06-34140/34139 *Data Structures, Algorithms & Databases* The University of Birmingham
Spring Semester 2021-22 School of Computer Science

Design Theory of Relational Databases

8 Functional dependencies

51. Where we are. Over the last few weeks we have looked at *conceptual design* via entity relationship diagrams, and last lecture we discussed how such a diagram can be translated into actual tables. If the conceptual model was sufficiently accurate and adequate, then the resulting tables will constitute a very good starting point for the *logical design*. It is unlikely, however, that the tables are already in the optimal shape for representing the real-world data. Indeed, it is very common that certain **redundancies** remain; such redundancies will lead to inefficiencies and are sources of potential errors.

As part of the logical design phase, therefore, we should have a close look at each table that results from ER modelling. The single most effective technique for doing so is to check the tables for potential **functional dependencies**. Once these have been established, the tables can be optimized in a fairly mechanical procedure, called **normalisation**. This will be the topic of the next section. The current section will do nothing more than present the necessary concepts and notation for normalisation.

52. Notation. We use English letter as symbols standing for various notions that arise in relational tables:

- T stands for tables.
- A, B, C, D stand for column names (or attribute names).
- X, Y, Z are variables ranging over column names. (The difference from A, B, C, D is that we allow variables to take any column name as a value. So, it is possible for X and Y to be equal, i.e., have the same column name as their value.)
- U, V, W stand for collections of column names.
- x, y, z and other lower case letters are variables ranging over records or parts of records.

53. Candidate keys. A candidate key in a table is a collection of attributes that has the property that any particular combination of values for its attributes will be unique in the table. Primary keys are necessarily candidate keys. There could be other candidate keys in a table too. For example, consider the schema for the "`course`" table:

```
course(cid, bc, name, level)
```

Since `bc` (short for "banner code") is the University-assigned id number for courses, we can be sure that every value of `bc` will be unique in the table. So, `bc` is another candidate key. Similarly, if we are sure that every course has a distinct name, then `name` will be a candidate as well. (However, be careful. Even though it is true that every course has a distinct name at any given time, when we compile historical data, we might have a past version of a course and a current version of the course having the same name, even though they should be considered distinct in the database.)

54. Functional dependencies. Let $S = (A, B, C, D, \ldots)$ be a **relational schema**, that is, the list of attributes for some table T. Let further U be a subcollection of attributes and X be a single attribute chosen from S. We say that the functional dependency $U \to X$ holds in the table T if the following is true for *all* pairs of tuples t_1, t_2 occurring in T:

> If t_1 and t_2 have the same value for every attribute mentioned in U, then they also have the same value for the attribute X.

In this case we say that the attribute X "*functionally depends*" on the attribute set U. We might also say that U "*determines*" X.

The presence of a functional dependency can be very visible in a table. For example, if we list the contents of "`lecturing`", ordered in a certain way, then we get the following output:

```
  cid   | sid | year | numbers            cid   | sid | year | numbers
-------+-----+------+---------          -------+-----+------+---------
  6995  |  22 | 2002 |       87           8762  |  38 | 2002 |      251
  6995  |  41 | 2002 |       87           8762  |  81 | 2002 |      251
  6995  |  57 | 2002 |       87           8762  |  85 | 2002 |      251
  6995  |  76 | 2002 |       87
                                         11580  |  14 | 2002 |      182
  6995  |  22 | 2003 |       80          11580  |  83 | 2002 |      182
  6995  |  89 | 2003 |       80          11580  |  86 | 2002 |      182
  6995  |  57 | 2003 |       80          (...)
  6995  |  76 | 2003 |       80
```

We observe that the number of students on a course depends on "`cid`" and "`year`" but not on "`sid`". In the formalism above we have the functional dependency `cid, year` $\rightarrow$ `numbers`. Make sure that you understand what it means: the number of students does *not* depend on the member of staff teaching it. It obviously depends on the course. It also depends on the year because student numbers change from year to year. However, it "functionally depends" on the combination of `cid` and `year`. That means that, given any particular pair of values of `cid` and `year`, the value of `numbers` is uniquely determined. Neither "`cid`" alone nor "`year`" alone functionally determines `numbers`.

55. Functional dependencies are constraints. In the example above, we observed that a particular "`lecturing`" table satisfies a functional dependency. Is it merely an accident? Or, do we expect that any "`lecturing`" table, i.e., any collection of rows that will ever make up the "`lecturing`" table, will *necessarily* satisfy the dependency?

Thinking about the *real-world meaning* of the "`lecturing`" table, we can see that the latter is the case. Any collection of rows that will ever make up the "`lecturing`" table should satisfy the dependency. Why? A given course in a given year will always have a fixed number of students. The number does not depend on whether one staff member taught the course or multiple staff members did. It does not vary from one staff member to another.

This means that we can regard the functional dependency as a *constraint* that the "`lecturing`" table should satisfy. If the "`lecturing`" table is ever modified in such a way that the constraint is not satisfied then the integrity of the table is violated. So, we should set up the database in such a way that the constraint will always be satisfied. Functional dependencies can be expressed as enterprise constraints, but it is a bit tricky.

Pause. Try it!

But luckily, they can be enforced more easily by *decomposing* tables into smaller ones. We will see how this works in the next section.

It is worth remembering that, as with all constraints, when we impose a functional dependency then we restrict the shape of the data that can be stored in the table. The restriction is likely to increase the efficiency and integrity but it might also make it impossible to adapt the database when the real-world situation changes.

In the example above, what could happen is that two lecturers share the teaching of a module but for some reason some students take only one half. In this case the numbers would depend on "`sid`" and the original functional dependency would no longer hold.

56. Candidate keys give functional dependencies. Whenever we declare a primary key for a table, we are declaring that every attribute of the table is functionally dependent on the primary key. For example, consider the "`staff`" table with the schema

```
    staff(sid, firstname, lastname, office, phone)
```

The fact that "`sid`" is the primary key means that we have the functional dependencies "`sid`" $\rightarrow$ "`firstname`", "`sid`" $\rightarrow$ "`lastname`", "`sid`" $\rightarrow$ "`office`" and "`sid`" $\rightarrow$ "`phone`". This is a bit of a funny situation. The functional dependencies mean that *if* two rows were to have the same "`sid`" attribute then they will have the same "`firstname`", the same "`lastname`" and so on. But no two distinct rows can have the same "`sid`" by virtue of the fact that it is the primary key. So the functional dependency is automatically satisfied.

What we said above for primary keys now holds more generally for all candidate keys. Every attribute in the table will be functionally dependent on every candidate key.

57. Playing with functional dependencies. Once some functional dependencies have been postulated, others can be inferred to hold as well. Here are some rules for this:

Trivial dependencies If the attribute X is actually a member of the collection U, then the functional dependency $U \to X$ is definitely and boringly true, as you can see from the definition.

Augmentation If the dependency $U \to X$ has already been postulated, and if V is a bigger collection of attributes than U, then $V \to X$ must also be true.

Transitivity If we already know that $U \to X$ holds for every X that is mentioned in the collection V, and if we further know that $V \to Y$ is true, then $U \to Y$ must also hold. Again, this follows easily from the definition.

These three rules are known as **Armstrong's Axioms**, and it can be shown that no further rule is needed to derive all functional dependencies that hold as a consequence of postulating some. One says that Armstrong's Axioms are *sound and complete*.

Conversely, if we are charged with stating all functional dependencies which should hold for a given relational schema, then it is sufficient to list a **basis** for them, that is, a collection of dependencies from which all others follow by Armstrong's Axioms.

Notation issues: Note that when we write a functional dependency $U \to X$, U is a collection of attributes whereas X is a single attribute, e.g., `cid,year` $\to$ `numbers`. So, the notation is *asymmetric*. We can extend the notation so that we have collections of attributes on both the sides of $\to$. We define $U \to V$ to mean that $U \to X$ holds for *every* X in V. Using this notation, we can see the transitivity rule in the more conventional form: if $U \to V$ and $V \to Y$ then $U \to Y$.

58. A closure operation for functional dependencies. Although Armstrong's Axioms are a complete method for finding additional dependencies, the derivation process itself is computationally expensive: in the worst case it runs in exponential time. For the following special case there is a simpler method.

Suppose we are given a family $\mathcal{F}$ of functional dependencies, and we ask whether a *particular* functional dependency $U \to X$ is a logical consequence of the given ones. To answer this question, we successively augment the set of attributes U with further attributes which functionally depend on (a subset of) U. So for each functional dependency $(V \to Y) \in \mathcal{F}$ we check whether V is a subset of U. If this is the case, then the attribute Y can be added to U, and the rule $V \to Y$ can be deleted from $\mathcal{F}$ (because it has made its contribution and will not be needed again). The process is repeated until no further rule remains in $\mathcal{F}$ whose left hand side is a subset of the current U. The new U contains all attributes which depend functionally on the original U. If X is among them, then $U \to X$ is true, otherwise it is false.

Here is an example. Suppose we are given $\mathcal{F} = \{AB \to C, CB \to D, A \to B\}$ and ask whether $A \to D$ follows from this. The completion process proceeds as follows:

	U	$\mathcal{F}$
start state	$\{A\}$	$\{AB \to C, CB \to D, A \to B\}$
	$\{A, B\}$	$\{AB \to C, CB \to D\}$
	$\{A, B, C\}$	$\{CB \to D\}$
final state	$\{A, B, C, D\}$	$\{\}$

We find that all four attributes depend on A, and in particular, $A \to D$ is a logical consequence of the given dependencies.

59. Exercises

a. Consider the following schema which describes M.O.T. inspections of motor vehicles.

```
inspection(date_of_inspection,
           owner,
           owner_address,
           owner_contact_phone,
           registration_number,
           model,
           year_of_first_registration,
           diesel_or_petrol,
           date_of_previous_MOT,
           engineer,
           garage,
           garage_address,
           garage_MOT_license_number,
           passed_or_not)
```

   i. Determine the candidate key(s).
   ii. Find plausible (and non-trivial) functional dependencies. In doing so, list your assumptions and discuss whether they are reasonable.

b. For each of the following assumed sets of functional dependencies determine whether they imply $A \to D$. If yes, justify your answer with a derivation based on Armstrong's Axioms; if no, give a table which satisfies the assumed dependencies but not $A \to D$.

   i. $AB \to C, C \to B, B \to D$
   ii. $AB \to C, CB \to D, A \to B$
   iii. $ABC \to D, AB \to C, A \to B$

c. List a basis from which all functional dependencies can be derived which are compatible with the following table:

a	b	c	d
s_1	t_1	u_1	v_1
s_1	t_2	u_2	v_1
s_2	t_2	u_3	v_1
s_3	t_3	u_4	v_2
s_3	t_3	u_5	v_3

9 Normalisation

60. The problem with unnormalised tables. Suppose that our design so far has produced a schema with attributes $A, B, C, D, \ldots$ and that we know from the intended interpretation of these attributes in the real world that the functional dependency $B \to C$ will always hold. This means that whenever two tuples are entered into this table, which agree in their B-value, then they must also agree in their C-value. Now, there are two possibilities: either we know that it will just never happen that two tuples agree in their B value, or otherwise we know that this could very well happen. The first case is true if B is in fact a candidate key for this table. It is only in the second case that we will have problems:

a. In case one of the dependent attributes has a numeric type, then the repetition can cause wrong results in aggregation queries. This is the case, for example, with the "`numbers`" field in the "`lecturing`" table. For instance, if we ask for the total number of course registrations in a particular year, we cannot simply sum up all the "`numbers`" attributes of the rows for that year. Some courses will get counted again and again. We call this phenomenon an "aggregation anomaly".

b. Whenever a new tuple is entered into the database, we should check whether there is already another tuple present which has the same B-value. It must then be checked that the corresponding C-values do indeed agree. Otherwise we may create an "insertion anomaly".

c. Whenever we need to update the C information in the table, we must make sure that we consistently replace all occurrences of that particular value. Otherwise we may create an "update anomaly".

d. When we delete the last tuple from the table which contains a specific B-value, then the corresponding C-value is lost from the table, too. This is called a "deletion anomaly".

For a concrete example, you may wish to think of the table as describing items held in a shop, and the dependency $B \to C$ as saying that the supplier name B always determines the supplier's contact phone number C. The insertion anomaly could arise when you enter a new tuple together with a new phone number for the supplier. All other items which are usually ordered from this supplier will then still hold the old phone number. The update anomaly is similar; when you want to change the phone number for a certain supplier then you have to make sure that you update all occurrences of it. The delete anomaly arises when you delete the last item which was usually ordered from a certain supplier; in this case the information about their contact phone number is lost.

61. Normalisation. The way to address the problem with non-trivial functional dependencies is to *split the table*. In general, if we have a table over the schema $S = (A, B, C, D, \ldots)$ and we have discovered the functional dependency $U \to X$, then we will first find all other attributes not in U which also depend on U with the closure mechanism described in para 58. Let's call this set V (which certainly contains X). We then split the table into two, one of which having all the attributes from $U \cup V$, and the other one having all attributes not in V.

In the example above, we would create a separate table which contains the supplier name and all details which depend only on the supplier. In another table, we include the supplier name and all other attributes from the original table that are not fully dependent on the supplier name.

The process of normalisation can now be continued with the new tables. There may still be other functional dependencies in the new tables that have nothing to do with the original dependency $U \to X$. The process ends when all remaining dependencies are induced by candidate keys (which, as we said above, can not lead to any anomalies). The relations are then said to be in **Boyce-Codd Normal Form**.

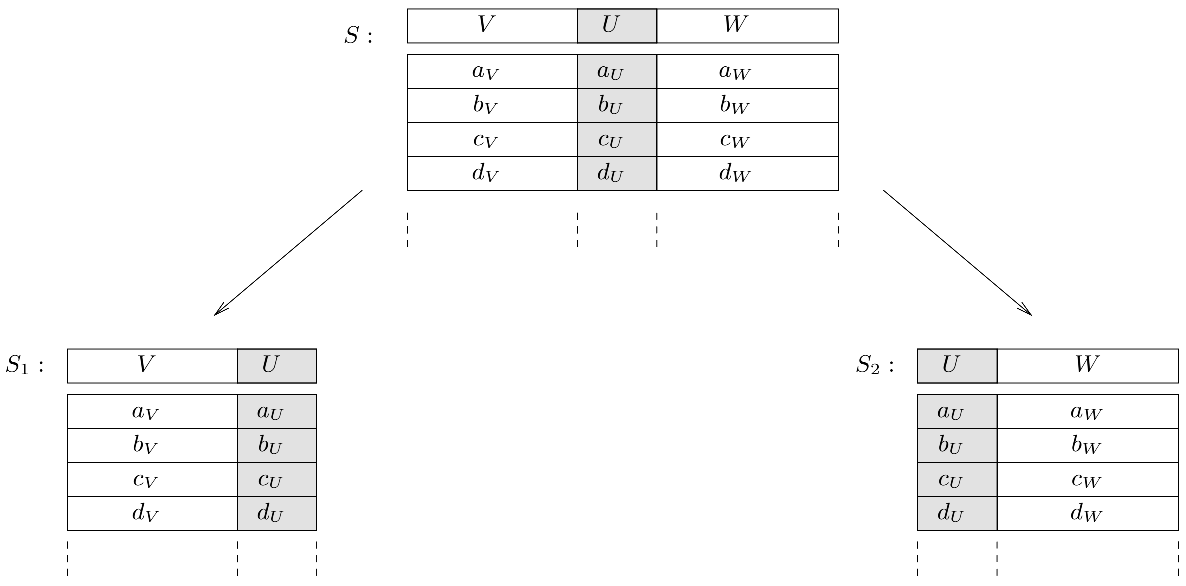
The example suggests that splitting the table is exactly the right way to go, and that the resulting tables are meaningful from a semantic point of view. Indeed, because the process of normalisation only ends when every attribute depends functionally on a candidate key for that table, it can be said that the attribute is therefore truly an attribute of the corresponding real-world object.

62. Boyce-Codd Normal Form: Definition. A table is said to be in Boyce-Codd normal form if, for every nontrivial functional dependency $U \to X$ in it, U is a superset of a candidate key.

Not that, if X is a member of U then $U \to X$ is a "trivial" dependency. (Any attribute will be functionally dependent on itself. Such a dependency is not of interest.) If $U \to X$ is nontrivial, i.e., X is not in U, then we want U to be either a candidate key or a superset of a candidate key. In that case, any particular combination of values for U will be unique in the table.

63. Properties of decompositions: losslessness. When we split a table we must make sure that this process does not lead to a loss of information. In database jargon, one speaks of **lossless decompositions**. Luckily, a decomposition in the process of normalisation, as described above, is *always* lossless. It is instructive for our understanding of both functional dependencies and natural join to study why this is the case.

Assume the attributes of the schema $S = (A, B, C, D, \ldots)$ are grouped into the three subsets V, U, and W, and that we are considering the decomposition of S into $S_1 = V \cup U$ and $S_2 = U \cup W$. Note that U represents the attributed that are repeated in both the tables, whereas the attributes V are only in S_1 and the attributes W are only in S_2. To make the discussion worthwhile we also assume that neither V nor W are empty sets, so that both S_1 and S_2 are strictly smaller than S. Graphically, the decomposition breaks every tuple over S into two shorter tuples over S_1 and S_2, repeating the information from the attributes in U:



When the two tables are combined together via the natural join operation, then pairs of tuples are combined whenever they agree in their U-part. Since all tuples arose from breaking up original tuples over S, we are guaranteed to get all the tuples back that we had in the original table. The only problem that could arise is that we get *more* tuples than we originally had.

Pause. So why on earth do database people talk of a "lossy" decomposition in that case?

The reason for this problem can only be that two tuples (x_V, x_U) and (y_U, y_W) fit together because they have the same value in their U-attributes, $x_U = y_U$, but the joined-together tuple (x_V, x_U, y_W) did not appear in the original table. Now, since (x_V, x_U) and (y_U, y_W) *do* appear in the smaller tables, there must be *some* partner tuple for each of them in the other table: Let's say the partner for (x_V, x_U) is (x_U, x_W), and the partner for (y_U, y_W) is (y_V, y_U). Assuming that (x_V, x_U, y_W) did not appear in the original table, it must be the case that $x_W \neq y_W$ and $x_V \neq y_V$. So in fact we get not only (x_V, x_U, y_W) but at least three more tuples the natural join: (x_V, x_U, x_W), (y_V, y_U, y_W), and (y_V, y_U, x_W). This means that neither the functional dependency $U \to W$, nor $U \to V$ holds in the joined-together table.

To summarise the preceding paragraph, a decomposition of a table with $V \cup U \cup W$ into two tables with $V \cup U$ and $U \cup W$ will be lossy if neither $U \to W$ nor $U \to V$ holds. Turning the argument around, if we have either $U \to W$ or $U \to V$, then the decomposition will be *lossless*. In the normalisation procedure, we always have $U \to W$ or $U \to V$ whenever a table is decomposed. So the decompositions carried out in the normalisation procedure are always lossless.

64. Properties of decompositions: redundancy reducing. Above we discussed the problems which arise from repeated information in unnormalised tables. Another word for "repeated information" is "redundancy". The purpose of normalisation is precisely to remove redundancies which arise as the result of a nontrivial functional dependency.

Keeping the terminology from before, our wish to split the table over the schema S is based on the expectation that there will be fewer rows in at least one of the smaller tables S_1 and S_2. Indeed, since the decomposition leads to a repetition of information in the U-columns, one of the two smaller tables had better be considerably shorter to make the whole enterprise worthwhile.

Formally, we can say that the decomposition into S_1 and S_2 will be redundancy reducing if and only if either S_1 or S_2 is *free* of any candidate key for the original schema S (and if both S_2 and S_2 are strictly shorter than S, as we assumed before). This situation is obtained if the set of common attributes, U, does not entail V or not entail W.

In the process of normalisation we consider non-trivial functional dependencies, and so if it was the case that U entails both V and W, then U would be a candidate key for S and we should not concern ourselves with the $U \to W$.

In the previous section we said that for the decomposition to be lossless we need that either $U \to V$ or $U \to W$ holds. For redundancy reducing we get that it must not be the case that both hold, so altogether *exactly one* of $U \to V$ and $U \to W$ should be valid.

So the situation for redundancy reducing is the same as for losslessness: If the decomposition arose as the result of a correct analysis of functional dependencies, then we are guaranteed to make an efficiency gain.

65. Properties of decompositions: dependency preserving. We say that a decomposition procedure is **dependency preserving** if all the functional dependencies of the original table are either present in the final tables or they can be derived from the functional dependencies present in the final tables.

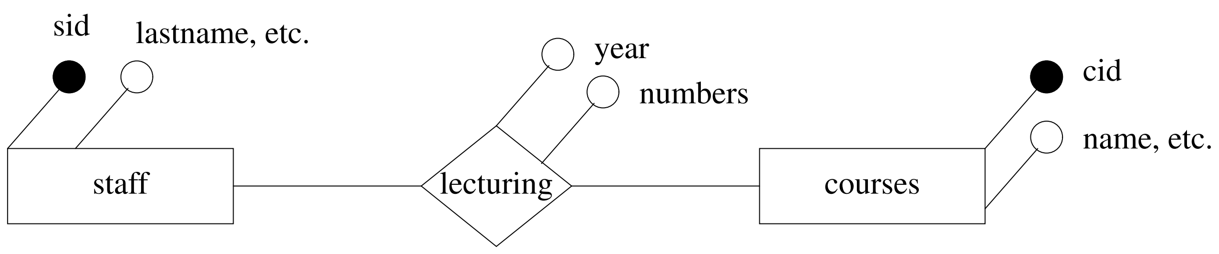
With this, finally, we are hitting a real obstacle, and there is no general guarantee that our decomposition will have the desired property. Let's explain: Functional dependencies are constraints that we observe during analysis of the real world situation, and that we impose on the database design. The way to guarantee that a functional dependency is always satisfied, is to decompose the tables into smaller ones, so that all dependencies arise from candidate keys. Unfortunately, there can be situations where we have a choice for the decomposition, and where decomposing one way will preclude decomposing the other. The net effect is that the second functional dependency is no longer visible in the smaller tables, and hence can no longer be guaranteed or checked.

Luckily, though, conflicts are rare. Here is a (specially construed) example: Consider the schema S = (<u>`part_number`</u>, `supplier`, `supplier_address`, `contact`), where "`contact`" is the name of a sales representative from the respective company responsible for this kind of part. We can therefore assume that we have the dependencies

`supplier` $\to$ `supplier_address` and
`contact` $\to$ `supplier_address`.

If we take out `contact` and `supplier_address` into a separate table, then the first functional dependency is no longer visible. If we separate out `supplier` and `supplier_address`, then the second one is no longer visible. If we separate out all three attributes `supplier`, `supplier_address`, and `contact`, then that table is not yet completely normalised. Whatever we try, we do not have a satisfactory solution. As mentioned at the beginning of this paragraph, this situation luckily is very rare; normally we can go to the Boyce-Codd normal form with only very few decompositions, because the candidate tables that arise from ER-modelling tend to be very accurate already.

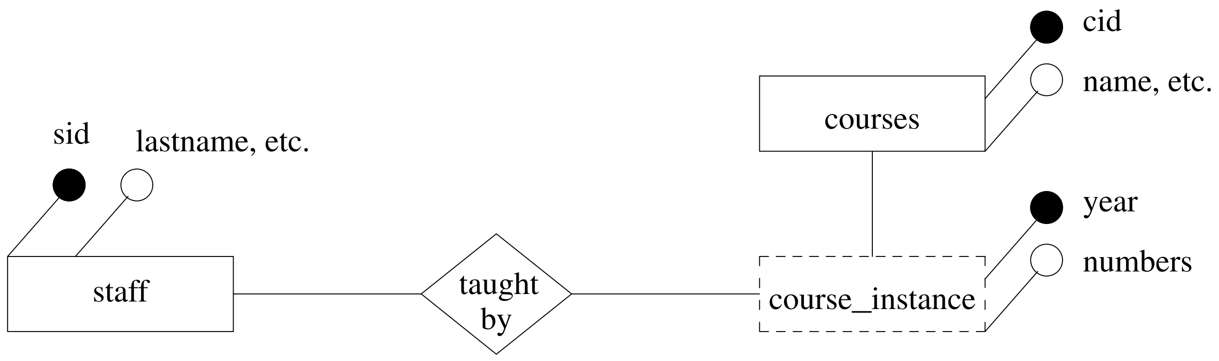
66. Case study: Achim's example tables. The three tables describing courses and staff in the School of Computer Science arose from translating the following ER diagram:



By looking at the "`lecturing`" table more carefully we discovered that there is the nontrivial functional dependency `cid, year` $\rightarrow$ `numbers`. It matters, because we can not assume the pair (`cid,year`) to be unique (as there are several courses which are taught by more than one lecturer). Normalisation, as explained here, would break the "`lecturing`" table into two:

`course_instances` = (<u>`cid`</u>`,` <u>`year`</u>`, numbers`) and
`taught_by` = (`cid, year, sid`).

By staring at this long enough, we discover that `course_instances` is a weak entity controlled by "`courses`". The second table establishes the relationship between course instances and lecturers. The correct entity-relationship diagram for this would have been



There is a lesson to be learned here: Very often one has a situation where there is a general *class* of things and also many instances of that class. There should then be a table for the class descriptions but also a separate table for the weak entity of class instances. Examples are many: Java classes and objects are an obvious one; another is stock categories and actual items in stock, as in "type of computer" and "actual machine".

67. Recording the results of normalisation. As you saw in the previous item, normalisation can be seen as correcting mistakes made during the conceptual modelling phase. It is very important to now go back to the ER diagram and correct it such that the usual translation rules (Section on Logical Design) will produce normalised tables.

If, later on, the design of the database has to be changed then you will have a correct ER diagram to start from, and you will not have to normalise on the same functional dependencies again.

68. Exercises

a. Assume that the functional dependencies $AB \rightarrow C$, $AB \rightarrow D$, and $BC \rightarrow D$, hold for the schema (A, B, C, D). Determine for each of the following decompositions whether they are redundancy reducing, lossless, and dependency preserving. If all three properties hold determine whether the resulting tables are in Boyce-Codd Normal Form.

   i. (A, B) & (B, C, D)
   ii. (A, B, C) & (B, C, D)
   iii. (A, B, D) & (B, C, D)
   iv. (A, B, C, D) & (B, C, D)

b. Like the previous exercise: Assume $A \rightarrow B$, $A \rightarrow C$, $A \rightarrow D$, $A \rightarrow E$, $B \rightarrow C$, and $CD \rightarrow E$ hold in schema (A, B, C, D, E), and consider the decompositions

   i. (A, B) & (B, C) & (C, D, E)
   ii. (A, B, D) & (B, C) & (B, D, E)
   iii. (A, B, C, D) & (B, C) & (C, D, E)