EDAF75
Database Technology

Lecture 4

`Christian.Soderberg@cs.lth.se`

January 29, 2026

Administration

- You must sign up your lab groups (2-3 students per group) *no later than 23:59 on Friday*
- On Monday I'll open a webpage where you can sign up for 10-minute lab sessions next week, *you must have a group before you sign up for the sessions*

A note about relations and tables

- Relational databases are based on *relational algebra* (a discipline of mathematics)
- In relational algebra we use the term *relation* to describe what we in a database call a *table*
- You'll often see the terms 'relation' and 'table' used interchangeably – for the purpose of this course, they are the same

Implementing entity sets

Q: How do we handle the data in the entity sets of our model (`Book`, `Author`, ...)?

A: We define a table for each entity set, with all its attributes

- We use the `CREATE TABLE` statement to create the table
- We mark our primary key with `PRIMARY KEY`

ToDo Define tables for the 'obvious' entity sets in the library example, and their 'obvious' columns

Implementing simple associations

Q: How do we implement a `*-1` association?

A: In the table on the `*` side we put attributes which uniquely points out the value on the 1 side, we call them *foreign keys* (and they typically constitute keys in the other table)

- We mark our foreign keys `FOREIGN KEY`, to make sure we have no ’loose ends’ in our database

ToDo Implement the `*-1` associations of the library example

Implementing `*-*` associations

Q: How do we implement a `*-*` association?

A: We add a new table, often called a *join table*, with foreign keys to the tables on both sides

- If we have an association class tied to our association, its attributes end up in the join table

ToDo Implement the `*-*` associations of the library example

Implementing ER models - special cases

- Some cases are not clear cut
  - `1-1` associations
  - `0..1-0..1` associations
  - `*-0..1` associations, where the `0..1` side is often 0

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 tables 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, but only few people have dogs:

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

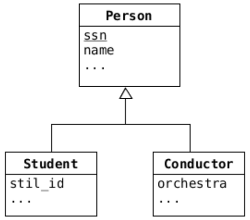
- If almost everyone own a dog:

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

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

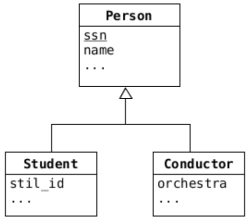
```
people(ssn, ...)
dogs(id, ...)
dog_ownerships(owner_ssn, dog_id)
```

Translating inheritance into tables



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

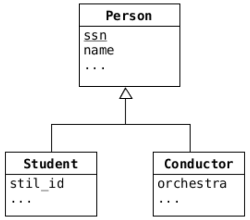
Translating inheritance into tables



- Create one table for each entity set, and reference from subclasses to superclasses using foreign keys:

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

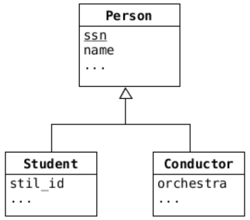
Translating inheritance into tables



- Create table only for concrete entity sets:

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

Translating inheritance into tables



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

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

Some constraints we can put in table definitions

- 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*)
- We can also use a SELECT statement to generate values to insert, and use WITH statements

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

Generating invented keys

- In SQLite3 we can get a somewhat 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)
);
```

- The database doesn't have to check if the generated value is unique, since the chance of a collision is ridiculously low