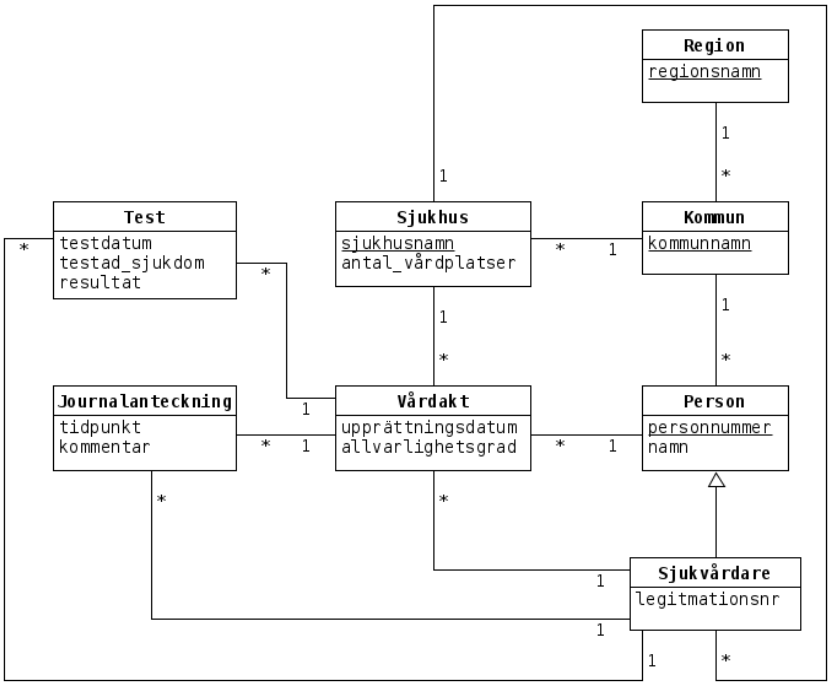
Lösningar till tentamen i EDAF75

17 mars 2020

Lösning 1

(a) Förslag till ER-modell (det finns alternativa lösningar):



(b) Relationerna `Vårdakt`, `Test` saknar primär nyckel och har fått surrogatnyckel *aktnummr*, resp. *testnummer*. Många relationer för förhållanden har förenklats bort då det är en till många.

```
regioner(regionnamn)
kommuner(kommunnamn, regionnamn)
sjukhus(sjukhusnamn, kommunnamn, antal_vårdplatser)
sjukvårdare(personnummer, legimationsnr, sjukhusnamn)
personer(personnummer, namn, kommunnamn)
vårdakter(aktnummer, personnummer, sjukhusnamn, upprättelsedatum, allvarlighetsgrad)
journalanteckningar(aktnummer, tidpunkt, kommentar, sjukvårdare_personnummer)
tester(testnummer, aktnummer, sjukvårdare_personnummer,
       testdatum, testad_sjukdom, resultat)
```

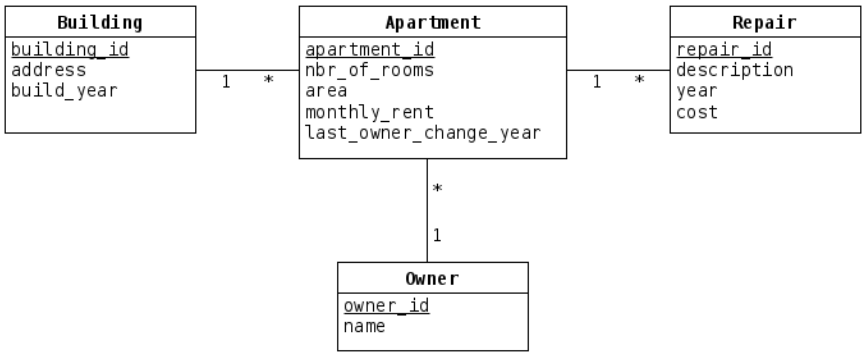
(c) Med de relationer vi har ovan får vi något i stil med:

```
SELECT  personnummer, kommunnamn
FROM    personer
JOIN    kommuner
```

```
USING   (kommunnamn)
JOIN    vårdakter
USING   (personnummer)
JOIN    journalanteckningar
USING   (aktnummer)
WHERE   regionsnamn = 'Skåne'
        AND kommentar LIKE '%inbillningssjuk%';
```

Lösning 2

(a) Ett sätt att rita diagrammet är:



Alla tabellerna har invented keys, och man kan betrakta dem som implementationsdetaljer, så det är OK att stryka dem ur diagrammet.

(b) Vi kan välja att representera `apartment_id`, `building_id` och `owner_id` som exempelvis text-strängar, eller som något slags heltalsräknare:

```
DROP TABLE IF EXISTS apartments;
CREATE TABLE apartments (
  apartment_id             TEXT,
  building_id              TEXT,
  owner_id                 TEXT,
  nbr_of_rooms             INT,
  area                     INT,
  monthly_rent             DECIMAL(10,2),
  latest_owner_change_year INT,
  PRIMARY KEY (apartment_id),
  FOREIGN KEY (building_id) REFERENCES buildings(building_id),
  FOREIGN KEY (owner_id) REFERENCES owners(owner_id)
);
```

(c) För att lista de senaste 10 nya medlemmarna i bostadsrättsföreningen räcker det med (eftersom vi slipper hantera problemet med duplicerade namn):

```
SELECT    name
FROM      apartments
JOIN      owners
USING     (owner_id)
ORDER BY  latest_owner_change_year DESC
LIMIT     10;
```

(d) När vi skall lista namnen på dem som äger enrumslägenheter på Storgatan 1, och vill undvika dubletter för dem som äger flera enrummare just där, så kan vi använda `SELECT DISTINCT`. Vi gör dock inga avdrag om ni inte gör det – det är alltså OK om de som har flera enrummare i huset kommer med flera gånger, det är i själva verket rimligt att man i listningen vill kunna se vilka som har flera lägenheter.

```
SELECT DISTINCT name
FROM     owners
JOIN     apartments
USING    (apartment_id)
JOIN     buildings
USING    (building_id)
WHERE    nbr_of_rooms = 1
         AND address = 'Storgatan 1';
```

(e) När vi skall höja hyran för en given adress, och vill få fram alla lägenheter på den aktuella adressen, så kan vi använda en subquery:

```
UPDATE   apartments
SET      monthly_rent = 1.018 * monthly_rent
WHERE    building_id IN (
            SELECT  building_id
            FROM    buildings
            WHERE   address = 'Storgatan 1'
         );
```

(f) För att lista de ägare som har mer än en lägenhet kan vi använda `GROUP BY`:

```
SELECT    owner_id, name
FROM      owners
JOIN      apartments
USING     (owner_id)
GROUP BY  owner_id
HAVING    count( ) > 1;
```

(g) När vi vill lista lägenhetsnummer och adress för de lägenheter i vilka inga reparationer har gjorts efter 2010-01-01, så kan vi välja mellan att använda en subquery:

```
SELECT  apartment_id, address
FROM    apartments
JOIN    buildings
USING   (building_id)
WHERE   apartment_id NOT IN (
            SELECT  apartment_id
            FROM    repairs
            WHERE   year > 2010
        );
```

eller att använda outer join (i detta fall måste vi se till att den tabell som vi joinar med innehåller rätt värden, vi kan använda en `WITH`-sats för det):

```
WITH
  recent_repairs(apartment_id, repair_id) AS (
    SELECT apartment_id, repair_id
    FROM   repairs
    WHERE  year > 2010
  )
SELECT     apartment_id, address
```

```
FROM       apartments
JOIN       buildings
USING      (building_id)
LEFT JOIN  recent_repairs
USING      (apartment_id)
WHERE      repair_id IS NULL;
```

(h) Även när vi listar byggnads-id och total reparationskostnad per hus för reparationer gjorda 2019, så kan vi välja mellan en outer join eller en subquery (en outer join känns dock mest naturlig).

Om vi gör det med en outer join så vill vi börja med alla byggnader, och först göra en outer join på alla lägenheter, och sedan en outer join på alla reparationer. Men eftersom vi sedan bara vill räkna reparationer under 2019 måste vi vara lite försiktiga om vi samtidigt vill ha kvar samtliga byggnader, vi kan inte först göra två outer joins och sedan använda en `WHERE`-sats, eftersom vi då riskerar att tappa byggnader igen – ett sätt att lösa uppgiften är att:

- först göra en `LEFT JOIN` mellan `buildings` och `apartments`, och sedan
- göra en `LEFT JOIN` med alla reparationer under 2019.

Vi kan göra det på olika sätt, ett sätt är att börja med att välja ut våra reparationer först:

```
WITH
  repairs_in_2019(apartment_id, cost) AS (
    SELECT   apartment_id, cost
    FROM     repairs
    WHERE    year = 2019
  )
SELECT     building_id, coalesce(sum(cost), 0) AS total_cost
FROM       buildings
LEFT JOIN  apartments
USING      (building_id)
LEFT JOIN  repairs_in_2019
USING      (apartment_id)
GROUP BY   building_id;
```

Vi kan även skapa vårt urval av reparationer ’på plats’ inne i vår `LEFT JOIN`:

```
SELECT     building_id, coalesce(sum(cost), 0) AS total_cost
FROM       buildings
LEFT JOIN  apartments
USING      (building_id)
LEFT JOIN  (SELECT apartment_id, cost
            FROM   repairs
            WHERE  year = 2019
           )
USING      (apartment_id)
GROUP BY   building_id;
```

Om vi förutsätter att det finns minst en lägenhet i varje byggnad så behöver vi inte börja med `buildings`, eftersom vi då kommer att få samtliga värden på `building_id` genom att bara gå igenom `apartments` (vi ger inget avdrag för detta).

Men en ’correlated subquery’ kan vi istället skriva frågan som:

```
SELECT    building_id,
          coalesce((
            SELECT  sum(cost)
            FROM    buildings AS i
```

```
            JOIN    apartments
            USING   (building_id)
            JOIN    repairs
            USING   (apartment_id)
            WHERE   o.building_id = i.building_id
                    AND year = 2019
            GROUP BY i.building_id), 0) AS total_cost
FROM      buildings AS o;
```

(i) För att ge en lista på samtliga fastigheter (adressen) och deras totala boyta kan vi återigen använda en outer join:

```
SELECT      address, coalesce(sum(area), 0) AS total_area
FROM        buildings
LEFT JOIN   apartments
USING       (building_id)
GROUP BY    building_id;u
```

Även i denna uppgift är det OK att förutsätta att det finns lägenheter i samtliga byggnader, vi kan då använda en inner join istället för vår `LEFT JOIN` ovan.

Lösning 3

(a) Vi har relationen $R(A, B, C, D, E, F, G)$, och de funktionella beroendena:

FD_1: $A \longrightarrow BC$

FD_2: $DE \longrightarrow F$

FD_3: $ADE \longrightarrow G$

Vi ser att A, D, och E måste ingå i alla nycklar, eftersom de inte förekommer i högerledet för något beroende.

Så vi börjar med att testa $\{ADE\}^+$, och får då:

$$\{ADE\}^+ = \{ABCDEFG\}$$

Så, $\{ADE\}$ är en nyckel. Vi vill hitta samtliga kandidatnycklar, men eftersom attributen A, D och E alla måste ingå i en nyckel, och de tillsammans utgör en nyckel, så kan det inte finnas några andra nycklar ($\{ADE\}$ skulle vara en del av en sådan nyckel, och den skulle därmed inte vara 'minimal').

(b) Vi är inte i BCNF eftersom FD_1 och FD_2 har vänsterled som inte är supernycklar.

(c) Vi är inte i 3NF eftersom FD_1 och FD_2 bryter mot BCNF, och deras högerled inte är delar av nycklar.

(d) Vi kan börja med att utveckla (alltså beräkna det transitiva höljen för) de funktionella beroenden som vi kan tänkas bryta upp på, dvs FD_1 och FD_2, men det gör ingen skillnad – de har båda sina transitiva höljen som högerled.

Sammanfattningsvis har vi:

Relation:	$R(A, B, C, D, E, F, G)$
Beroenden:	$A \longrightarrow BC$, $DE \longrightarrow F$, $ADE \longrightarrow G$
Nycklar:	$\{ADE\}$

Vi kan välja att bryta ner R med hjälp av $A \longrightarrow BC$, som ju bryter mot BCNF, och får då relationerna $R_1(A, B, C)$ och $R_2(A, D, E, F, G)$. För R_1 får vi:

Relation:	$R_1(A, B, C)$
Beroenden:	$A \longrightarrow BC$
Nycklar:	$\{A\}$

Det enda funktionella beroendet har ett vänsterled som är en superkey, så vi är i BCNF.

För R_2 har vi (vi stryker BC, eftersom vi nu kan slå upp dem i R_1 med hjälp av A):

Relation:	$R_2(A, D, E, F, G)$
Beroenden:	$DE \longrightarrow F$, $ADE \longrightarrow G$
Nycklar:	$\{ADE\}$

Denna är inte i BCNF, eftersom $DE \longrightarrow F$ har ett vänsterled som inte är en superkey – vi får därför bryta upp på detta beroende och får $R_{2a}(D, E, F)$ och $R_{2b}(A, D, E, G)$.

För R_{2a} gäller:

Relation:	$R_{2a}(D, E, F)$
Beroenden:	$DE \longrightarrow F$
Nycklar:	$\{DE\}$

Här är vi i BCNF, eftersom vänsterledet i dess enda funktionella beroende, $DE \longrightarrow F$, är en superkey.

För R_{2b} får vi:

Relation:	$R_{2b}(A, D, E, G)$
Beroenden:	$ADE \longrightarrow G$
Nycklar:	$\{ADE\}$

som även den är i BCNF, eftersom dess enda funktionella beroende har ett vänsterled som är en superkey.

Så, vi delar ner $R(A, B, C, D, E, F, G)$ till $R_1(A, B, C)$, $R_{2a}(D, E, F)$ och $R_{2b}(A, D, E, G)$, och är därmed i BCNF.

Om vi istället bryter upp R på $DE \longrightarrow F$, så kommer vi att få precis samma relationer – det är inte alltid det blir så, men i detta exempel blir det så:

För R_1 får vi:

Relation:	$R_1(D, E, F)$
Beroenden:	$DE \longrightarrow F$
Nycklar:	$\{DE\}$

Det enda funktionella beroendet har ett vänsterled som är en superkey, så vi är i BCNF.

För R_2 har vi (vi stryker F, eftersom vi nu kan slå upp den i R_1 med hjälp av DE):

Relation:	$R_2(A, B, C, D, E, G)$
Beroenden:	$A \longrightarrow BC$, $ADE \longrightarrow G$
Nycklar:	$\{ADE\}$

Denna är inte i BCNF, eftersom $A \longrightarrow BC$ har ett vänsterled som inte är en superkey – vi får därför bryta upp på detta beroende och får $R_{2a}(A, B, C)$ och $R_{2b}(A, D, E, G)$.

För R_{2a} gäller:

Relation:	$R_{2a}(A, B, C)$
Beroenden:	$A \longrightarrow BC$
Nycklar:	$\{A\}$

Här är vi i BCNF, eftersom vänsterledet i dess enda funktionella beroende, $A \longrightarrow BC$, är en superkey.

För R_{2b} får vi:

Relation:	$R_{2b}(A, D, E, G)$
Beroenden:	$ADE \longrightarrow G$
Nycklar:	$\{ADE\}$

som även den är i BCNF, eftersom dess enda funktionella beroende har ett vänsterled som är en superkey.

Så, vi delar ner $R(A, B, C, D, E, F, G)$ till $R_1(D, E, F)$, $R_{2a}(A, B, C)$ och $R_{2b}(A, D, E, G)$, och är därmed i BCNF. Och vi får alltså precis samma resultat som när vi bröt ner på $A \longrightarrow BC$, bortsett från namngivningen (men observera att vi alltså inte alltid vi får samma uppdelning oavsett vilket trilskande beroende vi väljer att bryta upp på, det bara råkade bli så i detta fall).

Lösning 4

(a) Transaktioner i databaser löser två olika klasser av problem:

- de kan skydda processer som kör samtidigt från att förstöra för varandra, och
- de kan hantera fel som uppstår, så att databasen kan hållas logiskt konsistent oavsett vad som händer.

I denna uppgift är det sätt att skydda processer från inverkan av samtidiga processer vi är intresserade av, eftersom det bara är då som begreppen `SERIALIZABLE` och `READ COMMITED` är relevanta.

Ett problem som kan uppstå är att flera processer läser samma data från databasen, gör parallella beräkningar, och sedan skriver sina resultat till databasen på ett sätt som gör att den sista processen skriver över de uppdateringar som de andra gjort.

Som ett enkelt exempel, låt oss anta att vi har en tabell `accounts`:

```
CREATE TABLE accounts (
  account_no   INTEGER,
  balance      DECIMAL(16,2),
  PRIMARY KEY  (account_no)
)
```

och att vi i två processer, p_1 och p_2 vill utföra uppdateringarna:

```
UPDATE  accounts
SET     balance = balance + <amount>
WHERE   account_no = 1010;
```

där vi på kontot 1010 från början har saldot (`balance`) 100, och p_1 vill ta ut 50, och p_2 vill ta ut 75.

Om dessa båda processer startar exakt samtidigt så kan de båda läsa saldot 100, och sedan vilja göra:

- Process 1:

  ```
  UPDATE  accounts
  SET     balance = 100 - 50
  WHERE   account_no = 1010;
  ```

- Process 2:

  ```
  UPDATE  accounts
  SET     balance = 100 - 75
  WHERE   account_no = 1010;
  ```

och vilket saldo som finns på kontot efteråt bestäms av vilken av de båda processerna som gjorde sin uppdatering sist.

Detta problem kräver att processerna körs *väldigt* samtidigt, men problemet är tillräckligt allvarligt för att man i en riktig bank måste skydda sig för det.

Och för banken i exemplet kan vi få liknande problem som uppstår med mycket större sannolikhet. Antag till exempel att vi i varje process först vill testa att det finns tillräckligt mycket pengar på ett konto för en transferering (så att vi slipper negativa saldon), och sedan gör uppdateringen – detta tar betydligt längre tid, så risken att det skall inträffa är mycket större.

Som exempel kan vi anta att vi har tre konton: A med 100 kr, B med 50 kr och C med 25 kr. Process 1 ska överföra 100 kr från A till B och process 2 ska överföra 50 kr från A till C.

- Process 1: Kontrollera att täckning finns för att överföra 100 kr från A, det finns 100 kr.
- Process 2: Kontrollera att täckning finns för att överföra 50 kr från A, det finns 100 kr.
- Process 1: Överför 100 kr till B, Subtrahera 50 kr från A, vilket gör att A nu har 0 kr kvar. B har 150 kr.
- Process 2: Överför 50 kr till C, Subtrahera 50 kr från A, vilket gör att A har nu saldot -50 kr, medan C har 75 kr.

Det faktum att processerna inte körs 'isolerat', dvs utan påverkan från varandra, gör att vi kan få negativa saldon – om vi hade kunnat garantera att vi antingen kör hela process 1 först, och sedan hela process 2, eller tvärtom, så hade vi sluppit dessa problem.

(b) `READ COMMITED` tillåter att två processer gör kontrollen av täckning samtidigt (att den bara läser värden som blivit 'commited' gör ingen skillnad i detta avseende). Värden kan ändras under en transaktion men de är inte synliga för andra förrän en `COMMIT` utförts. `SERIALIZABLE` ger en starkare garanti: värden ska inte kunna ändras efter man läst det. Vissa implementationer av databashanterare hanterar detta genom att ge fel när en samtidig uppdatering sker – det medför att samma värde kan läsas parallelt men att parallell uppdatering inte tillåts.

Lösning 5

I SQLite3 kan vi lösa det med:

```
CREATE TRIGGER repair_as_investment
AFTER INSERT ON repairs
BEGIN
  INSERT
  INTO    investments(amount, description, repair_id)
  VALUES  (NEW.cost, "repair", NEW.repair_id);
END;
```

I uppgiftstexten stod det:

> *"...det är OK om du inte får helt rätt på syntaxen för triggers."*

så vi accepterar mindre avvikelser från korrekt syntax utan poängavdrag, men man måste skriva något som ser ut ungefär som lösningen ovan (vi fick in ett antal väldigt innovativa lösningar, varav några var ganska eleganta, men för långt ifrån korrekt syntax för att ge mer än tröstpoäng).

Lösning 6

I uppgiftstexten stod det:

> *"Delfrågorna nedan har alla väldigt specifika svar,"*

så för att man skall få poäng på uppgiften måste man svara ungefär som nedan – svar som beskriver allmäna egenskaper hos funktionella beroenden och normalformer kan bara ge någon tröstpoäng. Det är ungefär som att på frågan "Hur dog Karl XII?" inte ge poäng för svaret "Hjärtat slutade slå".

(a) De saknar redundans som beror av funktionella beroenden.

(b) Metoden gör att vi kan återskapa vår ursprungliga relation utan att vi tappar eller lägger till någon information när vi joinar ihop våra mindre relationer.

(c) I den ena relationen lägger vi det funktionella beroende som bryter mot BCNF, så att vi kan plocka bort högerledet ur detta beroende från den andra relationen. Det innebär att vi i den andra relationen bara har beroendets vänsterled, och att vi kan hämta högerledet genom att joina med den första relationen (vi joinar då med vänsterledet i det funktionella beroendet som join-predikat). Eftersom vänsterledet i det funktionella beroendet inte är en superkey (för då hade vi varit i BCNF), så kan samma värden upprepas på flera ställen i vår ursprungliga relation, och då skulle högerledet i det funktionella beroendet ha gett redundans varje gång det inträffade.