EDAF75
Database Technology

Lecture 4

Christian.Soderberg@cs.lth.se

January 30, 2020

Two remaining queries from Tuesday

- Solve the *'average-highschool-size'* problem with a subquery instead of a view
- Solve the *'exactly-two-chemistry-laureates-when-summer-olympics-in-Europe'* problem without a subquery (but use the CTE's we wrote)

Database design

- To design a database, we first create a model which captures:
  - objects (entity sets),
  - the properties of the objects (attributes)
  - associations (relationships) between our objects – we're meticulous when it comes to multiplicities
- We then translate the model into a set of *relations* – these relations will become tables in our database
- We write relations as:

  cars(<u>license_plate</u>, *brand_id*, model)

  where primary keys are underlined, and foreign keys are italicized
- We'll sometimes use the terms 'relation' and 'table' interchangeably (there is a 1 – 1 correspondence, but they're not really the same thing)

Exercise

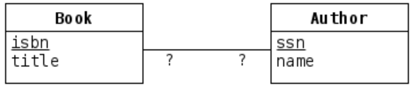
At LTH, courses are given every year, and each year a number of students register to take courses. Describe this in the simplest possible ER-model.

Translation from model to relations

- Classes (entity sets) will become relations – we use *snake_case* and plural form for the relations
- Attributes of our classes will become attributes of our relations
- Associations are handled according to multiplicity

Translation from model to relations – example

We have two entity sets, `Book` and `Author`, and an association between them:



Create relations for them if:

- each book is written by one author
- a book can be written by several authors

Translation from model to relations – example

Define relations based on the model of students taking courses at LTH which we just developed.

Translation of associations

- $*-1$ associations are translated into *foreign keys* on the $*$ side
- $*-*$ associations are translated into relations with foreign keys referencing both sides
- $*-0..1$ associations are a bit special:
  - if it's mostly 1 on the right side, we can use the first method above, and use `NULL` where we have no associated object
  - otherwise we can use the second method above (a new relation with two foreign keys)
- Other multiplicities require some handiwork

Translating 0..1 – 0..1 associations – example

Exercise: *We* want to keep track of people and dogs, and assume a person can only own one dog, and that a dog can be owned by at most one person.
What relations do we use if:

- Almost all dogs have an owner
- Almost every person have a dog
- Only some people own dogs, and many dogs are without an owner

Translating 0..1 – 0..1 associations

- If almost all dogs have owners:

```
people(ssn, ...)
dogs(id, ..., owner_ssn)
```

- If almost everyone own a dog:

```
people(ssn, ..., dog_id)
dogs(id, ...)
```

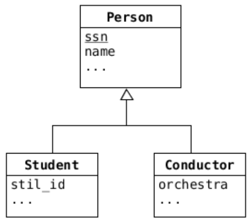
- If only some people own dogs, and many dogs are without an owner:

```
people(ssn, ...)
dogs(id, ...)
dog_ownerships(owner_ssn, dog_id)
```

Translation of association classes

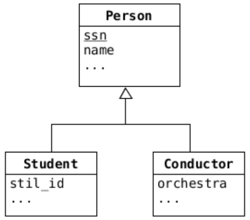
- For $* - *$-relations:
  - the $* - *$-relation itself will give us a new relation
  - the attributes of the association class will be attributes of this relation
- For $* - 1$-relations:
  - the attributes of the association class will end up together with the foreign key on the $*$-side

Translating inheritance into relations



- Create one relation for each class (entity set), and reference from subclasses to superclasses using foreign keys
- Create relations only for concrete classes (entity sets)
- Create one big relation, with all possible attributes (with a lot of NULL values)

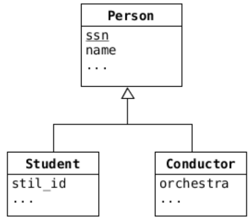
Translating inheritance into relations



- Create one relation for each class (entity set), and reference from subclasses to superclasses using foreign keys:

```
people(ssn, name, ...)
students(ssn, stil_id, ...)
conductors(ssn, orchestra, ...)
```

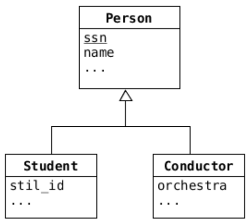
Translating inheritance into relations



- Create relations only for concrete classes (entity sets):

```
students(ssn, name, stil_id, ...)
conductors(ssn, name, orchestra, ...)
```

Translating inheritance into relations



- Create one big relation, with all possible attributes (with a lot of NULL values)

```
people(ssn, name, stil_id, orchestra, ...)
```

Defining tables in SQL

- To create a table, we use the CREATE TABLE command (and an optional DROP TABLE first):

```
DROP TABLE IF EXISTS books;

CREATE TABLE books (
  isbn     TEXT,
  title    TEXT,
  year     INT,
  ...
);
```

- For each row we define a type (see next slide)
- For each row we can add *constraints*

Types in our tables

- integers: `INT`, `INTEGER`, ...
- real numbers: `REAL`, `DECIMAL(w,d)`, ...
- strings: `TEXT`, `CHAR(n)`, `VARCHAR(n)`
- dates: `DATE`, `TIME`, `TIMESTAMP`, ...
- blobs (binary largs objects): `BLOB` (only in some databases)

Primary keys in table definitions

- We can declare an attribute to be primary key 'in place':

```
CREATE TABLE books (
  isbn     TEXT PRIMARY KEY,
  title    TEXT,
  year     INT,
  ...
);
```

- We can also declare it separately (especially useful when the key contains several attributes):

```
CREATE TABLE books (
  isbn     TEXT,
  title    TEXT,
  year     INT,
  ...
  PRIMARY KEY (isbn);
);
```

Foreign keys in table definitions

- We can declare foreign keys 'in place':

```
CREATE TABLE books (
  isbn     TEXT PRIMARY KEY,
  title    TEXT,
  author   TEXT REFERENCES authors(author),
  ...
);
```

- We can also declare it separately:

```
CREATE TABLE books (
  isbn     TEXT,
  title    TEXT,
  author   TEXT,
  ...
  FOREIGN KEY (author) REFERENCES authors(author);
);
```

Exercise

Create tables for the "students and courses at LTH" database which we designed earlier

Exercise, from Tuesday

Solve the first part of problem 1 on the exam from april 2017

Exercise

Create relations for the model we just developed

Exercise

Create tables for the database we just designed

Some other constraints

- We can declare a column to be:
  - `NOT NULL`
  - `UNIQUE`
  - `DEFAULT <value>`
  - `CHECK <condition>`
- These properties are enforced by the database, but the enforcement can often be temporarily turned off (it does take time to check everything all the time).
- We can also define *triggers* to enforce constraints, we'll return to this later in the course

Inserting values

- We can insert values using `INSERT`:

```
INSERT
INTO    students(s_id, s_name, gpa, size_hs)
VALUES (123, 'Amy', 3.9, 1000),
       (234, 'Bob', 3.6, 1500),
       ...
```

- We don't have to provide values for columns with default values
- We also don't have to provide values for primary keys which are declared as `INTEGER` – they will get a new unique integral value (hence the moniker *database sequence number*)

Updating values

- We can update values using `UPDATE`:

```
UPDATE students
SET    gpa = min(1.1 * gpa, 4.0)
WHERE  s_name LIKE 'A%';
```

- All rows are updated if we don't provide a `WHERE` clause

Deleting values

- We can delete values using `DELETE`:

```
DELETE
FROM   applications
WHERE  s_id = 123
```

- Beware that the innocent looking:

```
DELETE
FROM   applications
```

  empties the whole table

Variants

- There are various variants of the `INSERT` and `UPDATE` commands, such as:
  - `INSERT OR REPLACE`
  - `INSERT OR IGNORE`
  - `INSERT OR FAIL`
  - `INSERT OR ROLLBACK`
  - `UPDATE OR REPLACE`
  - `UPDATE OR IGNORE`
  - `UPDATE OR FAIL`
  - `UPDATE OR ROLLBACK`
- They are useful when an insertion or update would break some constraint

Keys, superkeys, and invented keys

- A *key* is a set of attributes which uniquely identifies each row in a table
- In our college application database, `s_id` is unique in the `students` table, but `s_name` isn't
- Of course, the tuple `(s_id, s_name)` is also unique, but here `s_name` is redundant, so we call `(s_id, s_name)` a *superkey* (which, despite its name, is way less fancy than a regular key)
- The `s_id` is an artificial value which we associate with each student, and we call it an *invented key* (or *surrogate key*)

Natural keys, and compound keys

- In the `colleges` table, `c_name` is unique in our small example, so we can use it as a key
- It is a value which occurs naturally in our problem domain, so we call it a *natural key*
- If we had a database with more colleges, we would soon end up with universities with the same name, but in different states – in that case we could use `(c_name, state)` as a key, it would be a *compound key*

Foreign keys

- In the `applications` table, we have two attributes, `s_id` and `c_name` which refers to keys in other tables, they are called *foreign keys*
- The key for the `applications` table is `(s_id, c_name, major)`

More about invented keys

- If a key ever changes, we may have to update in many places in our database – so we normally avoid natural keys which might change, and use an invented key instead
- Having a compound key can be a bit unwieldy, it is sometimes better to replace it with a simple invented key
- On the other hand: having an invented key requires us to do more joins

Different kinds of invented keys

- We can use an increasing sequence of integers as invented key – it is very common, but it's predictible, and reveals something about the state of the database
- A *uuid* (universally unique identifier) is a ≈ 128-bit random number, and it's a good choice for an invented key – we can safely assume it will be unique, and it doesn't reveal anything about the state of our system
- In `SQLite3` we can use `lower(hex(randomblob(16)))` to create something akin to a uuid (but it takes up more memory)

Keys, technically

- A key is a (minimal) set of attributes which uniquely identifies each row in a table
- For some relations we don't care about keys (e.g., a log of events)
- For some relations we may have several possible (*candidate*) keys – we then choose one of them as primary key
- For some relations there is no key using their own attributes, we call the corresponding entity set *weak*
- For a weak entity set, we can often use keys in associated entity sets (called *supporting entity sets*) to uniquely define a row, or just use an invented key

Generating invented keys

- In `SQLite3` we can get a uuid-lookalike using:

```
CREATE TABLE students (
  s_id     TEXT DEFAULT (lower(hex(randomblob(16)))),
  s_name   TEXT,
  gpa      DECIMAL(3,1),
  size_hs  INT,
  PRIMARY KEY (s_id);
);
```

- We'll soon see how the `DEFAULT` clause is used
- The database doesn't have to check if the generated value is unique, since the chance of a collision is ridiculously low
- The most recent version of `SQLite3` (Sqlite 3.31) has a `uuid()`-function