### Individuell inlämning 3

**Normalisering**  
*Normalisering* av relation syftar på processen att modellera/strukturera om på sådant sätt att varje relation får ett eget tema - en mer specifik representation än originalet. Målet är att skapa en mängd av relationer deriverade av en större relation som är stabilare och flexiblare.

En *anomali* i normaliserings mening syftar på något inkonsekvent, ofullständig eller motsägelsefull stadie av en databas, sådant att t.ex. information vid vissa uppdatering kan gå förlorad. Detta är precis vad normalisering ämnar åt att hämma. När relationer är *normaliserade* innefattar detta en design som är fri från *anomali*.

**Olika typer av anomalier**

* *Uppdateringsanomali*  
  Attribut finns på olika ställen så att det finns risk för att attribut som ska vara samma fast på olika ställen, inte blir uppdaterad på alla ställen (flera olika “versioner” av samma data).
* *Insättningsanomali*Vid insättning av information som kräver att annan data existerar, som i speciella fall inte kan existera vid tillfället.
* *Borttagningsanomali*När fält inte kan raderas för att relationen kräver att den existerar. T.ex. en *Klass* måste ha *stuID*, vilket gör att om sista studenten skulle raderas skulle databasen befinna sig i ett ogiltigt stadie.

**Funktionellt beroende**

Bortsett från beroende mellan tabeller kan även beroende mellan attribut definieras. Ett *funktionellt beroende* är ett “attributberoende” inom en relation som innebär att, för mängd A och B i relationsschemat R, så är varje värde i A associerad med exakt ett värde i B. Ett funktionellt beroende skrivs likt matematikens funktionslära *A -> B*.  
  
Det som är viktigt att poängtera med A -> B är att det inte medför att B -> A, utan det motsäger definitionen av funktionellt beroende på sådant sätt att det kan finnas flera värden i mängden B av R som är associerat med ett värde i mängden A av R (t.ex. studentID -> {fName, lName, soc}). Ett funktionellt beroende kan med andra ord ses som ett *many-to-one* förhållande - en *integritetbegränsning*.   
 **Normalisering med primärnycklar**

När en databas modelleras är första stadiet vanligen att översätta ER-modellen till en relationsmodellen. Detta är en bra tid att identifiera funktionella beroenden och välja ut passande primärnycklar till varje relation. För varje relation som ska normaliseras börjas det längst ner i 1NF, sedan 2NF och sist 3NF. Det finns ytterligare en *Boyce-Codd*.

**First Normal Form**

För att en relation ska få ha *First Normal Form*, *1NF* krävs det att varje attribut i relationen är *single-value*. Detta innebär att inget attribut får vara *multi-valued*, utan då måste andra tabeller skapas som innehåller detta multi-value-attribut. Attributet plockas alltså ut ur originalrelationen och sätta in i en nyskapad relation.  
  
**Second Normal Form (och fullt funktionellt beroende)**För att en relation ska få ha *Second Normal Form, 2NF* måste varje attribut i mängden B vara beroende av en nyckel *X*, men inte på något submängd av X. Om nyckeln X är en sammansatt nyckel (composite key) vara alla värden i B vara beroende av hela X, och inte på något primattribut (prime attribute) - *fullt funktionellt beroende*. Om något värde i B inte är fullt funktionellt beroende av nyckeln X, ska detta försättas i en ny tabell som ha primattributet av X som en primärnyckel.   
Ex.  
{nyckel1, nyckel2} -> {värde1, värde2, värde3} men också {nyckel2) -> {värde3} EJ 2NF  
Skriv om till:  
{nyckel1, nyckel2} -> {värde1, värde2} och {nyckel2} -> {värde3} 2NF

**Third Normal Form (och transitivt beroende)**

För att en relation ska få ha *Third Normal Form, 3NF* får inte relationen innehålla några *transitiva beroenden*. Ett transitivt beroende definieras som ett beroende där ett värde X i mängden B av relationen R är en nyckel i sig mot ett annat attribut Y i B. Sådant att värden i mängden A i relationen R är associerad med både X och Y i B, men är även associerad till Y via X. Ett vanligt exempel är när en adress har en postkod som associeras med en postort, men både postkod och postort är värden i mängden B. För att ta bort eventuella transitiva beroenden kan de placeras i en ny tabell.

**Boyce-Codd Normal Form**  
Denna normalformen är en lite striktare version av *Third Normal Form* som beskrev ovan. För att en relations ska få ha *Boyce-Codd Normal Form* ska den, förutom att uppfylla 3NF, uppfylla att alla dess determinanter är supernycklar. Om de inte är supernycklar ska de brytas upp. Om det endast finns en kandidatnyckel kommer kraven för att ha 3NF och Boyce-Codds normalform vara samma.  
  
**Egenskaper hos Relationsdekompositioner**

Mer formellt brukar normalisering ske genom en normaliseringsalgoritm som går ut på att börja med en *universalrelation* och jobba sig till via dekomposition tills att önskvärda relationer har uppnåtts. En universalrelation är en sorts sammansatt relation som innehåller alla attribut i en databas. Genom att successivt skapa mindre och mindre relationer kan normalformer uppnås och anomalier minimeras.   
  
Resultatet av den successiva dekompositionen har önskvärda egenskaper:

* *Attributbevarande*Attribut i relation ska finnas i minst en av de nyskapade mindre relationer. Det är vanligt att en databas innehåller en del repetition av något attribut.
* *Bereondebevarande*Funktionella beroende från originalrelation (relationen innan den gick igenom dekomposition) finns kvar i de nyskapade relationerna.
* *Förlustfri dekomposition*  
  Nyskapade relationer ska kunnas sammanfogas (natural join) tillbaka efter dekomposition utan att någon data försvinner eller läggs till. Förlustfri dekomposition är ofta svår att utföra på något annat än en *binär dekomposition*, och därför används oftast enbart binär dekomposition stegvist - om varje steg ses till att bli förlustfri kommer resultatet som helhet också vara förlustfri.

**Normaliseringsprocessen**  
Processen att normalisera kan strikt sett göras på två olika sätt, *analys* och *syntes*.  
  
*Analys* går ut på att lista attribut som databasen ska innehålla, identifiera funktionella beroenden bland attributen och använda dekomposition för att dela upp en universalrelation i mindre relationer. Detta är ett stadie där egenskaper såsom *förlustfri dekomposition* bör tas hänsyn till.

*Syntes* går ut på att identifiera grupper och gemenskap bland attribut för att skapa normaliserade relationer. Om analys ses som top-down är syntesprocessen bottom-up.

I praktiken behöver normalisering inte strikt följa analys eller syntes, utan ER-diagram används ofta och anses vara mer effektiv för normalisering. Normalisering med hjälp av ER-diagram kan ses som en sorts hybridlösning av analys och syntes.

**Prestandapåverkan av Normalisering**

När en databasdesigner modellerar en databas bör det finnas i åtanke vad normalisering orsakar i form av användning. Ett table innehållande samtliga attribut i en databas (universalrelation) kommer ha en snabb lästid då ingen sammanslagning (join) behöver göras. Om ett attribut hämtas mycket ofta måste det inte alltid vara värt prestandavis att normalisera vidare.   
Ex. Adress lagras enligt Student(stuID, gata, postkod, postadress, … ). Denna har inte 3NF då { stuID } -> { postadress } och { postkod } -> { postadress } (transitivt beroende).  
Men om adressen hämtas mycket ofta är det kanske inte bra att låta {postkod}->{postadress} finnas i en egen tabell (då detta skulle kräva sammanslagning varje gång), utan då nöjer man sig istället med att relationen har 2NF.  
En alternativt lösning är att använda *derived attribute* från ER-modellen som kan implementeras med SQL-procedures.

**Icke-normaliserade Databaser**

Traditionellt sett har relationsdatabaser varit industristandard för en god databas. Men på senare tid har det framkommit fall där en relationsdatabas inte är det bästa alternativet för att lagra data. Det gäller alltså data som inte passar inte i det traditionella sättet att se data i relation till varandra, data som inte behöver relationer eller data som prestandamässigt väljs att inte ha relationer. Kostnad av lagring av data är inte längre ett stort problem.

**Transaktioner**  
En *transaktion* i databasmening syftar på en enhet innehållande en mängd SQL-kommandon. Detta kan vara ett helt program eller en enskild förfrågan. En simpel transaktion mot en databas kan vara att uppdatera ett attribut för en viss rad i en viss tabell.   
Det som är viktigt att förstå är att varje SQL-kommando i sig består av instruktioner.   
Ex. UPDATE måste hitta -> hämta -> uppdatera -> skriv tillbaka.   
  
En transaktion kan bli *committed* (en slags färdigmarkering) eller *rolled back* (återläst). Rolled back innebär att databasen återställs tills det stadie den befann sig i innan transaktioner började. Efter en transaktion har blivit committed kan den inte rullas tillbaka, utan endast det som sker innan.  
  
En transaktion början medkommandot BEGIN TRANSACTION och avslutas END TRANSACTION.  
  
**ACID-egenskaper**

Alla transaktioner som sker bör följa fyra egenskaper som akronymen *ACID* definierar för att försäkra ett korrekt stadie av databasen.

* *Atomicity*Varje transaktion ska ske som “allt eller inget”. Antingen sker hela transaktioner eller så sker ingenting. Då varje transaktion kan vara i flera steg måste databasen har ett dedikerat *Recovery System* ifall något skulle gå fel mitt i transaktionen.
* *Consistency*Flera användare ska kunna använda (både skriva till och läsa från ) databasen samtidigt. Det ska inte vara upp till användare att deras transaktioner ska kunna ske samtidigt som andra användares, utan ett dedikerat *Currency Control System* ska finnas.
* *Isolation*Flera transaktioner ska kunna köras samtidigt utan att de påverkar varandra. Transaktioner ska inte kunna övervakas under tiden de körs, utan de ska exekveras som om det körs seriellt (det ska verka som om ingen concurrency finns). *Currency Control System* ska garantera isolation.
* *Durability*Varje transaktion som har blivit committed måste registreras i en sorts log. Om databasen skulle krascha mitt i exekvering av transaktionen ska ett *Recovery System* kunna läsa tillbaka vad som skulle ha hänt innan databasen kraschade, och börja där.

**Flera användare i en databas (motivation till Concurrency System)**  
Som beskrevs ovan beskriver ACID en sorts best-practice vid *multiprogramming* för en databas (C:et i ACID). Men hur löses concurrency-problem? Det finns huvudsakligen tre problem med concurrency (boken tar endast upp tre, finns fler):

* *Lost update problem*Alla uppdateringar av data sker huvudsakligen i LÄS-UPPDATERA-SKRIV  
  Om användare 1 ska skriva till gemensam data, och just efter LÄS kommer användare 2 och också ska uppdatera samma data. Då kommer användare 2 att läsa från original värdet som ännu inte uppdaterats av användare 1. Detta kan resultera i att uppdatering från användare 1 går förlorad.
* *Uncommitted update problem*Om uppdatering från användare 1 inte blivit committed innan användare 2 vill läsa av denna data, och användare 1 sedan väljer ROLL BACK av sina ändringar, kommer användare 2 inte bry sig om denna ROLL BACK.
* *Problem of inconsistent analysis*  
  Data som finns tillgänglig för stunden för analys (t.ex. för att ta SUM av olika kolumner) förändras under tiden beräkningen sker.

## Individuell uppgift

1. { städObjekt, datum, städareId } -> { städTid } **FFB**  
   { städareID } -> { städareNamn, förmanID, förmanNamn } **Partial**  
   { förmanID } -> { förmanNamn } **Transistivt**
2. Normalform som relationen befinner sig i  
   Endast 1NF, då den inte har några multivariabel-attribut  
   Inte 2NF då *städareID* kan självt bestämma *städareNamn, förmanID* och *förmanNamn*.  
   Inte 3NF då den innehåller en transitiv relation: {förmanID} -> {förmanNamn}
3. Exempel på anomalier  
   - Uppdateringsanomali  
   Byta förmanNamn. Då varje städare har en kopplad förmanID behöver varje fält uppdateras. Om endast en rad för förmanNamn uppdateras kommer databasen befinnas i ett ogiltigt stadie där flera förmanID pekar på olika förmanNamn.  
     
   - Insättningsanomali  
   Ny förman. En förman kan inte sättas in utan att en städare existerar.  
     
   - Borttagningsanomali  
   Ta bort städobjekt. Om en städobjekt försvinner kommer all data försvinna kopplad till det städobjektet. Skulle en städare endast ha städat på samma objekt vid tillfället kommer all data tas bort om denna städare. Om detta var den enda städaren på sin förman skulle det innebära att en förman inte har några städare kopplade till sig.
4. Normalisera relationen  
   - 1NF -> 2NF  
   Vi måste bryta ut städareID ur primärnyckeln.  
   Vi får då två tabeller:  
     
   Personal(städareID, städareNamn, förmanID, förmanNamn)  
   Städ(städObjekt, datum, *studentID*, städTid)  
     
   - 2NF -> 3NF  
   Vi måste bryta ut förmanID ur Städare-tabellen för att ta bort det transitiva förhållandet ( {förmanID} -> {förmanNamn} )  
     
   Förman(förmanID, förmanNamn)  
   Städare(städareID, städareNamn, *förmanID*)  
   Städ(städObjekt, datum, *studentID*, städTid)  
     
   Test( key1, key2, att1, att2 )