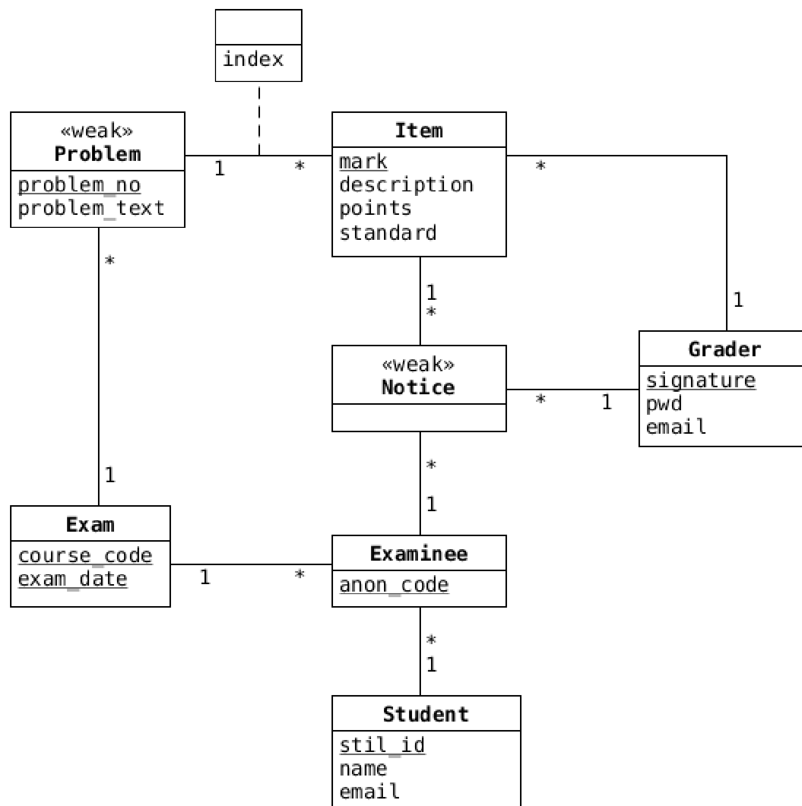
Lösningar till tentamen i EDA216

14 mars 2017

Lösning 1

(a) Som vanligt kan man rita sin E/R-modell på många olika sätt – i nedanstående lösning använder vi ett 'entity set' för alla 'items', och markerar 'standard items' med hjälp av ett boolskt attribut, man kan tänka sig att dela upp i två olika entity sets, exempelvis genom att använda arv.

Här är en lösning med bara ett entity set för 'items':



Ett naturligt alternativ är att rita `Notice` som en associationsklass mellan `Examinee` och `Item` (och med en association till `Grader`) – resultatet skulle bli detsamma nedan. Och det finns flera andra rimliga sätt att tolka uppgiftstexten, så du behöver inte vara orolig bara för att du inte kommit fram till just denna modell.

I figuren har vi utgått från att varje problem bara hörde till en tenta, ni får naturligtvis inget avdrag om ni istället gör det möjligt att låta en uppgift förekomma på flera tentor.

(b) Ur figuren kan vi direkt hämta följande relationer (primärnycklar understrukna, främmande nycklar kursiverade):

```
exams(course_code, exam_date)
problems(problem_no, problem_text)
```

```
items(mark, description, points, standard)
graders(signature, pwd, email)
examinees(anon_code)
students(stil_id, name, email)
notices()
```

Relationen `notices` ser lite lustig ut just nu, eftersom den inte har några attribut – den kommer att se mer naturlig ut efter att vi strax lagt in våra främmande nycklar.

Dessutom verkar `problems` få en jobbig nyckel (en naturlig nyckel skulle behöva innehålla både kurskod och datum för tentamen), så vi kan införa en egen nyckel[1]:

```
problems(problem_id, problem_no, problem_text)
```

Det är så dags att ta hand om våra 'många-till-1'-associationer, vi gör det genom att införa ett antal främmande nycklar i våra relationer:

```
exams(course_code, exam_date)
problems(problem_id, problem_no, problem_text, course_code, exam_date)
items(mark, description, points, standard, problem_id, grader_signature)
graders(signature, pwd, email)
examinees(anon_code, stil_id, course_code, exam_date)
students(stil_id, name, email)
notices(anon_code, mark, grader_signature)
```

Slutligen tar vi hand om våra övriga associationer och associationsklasser, i detta fall har vi bara en ordningen av våra items att ta hand om[2]:

```
items(mark, description, points, standard, problem_id, grader_signature, idx)
```

(c) Lösningen till just denna uppgift blir faktiskt lite enklare om vi inte inför någon 'invented key' (`problem_id`) för `problems`, eftersom då både kurskod och datum skulle vara foreign-keys i `items`.

Om vi gör som ovan, dvs använder `problem_id`, så måste vi join-a in `problems` för att få kurskod och datum för våra `item`-s. Vi kan i så fall börja med att skapa en vy som innehåller alla 'items' för denna tenta:

```sql
DROP VIEW IF EXISTS current_exam_items;
CREATE VIEW current_exam_items AS
SELECT *
FROM   items
JOIN   problems
USING  (problem_id)
WHERE  course_code = 'EDA216'
       AND exam_date = '2017-03-14';
```

För att räkna ut maxpoängen på uppgift '1b' kan vi nu summera alla standardpunkter för uppgiften:

```sql
SELECT SUM(points)
FROM   current_exam_items
WHERE  problem_no = '1b'
       AND standard;
```

Vi antar här att `standard` är ett logiskt attribut som anger om ett 'item' är ett 'standard-item' (dvs ett sådant item som bör ingå i en helt korrekt lösning).

[1] Ungefär samma argument kan användas för `exams` och `notices`, men vi låter det bero i denna lösning.

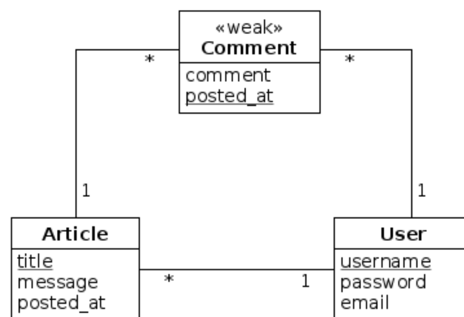
[2] Och vi kunde mycket väl ha lagt in ett index i `items` redan tidigare, utan att ha den i en associationsklass först.

Lösning 2

(a) Designen av relationerna kan verka lite märklig, `title` i `articles`, och `username` i `users` skulle kunna vara nycklar i sina respektive relationer, utan att vi blandar in `article_id` och `user_id` (de är båda 'invented keys').

I uppgift 1 använde vi en 'invented key' för att vi skulle slippa släpa runt en stor nyckel som foreign key i flera tabeller – här är våra 'invented keys' lika stora som våra naturliga nycklar, men man inför ibland invented keys ändå, om det finns en risk att ett värde som vi använder som nyckel skulle kunna ändras. Exempelvis kan man tänka sig att någon ändrar titeln på en artikel (fortfarande med kravet att den skall vara unik), och om vi då använt titeln som nyckel så måste vi uppdatera databasen på flera ställen (enklast med hjälp av en `ON UPDATE CASCADE`). Med invented keys här kan vi ändra både titlar och användarnamn enklare[3], en nackdel är att vi måste join-a mer (för uppgiftens del är det en fördel...).

Snyggast är egentligen att inte rita ut våra invented keys i E/R-diagrammet:



I detta fall hade det även varit OK att rita skriva ut `article_id` och `user_id` (observera dock att vi *inte* vill ha våra foreign keys i `comments` i diagrammet, de täcks redan av associationerna):

(b) I uppgiften stod från början att kommentarerna innehåll uppgift om vilket *datum* de var skrivna, jag har ändrat det till tidpunkt här (men du får naturligtvis inget avdrag för att använda datum...):

```
CREATE comments(
  article_id   INT,
  user_id      INT,
  comment      TEXT,
  posted_at    TIMESTAMP,
  PRIMARY KEY  (article_id, user_id, posted),
  FOREIGN KEY  (article_id) REFERENCES articles(article_id),
  FOREIGN KEY  (user_id) REFERENCES users(user_id)
);
```

(c)

```
SELECT    title
FROM      articles
ORDER BY  posted_at DESC
LIMIT     10;
```

(d)

```
SELECT  title
FROM    articles
JOIN    users
USING   (user_id)
WHERE   username = 'niklas';
```

[3] Vi kan exempelvis ändra exempelvis `title` i en artikel utan att behöva ändra i `comments`-relationen.

(e)

```sql
SELECT  COUNT()
FROM    comments
JOIN    articles
USING   (article_id)
WHERE   title = 'Det finns inget roligare än databaser!';
```

(f)

```sql
SELECT  username
FROM    users
WHERE   user_id NOT IN
        (SELECT  user_id
         FROM    articles)
        AND
        user_id NOT IN
        (SELECT  user_id
         FROM    comments);
```

eller, med en temporär vy:

```sql
WITH
  active_users AS (
    SELECT DISTINCT user_id
    FROM            articles
    UNION
    SELECT DISTINCT user_id
    FROM            comments
)
SELECT username
FROM   users
WHERE  NOT user_id IN (
   SELECT user_id
   FROM   active_users
);
```

(g)

```sql
SELECT    username, COUNT() as count
FROM      comments
JOIN      users
USING     (user_id)
GROUP BY  user_id
HAVING    count > 5;
```

Lösning 3

(a) A måste ingå i nyckeln eftersom A inte finns på högersidan i någon FD, så vi beräknar det transitiva höljet (*transitive closure*) av $\{A\}$:

- $\{A\}^+ = \{A\}$

$\{A\}$ är alltså inte en nyckel.

Vi går sedan vidare med alla tänkbara nycklar med två attribut, och eftersom A måste ingå, så är de enda kandidaterna:

- $\{A, B\}^+ = \{A, B\}$

- $\{A, C\}^+ = \{A, B, C, D, E\}$
- $\{A, D\}^+ = \{A, B, C, D, E\}$
- $\{A, E\}^+ = \{A, E\}$

Detta visar att både $\{A, C\}$ och $\{A, D\}$ är nycklar.

Därefter testar vi potentiella nycklar med tre attribut, och vi kan återigen konstatera att A måste ingå, men att varken C eller D kan ingå (vi skulle annars få en supermängd av $\{A, C\}$ eller $\{A, D\}$).

Den enda kandidat vi behöver testa är $\{A, B, E\}$:

- $\{A, B, E\}^+ = \{A, B, E\}$

Nycklarna för R är alltså: $\{A, C\}$ och $\{A, D\}$.

(b)
- FD_1 bryter ej mot BCNF, eftersom $\{A, D\}$ är en supernyckel.
- FD_2 bryter ej mot BCNF, eftersom $\{A, C\}$ är en supernyckel.
- FD_3 bryter mot både BCNF och 3NF, eftersom vänsterledet, $\{C\}$, inte är en supernyckel och högerledet, $\{E\}$, inte är medlem i någon nyckel.

(c) Vi väljer att bryta upp relationerna utgående från FD_3, eftersom den bryter mot BCNF – vi skapar därför relationerna:

- $R_1(C, E)$, och
- $R_2(A, B, C, D)$.

R_1 måste vara i BCNF, eftersom den bara har två attribut. R_2 har två funktionella beroenden, FD_1 och FD_2[4], och eftersom både $\{A, C\}$ och $\{A, D\}$ är nycklar i R_2, så är vi i BCNF.

Lösning 4

Normalisering används för att undvika redundans, och göra det enklare att sätta in och ta bort värden med bibehållen data-integritet.

Normalisering innebär att man bryter upp relationer med många attribut till fler relationer som vardera har färre attribut – detta innebär att man efteråt måste 'joina' ihop flera tabeller för att hämta data som tidigare fanns i en tabell, och det kan ibland ta tid.

Så om vi väldigt ofta använder värden i en icke normaliserad tabell, och tabellen sällan ändras, kan det ibland av effektivitetsskäl vara värt att låta den vara onormaliserad.

Lösning 5

(a) Traditionell SQL (utan rekursiva queries) har inte `if`-satser, och kan inte loopa ett variabelt antal steg, exempelvis för att hitta roten i det träd en given nod tillhör. Om vi har

```
people(id, name, birth_year, mom_id, dad_id)
```

så kan vi till exempel inte hitta vår den första av våra 'mor-mor-mor...-mödrar' som finns i registret.

(b) En lagrad procedur är kod som körs på databas-servern, skrivet i någon variant av SQL/PSM. Det finns flera poänger med att använda en lagrad procedur, exempelvis att vi faktiskt kan loopa och använda `if`-satser, och att vi kan minska vår nätverkstrafik (istället för att skicka delresultat fram och tillbaka mellan klient och server kan vi nu göra allting på servern).

Lösning 6

Isolation level anger graden av isolation som en transaktion upplever – med repeatable read är vi garanterade att varje gång vi anropar en given `SELECT`-sats få samma resultat, även om den underliggande tabellens värden ändras av någon annan (vi kan dock få *phantom reads*, dvs rader som *lagts till* i tabellen kommer att dyka upp i efterföljande läsningar).

[4] R_2 påverkas vare sig av FD_3, eller något transitivt beroende från FD_3.