EDAF75
Database Technology

Lecture 3

Christian.Soderberg@cs.lth.se

January 22, 2024

Administration

- Course representatives?
- From tomorrow you can register your lab group on the course website:
  - *make sure you enter your whole group at once*, and
  - ***don't register only yourself!*** (you will be removed)
- This week we'll discuss *database modeling*, and then see how you can translate a model into a database
- Lab 1 is next week, it's an exercise in SQL queries
- Lab 2 is the week after next, and it lets you practice modeling
- If you want a QA-session tomorrow, please sign up on Moodle – it's totally fine to ask question about the labs during the QA sessions

Short recapitulation

- A *relational database* is a collection of one or more *tables*, where each table has a fixed set of *columns*, and a varying number of rows – *all cells contain primitive values*
- Simple queries (`SELECT-FROM-WHERE`)
- Set operations (`UNION`, `INTERSECT`, `EXCEPT`)
- Simple functions and aggregate functions
- Grouping (`GROUP BY-HAVING`)
- Window functions (`OVER`)
- Subqueries, views and CTEs (`WITH` statements)
- Joins (`CROSS-`, `INNER-`, `OUTER-`, …)

Modeling

- To design a database, we'll start out with what's called an Entity/Relation Model (E/R Model)
- There are many 'standards' for drawing E/R diagrams, we'll use UML class diagrams – it's becoming increasingly popular for database modeling

Elements of an E/R Model

- *Entity Sets*: these are the 'objects' of our model, they correspond to classes in a traditional object oriented model
- *Attributes*: properties of our objects – must be primitive values (see the next slide)
- *Relationships*: associations between our entity sets (with cardinality)
- We will typically convert entity sets to tables (*relations*), and attributes to columns in our tables – relationships will be dealt with according to their cardinality

'Primitive' values in our models

- integers: `INT`, `INTEGER`, ...
- real numbers: `REAL`, `DECIMAL(w,d)`, ...
- strings: `TEXT`, `CHAR(n)`, `VARCHAR(n)`
- boolean values: `true`, `false`
- dates: `DATE`, `TIME`, `TIMESTAMP`, ...
- blobs (binary largs objects): `BLOB` (only in some databases)
- JSON objects (not really primitive..., only in some databases)

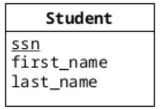
UML class diagrams

- We'll use UML classes in approixmately the same way as you may have seen them used in earlier courses, with some caveats:
  - There will be no methods in our classes
  - All our attributes will be primitive and public
  - We won't bother much with aggregates and compositions, we'll use plain associations instead
  - We'll be very careful to mark cardinalities everywhere
  - We will think carefully about what constitutes a *key* for each entity set

Class diagrams for ER modeling

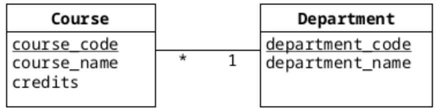
- We'll use:
  - classes (entity sets)
  - associations with cardinality (relationships)
  - association classes
  - inheritance (sometimes)
- We have simple rules of how to translate each kind of element into our tables
- There isn't much theory behind our ER-models, creating them is mostly an art to learn

UML class diagrams – classes



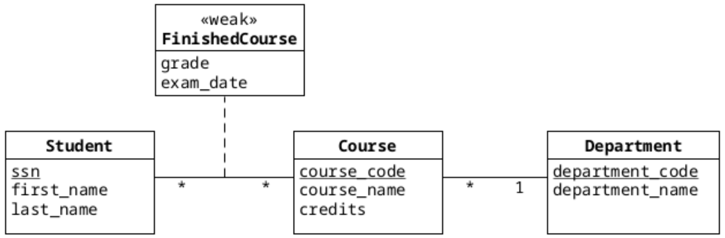
- The entity/class name in singular
- Only one box (since we have no methods)
- We will underline keys

UML class diagrams – associations



- We always mark cardinality on our associations
- We use associations instead of attributes whenever the value is a reference to another object

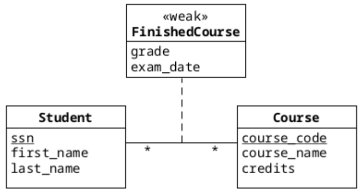
UML class diagrams – association classes



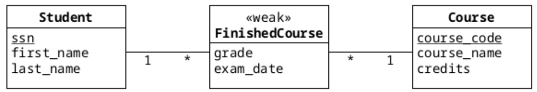
- Sometimes the association between two entity sets contains data itself
- We use an *association class* to capture that data

UML class diagrams – association classes

Normally we can use either an association class between two entity sets:



or another entity set ’between’ them



Example

Exercise: *Create a model for students applying for college*

Keys, candidate keys and 'super'-keys

- A *superkey* is a set of attributes for which all rows in a table are guaranteed to be unique
- A *key*, or *candidate key*, is a *minimal* set of attributes which uniquely identifies each row in a table – by minimal we mean that no attribute in the set is superfluous (it does *not* mean there can't be another key with fewer attributes)
- A table can have several candidate keys – when we model our database we pick one of them and call it our *primary key*
- If we add attributes to a key, the row will still be unique, which means that the key plus extra attributes is a superkey – so although being a superkey may sound impressive, it's actually less impressive than being a (candidate) key
- A key with more than one attribute is called a *composite key*

Example

Exercise: *What would be keys in a table of our children's classmates?*

Keys and foreign keys

- When a row in one table needs to refer to a specific row in another table (i.e., we have a $* - 1$ association), it keeps a key to the other table (as one or more columns in this table) – this key is called a *foreign key*
- The database will ensure that there are no duplicate primary keys in a table, and it will create an *index* to speed up searches for it (more about that later), so using a primary key in another table as a foreign key makes sense
- We can also ask the database to make sure there always is a corresponding row for a foreign key (we'll return to that in a few weeks time)

Natural keys and invented keys

- Sometimes keys occur naturally in the problem domain, we call such keys *natural keys* or *business keys*
- Entity sets which can't be uniquely identified by its attributes alone is sometimes called a *weak entity sets* (they need to use foreign keys to create a primary key) – for the sake of this course, calling them ”weak” is not a big deal (it doesn't effect our designs *at all*)
- Sometimes we invent keys by introducing some artificial attribute, these keys are called *invented keys*, *surrogate keys*, *synthetic keys*, …

Natural keys and invented keys

- Whether to use an invented key or not is often a question of simplicity vs effeciency:
  - Without an invented key we sometimes get an unwieldy key (either because it contains many attributes, or because a single attribute might me easily mistyped)
  - With an invented key our tables and queries only need a single key column, *but finding the key may require additional joins*
- If an attribute might change over time, it's not a good choice for a key – it would require us to update all tables which uses the old value

Weak entity sets and compound keys

- Association classes are typically 'weak', but using its foreign keys we can get a key – this is sometimes called a *compound key* (it is also a *composite key*)
- We sometimes add an invented key to a ”weak” entity set – technically it's still a weak entity set (since the invented key isn't really a proper attribute)

Example

Exercise: *Solve the library example from the preparation web page.*

Updating or accumulating?

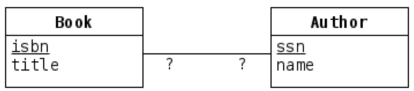
- How would you keep track of the balance of a bank account?
- Two ideas:
  - updating a balance attribute
  - saving all transactions, and then calculate the balance each time
- Saving the transactions allows us to track and explain the current state – it's called *Event Sourcing*, and is becoming increasingly popular
- When we update a single attribute, we need to make sure no one else updates it at the same time
- Adding a new transaction requires less locking

Translating E/R models to databases

- For each entity set (class) we create a relation (table) with the same attributes (columns) as the entity set
- For relationships (associations), what we do depend on their cardinality
- For inheritance, there are several alternative implementations, we'll look at it next time

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 of associations

- '1 – 1'-associations typically means that the two entity sets can be merged into one entity set
- $*$ – 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 cardinalities 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, ...)
```

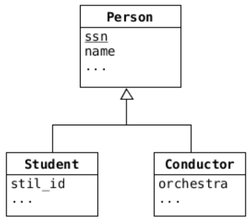
- If only some people own dogs, and many dogs are without an owner:

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

Translation of association classes

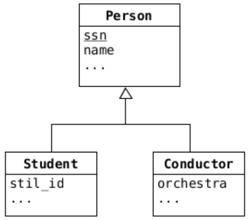
- For $*$ – $*$-relations:
  - the $*$ – $*$-relation itself will give us a new relation
  - the attributes of the association class will be attributes of this relation
- For $*$ – 1-relations:
  - the attributes of the association class will end up together with the foreign key on the $*$-side

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)

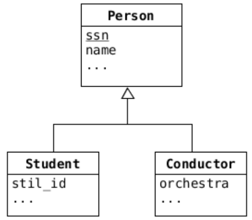
Translating inheritance into relations



- 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, ...)
```

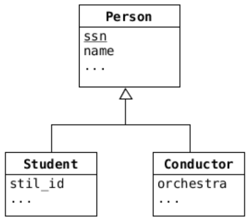
Translating inheritance into relations



- 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), ...
- booleans: not in SQLite, instead we use INTs, where 0 = false, and 1 = true
- strings: TEXT, CHAR(n), VARCHAR(n)
- dates: DATE, TIME, TIMESTAMP, ...
- blobs (binary largs objects): BLOB (only in some databases)
- JSON objects (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)
);
```

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

Variants

- The documentation of SQLite3 gives lots of variants of the `INSERT`, `UPDATE`, and `DELETE` commands
- We can use `WITH` statements to create useful tables during inserts, updates and deletions
- We can use various forms of subqueries, etc.

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)
);
```

- The database doesn't have to check if the generated value is unique, since the chance of a collision is ridiculously low
- PostgreSQL has `uuid` as a type, SQLite3 has some library support for it, but for now it's easier to use `randomblobs`
- When we insert a row with generated attributes, we can have the `INSERT` statement return the values (using `INSERT` - `RETURNING`)