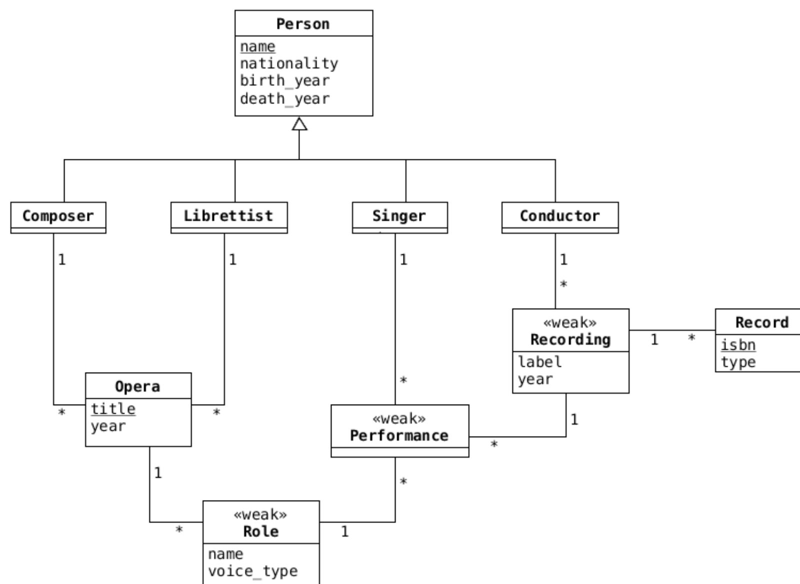
Lösningar till tentamen i EDA216

16 augusti 2017

Lösning 1

(a) En möjlig ER-modell, man kan få poäng även för andra lösningar:



(b) Förslag till relationer baserat på modellen (primärnycklar understrukna, främmande nycklar kursiverade):

people(name, nationality, birth_year, death_year)
composers(*name*)
librettists(*name*)
singers(*name*)
conductors(*name*)
operas(title, *composer_name*, *librettist_name*, year)
roles(*opera_title*, role_name, voice_type)
performances(*singer_name*, *opera_title*, *role_name*, *recording_id*)
recordings(id, label, year)
records(isbn, *recording_id*, media_type)

Man kan tänka sig andra sätt att implementera subklasserna till `Person`, på detta sätt är det enkelt att låta en person få flera roller utan att vi behöver duplicera information (exempelvis är Placido Domingo både sångare och dirigent).

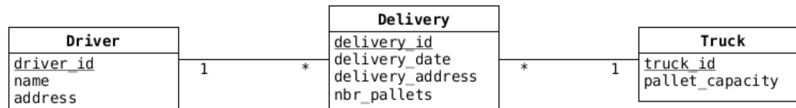
(c) SQL-kod:

```
SELECT  roles.opera_title, roles.role_name, recordings.year, media_type
FROM    records
JOIN    recordings
ON      records.recording_id = recordings.id
JOIN    performances
```

```
ON      performances.recording_id = recordings.id
JOIN    roles
ON      roles.opera_title = performances.opera_title AND
        roles.role_name = performances.role_name
JOIN    operas
ON      operas.title = roles.opera_title
WHERE   performances.singer_name = 'Angela Gheorghiu' AND
        operas.composer_name = 'Giacomo Puccini';
```

Lösning 2

(a) E/R-diagram:



(b)

```
DROP TABLE IF EXISTS deliveries;
CREATE TABLE deliveries (
  delivery_id       INT,
  truck_id          INT,
  driver_id         INT,
  delivery_date     DATE,
  delivery_address  TEXT,
  nbr_pallets       INT CHECK (nbr_pallets > 0),
  PRIMARY KEY (delivery_id),
  FOREIGN KEY (truck_id) REFERENCES trucks(truck_id),
  FOREIGN KEY (driver_id) REFERENCES drivers(driver_id)
);
```

(c)

```
-- Skriv ut namn och adress för alla förare, i alfabetisk ordning
-- efter namn.
SELECT    name, address
FROM      drivers
ORDER BY  name;
```

(d)

```
-- Skriv ut medelvärdet av antalet pallar som lastbilarna kan lasta.
SELECT  AVG(pallet_capacity)
FROM    trucks;
```

(e)

```
-- Skriv ut alla uppgifter om de leveranser där lastbilens hela
-- kapacitet inte utnyttjades.
SELECT  *
FROM    deliveries
JOIN    trucks
USING   (truck_id)
WHERE   nbr_pallets < pallet_capacity;
```

(f)

```sql
-- Skriv ut namnet på den förare som har gjort flest leveranser
-- (namnen, om de är flera).
DROP VIEW IF EXISTS delivery_counts;
CREATE VIEW delivery_counts AS
SELECT  driver_id, COUNT(*) AS c
FROM    deliveries
GROUP BY  driver_id,
ORDER BY  c DESC;

SELECT  name
FROM    delivery_counts
JOIN    drivers
USING   (driver_id)
WHERE   c = (SELECT MAX(c)
             FROM   delivery_counts);
```

(g)

```sql
-- Skriv ut namnen på de förare som inte har gjort någon leverans.
SELECT     name
FROM       drivers
LEFT JOIN  deliveries
USING      (driver_id)
WHERE      delivery_date IS NULL;
```

alternativt:

```sql
SELECT  name
FROM    drivers
WHERE   driver_id NOT IN
        (SELECT driver_id
         FROM deliveries);
```

Lösning 3

(a) Vi har följande funktionella beroenden:

FD1. passport_no → name

FD2. flight_no → departure arrival aircraft_type

FD3. aircraft_type → capacity

(b) Vi hittar inte passport_no, flight_no eller flight_date i högerledet i något funktionellt beroende, så de måste alla ingå i en nyckel. Och om vi beräknar det transitiva höljet av dessa tre attribut, så får vi samtliga attribut, så de utgör en nyckel. Eftersom vi dessa attribut både måste ingå i en nyckel, och räcker för att få en nyckel, så är det den enda tänkbara nyckeln.

(c) Relationen är inte i BCNF eftersom vänsterledet i FD1 inte är en super-key, detta gäller även vänsterleden i FD2 och FD3 (så samtliga funktionella beroenden bryter mot BCNF).

(d) Relationen är inte heller i 3NF, samtliga funktionella beroenden bryter mot 3NF eftersom deras högerled inte är en del av en superkey.

(e) Vi kan börja med att bryta upp med avseende på FD1, och får då:

bookings(passport_no, flight_no, departure, arrival, flight_date, aircraft_type, capacity)
names(passport_no, name)

Vi kan fortsätta med FD2, och får:

bookings(passport_no, flight_no, flight_date, capacity)
names(passport_no, name)
flights(flight_no, departure, arrival, , aircraft_type)

Slutligen bryter vi även upp med FD3, och får:

```
bookings(passport_no, flight_no, flight_date)
names(passport_no, name)
flights(flight_no, departure, arrival, aircraft_type)
capacities(aircraft_type, capacity)
```

Dessa relationer är alla i BCNF.

Lösning 4

(a)
```
-- (i): define table

DROP TABLE IF EXISTS accounts;
CREATE TABLE accounts (
  acc_no    INT,
  balance   DECIMAL(12,2) DEFAULT 0,
  PRIMARY KEY (acc_no)
);

-- (ii): create account

INSERT
INTO   accounts(acc_no)
VALUE  (1234);

-- (iii): deposits and withdrawals

UPDATE accounts
SET    balance = balance + 10
WHERE  acc_no = 1234;

UPDATE accounts
SET    balance = balance + 30
WHERE  acc_no = 1234;

UPDATE accounts
SET    balance = balance - 20
WHERE  acc_no = 1234;

-- (iv): show current balance

SELECT balance
FROM   accounts
WHERE  acc_no = 1234;
```

(b) I detta fall behöver vi egentligen bara en tabell, för att hantera transaktionerna (`transactions` nedan), i denna lösning finns även en kontoklass (`accounts`, som vi kan använda för att hantera information om kontohavaren), men den är alltså inte helt nödvändig för uppgiftens skull.

```
-- (i): define tables

DROP TABLE IF EXISTS accounts;
CREATE TABLE accounts (
  acc_no    INT,
  PRIMARY KEY (acc_no)
);

DROP TABLE IF EXISTS transactions;
CREATE TABLE transactions (
```

```
  acc_no    INT,
  amount    DECIMAL(12,2),
  FOREIGN KEY (acc_no) REFERENCES accounts(acc_no)
);

-- (ii): create account

INSERT
INTO   accounts(acc_no)
VALUE  (1234);

-- (iii): deposits and withdrawals

INSERT
INTO   transactions(acc_no, amount)
VALUES (1234, 10),
       (1234, 30),
       (1234, -20);

-- (iv): show current balance

SELECT SUM(amount)
FROM   transactions
WHERE  acc_no = 1234;
```

(c) Den första metoden kräver mycket lite arbete när skall räkna ut saldot, men vi kommer inte att kunna se varför saldot är som det är. Omvänt kräver den andra metoden en del arbete vid uträkning av saldot, men, men vi får full kontroll på historiken hos varje konto.

Med den första metoden kan vi med hjälp av en trigger enkelt kontrollera att saldot aldrig blir negativt – med den andra metoden behöver vi aldrig ändra något värde i någon av tabellerna, vilket gör systemet mindre känsligt för samtidiga uppdateringar.

Den andra metoden blir mer och mer populär – tekniken kallas *event sourcing*, och används idag i många olika sammanhang, exempelvis:

- Det är en vanlig teknik att hantera tillstånd i program, bibliotek som `redux` sparar 'actions' som beskriver tillståndsändringar, och beräknar aktuellt tillstånd genom att 'reducera' ihop dessa actions (man kan även se det som en tillämpning av *Command Pattern*).
- Det ligger till grund för idén med *blockchains*, som är fundamentet för cryptocurrencies som *Bitcoin* och *Ethereum* (där pengar lagras som summan av alla transaktioner).

En poäng med att använda event sourcing är att vi aldrig ändrar något värde, vilket gör det mycket enklare att resonera kring våra program, och gör det mycket enklare att parallellisera vårt arbete.

Lösning 5

En *natural key* är en nyckel som förekommer naturligt i problemdomänen, exempelvis namnet på en opera i uppgift 1. En *invented key* är ett artificiellt nyckel-värde som man genererar – för operadatabasen skulle vi kunna koppla ett vanligt heltal till varje roll, så att vi slipper hålla reda på både operanamn och rollnamn.

Vi använder typiskt naturliga nycklar när vår modell innehåller unika enkla värden som inte ändras, och inför 'invented keys' när det antingen finns risk att nyckeln kommer att ändras, eller att det krävs flera attribut som nyckel.

För/nackdelar med att använda en invented key:

+ Den kan vara enkel, exempelvis bara ett heltal, istället för att vi (som för rollerna i operadatabasen) har två eller flera attribut som måste kombineras.
+ Den behöver aldrig ändras, även om det underliggande värdet ändras.

- Det gör att vi ofta måste joina flera tabeller.

För/nackdelar med att använda en naturlig nyckel:

+ Det är lättare att läsa och förstå våra SQL-queries.
+ Vi slipper ett antal joins.
- Vi riskerar att behöva ändra främmande nycklar i många tabeller om det underliggande värdet ändras.

Våra personnummer är en lite lustig blandning mellan naturlig nyckel och påhittad nyckel – de är ursprungligen en påhittad nyckel[1] men används idag i så många sammanhang att man börjar ta dem för naturliga (alla känner igen ett personnummer, och alla vet vilket personnummer de har). De är faktiskt ingen särskilt bra nyckel – de är visserligen unika, men de kan ändras (exempelvis när man får skyddad identitet, eller när man som invandrad byter från samordningsnummer till sitt ordinarie personnummer), och det finns en risk att man måste uppdatera på många ställen om vi använder personnummer som främmande nycklar i olika tabeller.

Lösning 6

Att skala upp innebär att vi skaffar en kraftfullare dator, på motsvarande sätt kan vi skala ner genom att byta till en mindre kraftfull dator.

Att skala ut innebär istället att vi använder flera datorer och fördelar arbetet mellan dem.

Horisontell skalning är samma sak som att skala ut/in, vertikal skalning är samma sak som att skala upp/ner.

Sharding är ett exempel på horisontell skalning, där vi distribuerar rader med data över flera olika servers (alla rader för en given tabell behöver inte ens ligga på samma kontinent).

[1] De första sex siffrorna är visserligen 'naturliga', men de sista fyra är en ren artifakt).