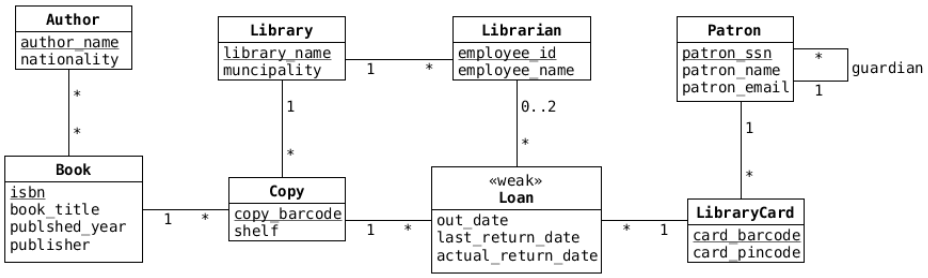
Lösningar till tentamen i EDA216

20 april 2017

Lösning 1

(a) En möjlig ER-modell (det finns alternativa lösningar):



(b) Relationer (primärnycklar understrukna, främmande nycklar kursiverade):

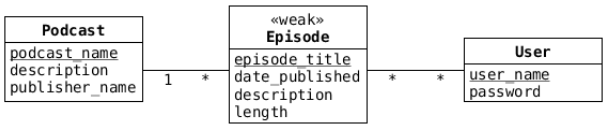
authors(author_name, nationality)
books(isbn, book_title, published_year, publisher)
authored_books(*author_name*, *isbn*)
copies(copy_barcode, *isbn*, *library_name*, shelf)
libraries(library_name, muncipality)
librarians(employee_id, employee_name, *library_name*)
loans(*copy_barcode*, *card_barcode*, out_date, *lending_librarian_id*,
last_return_date, actual_return_date, *receiving_librarian_id*)
patrons(patron_ssn, patron_name, patron_email, *guardian_ssn*)
library_cards(card_barcode, *patron_ssn*, card_pincode)

(c) SQL-kod:

```
SELECT title
FROM   loans
JOIN   copies
USING  (copy_barcode)
JOIN   books
USING  (isbn)
JOIN   library_cards
USING  (card_barcode)
JOIN   patrons
USING  (patron_ssn)
WHERE  name LIKE 'Niklas%' AND out_date = '2017-04-20';
```

Lösning 2

(a) E/R-diagram:



(b)

```sql
DROP TABLE IF EXISTS episodes;
CREATE TABLE episodes (
  episode_title        TEXT,
  podcast_name         TEXT,
  date_published       DATE,
  episode_description  TEXT,
  episode_length       INTEGER,
  PRIMARY KEY (podcast_name, episode_title),
  FOREIGN KEY (podcast_name) REFERENCES podcasts(name)
);
```

(c)

```sql
SELECT    episode_title
FROM      episodes
WHERE     episode_length > 60
ORDER BY  episode_length DESC;
```

(d)

```sql
SELECT  episode_title
FROM    episodes
JOIN    podcasts
USING   (podcast_name)
WHERE   date_published = '2017-04-20'
        AND publisher_name = 'BBC';
```

(e) Vi kan antingen använda en 'left outer join':

```sql
SELECT     podcast_name
FROM       podcasts
LEFT JOIN  episodes
USING      (podcast_name)
WHERE      date_published IS NULL;
```

eller en subquery:

```sql
SELECT  podcast_name
FROM    podcasts
WHERE   podcast_name NOT IN
        (SELECT podcast_name
         FROM   episodes);
```

(f) Om vi antar att varje avsnitt har åtminstone en lyssnare så räcker:

```sql
SELECT    episode_title, COUNT(user_name) AS cnt
FROM      listenings
WHERE     podcast_name = 'The Inquiry'
GROUP BY  episode_title
ORDER BY  cnt DESC;
```

Annars måste vi behandla varje avsnitt, och standardmetoden skulle i så fall vara:

```
SELECT     episode_title, COALESCE(COUNT(user_name), 0) AS c
FROM       episodes
LEFT JOIN  listenings
USING      (podcast_name, episode_title)
WHERE      podcast_name = 'The Inquiry'
GROUP BY   episode_title
ORDER BY   c DESC;
```

(g) Detta kan uttryckas på ganska många sätt, för att bryta ner uppgiften i mindre delar kan vi börja med att definera en vy för hur mycket varje lyssnare faktiskt har lyssnat:

```
DROP VIEW IF EXISTS listened_time;
CREATE VIEW listened_time AS
SELECT    user_name, SUM(episode_length) AS total_time
FROM      listenings
JOIN      episodes
USING     (podcast_name, episode_title)
GROUP BY  user_name;
```

Därefter kan vi ta med även 'lyssnare' som inte har lyssnat:

```
SELECT     users.user_name, COALESCE(total_time, 0) AS total_time
FROM       users
LEFT JOIN  listened_time
GROUP BY   users.user_name
ORDER BY   users.user_name;
```

Lösning 3

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

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

$\{D\}$ är alltså inte en nyckel.

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

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

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

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

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

(b)

- FD_1 bryter mot BCNF, eftersom vänsterledet, $\{A\}$, inte är en supernyckel. Däremot är högerledet, $\{B\}$, medlem i en nyckel och bryter därför ej mot 3NF.
- FD_2 bryter ej mot BCNF, eftersom $\{A, D\}$ är en supernyckel.
- FD_3 bryter mot BCNF, eftersom vänsterledet, $\{B\}$, inte är en supernyckel. Däremot är högerledet, $\{A\}$, medlem i en nyckel och bryter därför ej mot 3NF.

- FD_4 bryter ej mot BCNF, eftersom $\{B, D\}$ är en supernyckel.

Således är relationen ej i BCNF, men däremot i 3NF.

(c) Vi väljer att bryta upp relationerna utgående från FD_1, eftersom den bryter mot BCNF. Vi skapar därför relationerna:

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

R_1 måste vara i BCNF, eftersom den bara har två attribut. R_2 har enbart ett funktionellt beroende, FD_2, och nyckeln för relationen blir då $\{A, D\}$ och relationen är i BCNF.

Lösning 4

(a) Ett *index* är en datastruktur, typiskt någon form av sökträd, som gör sökning efter ett eller flera attribut effektivare (så länge dessa attribut inte ändrar värde(n)).

Fördelarna med ett index är att sökningar och joins kan gå snabbare, nackdelarna är att insättning, borttagning och ändring av värdet kräven att vi uppdaterar vår datastruktur, och det kan ta tid.

(b) Sökning på nycklar är vanliga, dessutom är primärnycklar unika, så det är lätt att hantera sökningar på dem (resultatet av en sökning är antingen ett värde, eller inget värde).

Lösning 5

(a) En *trigger* är kod som exekveras i vår DBMS (servern) när någon specificerad händelse inträffar, de används framförallt för att se till att våra data uppfyller givna villkor (dataintegritet). Händelser som kan aktivera triggers är typiskt insättningar, uppdateringar, och borttagningar av data och scheman.

Lösning 6

(a) SQL-injicering ett sätt för någon utomstående att stoppa in (’injicera’) kod i våra SQL-satser, för att få vår DBMS att göra något som vi inte hade tänkt att den skulle göra.

Ett enkelt exempel är satserna:

```
String userId = getRequestString("user_id");
String sql = "SELECT * FROM users WHERE user_id = " + userId;
Statement s = conn.createStatement();
ResultSet rs = s.executeQuery(sql);
```

där `getRequestString()` hämtar en sträng som användaren skickar in.

Om denna sträng är exempelvis

```
1 OR 1 = 1
```

så kommer vår databas att exekvera SQL-satsen

```
SELECT * FROM users WHERE user_id = 1 OR 1 = 1
```

och den kommer att returnera samtliga rader i tabellen `users`, inte bara den med rätt `user_id`.

Vi kan skydda oss mot SQL-injicering genom att noga undersöka de strängar som vi använder som SQL-frågor, eller genom att använda någon form av bibliotek som gör det åt oss – i JDBC kan vi använda `PreparedStatement`-objekt:

```
String userId = getRequestString("user_id");
String sql = "SELECT * FROM users WHERE user_id = " + userId;
PreparedStatement ps = conn.prepareStatement(sql);
ResultSet rs = ps.executeQuery();
```

Här skulle `conn.prepareStatement(sql)`-anropet upptäcka om strängen `sql` var ’farlig’.