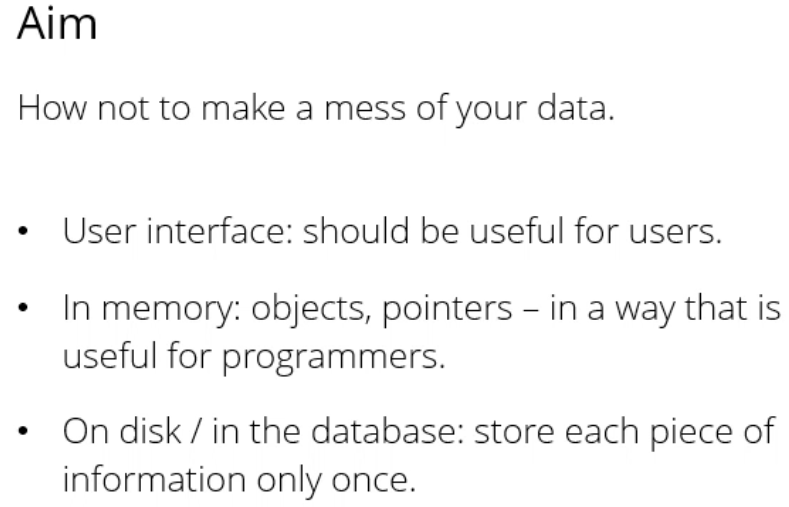
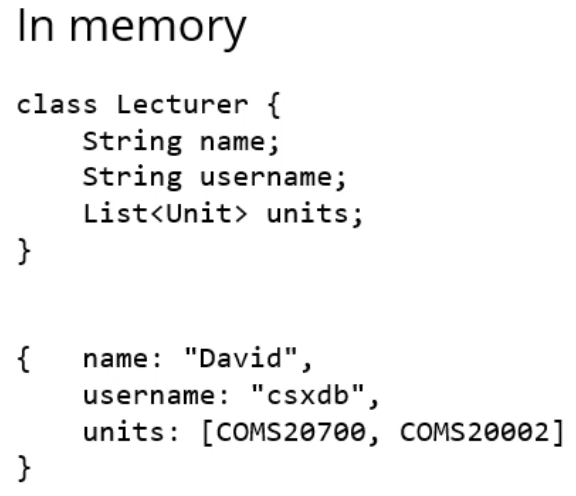
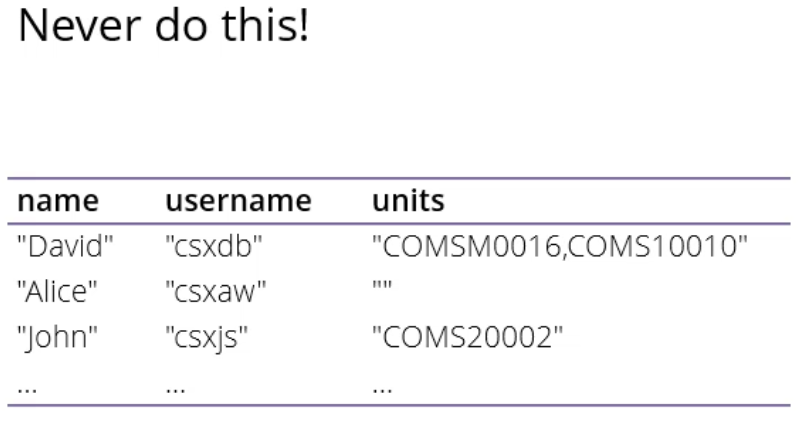
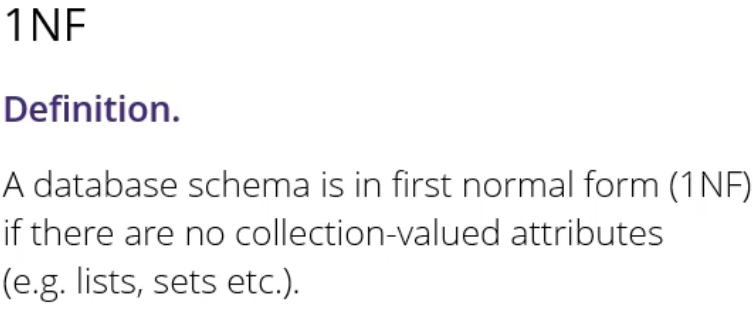
**Normal Forms**

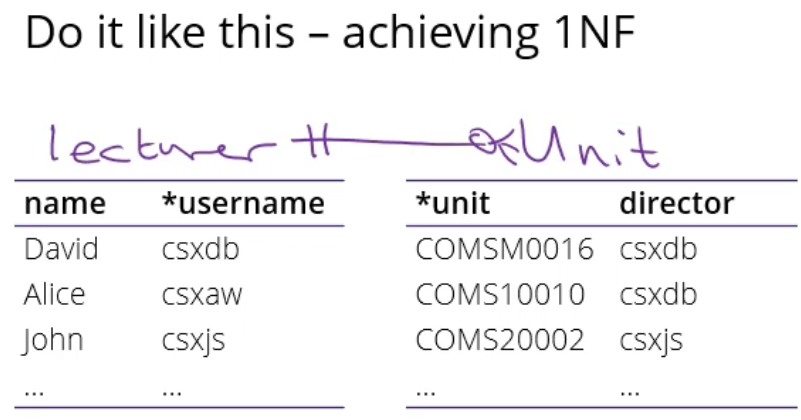
* On disk, never have a comma separated list of units if you have a lecturer teaching multiple units, for example:



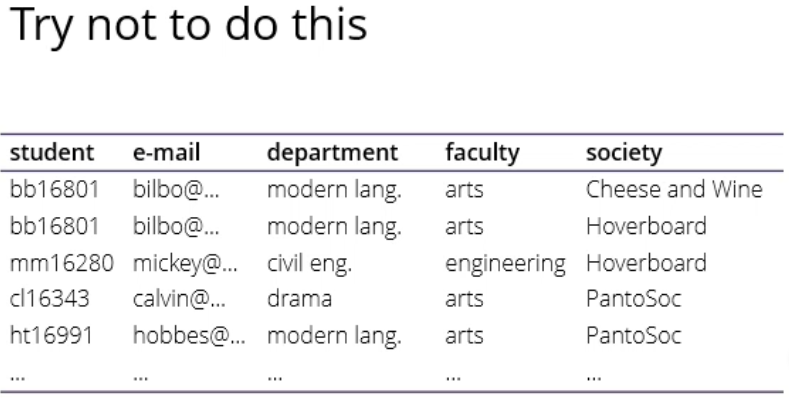
* Never store lists etc. in a single table cell



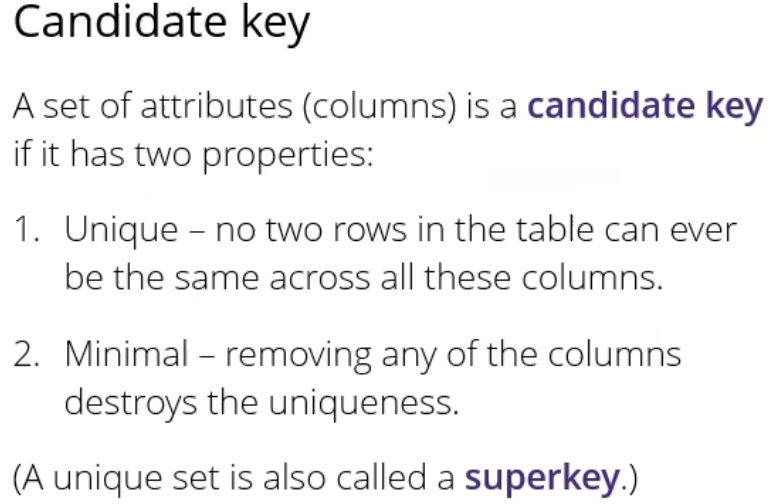
* Sequence of normal forms, each one adds new condition
* Normal form = good design looks like this



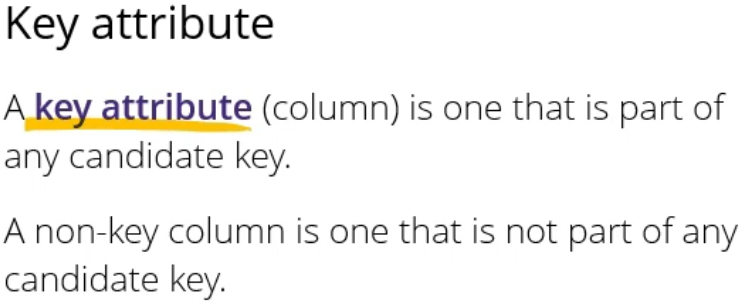
* Figure out what the relationship is between each table e.g. first table is lecturer, second table is unit
  + E.g. maybe each unit is taught by exactly one lecturer, each lecturer can teach 0 or more units
* In the database that becomes two tables, and in the database you implement that with a foreign key
* Horrible schema:



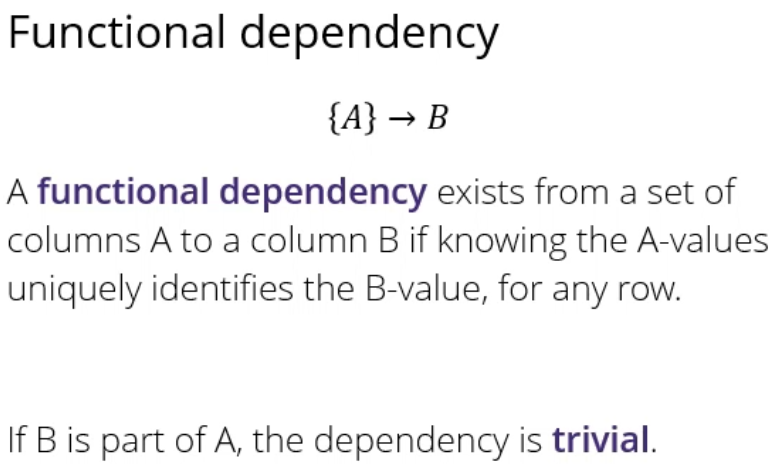
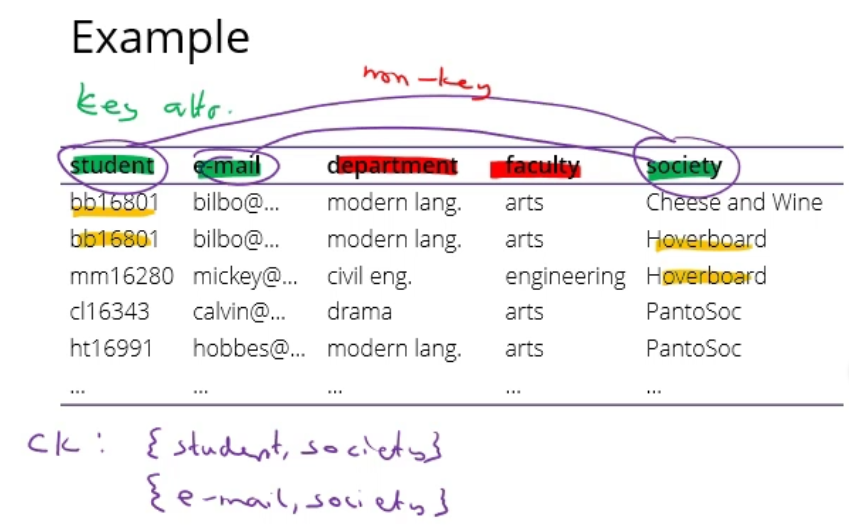
* For each student, we’re storing their username and email address as a way to contact them, also their department and faculty, and name of society they’re a member of
* Storing information multiple times – repeated and duplicated everywhere so can quickly get out of sync
* Relational design = storing each bit of information only once



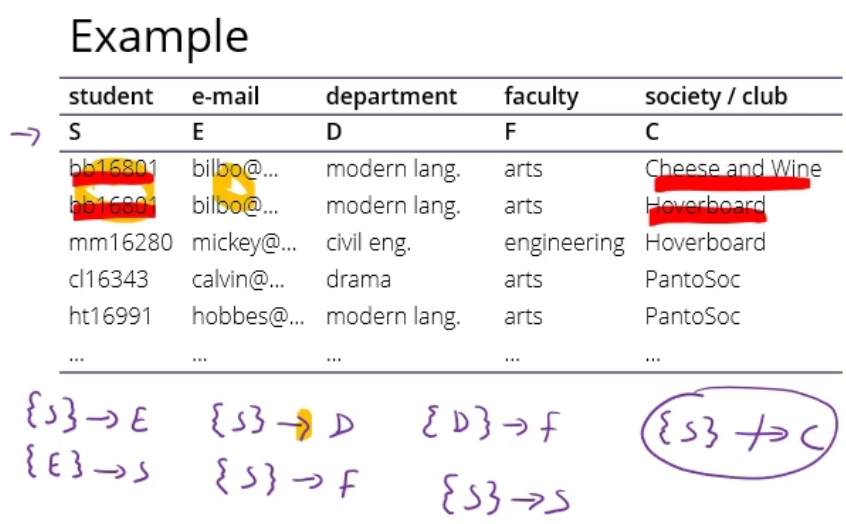
* Candidate key = set of attributes (1 or more columns in a table) with two properties:
  + Uniqueness – never have 2 different rows in the table that are the same across all the columns
    - Superkey = set of columns that only reach this condition
  + Candidate key is a superkey which is also minimal in the sense that if you remove any one of these columns from the set again, you break the uniqueness property



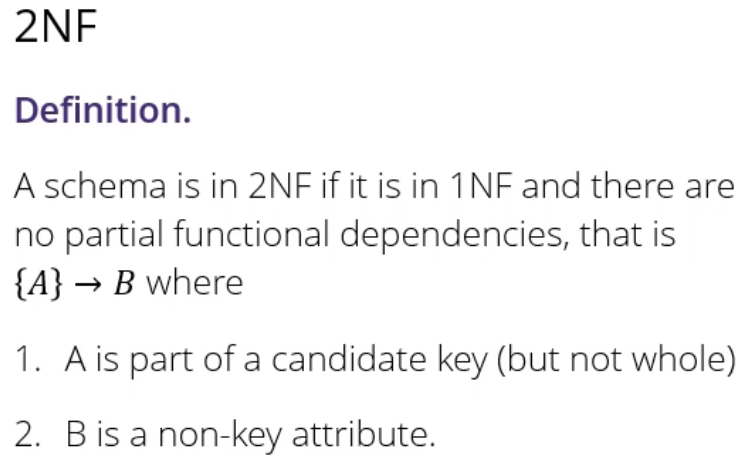
* Key attribute = a column that is part of any candidate key at all
* When you have a schema you can identify the candidate keys from the schema alone, based on that anything that’s part of a candidate key is called a key attribute, anything that’s left over are called non-key columns (attributes)
* Candidate key is part of your schema
* Something is a candidate key or not depending on how your schema works
* The same table can have multiple candidate keys
* A primary key is a choice – you as a database schema designer choose for each table what to make the primary key
* A primary key is always going to be a candidate key
* If your table has multiple candidate keys, it’s up to you which one you make primary
* So a candidate key is part of the schema, primary key is a choice by the database developer
* In our example schema, it’s trying to track memberships of societies
* There is one row per membership per society
  + Won’t be multiple rows for the same student in the same society, but might be multiple rows of the same student if they’re in different societies
* Assume that student names, emails and societies are unique
* Student and society together are a candidate key – there are never going to be two rows of the same student and the same society (that’s the uniqueness property)
  + If you take any attributes out (e.g. student) then you will find examples where there are multiple rows of the same data (e.g. society)
* For the same reason, because student’s emails are unique, email and society are also a candidate key
* With that, we can identify the key attributes (in green) (anything that’s part of a candidate key)
* Non-key attributes (in red) = everything left over



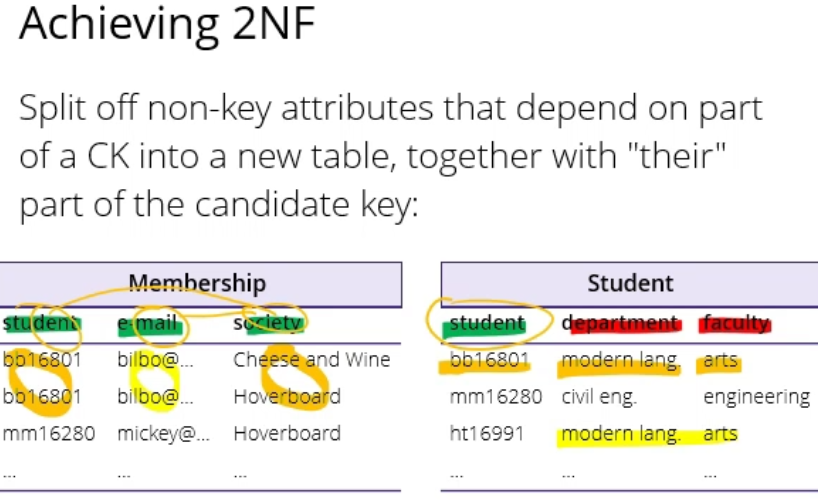
* We want to look at cases where columns depend on other columns
* Functional dependency goes from a set of columns (might be a single one or more) to a single column
* Trivial dependency = dependency where {A} -> A means that for each row in the database, if I tell you the value of column A, you can go in the table and look up the value of column A
  + Not very interesting
  + Functional dependency called trivial if column on the right is already a member of the set of columns on the left
* Functional dependency from {S} -> E : if I tell you a student’s name, you can look up their email
  + There are several rows of the same student – not a problem because they also have the same email
  + Means that if there’s more than one row of the same S, then they will also have the same E
* For the same reason, if I tell you E then you can look up S so {E} -> S
* Students are part of departments, and the department depends on the student
  + So if I tell you the student, then you can look up the department so {S} -> D
* If I tell you the student, you can also look up the faculty because departments belong to faculties so {S} -> F
* For the same reason, {D} -> F
* Not a functional dependency = {S} -/-> C
  + If I tell you the name of the student, you can’t tell me what society is in the same row because they’re in different/multiple societies
* These are all the non-trivial dependencies



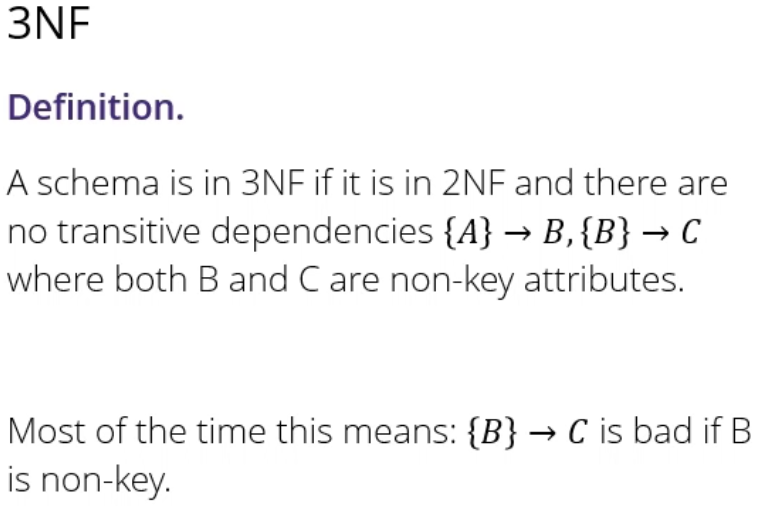
* 2nd normal form = definition based on functional dependencies



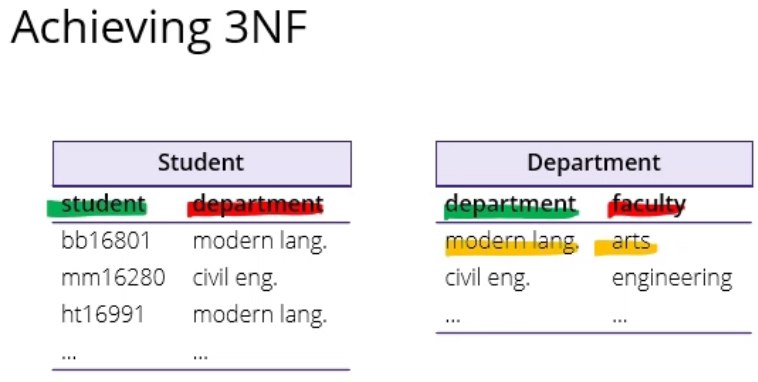
* Only happens if you have candidate keys with more than one attribute in them
* So in the example, there is a functional dependency from student to department
  + Example of a partial functional dependency
  + S is part of a candidate key (student & society) but not the whole candidate key
  + D is a non-key attribute
  + So {S} -> D is a 2NF violation
* One of the reasons that this schema is bad, is that it is trying to do too many things at once – it’s trying to track which students are in which society, and also which students are in which department
* What this functional dependency here shows is that the department attribute is only an attribute of the student, but not of the society
* Trying to put an attribute that only depends on the student in a table where the candidate key is tracking both students and societies
  + That’s what’s causing the relationship that bilbo is in modern languages to be repeated once for every society that bilbo is in
* The solution to this problem: split off all attributes that depend only on the student into a new table called student
* The attributes that used to be non-key (department and faculty) go in the student table
* We need to identify the student, so will copy over one of the student identifiers (username)
* Now have a separate student table, and only the attributes that depend on the student are in that table
* So each student only needs to appear once in this table
* Still have a problem with modern languages and arts appearing multiple times, but we will come back to this
* In the membership table, we still have two rows for bilbo if he is a member of two societies, but each of these rows now contains different information (namely each one talks about a different society)
* We will also deal later with the fact that the emails are repeated



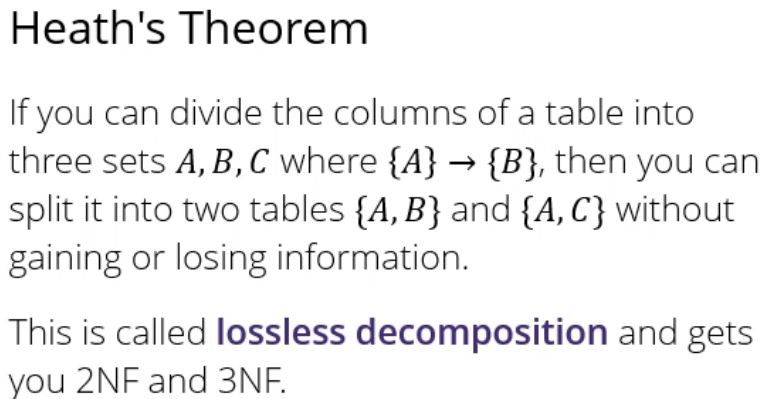
* In the membership table, there are still 2 candidate keys: student society and email society
  + They are key attributes
* In the student table it’s just student as a key attribute on its own
* Another problem: modern languages is part of arts, but we’re still storing that at least twice in the table
* Problem because the fact that modern languages is a part of arts is a fact about the university, not a fact about the student
* Functional dependencies:
  + {S} -> D student implies department – not a problem in the student table
  + {D} -> F department implies faculty – that is a problem in the student table because it isn’t a department table
* Formally, to deal with this problem we need third normal form



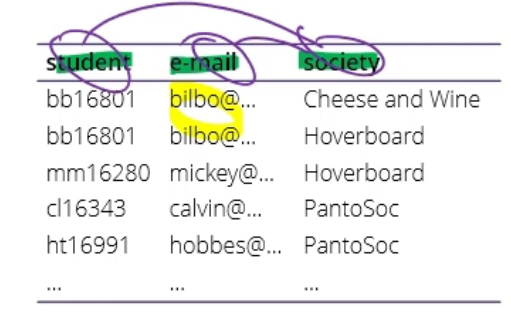
* Transitive dependencies = a chain of two dependencies where A implies B and B implies C, but both B and C are non-key
* Most of the time this means it’s bad to have a functional dependency from B to C if B (on the left) is non-key
* Way to fix this: to put department information in a separate table called department
* In the student table, we have a key attribute student (a candidate key, our only one) and the non-key attribute department
  + The department is information about the student, so it belongs in the student table
  + Now, to reach student, their information is stored only once
* In the department table now, the key is department
  + The formally non-key attribute in the student table department has been promoted to key
  + Means that each department only needs to appear once
  + Now the non-key attribute is faculty, and only need to record once in a whole database that modern languages is a part of arts



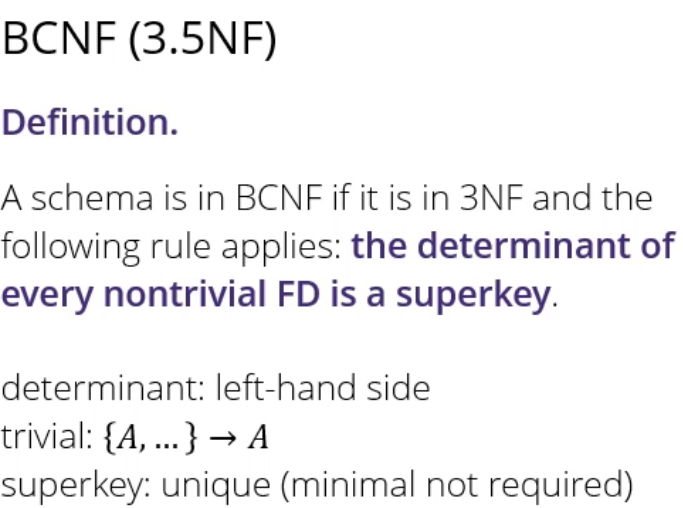
* The whole principle here is that when we store the information on disk, we split it apart into multiple tables
* Then when we want to do something useful with the information like answer a question, we join them back together again
* Databases are written to make joins as efficient as possible
* More formal way of expressing how to get the schema into second and third normal form:



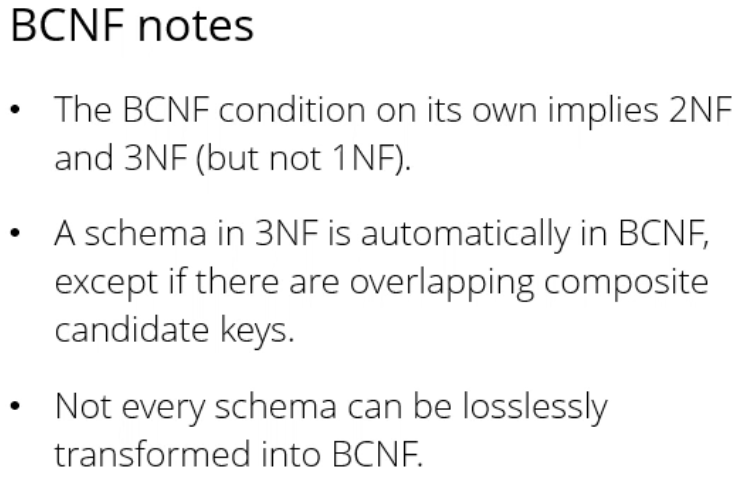
* Heath’s theorem = if you have a functional dependency from A to B which is annoying you (B can be a set of columns if you want), if you divide your columns into three sets, A are the things on the left, B the things on the right, and C is everything else
* In this case, you split your columns into two tables – one just the A and the B columns, the other one A and C
* The functional dependency on the left A lives in two tables, and that’s how you get the relationship back
* Heath’s theorem says that if you do it that way, you get a property called lossless decomposition – means that the information in your new tables is exactly the same as in your old ones – don’t gain or lose any information
* In the last step, because everything is key, second normal form can’t be violated because it says you’re not allowed to have dependencies out of part of a key into a non-key attribute, but there’s no non-key attributes left



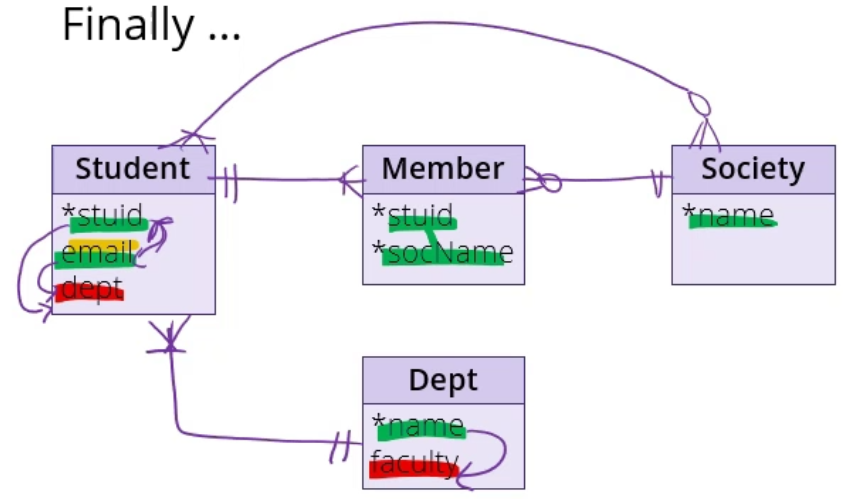
* Third normal form says you’re not allowed to have dependencies between non-key attributes, but there’s no non-key attributes left
* Not a good schema because you’re still repeating bilbo’s email address multiple times
* Boyce and Codd = two big names in database and database theory
  + Said that the reason that you can follow all the normal forms and still have a rubbish schema is because you’re doing third normal form wrong
  + Need another rule that fixes that



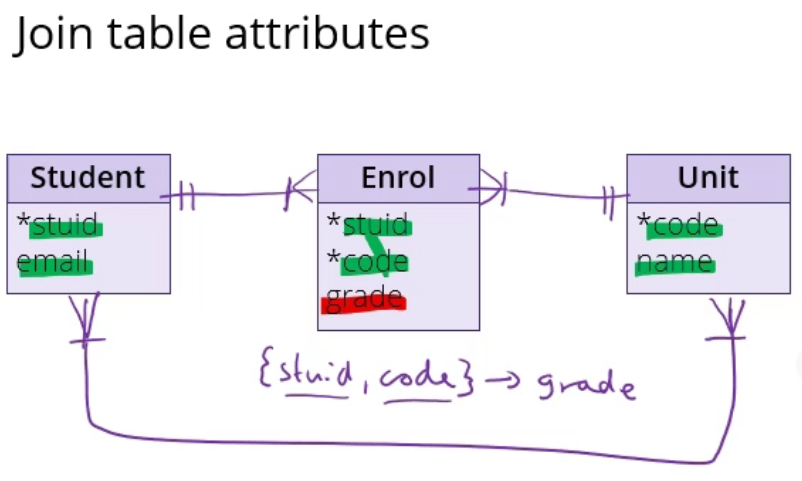
* Some people call BCNF 3.5NF as it’s an extension to 3rd normal form
* Generally speaking the level of normalisation you want to aim for in your schema
* A schema is in BCNF if it’s in 3rd normal form and you have a new rule that says the determinant of every nontrivial functional dependency is a superkey
* Non-trivial = column on the right isn’t part of the set on the left
* Superkey = key with uniqueness property (can’t be two rows which are the same in all the columns, but doesn’t say anything about minimality)
* Whereas 2nd and 3rd normal form both say that particular kinds of functional dependencies are bad, BCNF says that these are the good kinds of functional dependencies, and anything not covered by our rule is bad
* The good ones are where the left hand side is a superkey
* Means that the only functional dependencies you’re allowed to have in a table are ones related to keys, and not even part of the keys as then it wouldn’t be a superkey, would be a partial key
* Another way of expressing BCNF is saying that functional dependencies must depend on the key, the whole key, and nothing but the key



* Because of the way it’s written, BCNF actually implies 2nd and 3rd normal form automatically
* 1st normal form is different – is to do with collection values not functional dependencies
* Checking something is in BCNF is hard because you have to check every possible functional dependency
* Most of the time 3NF implies BCNF, except in some edge cases – if there’s more than one candidate key which has more than one attribute in it (called composite candidate keys) and they overlap
* So in our schema before, we had student ids, societies and email
  + With 2 candidate keys of student and societies, and society and emails
  + Both candidate keys had more than one attribute and overlapped in more than one element of society
  + That’s the reason why we’re still repeating bilbo’s email address even though we inferred normal form
* Heath’s theorem is automatically going to transform anything into second and third normal form
  + Most of the time works for BCNF unless you have multiple overlapping candidate keys – then need to fix by hand
* Many-to-many relationship between student and society, implemented by members table
* Separate relationship between students and departments
* Every table has one candidate key – student dept and society have name, member is a join table so it has a composite candidate key of student id and society name (only one here so not breaking any normal form)
* Any non-key keys are just attributes in the table
* You can see, if you look at the arrows, that the determinant (the place where the arrow starts) is a candidate key, so it’s fully normalised
* Now you don’t need to repeat bilbo’s email address, because there will be one entry in the student table for bilbo which will give his email address exactly once



* Brings us back to the question of when to put an attribute on a join table
* Say many-to-many relationship between students and units, so every unit has at least one student and vice versa, but doesn’t change where the attributes go
* Implemented by the join table enrol – every unit enrols at least one student, every enrol entry refers to at least one student
* Grade is non-key because there can be two students with the same grade
* Grade goes on the join table because you can’t put it on either the student or unit table – there is one grade per student per unit
* Student id on its own isn’t going to imply grade, because the same student might have different grades for different units
* The unit code on its own doesn’t imply grade, because the same unit might have different grades for different students
* Functional dependency where the left side is a composite candidate key, but it’s the whole key so is fine



**Summary**

* Prime attribute = attributes of the candidate key that define uniqueness
* Non-prime attribute = attribute that doesn’t occur in any candidate key
* Super key = set of attributes in a table whose values can be used to uniquely identify a row (tuple)
* Candidate key = minimal superkey of the relation schema
  + Set of attributes such that each instance relation doesn’t have two distinct rows with the same values for these attributes
  + The minimal set necessary (non-minimal is a super key)
* Primary key = chosen candidate key
* Foreign key = set of attributes in a table that refers to the primary key of another table
  + Foreign key links these two tables
* Functional dependency = a constraint between two sets of attributes in a relation from a database i.e. a constraint between two keys
  + X -> Y (X functionally determines Y) iff each X value is associated with precisely one Y value
  + If the values of X are known, the values for Y corresponding to X can be determined by looking them up in any row containing X
* Transitive dependency = a functional dependency is said to be transitive if it is indirectly formed by two functional dependencies
  + E.g. X -> Z if:
    - X->Y
    - Y does not -> X
    - Y -> Z
  + Can only occur in in a relation of three or more attributes
  + When there is an attribute that depends on some non-prime attribute and not the prime attribute
* Partial dependency = the proper subset of candidate key determines non-prime attribute
  + Any attribute in the table depends only on a part of the primary key and not on the complete primary key
* Determinant = any attribute you can use to determine the values assigned to other attributes in the same row
* 1NF = Only one data attribute per column
* 2NF = already in 1NF and there is no *partial functional dependency*: every non-primary-key attribute is fully functionally dependent on the primary key
  + Non-prime attributes are allowed to be functionally dependent on non-prime attributes
  + Goal to eliminate redundant data
* 3NF = already in 2NF and 1NF, and every non-prime attribute is *non-transitively dependent on superkey of relation*
  + There is no transitive functional dependency (no non-key attribute is transitively dependent on the primary key)
  + Non-prime attributes are only allowed to be functionally dependent on super key of any relation
  + Goal to ensure referential integrity
  + For the functional dependency C -> D:
    - C should be the super key
    - D should be a prime attribute, i.e. D should be a part of the candidate key
* BCNF = 3NF may not remove 100% redundancy because of X?Y functional dependency, if X is not a candidate key of given relation
  + A relation is in BCNF iff X is superkey for every functional dependency (FD) X?Y in given relation
  + A relation is in BCNF iff every determinant is a Form (BCNF) candidate key
  + To test if a relation is in BCNF, identify all determinants and make sure they are candidate keys
  + For a functional dependency say P -> Q, P should be a super key
    - P cannot be a non-prime attribute if Q is a prime attribute