Relationsalgebra går översiktligt ut på att utföra matematiska operationer på relationer och generera en ny relation. Exempel på operationer som kan utföras på relationer är SELECT, JOIN och PROJECT.

Denna operation väljer ut en tupel (som en rad i ett table)

Denna operation väljer ut ett vertikalt submängd (som en kolumn i ett table)

*JOIN*

Denna operation kopplar ihop tables, men den finns i olika typer:

* *Theta Join*Den mest grundläggande som går ut på att ta fram en kartesisk produkt av relationerna. Från ett table-perspektiv innebär detta kolumnerna kombineras (adderas) och sedan tas alla olika kombinationer av de två tables rader fram. Theta Join har i praktiken få tillämpningsområden då den matchar ihop alla former av rader med den kartesiska produkten.
* *Equijoin*Som theta join men den skalar ner alla kombinationer som inte har ett gemensam värde på en förbestämd kolumn (gemensam key). Denna typen av join är mer användbar då den matchar två tables bättre (de som har en sorts gemensam key).
* *Natural join*Som Equijoin men den tar bort den dubbla kommunen som var en gemensam key.   
  Den mest förekommande typen av join. Så pass vanlig att en *natural join* ofta är vad som menas när endast *JOIN* anges.

Det finns ytterligare typer av join, såsom *Semijoin* och *Outer Join* som är en form av specialiserad *natural join*. Semijoin är en natural join som sedan projiceras på prejudikatet. Outer join är i princip motsatsen, den matchar prejudikatets table och sedan även lägger till prejudikatets alla icke-matchade entries (mer precist är detta en *Left Outer Join*. En *Full Outer Join* skulle istället lägga till omatchade entries från båda termernas tables).

Mängdoperationer

Då relationer i databaskontext är en uppsättning av tuplar (raderna i ett table) innehåller relationsalgebra även operationer som görs på *mängder*. Därav kopplas *mängdlära* genom dess operationer såsom *Union, Snitt* och *Mängddifferens*. För att de ska gå att använda mängdoperationer på relationer måste relationerna ingående i operationen vara *Unionkompatibla* - samma relationsgrad (samma antal kolumner) och varje par av kolumn ( [ACol1, BCol1], [ACol2, BCol2] … ) ska ha samma domän.

* Union  
  En mängd som består av alla element som finns i minst en av . Detta är en mängd som endast innehåller unika element.  
  T.ex. Administrator UNION Users innehåller att tuplar som är antingen admin eller user.
* Snitt  
  En mängd som består av alla element som finns i både . Detta är en mängd som endast innehåller unika element.
* Mängddifferens  
  En mängd som består av alla element i *A* men filtrerat bort på de tal som är gemensamma med mängd *B*.

**Relationsdatabas**

En databas som lagra data genom *relationsmodellen*. Databasen består till grunden (*internal level*) av filer som innehåller all data. Denna data representeras på den *logiska nivån* som *base tables*. Detta är alla databasens faktiska tables som all data komma manipuleras i och brukar ofta benämnas som den “fysiskt” lagrade datan i databasen. Ovanför detta lager finns *external level* som är eventuella *vyer* skapade för användarna av databasen (användare kan använda base tables direkt om detta skulle passa). *Vyer* kan peka på flera olika base tables.

Vyer

En databas har innehåller ett antal tables (*base tables*) som representerar all data som finns i databasen. Dock är det inte alltid praktiskt uppsatt för alla som ska använda databasen eller bara omöjligt att göra det praktiskt inom ett visst användningsområde - olika användningsområden kräver olika data. Istället för att skapa flera uppsättningar av tables för att gynna alla användare kan *vyer* skapas. En sorts mekanism som används för att dölja eller slå samman data i tables, utan att skapa nya base tables.

* Om en användare endast är intresserad av några kolumner i table av många fler kan en vy lösa detta genom att dölja den onödiga kolumnerna.
* Om en användare behöver komma åt kolumner från olika tables kan en vy skapas som *virtuellt table* som agerar som ett vanligt table\* men är egentligen bara en sammanslagning.  
  \* Beroende på vilken data som finns i vyn kan olika operationer inte göras. T.ex. om en vy skapas för ett table men PK dölj, kommer inte databasen att tillåta INSERT.

*Vyer skapas* i SQL med *CREATE VIEW* följt av vilka kolumner från vilka tables som vyn ska täcka.

Manipulera SQL-databas

SQL-kommandon används för att manipulera en relationsdatabas. De finns kommandon för att välja ut rader/kolumner, slå samman tables, skapa tables, sätta in rader/kolumner osv.

Vanliga *SQL-kommandon* för att manipulera en relationsdatabas är:

* CREATE DATABASE  
  Skapar en ny databas men angivet namn. För att sedan ansluta till den databas för att kunna manipulera den används *USE* (som heter *CONNECT* för vissa system)
* CREATE TABLE   
  Skapar ett nytt table i en databas och anger namn och vilka kolumner som ska finnas.
* CREATE INDEX  
  Skapar ett nytt index för ett kolumn i ett table. Index används för att snabbare åtkomst till ofta återkommande hämtande av specifik data.
* ALTER TABLE / RENAME TABLE   
  Ändrar ett table, såsom lägga till/ta bort kolumner respektive ändra namn på ett table
* DROP TABLE / DROP INDEX  
  Ta bort ett table och alla data i den / Tar bort ett index  
  Att ta bort en index kan påverka prestandan om det användes ofta för att hämta en data som indexet var kopplat till.

Hämta data i SQL-databas

För att hämta data med SQL-kommando används SELECT. Den kan hämta från valda kolumner, rader eller speciella givna villkor. SELECT kan utföra relationsalgebrans tre SELECT, PROJECT och JOIN taget att rätt villkor specificeras. *SELECT* som SQL-kommando används alltså i sin breda mening att hämta data och är därmed ett mycket vanligt kommando.  
  
När SELECT används anges vilka kolumner från vilket table som ska hämtas. Samt ett *predikat*, vilket är ett villkor som måste uppfyllas för att en tupel ska inkluderas i resultatet.   
*- > SELECT Col1, Col2 FROM Table WHERE Col3 == “Value”*

För att hämta data från flera tables anges detta enkelt som *SELECT Col1, Col2, Col3 FROM Table1, Table2…* Detta kan jämföras med *JOIN* i relationsalgebra då kommandot med dessa parametrar returnerar ett nytt table med givna kolumner, en sorts sammanslagning (fast endast innehållande specificerade kolumner).

*DISTINCT* är en parameter som läggs efter SELECT för att ange att resultatet måste innehålla unika tuplar. T.ex. Om en *query* (förfrågan) görs om att hämta en data endast från en enda kolumn kan det lätt hända att flera tuplar (rader) får samma värde. Om *DISTINCT* anges kommer alla duplikat rensas.

*NOT EXISTS* är en parameter som används tillsammans med *WHERE* i en SELECT-query och fungerar som en icke-sats. Kan beskrivas som “Hämta alla tuplar i raderna R1, R2 från tables T1, T2 som inte finns i ett annat table”.

Det finns många fler parametrar för SELECT som alla anger olika sätt att hämta/filtrera tuplar. T.ex. *INTERSECT*, *UNION*, *MINUS* som representerar mängdoperationerna snitt, union och mängddifferens.

Uppdatera data i SQL-databas

Bland SQL-kommandon finns det tre kommandon som används för att uppdatera data i tables:

* UPDATE  
  Ändrar data i en kolumn i ett table. Oftast anges ett predikat: “Ändra Col1 i Table1 där Name=MyName”
* INSERT  
  Lägger till en ny tupel (rad) i ett table. INSERT kan specificera alla värden som behövs för en rad (dvs. ett värde för varje kolumn) eller låta ett standardvärde sättas in (dock kan standardvärden inte sätta in i kolumner som är markerade med “NOT NULL”, såsom Primary Keys)
* DELETE  
  Tar bort en rad i en table. Oftas anges ett predikat för att ta bort de rader/raden som uppfyller ett villkor.

**Begränsningar och aktiv databas**

En databas som aktivt gör kontroller för att se till att data i databasen ligger inom *begränsningarna* (*constraints*, t.ex. *domän* eller kolumn parametrar NOT NULL) kallas för en aktiv databas. Detta görs för att databasen ska kunna försäkra sig om att den hela tiden befinner sig i ett *giltig stadie*. En databas som inte kan garantera att den befinner sig i ett giltigt stadie kan inte garantera sin funktionalitet såsom att *triggers* (en typ av *OnEvents* som beskrivs längre ner) fungerar som de ska.

Det finns dock tillfällen *begränsningar* kan vara problem. Om två tables refererar till varandra (t.ex. via en *Foreign Key - Referential Integrity*) på sådant sätt att fältet är markerat med *CONSTRAINT* kommer det vara omöjligt att skapa det första elementet, för vad ska den första tupeln referera till ifall referatet inte hunnits skapas i det andra tablet? Genom att temporärt låta *inaktivera begränsningar* kan databasen temporärt (under kontroll) vara i ett *ogiltigt stadie* och tables som beskrevs ovan kan användas utan problem. Begränsningar inaktiveras och aktiveras med hjälp av kommandot *DISABLE CONSTRAINT* respektive *ENABLE CONSTRAINT*.

**Triggers**

Det kan ibland vara praktiskt att automatisera operationer i sin databas beroende på olika typ av händelser. I SQL finns *Triggers* som praktiskt taget är en sorts event listener med en kopplad handling till när händelse inträffar - likt designmönstret *Subject-Observer*. För att använda en *trigger* registreras den genom kommandot *CREATE OR REPLACE TRIGGER* följt av dess namn, vilket typ av händelse som lyssnas efter (INSERT, DELETE, UPDATE) och en definition av handling som ska utföras i samband med att händelsen inträffar.  
  
Exempel på användsområde för en trigger är att hålla räkningen på en något som inte får överstiga eller understiga ett värde: “När <entity> läggs till i ett table, öka noEntities med 1 i ett annat table”.

**Tidsdatabaser**

En databas som på något vis integrerar konceptet tid kallas för en *tidsbdatabas* (*temporal database*). Idén grundar sig i att all data som finns att hämta “just nu” och representerar nuläget i en databas bör inte vara den enda data som finns där, utan “tidigare stadie” av en databas bör finnas lagrat i den själv. Sådant att databasen agerar även som ett arkiv utöver att spara som en *snapshot* av det den beskriver.

SQL har inbyggda datatyper och funktioner som möjliggör användning av tid i en databas. Detta inkluderar *tidsdatatyp* i form av *TIMESTAMP* (även *TIMESTAMP WITH TIMEZONE* och *TIMESTAMP WITH LOCAL TIMEZONE*) och den kan skapas genom att använda *TO\_TIMESTAMP* och ange en datumsträng:  
T.ex. TO\_TIMESTAMP(‘15-JAN-2020’, ‘dd-mon-yy’’).

Data som lagras i dessa tidsdatatyper har sedan unik användning som parametrar i vissa SQL-kommandon. T.ex. SELECT har nyckelorden *BETWEEN* *AND* där två tidsdatatyper kan sättas in sådant att tuplar inom tidsramen blir valda, något som är särskilt användbart när arkiverad data behöver hämtas. SELECT Col1, Col2 FROM Tab1 WHERE Col3Date BETWEEN FirstDate AND SecondDate

Det finns fler kommandon specifikt för en tidsdatabas såsom *PERIOD FOR* som representerar en tidsperiod, dvs. tiden mellan två *TIMESTAMP*.

**Java Database Connectivity (JDBC)**

För att kontakta en relationsdatabas kan man i programmeringsspråket Java använda JDBC-APIn. Dess gränssnitt inkluderar en uppsättning av klasser och interfaces som används för att prata med en databas. Så sådan API används både för att det är praktiskt och tidsparande, men också för att minska risken att något blir fel.

JDBC innehåller bland annat *Connection* som håller anslutning till databasen, *Statement* som är en som en *wrapper* runt ett parameterfritt SQL-kommando (*PreparedStatement* är med parametrar), samt *CallableStatement* som liknar en funktion med input och output.

**Systemkatalogen (The System Catalog)**

När en databas växer behöver information om databasen i sig lagras någonstans. Detta lagras i *systemkatalogen* - alltså en databas med information om en databas. Systemkatalogen består bland annat av tables med info om databasen struktur och hur den ser ut vid en viss tidpunkt. Systemkatalogen måste därför uppdateras varje gång databasen ändras med t.ex. UPDATE, INSERT, DELETE.

**Uppgift**

**A**

CREATE\_TABLE T\_MEMBER(

mID INT NOT NULL AUTO\_INCREMENT,

name VARCHAR(50) NOT NULL,

email VARCHAR(50) NOT NULL,

CONSTRAINT memberPK PRIMARY KEY (mID)

)

CREATE\_TABLE T\_LOAN(

mID INT NOT NULL,

kID INT NOT NULL,

date VARCHAR NOT NULL

CONSTRAINT loanPK PRIMARY KEY (mID, kID, date),

CONSTRAINT loanFK\_1 FOREIGN KEY (mID) REFERENCES T\_MEMBER (mID) ON DELETE CASCADE

CONSTRAINT loanFK\_2 FOREIGN KEY (kID) REFERENCES T\_KAYAK (kID)  
 ON DELETE CASCADE

)

CREATE\_TABLE T\_MEMBERPHONE(

mID INT NOT NULL,

phoneNumber VARCHAR(20) NOT NULL,

CONSTRAINT phonePK PRIMARY KEY (mID),

CONSTRAINT phoneFK FOREIGN KEY (mID) REFERENCES T\_MEMBER (mID)

)

**B**

INSERT INTO T\_MEMBER(name, email) VALUES (“Ari Ek”, “[Ari.Ek@gmail.com](mailto:Ari.Ek@gmail.com)”)  
  
SELECT DISTINCT mem.mID, mem.name, mem.email  
FROM T\_LOAN loan, T\_MEMBER mem  
WHERE mem.mID == loan.mID AND loan.kID == 42