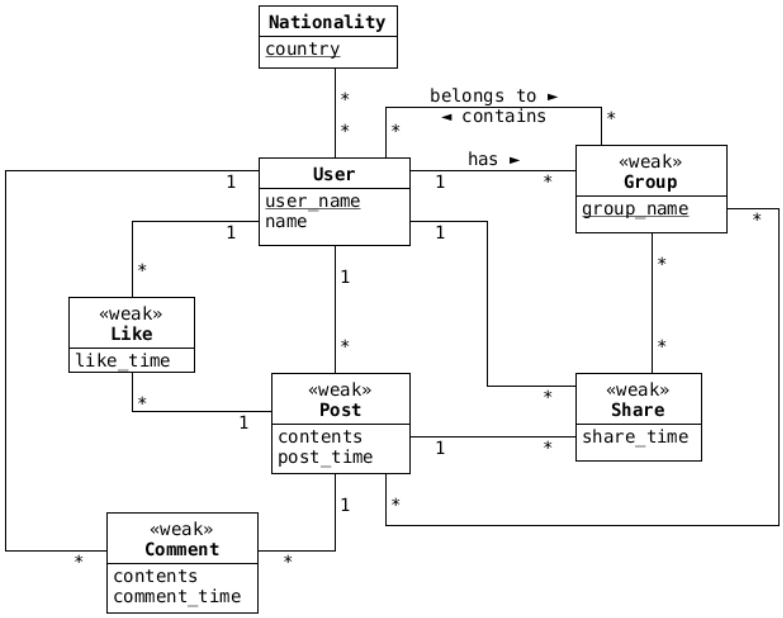
Lösningar till tentamen i EDAF75

4 april 2018

Lösning 1

(a) Här är ett förslag till E/R-modell:



Det finns flera rimliga alternativa sätt att modellera, så du behöver inte vara orolig bara för att du inte gjort precis som i figuren ovan.

(b) Det finns ganska många svaga 'entity sets', nedan har vi infört flera 'surrogate keys':

```
users(user_name, name)
nationalities(country)
citizenships(user_name, country)
groups(group_id, user_name, group_name)
group_members(group_id, user_name)
posts(post_id, user_name, contents, post_time)
accessible_posts(post_id, group_id)
likes(like_id, post_id, user_name, like_time)
comments(comment_id, post_id, user_name, contents, comment_time)
shares(share_id, post_id, user_name, share_time)
group_shares(share_id, group_id)
```

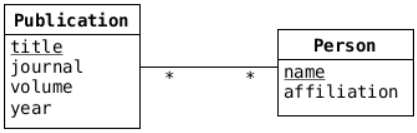
(c) Som vanligt finns det flera sätt att lösa uppgiften (man kan till exempel ha ett `DISTINCT` inne i en `COUNT`), ett sätt att göra det i steg är följande:

```
WITH
  complicit_users AS (
```

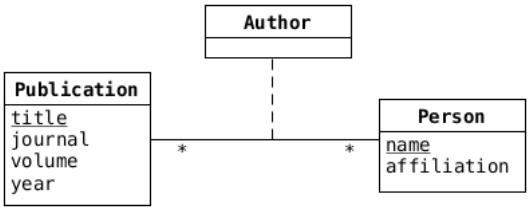
```
    SELECT DISTINCT user_name
    FROM   users
           JOIN citizenships USING (user_name)
           JOIN likes USING (user_name)
           JOIN posts USING (post_id)
    WHERE  country = 'Australia'
           AND contents LIKE '%#SANDPAPER%'
  )
SELECT COUNT()
FROM   complicit_users
```

Lösning 2

(a) E/R-diagram: Den enkla lösningen är:



Om vi vill så kan vi använda en tom associationsklass för aassociationen mellan `Publication` och `Person`:



(b) Attributen `name` och `title` är strängar, och de är båda främmande nycklar – dessutom utgör de tillsammans nyckel för tabellen:

```
CREATE TABLE authors (
  name    TEXT,
  title   TEXT,
  PRIMARY KEY(name, title),
  FOREIGN KEY(name) REFERENCES persons(name),
  FOREIGN KEY(title) REFERENCES publications(title)
);
```

(c) Här räcker det med en vanlig SFW (*select-from-where*):

```
SELECT    title, journal
FROM      publications
WHERE     year = 2017
ORDER BY  title;
```

(d) Eftersom titeln är naturlig nyckel i `authors`, så räcker det att vi joinar `authors` och `persons` (vi behöver inte titta vidare in i `publications` – det hade vi behövt göra om vi istället hade haft en surrogatnyckel i `publications`):

```
SELECT DISTINCT title
FROM   authors
       JOIN persons USING (name)
WHERE  affiliation = ”Lund University”;
```

(e) Vi kan lösa uppgiften antingen med en outer join:

```
SELECT     name
FROM       persons
           LEFT JOIN authors USING (name)
WHERE      title IS NULL
           AND affiliation = ”Lund University”;
```

eller med en subquery:

```
SELECT  name
FROM    persons
WHERE   affiliation = ”Lund University”
        AND name NOT IN
        (SELECT name
         FROM   authors);
```

(f) Vi kan gruppera författarna på titel, och se vilka titlar som har mer än en författare:

```
SELECT    title
FROM      authors
GROUP BY  title
HAVING    COUNT() > 1;
```

(g) Vi gör en left outer join på `persons` och `authors`, och får på så vis med samtliga personer (vi anropar `count` med `title` som argument, eftersom vi vill att de författare som inte skrivit något, och blivit ’paddade’ med `NULL` skall räknas som 0): med :

```
SELECT     name, COUNT(title) AS cnt
FROM       persons
           LEFT JOIN authors USING (name)
GROUP BY   name
ORDER BY   cnt DESC;
```

Lösning 3

(a) Vi kan inte på rak arm säga att något attribut måste ingå i en nyckel (alla attribut kan härledas med hjälp av något/några av de andra), så vi testar alla 1-attributsnycklar:

$$\{A\}^+ = \{AB\}$$
$$\{B\}^+ = \{B\}$$
$$\{C\}^+ = \{CD\}$$
$$\{D\}^+ = \{D\}$$

Vi testar därefter alla möjliga två-attributsnycklar:

$$\{AB\}^+ = \{AB\}$$
$$\{AC\}^+ = \{ABCD\}$$
$$\{AD\}^+ = \{ABCD\}$$
$$\{BC\}^+ = \{ABCD\}$$
$$\{BD\}^+ = \{ABCD\}$$

$\{CD\}^+ = \{CD\}$

Så, $\{AC\}$, $\{AD\}$, $\{BC\}$ och $\{BD\}$ är nycklar. Det går inte att bilda kombinationer med tre eller fyra attribut som inte har någon av nycklarna ovan som delmängder, så vi har hittat alla våra nycklar.

(b) FD_1 och FD_4 bryter mot BCNF, eftersom deras vänsterled inte är supernycklar – FD_2 och FD_3 bryter inte mot BCNF eftersom deras vänsterled är nycklar (och därmed även supernycklar).

(c) FD_2 och FD_3 är i BCNF, och därmed även i 3NF, och i både FD_1 och FD_4 har vi högerled som ingår som delar av nycklar, så relationen R är i 3NF.

(d) Vi kan baka ihop FD_2 och FD_3 till ett funktionellt beroende, och har därefter:

Relation:	$R(A, B, C, D)$
Beroenden:	$A \to B$, $BD \to AC$, $C \to D$
Nycklar:	$\{AC\}$, $\{AD\}$, $\{BC\}$, $\{BD\}$

och kan här välja att bryta upp på antingen $A \to B$ eller $C \to D$, eftersom de är de enda som bryter mot BCNF. Vi börjar med $A \to B$:

Relation:	$R_1(A, B)$
Beroenden:	$A \to B$
Nycklar:	$\{A\}$

  Denna är i BCNF, eftersom vänsterledet i vårt enda funktionella beroende är en supernyckel (alla relationer med två attribut är i BCNF, för om det finns något funktionellt beroende i relationen $R(A, B)$, så måste det vara av formen $A \to B$ eller $B \to A$, och A eller B skulle i så fall vara en nyckel).
Relation:	$R_2(A, C, D)$
Beroenden:	$C \to D$
Nycklar:	$\{AC\}$

  Denna är inte i BCNF, eftersom $C \to D$ har ett vänsterled som inte är en supernyckel. Vi måste alltså bryta upp ett steg till, och kan bara göra det på $C \to D$:
Relation:	$R_3(C, D)$
Beroenden:	$C \to D$
Nycklar:	$\{C\}$

    Vänsterledet i det enda beroendet är en supernyckel, alltså är relationen i BCNF.
Relation:	$R_4(A, C)$
Beroenden:	inga
Nycklar:	$\{AC\}$

    Vi har inga beroenden, alltså är vi i BCNF.

Sammanfattningsvis får vi relationerna:

- $R_1(A, B)$
- $R_3(C, D)$
- $R_4(A, C)$

Om vi istället väljer att bryta upp vår ursprungliga relation på $C \to D$, så får vi:

Relation:	$R_1(C, D)$
Beroenden:	$C \to D$
Nycklar:	$\{C\}$

  Denna är i BCNF, eftersom den bara har två attribut (vänsterledet i vårt enda funktionella beroende är en supernyckel).
Relation:	$R_2(A, B, C)$
Beroenden:	$A \to B$
Nycklar:	$\{AC\}$

  Denna är inte i BCNF, eftersom $A \to B$ har ett vänsterled som inte är en supernyckel. Vi måste alltså bryta upp ett steg till, och kan bara göra det på $A \to B$:

–

Relation:	$R_3(A, B)$
Beroenden:	$A \to B$
Nycklar:	$\{A\}$

Vänsterledet i det enda beroendet är en supernyckel, alltså är relationen i BCNF.

–

Relation:	$R_4(A, C)$
Beroenden:	inga
Nycklar:	$\{AC\}$

Vi har inga beroenden, alltså är vi i BCNF.

Sammanfattningsvis får vi relationerna:

- $R_1(C, D)$
- $R_3(A, B)$
- $R_4(A, C)$

Detta är samma relationer som vi fick ovan (med lite annan numrering).

Lösning 4

(a) Om attributen $B_1, B_2, \ldots B_m$ entydigt bestäms av värdet på attributen $A_1, A_2, \ldots A_n$, så sägs $B_1, B_2, \ldots B_m$ vara *funktionellt beroende* av $A_1, A_2, \ldots A_n$, och vi skriver

$$A_1, A_2, \ldots A_n \to B_1, B_2, \ldots B_m$$

Man säger även ibland att $A_1, A_2, \ldots A_n$ *funktionellt bestämmer* $B_1, B_2, \ldots B_m$.

En *supernyckel* i en relation är en uppsättning attribut som entydigt bestämmer värdet på samtliga övriga attribut i relationen – en *nyckel* är en supernyckel som inte innehåller något attribut som själv beror på övriga attribut i nyckeln (den kallas då *minimal*).

För en nyckel gäller alltså att samtliga övriga attribut i relationen är funktionellt beroende av dem.

(b) Vi kan ta en tabell med information om våra vänner, vi kan ha attributen:

- `namn`, vi antar att namnen är unika
- `telefonnummer`, vi antar att varje person bara har ett telefonnummer, och att inga vänner delar telefon med varanda
- `gatuadress`, vi antar att vännerna bara har en bostadsplats, men att flera vänner kan dela en adress
- `postnummer`

Här har vi följande funktionella beroenden:

FD_1: `name` → `telefonnummer`, `gatuadress`, `postnummer`
FD_2: `telefonnummer` → `namn`, `gatuadress`, `postnummer`
FD_3: `gatuadress` → `postnummer`

(Här är `namn` och `telefonnummer` var för sig nycklar för hela relationen – relationen är inte i BCNF, eftersom FD_3 har ett vänsterled som inte är en supernyckel.)

Lösning 5

(a) Vi använder transaktioner av två olika skäl:

- Vi vill kunna ge samtidig access till databasen från flera processer.
- Vi vill kunna hantera olika slags fel som kan uppstå.

Begreppet ACID (*Atomicity-Consistency-Isolation-Durability*) kretsar kring både samtidighet och felåterhämtning.

Ett exempel då vi vill använda transasktioner är när vi i en bank-databas vill flytta pengar mellan två konton[1]. Vi startar en transaktion när vi börjar transfereringen, och avslutar transaktionen när vi ökat det ena saldot, och minskat det andra. Det gör att:

[1] Detta är något man numera ofta försöker undvika genom att man lagrar transfereringar snarare än uppdaterar saldon.

- ingen annan kan ändra saldot på våra konton under själva transaktionen (isolation),
- vi kan göra en roll-back om något saldo skulle bli negativt (atomicity, consistency),
- vi kan återställa databasen om vi får något systemfel (atomicity, consistency).

(b) Vi kan starta en transaktion med `START TRANSACTION` (syntaxen skiljer mellan olika databaser, i några skriver man istället `BEGIN`), och vi kan avsluta den med `COMMIT` eller `ROLLBACK` (vi skulle ange två kommandon, välj två av ovanstående tre). `COMMIT` markerar att vår transaktion har avslutats som vi planerade, och att våra tabeller skall uppdateras. `ROLLBACK` gör att vi tar tillbaka de uppdateringar vi har gjort i transaktionen.

(c) Vi kan välja olika grader av isolering för våra transaktioner, minimal isolering får vi med `ISOLATION LEVEL READ UNCOMMITTED` – här riskerar vi att läsa data som ännu inte 'commit'-ats i någon annan transaktion (och som kanske kommer att tas tillbaka i en `ROLLBACK`). Fördelen med att använda `READ UNCOMMITTED` är att vi inte behöver låsa databasen, nackdelen är att vi har färre garantier rörande våra data, vi kan rent av få data som aldrig skulle varit i databasen (som vid en `ROLLBACK`).

Om vi istället väljer `ISOLATION LEVEL READ COMMITTED`, så får vi bara data som har 'commit'-ats, vi ökar därmed kvaliteten på våra data, men detta kräver att vi måste låsa delar av databasen.

I `REPEATABLE READ` garanteras att vi inne i vår transaktion får samma svar på samma query, så länge vi inte själva ändrar i tabellen, *eller någon annan lägger till nya rader i den underliggande tabellen* (detta kallas *"phantom reads"*).

Lösning 6

En inre join kombinerar två tabeller R_A och R_B med avseende på något *join-predikat* – ett sådant predikat tar attribut från båda tabellerna och ser var raderna matchar varandra: $p(a_{i1}, a_{i2}, \ldots, b_{j1}, b_{j2}, \ldots)$. Den resulterande tabellen får i sina rader samtliga projicerade kolumner från de båda tabellerna, för de kombinationer av rader i R_A och R_B för vilka join-predikatet gäller – rent logiskt är detta samma som om vi tar en cross join mellan R_A och R_B, och sedan använder join-predikatet för selektion (men vi kan slippa upprepade kolumner, och frågeoptimeraren kan ibland konstruera effektivare sökningar om den vet att vi vill göra en inner join – dessutom blir frågan lättare att läsa om den struktureras med inner joins).

Skillnaden mellan en inner join och en outer join är att vi i en outer join matchar *samtliga* rader i den ena (left eller right outer join) eller båda (full outer join) – de rader som saknar motsvarighet i den andra tabellen kombineras ihop med `NULL`-värden.

Som exempel kan vi ta en databas liknande den med vänner som vi hade i problem 4b – men nu även med böcker som kan lånas ut:

```
friends(name, phone, street_addr)
books(isbn, title)
loans(name, isbn, borrowed_on)
```

Vi kan generera en lista på alla de böcker våra vänner lånat, antingen bara en lista med namn och titlar:

```
SELECT  name, title
FROM    loans
        JOIN books USING (isbn);
```

eller en lite snyggare utskrift (där vi slår samman alla böcker på en rad i utskriften, så att vi får en rad per vän):

```
SELECT    name, GROUP_CONCAT(title, ", ") AS borrowed
FROM      loans
          JOIN books USING (isbn)
GROUP BY  name;
```

I dessa båda queries får vi bara med de vänner som har lånat någon bok, de vänner som inte kan kombineras med en rad i `loans` kommer inte med.

Om vi vill ha med *alla* vänner, oavsett om de lånat eller ej, så kan vi göra en *left outer join* med `friends`-tabellen till vänster:

```
SELECT    name, COALESCE(GROUP_CONCAT(title, ", "), "no loans") AS borrowed
FROM      friends
          LEFT JOIN loans USING (name)
          LEFT JOIN books USING (isbn)
GROUP BY  name;
```

Vi kan även använda denna teknik för att hitta just de vänner som inte har lånat någon bok – de har parats ihop med `NULL`-värden:

```
SELECT  name
FROM    friends
        LEFT JOIN loans USING (name)
WHERE   isbn IS NULL
```