EDAF75
Database Technology

Lecture 3

Christian.Soderberg@cs.lth.se

January 27, 2025

Administration

- From Wednesday, 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`)
- Simple functions and aggregate functions
- Grouping (`GROUP BY` – `HAVING`)
- Window functions (`OVER`)
- Subqueries, views and CTEs (`WITH` statements)
- Joins (`INNER`, `CROSS`, `OUTER`)
- Set operations (`UNION`, `INTERSECT`, `EXCEPT`)

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)

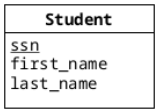
UML class diagrams

- 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 (essentially, they're some combination of attributes which will make each entity unique)

Class diagrams for ER modeling

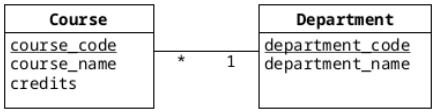
- We'll use:
  - classes (entity sets)
  - associations with cardinality (relationships)
  - association classes
  - inheritance (sometimes)
- We have simple rules of how to translate each kind of element into our tables
- There isn't much theory behind our ER-models, creating them is mostly an art to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

UML class diagrams – classes



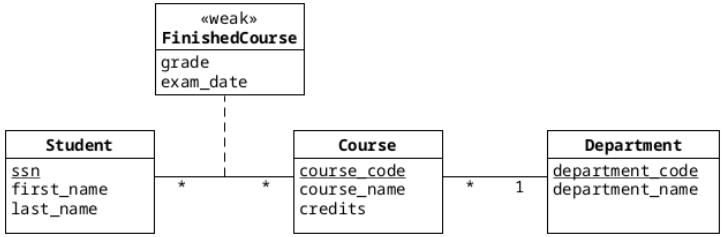
- The entity/class name in singular
- Only one box (since we have no methods)
- We will underline keys

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

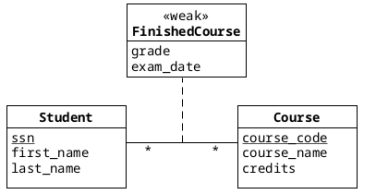
UML class diagrams – association classes



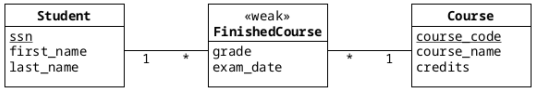
- Sometimes the association between two entity sets contains data itself
- We use an *association class* to capture that data

UML class diagrams – association classes

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



or another entity set 'between' them

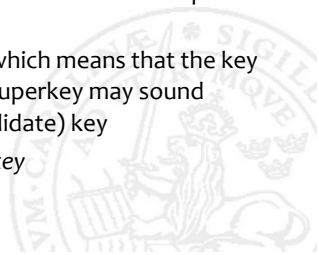


Example

Exercise: *Create a model for the students applying for college we saw last week*

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*

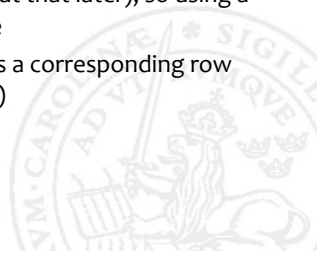


Example

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

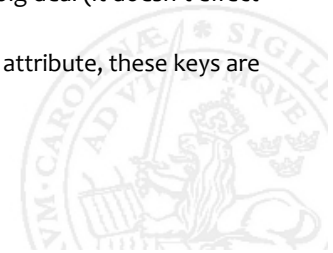
Keys and foreign keys

- 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)



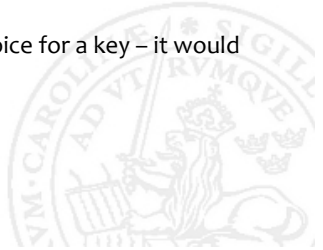
Natural keys and invented keys

- 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