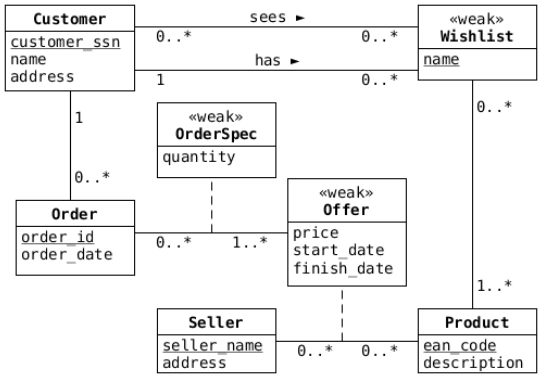
Lösningar till tentamen i EDAF75

13 mars 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) Här är `Offer`, `OrderSpec` och `Wishlist` 'weak', vi kan införa några 'surrogate keys' för att slippa släpa runt stora nycklar (nedan får `offers` och `wishlists` surrogate keys):

```
products(ean_code, description)
sellers(seller_name, address)
offers(offer_id, seller_name, ean_code, price, start_date, finish_date)
orders(order_id, customer_ssn, order_date)
order_specs(order_id, offer_id, quantity)
customers(customer_ssn, customer_name, address)
wishlists(wishlist_id, name, customer_ssn)
shared_wishlists(wishlist_id, sharee_ssn)
wishlist_products(wishlist_id, ean_code)
```

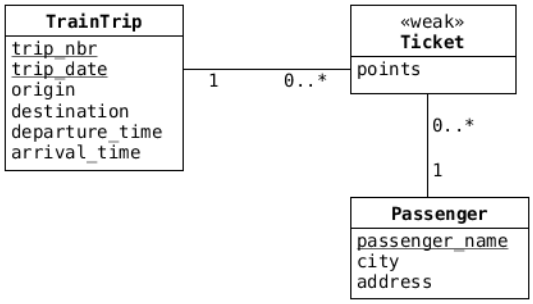
(c) Om vi låter `sellers` ha den naturliga nyckeln `seller_name` (som ovan), så blir `seller_name` en foreign key i `offers`, vilket gör att det räcker med:

```
SELECT price, description
FROM   products
JOIN   offers
USING  (ean_code)
WHERE  seller_name = 'Acme Explosives'
       AND '2018-03-13' BETWEEN start_date AND finish_date;
```

Om vi inför en "surrogate key" i `sellers`, så måste vi göra ytterligare en `JOIN` (på `sellers`) för att få `seller_name`.

Lösning 2

(a) E/R-diagram:



(b)

```sql
CREATE TABLE tickets(
  trip_nbr        INTEGER,
  trip_date       DATE,
  points          INT,
  passenger_name  TEXT,
  PRIMARY KEY(trip_nbr, trip_date, passenger_name),
  FOREIGN KEY(trip_nbr, trip_date) REFERENCES train_trips(trip_nbr, trip_date),
  FOREIGN KEY(passenger_name) REFERENCES passengers(passenger_name)
);
```

(c)

```sql
SELECT    trip_nbr
FROM      train_trips
WHERE     trip_date = '2018-03-13'
ORDER BY  departure_time;
```

(d)

```sql
SELECT  trip_nbr, departure_time, origin, passenger_name
FROM    train_trips
JOIN    tickets
USING   (trip_nbr, trip_date)
WHERE   trip_date = '2018-03-13';
```

(e) Vi kan lösa denna antingen med en subquery:

```sql
SELECT  passenger_name
FROM    passengers
WHERE   city = 'Lund'
        AND passenger_name NOT IN
          (SELECT passenger_name
           FROM tickets);
```

eller med en outer join:

```sql
SELECT     passenger_name
FROM       passengers
LEFT JOIN  tickets
USING      (passenger_name)
WHERE      city = 'Lund'
           AND trip_nbr IS NULL;
```

(f)

```
SELECT    trip_date, COUNT()
FROM      train_trips
GROUP BY  trip_date
ORDER BY  trip_date;
```

(g)

```
SELECT     passenger_name, COALESCE(SUM(points), 0) AS total_points
FROM       passengers
LEFT JOIN  tickets
USING      (passenger_name)
GROUP BY   passenger_name
ORDER BY   total_points DESC, passenger_name;
```

Lösning 3

(a) B finns inte i något högerled, så den måste ingå i en nyckel. Vi testar alla möjliga två-attributsnycklar:

$$\{AB\}^+ = \{ABCDEF\}$$
$$\{BC\}^+ = \{BC\}$$
$$\{BD\}^+ = \{BDAEFC\}$$
$$\{BE\}^+ = \{BEF\}$$
$$\{BF\}^+ = \{BF\}$$

Så, $\{AB\}$ och $\{BD\}$ är nycklar. De enda möjliga nycklarna med tre eller fyra attribut är:

$$\{BCE\}^+ = \{BCEF\}$$
$$\{BCF\}^+ = \{BCF\}$$
$$\{BEF\}^+ = \{BEF\}$$
$$\{BCEF\}^+ = \{BCEF\}$$

De enda nycklarna är alltså $\{AB\}$ och $\{BD\}$.

(b) FD_2, FD_3 och FD_4 bryter alla mot BNCF, eftersom deras vänsterled inte är supernycklar. Högerleden i FD_3 och FD_4 är inte del av några nycklar, så relationen är heller inte i 3NF.

(c) Vi har tre trilskande funktionella beroenden ($\text{FD}_2 \ldots \text{FD}_4$) att välja bland när vi skall bryta upp R, och vi kan göra på flera olika sätt. Efter den första uppdelningen kommer vi att få två relationer, varav minst en kommer att behöva delas upp ytterligare en gång (oavsett vilket funktionellt beroende du väljer att dela upp på i första steget) – så det finns många sätt att följa den algoritm som beskrevs under föreläsning 6 och 7 (vilket innebär att uppgiften kommer att bli väldigt jobbig att rätta...). Ett sätt att göra uppdelningen är att bygga upp FD_2 till $A \to DEF$ (genom att använda FD_3 och FD_4 transitivt), dvs att börja i:

Relation:	$R(A, B, C, D, E, F)$
Beroenden:	$AB \to C, A \to DEF, D \to AE, E \to F$
Nycklar:	$\{AB\}, \{BD\}$

och sedan med hjälp av $A \to DEF$ (vars vänsterled inte är en superkey, och därför bryter mot BCNF) bryta ner R till:

Relation:	$R_1(A, D, E, F)$
Beroenden:	$A \to DEF, D \to AE, E \to F$
Nycklar:	$\{A\}, \{D\}$

Eftersom vänsterledet i $E \to F$ inte är en superkey, så är R_1 inte i BCNF.

Relation:	$R_2(A, B, C)$
Beroenden:	$AB \rightarrow C$
Nycklar:	$\{AB\}$

  Här har $AB \rightarrow C$ en superkey som vänsterled, så R_2 är i BCNF.

Vi kan bryta ner R_1 med hjälp av $E \rightarrow F$ (vars vänsterled inte är en superkey för R_1), och får då:

Relation:	$R_{1a}(E, F)$
Beroenden:	$E \rightarrow F$
Nycklar:	$\{E\}$

  Denna är uppenbarningen i BCNF (liksom alla relationer med två attribut)

Relation:	$R_{1b}(A, D, E)$
Beroenden:	$A \rightarrow DE, D \rightarrow AE$
Nycklar:	$\{A\}, \{D\}$

  Vänsterleden i båda våra funktionella beroenden är superkeys, så R_{1b} är i BCNF.

Detta ger oss uppdelningen $R_{1a}(E, F)$, $R_{1b}(A, D, E)$ och $R_2(A, B, C)$, men det finns alltså många andra sätt att lösa uppgiften.

Lösning 4

(a) En *trigger* är kod som exekveras i vår databas (servern) när någon specificerad händelse inträffar. Händelser som kan aktivera triggers är typiskt insättningar, uppdateringar, och borttagningar av data och scheman. Triggers kan användas för att se till att vår databas uppfyller givna integritetsvillkor, exempelvis

- att säkerställa att värden i en enstaka kolumn uppfyller vissa villkor, eller
- att låta kolumner i flera tabeller uppdateras 'tillsammans', som i en databas för en bank, där vi kan uppdatera en tabell med saldon när nya rader sätts in i en tabell med transfereringar.

Lösning 5

Vi behandlade detta under föreläsning fyra, och har även stött på det i samband med att vi löst extentor på tavlan (vid genomgångarna har vi varje gång valt det första alternativet nedan).

- Alternativ 1: Skapa en tabell för alla gemensamma attribut (dvs de som finns i superklassen), och sedan en tabell för varje subklass, med nyckeln i superklassen som foreign key:

  ```
  X(a, b, c)
  Y(a, d, e)
  Z(a, f)
  ```

- Skapa en separat tabell för varje subklass, lägg alla attribut i superklassen i var och en av subklasserna:

  ```
  X(a, b, c)
  Y(a, b, c, d, e)
  Z(a, b, c, f)
  ```

- Skapa en enda stor tabell, med samtliga möjliga attribut:

  ```
  X(a, b, c, d, e, f)
  ```

  Denna lösning kräver att vi ändrar tabellens struktur varje gång en ny subklass skall hanteras (så lösningen bryter mot *Open-Closed-Principle*), och kommer att innebära att vi förmodligen får ganska många `NULL`-element i vår tabell.

Lösning 6

En möjlighet är att först gruppera efter värden, och se vilka som förekommer i lika många examplar som antalet grupper:

```sql
WITH
  unique_set_ids AS (
    SELECT DISTINCT set_id
    FROM            sets
  ),
  element_counts AS (
    SELECT   element, COUNT() AS cnt
    FROM     sets
    GROUP BY element
  )
SELECT element
FROM   element_counts
WHERE  cnt = (
  SELECT COUNT()
  FROM   unique_set_ids
);
```

Vi kan även trixa lite med `DISTINCT` inne i en `COUNT`:

```sql
SELECT    element
FROM      sets
GROUP BY  element
HAVING COUNT() = (
  SELECT COUNT(DISTINCT set_id)
  FROM   sets
);
```

Det finns många olika sätt att lösa uppgiften, vi är nöjda om du löser den på något sätt överhuvud taget.