

# Anteckningar Databasteknik

## Vecka 1

Vyer

## Vecka 2

### Programmering i DBMS

Funktioner, Procedurer och Triggers.

- Inbyggda funktioner så som `SUM`, `MAX`, `MIN`

Det finns en stor mängd inbyggda funktioner för stränghantering, matematiska funktioner, 'aggregate-funktioner', datum och tid, säkerhet, system, statistik o.s.v.

- Användardefinierade funktioner som tar en parameter och resulterar ett värde - får inte ändra data
- Procedurer är "småprogram" som körs med `CALL` - kan ha 0 eller flera input och 0 eller flera output - kan ändra data
- Trigger är något som körs vid ett visst event så som `DELETE` i en tabell o.s.v.

Detta görs bland annat för att se till att data är riktig, det går att tvätta data o.s.v. Det går även att använda för att förbereda data för en ev. front-end. Det kan även användas för att ge ett rent API.

### Exempel - ålder i databas

Vi lägger till en födelsedag till en Table.

```
1  # NOW() ger den nuvarande tiden ner till millisekunden
2  ALTER TABLE User ADD COLUMN birthdate DATE DEFAULT NOW();
3
4  UPDATE User SET USER.birthdate = '1981-05-18' WHERE id = 4;
5
6  INSERT INTO User (id, birthdate) VALUES
7      (5, '1976-08-21'),
8      (6, '1989-08-21'),
9      (7, '1987-08-21')
10 ON DUPLICATE KEY UPDATE
11     id=VALUES(id), birthdate=VALUES(birthdate);
12
13 SELECT * FROM User;
```

Vi använder följande kod får att hämta ålder i år från användarna

```
1 SELECT TIMESTAMPDIFF(YEAR, birthdate, NOW()) AS "age", name FROM USER;
```

## Vi skapar en funktion för att hämta åldern.

Används för bland annat datavalidering. En funktion returnerar ett värde.

```
1 DROP FUNCTION IF EXISTS AGE;
2 CREATE FUNCTION AGE(birthdate, DATE)
3     RETURNS INTEGER NOT DETERMINISTIC
4     RETURN TIMESTAMPDIFF(YEAR, birthdate, NOW());
5
6 SELECT AGE(birthdate) AS "age", name FROM User;
```

Vi kan även skapa en query som hämtar alla användare som har födelsedag idag.

```
1 SELECT * FROM User WHERE DATEDIFF(DATE_ADD(birthdate, INTERVAL
TIMESTAMPDIFF(YEAR, birthdate, NOW()) YEAR, NOW())=0)
```

```
1 DROP FUNCTION IF EXISTS HAVE_BIRTHDAY;
2 CREATE FUNCTION HAVE_BIRTHDAY(birthday DATE)
3     RETURNS BOOLEAN NOT DETERMINISTIC
4     RETURN DATEDIFF(
5         DATE_ADD(birthday, INTERVAL AGE(birthday) YEAR),
6         NOW()) = 0;
7
8 SELECT name, HAVE_BIRTHDAY(birthdate) as "have_birthday",
CONCAT(AGE(birthdate), " years") AS "age" FROM User;
```

## Stored Procedures

En procedur får ändra data. Körs mer som ett program - kan ha flera argument.

### Exempel

```
1 ALTER TABLE User ADD COLUMN Country VARCHAR(60),
2     ADD COLUMN Town VARCHAR(60);
3
4 CREATE TABLE Allowed_Towns (
5     Country VARCHAR(60),
6     Town_Name VARCHAR(60)
7 );
8
9 INSERT INTO Allowed_Towns VALUES
10     ('Sweden', 'Ronneby'), ('Sweden', 'Karlskrona'), ('China',
'Hongkong');
```

```

11
12 SELECT * FROM Allowed_Towns;
13
14 DELIMITER //
15 CREATE PROCEDURE PShowAllowed_Towns()
16 BEGIN
17     SELECT * FROM Allowed_Towns;
18 END //
19 DELIMITER ;
20
21 CALL Allowed_Towns();

```

## Trigger

```

1 DELIMITER //
2 CREATE TRIGGER Message_Reactions_insert AFTER INSERT
3 ON Reaction FOR EACH ROW
4 BEGIN
5     UPDATE Message SET reactions = reactions + 1 WHERE id =
NEW.to_message;
6 END //
7 DELIMITER ;
8
9 DELIMITER //
10 CREATE TRIGGER Message_Reactions_insert AFTER DELETE
11 ON Reaction FOR EACH ROW
12 BEGIN
13     UPDATE Message SET reactions = reactions - 1 WHERE id =
OLD.to_message;
14 END //
15 DELIMITER ;

```

## Transaktioner

Exempel: Kalle vill överföra 100 SEK från hans ena konto till hans andra konto. En transaktion är en grupp av kommandon som utförs. Vanligtvis används begreppet *ACID*. En transaktion säger helt enkelt: "antingen ska allt genomföras eller så ska inget genomföras". Uppstår ett problem längs vägen utförs en s.k. *rollback* (gå tillbaka till det tillstånd som fanns innan vi påbörjade ändringar).

### ACID

Atomicity: allt körs eller inget körs. För detta ansvarar databasservern

Consistency: en transaktion ska ta databasen från ett konsekvent läge till ett annat. För detta ansvarar dels databasservern men även applikationsprogrammerare. "Om jag är gift med någon, är den personen i sin tur gift med mig".

Isolation: en transaktion kan inte använda andra transaktioner.

Durability: om någonting sker (så som att anslutning till databasen tappas) så bör vi se till att transaktionen körs ändå. För detta ansvarar applikationsprogrammerare.

## Implementation

1. `BEGIN TRANSACTION name`
2. `COMMIT TRANSACTION name`
3. `ROLLBACK TRANSACTION name`

För applikationsskapare är det rekommenderat att vi sparar all data om användaren som behövs, sedan skickar vi en transaktionslista innehållande `BEGIN`, `COMMIT` och `ROLLBACK` så att allt sker på en gång. Annars är risken att vi låser databasen så att andra ej kan använda den.

```
1 BEGIN TRANSACTION
2     UPDATE staff
3     SET salary = salary + 200
4     WHERE salary >= 800;
5
6     UPDATE staff
7     SET salary = salary * 1.25
8     WHERE salary < 800;
9 COMMIT
```

Förändringarna sker först när vi kallar på `COMMIT`.

## Auto commit transaction

```
1 INSERT INTO Person (Name, Address) VALUES ('Bob', 'Kalmar'), ('Alice',
'Tingsryd')
```

Antingen lägger vi till både Bob och Alice, eller ingen av dem.

## Concurrency

Vi behöver många användare som jobbar mot databasen. Detta kan röra transaktioner som ska kunna utföras samtidigt. En kraftfull databasserver kan hantera ungefär 60000 transaktioner per sekund.

### Problem som orsakas av concurrent transactions

- Lost Updates - uppdateringar sker inte
  - Kalle har 100 SEK i ett konto (både Kalle och hans far läser)
  - Kalle vill dra 30 SEK för att köpa pasta (Kalle skriver 70)
  - Kalles far vill överföra 200 SEK till kontot (Kalle skriver 300)
- Uncommitted Dependency (Dirty read) - vi läser saker som inte är ordentligt uppdaterade
  - Kalle har 100 SEK i ett konto (banken läser)

- banken råkar sätta in 100 SEK (banken skriver 200, Kalle läser)
- banken tar bort 100 SEK (banken skriver 100)
- Kalle drar ut 20 SEK (Kalle skriver 180)
- Inconsistent Analysis (Non-repeatable Read) - vi läser saker som bara är tillfälliga
  - $T_1$  och  $T_2$  är concurrent transactions
  - $T_2$  kalkylerar summan av konton  $X$ ,  $Y$  och  $Z$
  - $T_1$  överför 10 SEK från kontot  $X$  till  $Z$
- Phantom Reads

### Phantom Reads:

```

1 SELECT COUNT(*) AS @numberOfEmployees
2   FROM Employees
3
4 UPDATE Employees
5   SET salary = salary + 100000 / @numberOfEmployees

```

Mellan det att vi läser antalet anställda tills dess att vi skriver dess lön kan vi få fler eller färre anställda.

### Hur vi undviker problemen

**Optimistisk kontroll:** vi jobbar utan att låsa något. Vi kontrollerar manuellt innan vi "committar" en transaktion. Vi räknar inte med problem. Få användare så är det antagligen korrekt.

**Pessimistisk kontroll:** vi jobbar genom att låsa en rad, tabell eller hela databasen. Våra transaktioner bör vara isolerad. Låsningar kan styra åtkomst för läsning, skrivning o.s.v. Exclusive lock (write lock), shared lock (read lock). Tidsstämpling.

MYSQL har stöd för flera olika *storage engines* så som Inno DB. Beroende på val av engine så hanteras låsning på olika vis.

## Låsning

Vi kan som sagt låsa en rad, tabell eller hela databasen. Vi kan låsa under en hel transaktion eller under vissa kommandon.

Låsningslägen:

- **Shared (S):** lås read only, så som SELECT
- **Exclusive (X):** lås INSERT, UPDATE, DELETE
- **UPDATE (U):** lås UPDATE

Ett problem som kan uppstå är ett så kallat *dead lock* där vi hamnar i ett läge där vi låser utan att någonsin komma ur låsningen. Detta kan ske när två transaktioner väntar på varandra. Servern tar vanligtvis hand om detta genom att sätta en *timeout* som låser upp efter en viss tid. Då utförs en rollback.

Vi kan minimera risken genom att alltid komma åt tabeller i samma ordning (så som bokstavsordning). Man kan använda procedurer som utför uppdatering så att samma ordning alltid följs. En procedur räknas dessutom som en egen transaktion automatiskt. Försök även att hålla transaktioner så korta som möjligt - endast då vi använder data som är beroende av annan data. Använd även en låg isoleringsnivå.

- En transaktion bör aldrig vänta på en användare
- Ett användarinterface väntar 99.9% av tiden på användaren
- Transaktioner ska vara snabba

Det går inte rimligtvis att låsa allt man behöver hela tiden (transaktioner överallt) - vi tappar på så vis prestanda.

Storlek på låsningen beror på vilken databas vi använder och hur vi sparar data i databasen. Vi kan använda det på databasnivå, tabellnivå, radnivå o.s.v.

## Databas recovery

Om något sker eller går fel ser vi till att ha backup på systemet. Denna del berör *A* och *D* i *ACID*. Vissa databaser sparar mycket i minnet, något som kan kräva en synkning till disk. Annars riskerar vi att tappa information om strömmen går eller systemet på annat vis får ett abrupt avslut. Transaktioner är en del i recovery - de har en backup-mekanism, logging-möjligheter, milstolpar etc.

## Vecka 3

---

### Funktionalitet

#### ANSI-SPARC

En modell över hur en databas bör se ut. Användarna använder externa scheman. En logisk / konceptuell nivå kan ses vara administratörer med konceptuella scheman. Den fysiska designen styrs av vilken produkt vi har - interna scheman och fysisk lagring. Externa scheman kan beröra virtuella tabeller. Grundtabeller ligger i ett logiskt schema. Interna schema styr index. Fysisk lagring styr filstruktur. Generellt vill man aldrig exponera de egentliga tabellerna för användare (applikationer) - utan man använder sig av virtuella tabeller (vyer).

### Designstadie

Se de andra anteckningarna.

1. En konceptuell nivå - vilka generella entiteter, potentiella relationer etc. har vi?
2. Logisk design - styra upp relationer etc.
3. Implementera systemet i en DBMS

### Att välja DMBS

I teorin gäller det att studera alternativ. I praktiken är detta oftast valt innan ett projekt påbörjas av exempelvis en chef. Detta kan vara dåligt - teknologiskt driven istället för driven av krav. Det är ofta dyrt att köpa en DBMS och att byta system tar lång tid. Vi kan behöva köpa in licenser som säljs till användare.

## Från logisk design till fysisk

Se separata anteckningar.

Regler för integritet bör följas. Ibland kan beräknad data behövas sparas för att öka prestanda (triggers för summor, reaktioner på meddelande etc.). Redundans måste kontrolleras. Definiera virtuella relationer (vyer) - det är så vi exponerar information till användare (applikationer). Enbart DBA (database administrator) bör kunna jobba direkt mot de ursprungliga tabellerna. Använd triggers eller liknande för att verifiera att data är riktig.

## Balansera användning av beroenden och flexibilitet

Beroenden (constraints):

- Regel om vilken specifik data en entitet måste följa

Regler som måste följas:

- Entitetintergritet

Regler som kan implementeras:

1	* Business constraints
---	------------------------

Viktigt att läsa på ordentligt. Diskutera. Svårt utan större kunskap.

## Redundans

Kontrollerad redundans och distribution kan förbättra prestanda och tillgänglighet. Men "kontrollerad" betyder fler beräkningar (overhead) och en större kostnad för kommunikation.

## Datakonvertering och inladdning

Om inte en helt ny produkt utvecklas kan det finnas ett system som vi vill ersätta. Detta innebär att vi kan behöva konvertera data från ett legacy system. Det är sällsynt att börja från start. Ibland kan det vara fördelaktigt att konvertera data i en körning under en kortare tid. Ibland kanske det äldre systemet behöver fortsätta köras framöver.

```

1 BULK INSERT Course
2   FROM 'C:\course.txt'
3   WITH
4   (
5       FIRSTROW = 2,
6       FIELDTERMINATOR = '\t',
7       ROWTERMINATOR = '\n',
8       CODEPAGE = 'UTF-8'
9   )
10
11 SELECT * FROM Course;

```

## Testa data

När ett system konstruerats - innan all data läggs in - kan det vara bra att testa en delmängd av datan. Testa funktion av begränsningar, funktionalitet och prestanda. Prestanda kan förbättras vid ett senare tillfälle - funktion är mycket svårare. Man vill givetvis få så mycket rätt som möjligt från början.











Tester måste ske på alla nivåer av ANSI-SPARC. Externa och konceptuella nivåer testas för funktionalitet. Den fysiska nivån kan testas gentemot prestanda.

## Att välja filstruktur




### Sökmeter

Låt oss ha 10 000 rader i en tabell ur vilken vi vill hämta värden. Om vi inte har någon vidare struktur på tabellen måste vi utföra en s.k. *sequential scan* (genomsökning av hela tabellen). metoden är långsam men fungerar. Vi utföra gärna en s.k. *binary search* istället där vi kan utföra successiva delningar av sorterad data där vi istället för att börja från största till minsta värde påbörja vår sökning nära värdet.

### Datastrukturer

- Heap-filer
  - vi lägger saker och ting på varandra eller under varandra (append)
  -  snabb insert (bra för loggar)
  -  bra för små datamängder
  -  långsam select
  -  långsam join
- Sorterade filer
  -  snabb select
  -  snabb join
  -  långsam insert
  -  vi kan bara sortera efter en attribut
- Hashade filer (mellanting av heap och sortering)
  -  snabb select
  -  snabb join



-  snabb insert (något långsammare än heap)
-  vi kan bara hasha en attribut
-  filen är konstant i storlek (vi måste "hasha om" när vi ändrar storleken)

## Index

Primära index används för sekventiellt ordnad data efter en *unik nyckel* (PK / Primary Key). Sekundära index kan vara flera och behöver ej vara unika (FK / Foreign Keys). Har vi för många index tar det tid att räkna om index varje gång en förändring sker - desto fler desto långsammare. Om möjligt kan det vara bra att utföra många inserts (batch / bulk inserts). Man kan alltså använda en procedur som avvaktiverar indexet - lägger in all data och sedan aktiverar indexet igen.

- Clustered index
  - Varje rad är sorterad
  - Bra för många `BETWEEN` etc.
  - Maximalt ett per tabell
- Non-clustered index
  - Bra för höga krav på inserts
  - Bra för många `WHERE`, `JOIN` etc.

I MYSQL kan man säga att en primär nyckel är samma sak som en clustered index. Man skulle kunna säga att främmande nycklar är non-clustered index.

Tips:

- Använd inte index för extremt små tabeller (de får plats i RAM-minnet)
- Lägg till sekundära index där vi använder `JOIN`, `SORT`, `ORDER BY` och andra aggregerande funktioner
- Använd `EXPLAIN`: `EXPLAIN SELECT * FROM Users`
- Undvik index för attribut som ofta uppdateras
- Undvik index för attribut som består av långa texter ( `VARCHAR` )
  - använd inte index där bättre lösningar finns
  - (Dock finns nya versioner av MYSQL ett s.k. FULLTEXT-index som kan fungera)
- Dokumentera val av index ( `DESCRIBE` -kommandon, kommentarer, extern dokumentation)

## Vecka 4

### NoSQL

Från början pratade man om "no SQL" men det har blivit "not only SQL". Drivkraften var att få bort den långsamma och dyra relationsorienterade databasen för något mer effektivt och billigt. NoSQL handlar om ostrukturerad eller semi-strukturerad data. Grundidén är hög prestanda, tillförlitlighet och skalbarhet (något som inte stöds av traditionella databaser utöver "att kasta hårdvara på problemet" - vertikal skalbarhet). Den moderna synen på skalbarhet berör horisontell skalbarhet - man lägger till billiga maskiner i ett moln istället för att öka

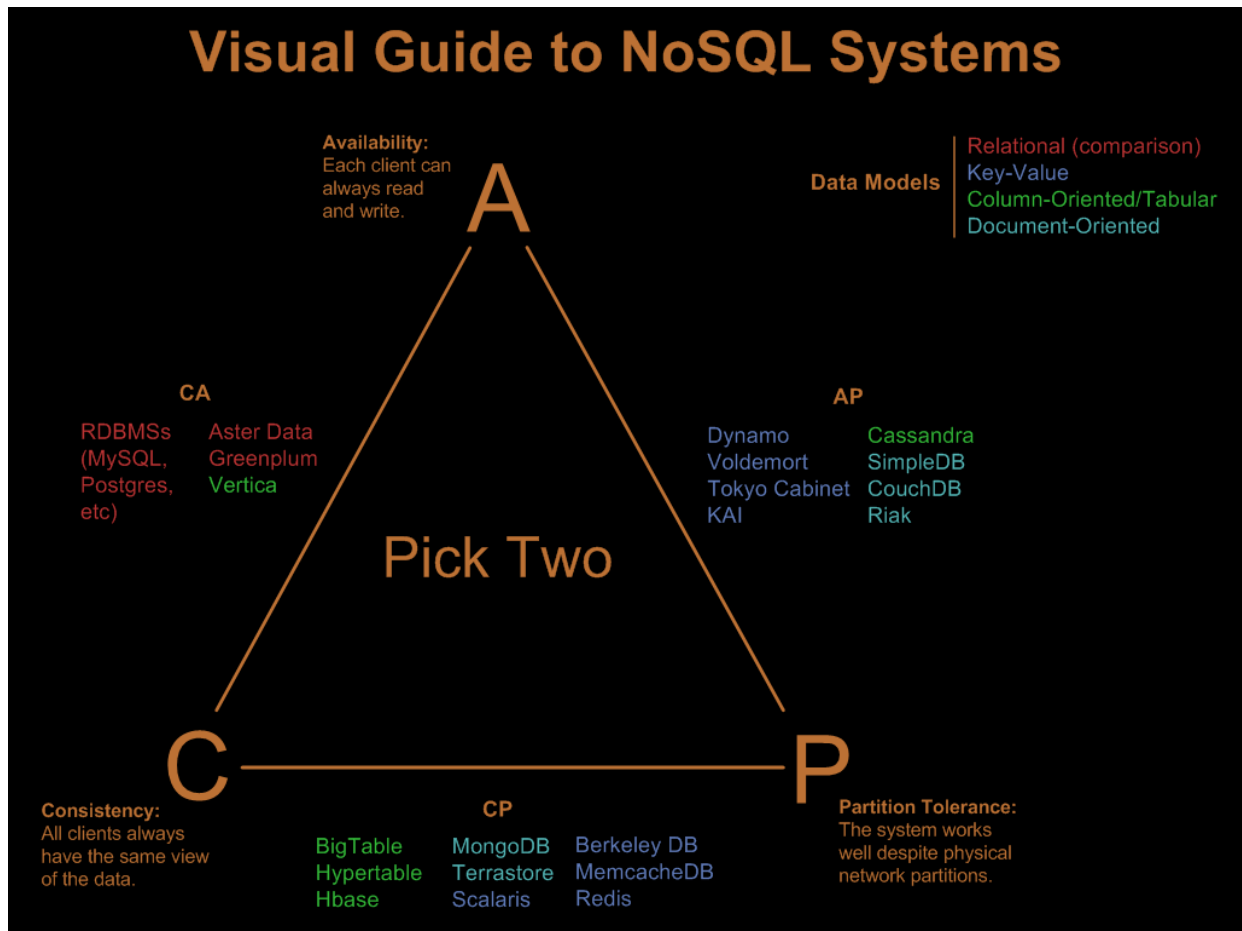
prestandan av en enda server. Normalt sett används inget SQL-språk och heller inget schema (tabeller med datatyper). Bland annat finns (S)CRUD - *Search, Create, Read, Update, Delete*. Joins sker vanligtvis manuellt. Vanligtvis stöds kontroll av historik av datan.

- Document Stores - MongoDB, CouchDB
  - JSON
  - Åtkomst till data via dokument-id eller innehåll
- Key-Value Stores - Redis, DynamoDB, Risk
  - Snabba, distribuerade hashtabeller
  - Associativa fält - data från id
- Graph - Neo4j, HyperGraphDB
  - Noder, kolumner
  - Optimerad för grafooperationer
- Column Stores - Cassandra, HBase (Hadoop)
  - Sparar data som {name, value, timestamp} där name är unikt

## CAP-teoremet

Det är omöjligt att tillfredsställa mer än två delar i CAP:

- Consistency - Varje gång vi läser får vi den senaste versionen av data eller ett felmeddelande
- Availability - Varje förfrågan får ett svar (som inte är ett felmeddelande) utan garanti för att det är den senaste versionen
- Partition tolerance - Systemet fortsätter att fungera även om en okänd mängd meddelanden tappas eller blir försenade i nätverket mellan databasens noder



([http://digbigdata.com/wp-content/uploads/2013/05/media\\_httpfarm5static\\_mevlk.png](http://digbigdata.com/wp-content/uploads/2013/05/media_httpfarm5static_mevlk.png))

## MongoDB

Dokumentorienterad, hashbaserad, schema-lös databas som använder BSON (JSON där B står för Binary). Det finns alltså inga beskrivningar för hur data definieras. API finns i många språk så som JavaScript, Python, o.s.v.

RDBMS	MongoDB
Database	Database
Table, View	Collection
Row	Document (BSON)
Column	Field
Join	Embedded Document
Foreign Key	Reference

## CRUD

- Create
  - `db.collection.insert(document)`
  - `db.collection.save(document)`

- `db.collection.update(query, update, {upsert: true})`
- Read
  - `db.collection.find(query, projection)`
  - `db.collection.findOne(query, projection)`
- Update
  - `db.collection.update(query, update, {upsert: true})`
- Delete
  - `db.collection.remove(query, justone)`

Projection är vilka fält vi vill hämta. Query går att likna vid en RDBMS *WHERE*.

```
1 db.message.insert({from: "Birgitt", to: "Fredrik", message: "Nice
lecture"})
2 db.message.insert({from: "Fredrik", to: "Christian", message: "Keep up
the good work!"})
3 db.message.find().pretty()
```