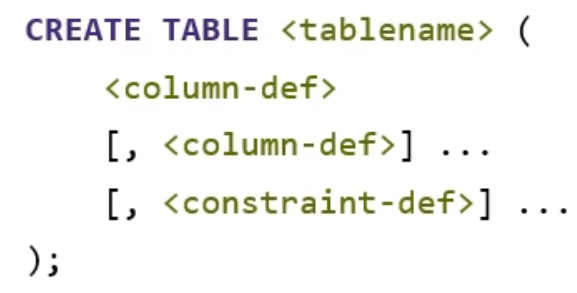
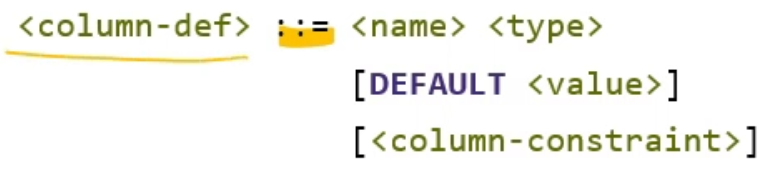
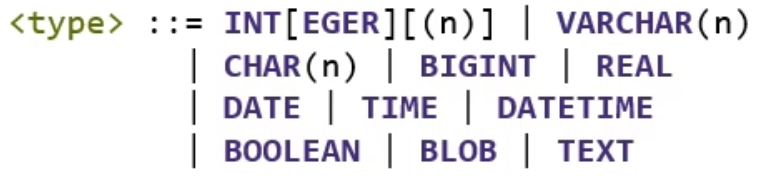
**SQL Create-Drop Scripts**

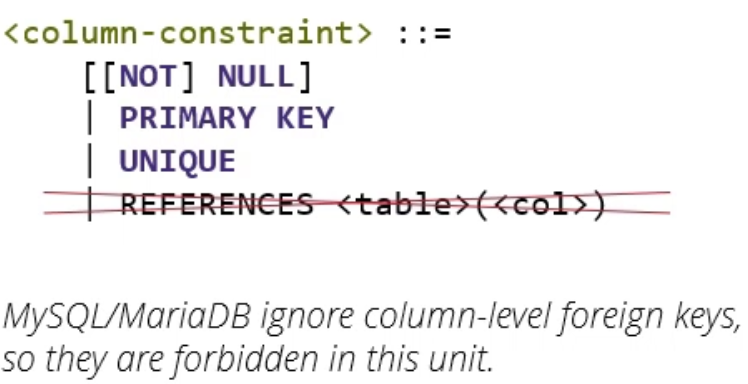
* Two main sub-languages in SQL: select (for working with data) and create
* Create: for creating, deleting, and modifying tables to create a table in the first place, before we put the data into it



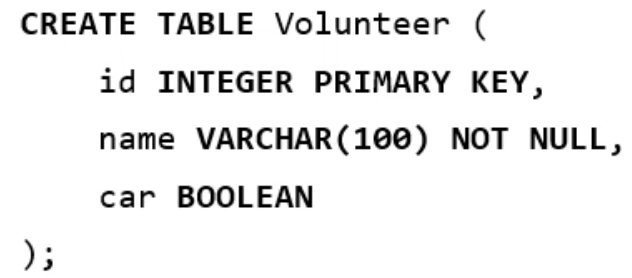
* SQL keywords don’t have to be capitalised, but by convention they are
* Green colours are parameters to be replace by actual values
* All SQL statements have to end with ;
* [] David uses to show options, … means repeat as many times as you want

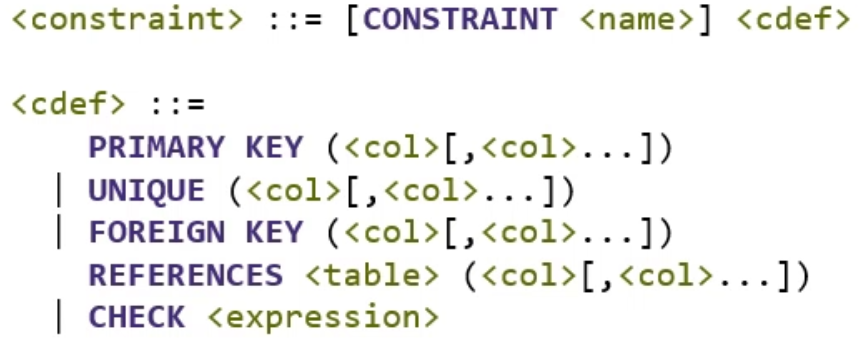
* : := means “this is how it works”
* Which kinds of data types are allowed depends on your database – although SQL is sort of standard, every database works slightly differently with what it implements
* VARCHAR = a variable field (CHAR was a fixed width field, VARCHAR is as much as it needs – now pretty much just use VARCHAR)
  + Something you can search, index on etc.
  + TEXT, as far as database concerned, a bunch of text which you can either retrieve the whole text or none of it (can’t search it or anything)
* BLOB = binary large object
  + Binary counterpart to TEXT
  + What you would use if you were storing files (e.g. images) in a database
  + Can’t search contents – can just request the whole thing or not
* DATETIME is everything together and represents a point on the arrow of time



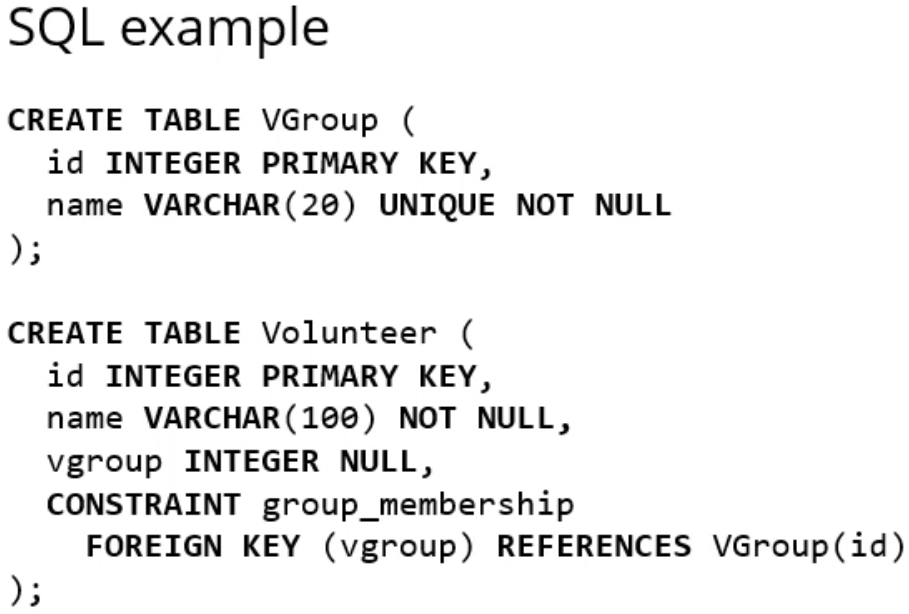
* At the end of a column definition, if you want you can put a column constraint
  + Restricts further the type of data that can go in this column
  + By default, every column is nullable – you can make this NOT NULL
  + If you want a column to allow NULL and that has some kind of meaning in your application, you explicitly declare a column NULL (for readability)
  + Sign to someone reading SQL script that there is a reason that there might be NULLs in this column (e.g. it is a foreign key which is optional)
* PRIMARY KEY constraint you can use exactly once per table
  + Declares which column is your primary key
* UNIQUE constraint says that values in this column are unique, so the database will refuse any attempt to insert or edit data that would cause two rows in this table to be equal in this column
  + E.g. prevents you from having two students with the same username
* SQL standard and most other databases will allow you to declare a column key on the foreign level, with the references keyword
  + MySQL and MariaDB will allow you to do that, but they will just be ignored, so we won’t do that here
* Example:



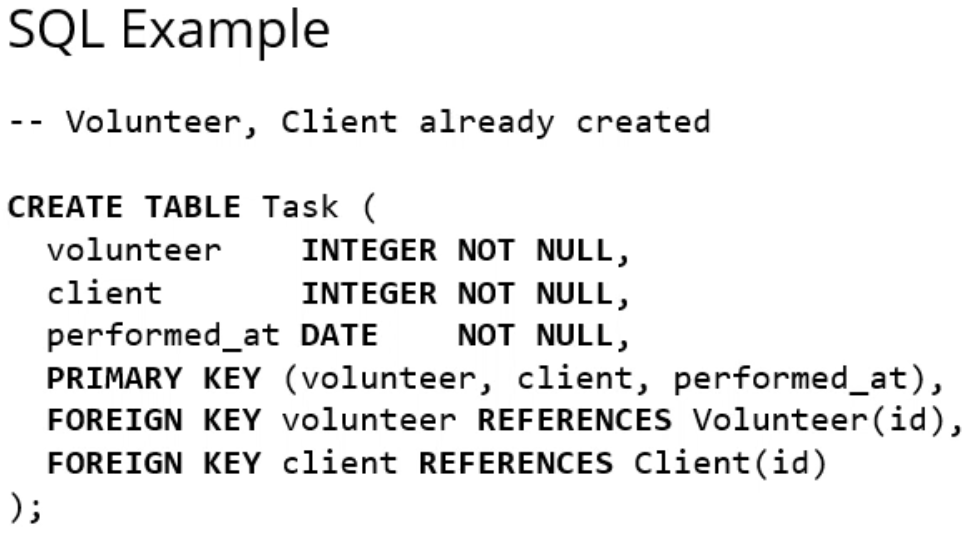
* Here, create table on the top line and then list all the columns with commas between
* SQL doesn’t care about line breaks, tabs etc. all just seen as spaces to SQL



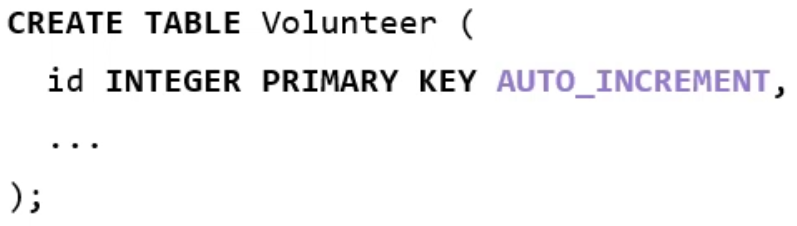
* Constraints at a table-level can do everything that constraints at a column level can do, with even more as they can cover constraints that apply to more than one column
* Syntax for constraints is optionally, you can give the keyword CONSTRAINT and then a constraint name
  + Constraint names have to be unique across the whole database
  + If you ever get an error in relation to the constraint, it will print out a useful error message that contains the constraint name
  + E.g. if there is a constraint that says the volunteers must have an address, and you call this constraint “volunteer must have address”, if you ever try to insert a volunteer without giving an address, it will say “error, constraint volunteer must have address was violated”
  + The constraint definition itself can be one of several options
  + A PRIMARY KEY constraint declares a primary key, and within brackets you can give a comma separated list of columns
    - So a primary key can span multiple columns
  + Similarly for a UNIQUE constraint, the combination of all these columns together has to be unique (e.g. first name and surname – although don’t really want to use human names)
  + FOREIGN KEY constraints followed by a column separated list of constraints in the current table which make up the foreign key, keyword REFERENCES, the name of the table which contains the primary key, then a comma separated list of columns that make up the primary key
  + This has to be in the same order in the sense that the column first in the list of the foreign key columns will be matched with the first column in the primary key
  + Finally a CHECK constraint takes any possible expression that can evaluate true or false, and it refuses any insert or update that violates that
  + E.g. if you want an integer that can’t be less than 0 or has to be in the range 1-10 you can put a check statement for that
* Example implementation of volunteers belong to group scenario:



* First create a table for groups with CREATE TABLE VGroup
* Then put in columns of id which is going to be an integer and the primary key, and name which is going to be a VARCHAR of at most 20 characters which will be unique, and not null (every group has to have a name)
  + Name is a candidate key not the primary key
* Example of a many-to-many relationship, of tasks performed by volunteers on behalf of clients:



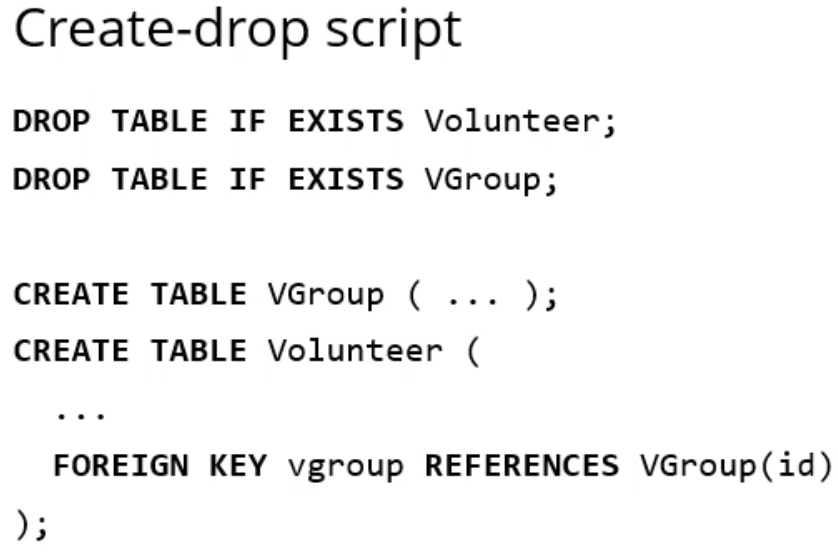
* Join tables are the standard example of composite primary keys – here we have a three way one
  + In a join table, the primary key covers the two or more foreign keys, sometimes it covers even more
* For MariaDB, if you have an integer primary key which is a single column, you can put the keyword AUTO\_INCREMENT which says that if I ever put an entry here and I don’t give you an ID, then just choose one yourself in such a way that it doesn’t clash with everything that already exists
  + Typically in the form that the database finds the highest one that exists and adds one to that



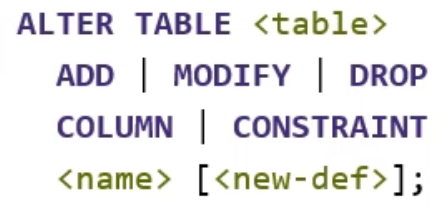
* DROP – removes tables again
* This syntax removes a table if it exists (if it’s allowed to), and creates an error message if it doesn’t exist:



* A create-drop script is a script to create and drop your script in such a sense that the effect of running this script, whether the tables already exist or not, is that the tables now exist and are empty



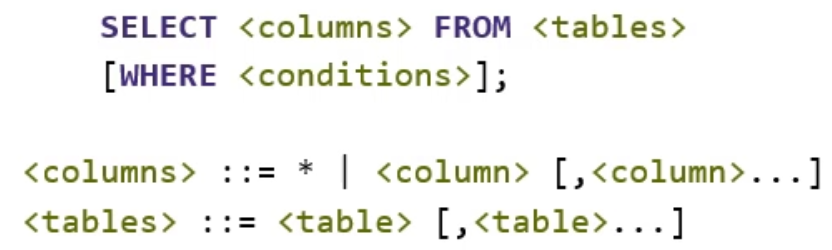
* Create part: have all create statements, they have to be in the correct order in the sense that if you have a foreign key reference in a table, that table must already exist beforehand
* In this case, in the volunteer table, we make a reference to the VGroup table, so the VGroup table has to be declared before the volunteer table
* For the same reason, if you have a join table it has to come after the two or more tables that it references
* If you ever have a circular dependency, in the way that table A references table B and vice versa, you’ve probably done something wrong with your data model
* Conversely, when you’re dropping tables, you can’t drop the group table while the volunteer table exists otherwise your database will complain that you have a foreign key reference to a table that doesn’t exist
  + So you have to drop the volunteer table before the group table
* So in a create-drop script you have to do three things
  + You have to have create statements to create your tables
  + Drop table if exist statements to make sure that even if the tables exist when you run this script, it will delete all the tables and make them new
    - E.g. to update data model, you would put update in SQL script and then run whole script through, remove old tables and add new ones
    - If you’re making a replica of your database, whether the tables exist or not, running the create-drop script will make sure they exist in the end
  + For that reason, you have your create statements at the end and you typically want to do your drop statements in the opposite order to your create statements
* Later on, when your application has launched and you already have data in production and you can’t just create and drop your tables, you will use the ALTER TABLE statement
* Syntax: ALTER TABLE, name of table, what you want to do to it, the name of what you are doing something to, then if you’re adding or modifying, the new definition



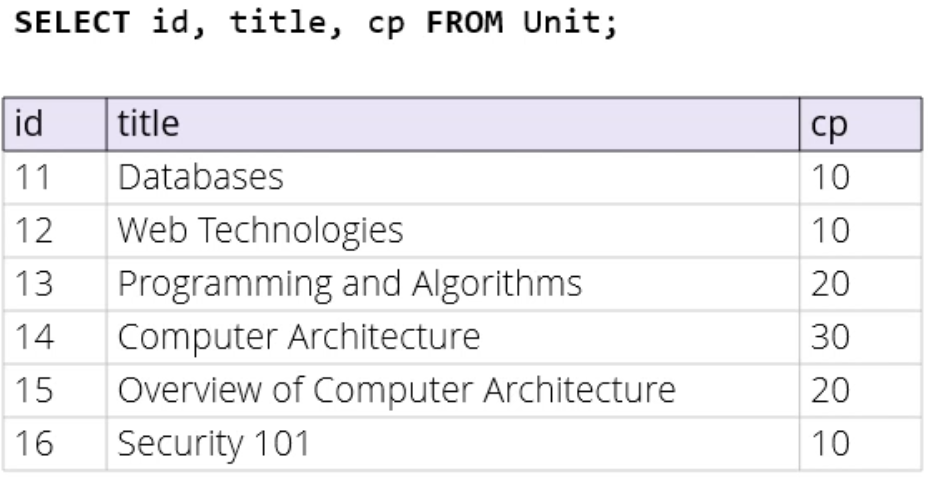
* Database will check if constraint is violated – if it is, it will refuse to add it

**SQL SELECT**

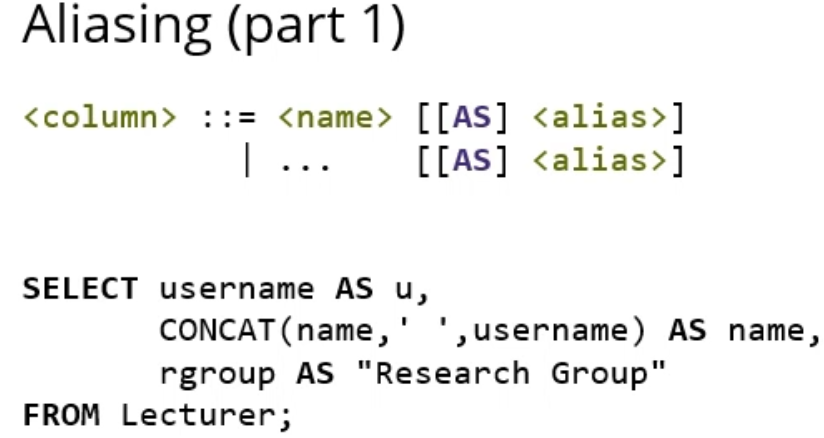
* SELECT statement = use to get data out of the database



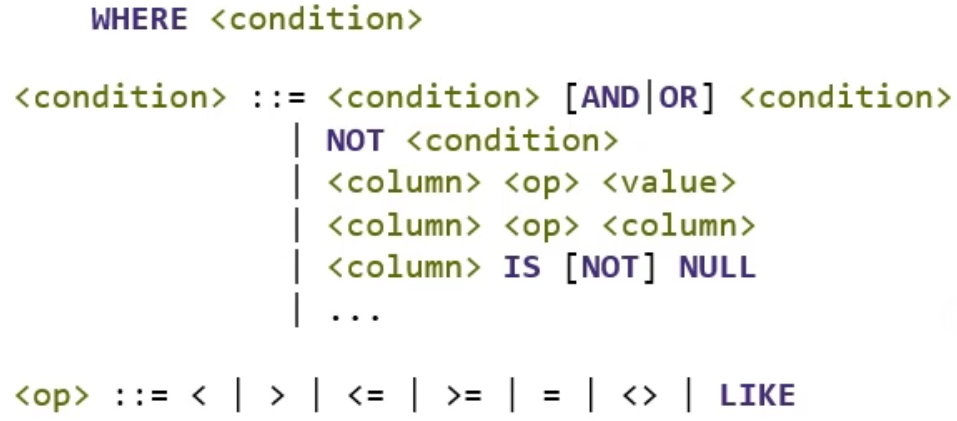
* \* means give back everything
  + E.g. SELECT \* FROM table; will give you back everything from the table
  + Fine on command line, bad style if you are using SQL from another program – save yourself some time and effort by only selecting the columns you actually need



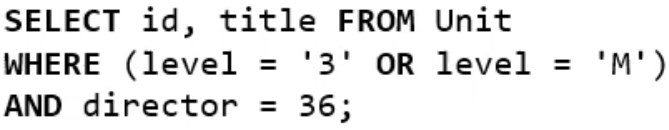
* Aliasing is a way that the column names in a table that come out of a query don’t have to be the same as names in the original table in the database



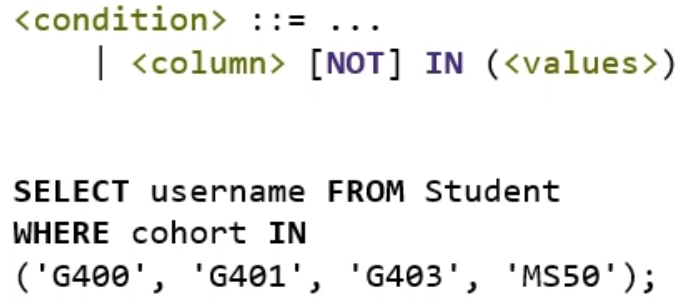
* One reason for alias = you might want to build a column out of something that’s not actually a column in the original table
* E.g. here you might want to CONCAT(inate) someone’s name, a space, then their username so that you have their name and username in the same column, and call the result name
* Also might want to give a column a descriptive name in the report
* One of the cases where in SQL you use double quotes – most of the time you use single quotes for strings
  + Strings that refer to names of columns will use double quotes
* WHERE cause is a filter on the rows
* Database first does SELECT, then after that takes WHERE clause, goes over all the ones you’ve selected and throws out the ones that don’t match
* For efficiency reason, the database actually uses an algorithm where rows that don’t match the WHERE clause aren’t constructed in the first place



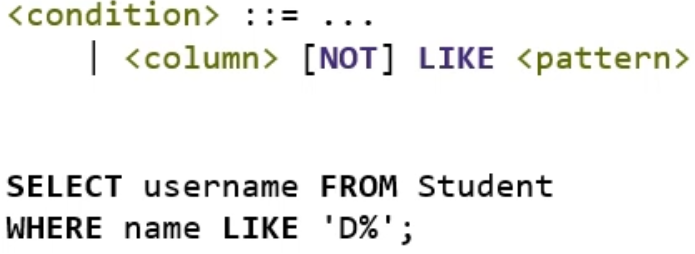
* WHERE clause takes a condition
  + Can build smaller conditions into larger ones with AND & OR & NOT operations
* Basic things you can do:
  + Compare a column to a value (usual operators, apart from equivalence is = and not is <>)
  + Compare a column to another column
  + Things with NULL
* Example:



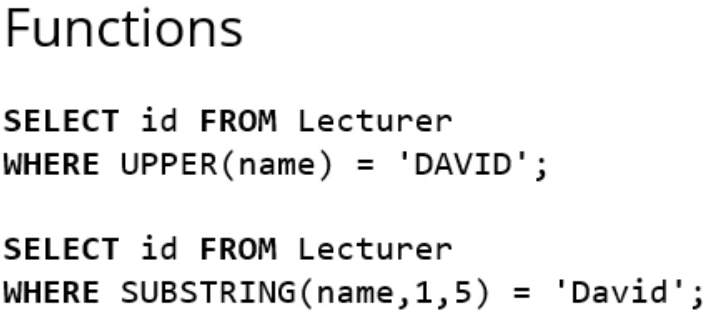
* Condition has several parts – imagine there’s a column level that has a string type
  + If you have a string datatype, you must put it in single quotes
* Another useful construction is (NOT) IN
  + Compares a column against a list of values



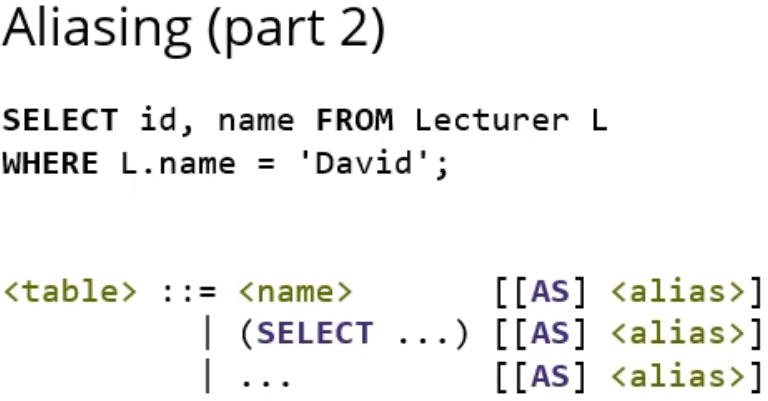
* Basic use is if you have a list of things you want to find
* Last line (…) is equivalent to a list of ORs
* LIKE is a pattern matching construct (with NOT LIKE for negative matches)



* % sign is a wildcard that matches any number of characters
  + Names starting with D: D%
  + Names with D anywhere: %D%
  + Names ending with D: %D
* Another wildcard is ? which matches a single character
  + E.g. ‘?an’ would match Dan and Jan
* Depending on your database there might be different pattern matching available, LIKE is the one defined in SQL standard and will have some version of it in pretty much any database
* SQL, like any programming language, has a number of built-in functions



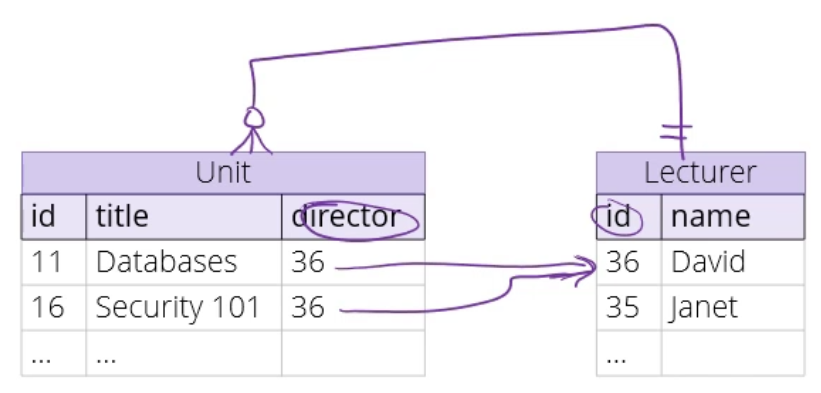
* You can select a column based on a function as well – above it takes names turns them to uppercase, then matches them to the name DAVID
* SUBSTRING function says, starting from the name column, pick a substring starting at the first character and ending at the 5th one, and match that with David
* You can alias tables as well as columns



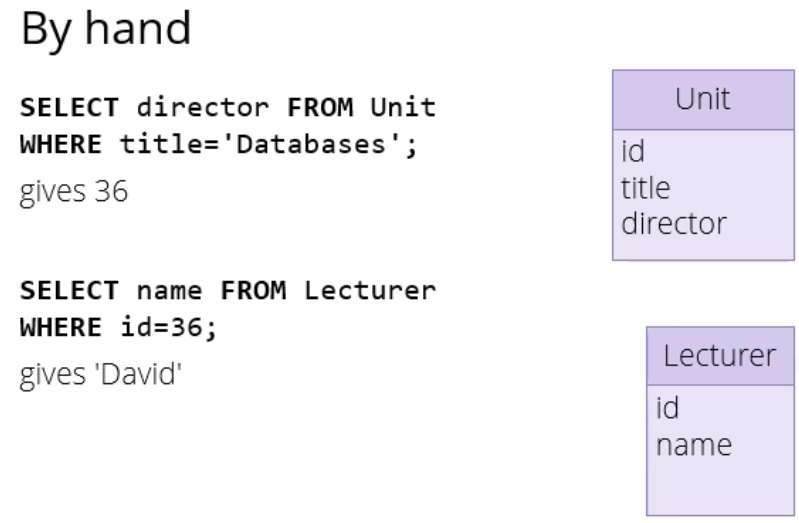
* By convention, people don’t usually write AS when it’s a table
  + So in the example above, selecting from table Lecturer but calling it L from now onwards
* Can use a sub-select – a SELECT that returns a table as a table in another select
* Many other things you can do that also return tables

**SQL JOIN**

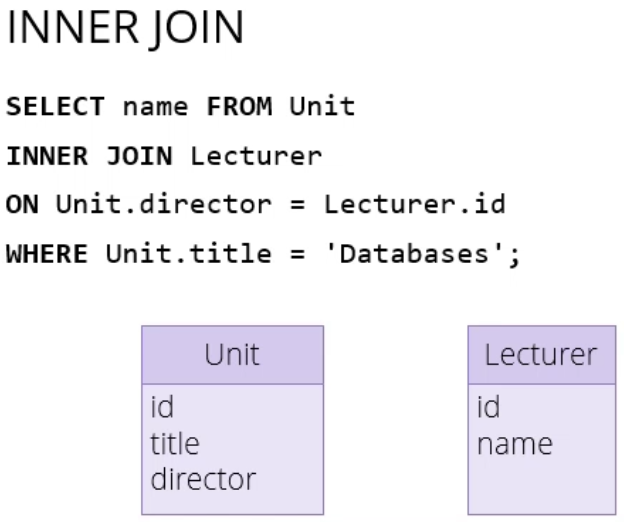
* Point of relational model = you store data apart, and if you want to do something with it you have to join it back together



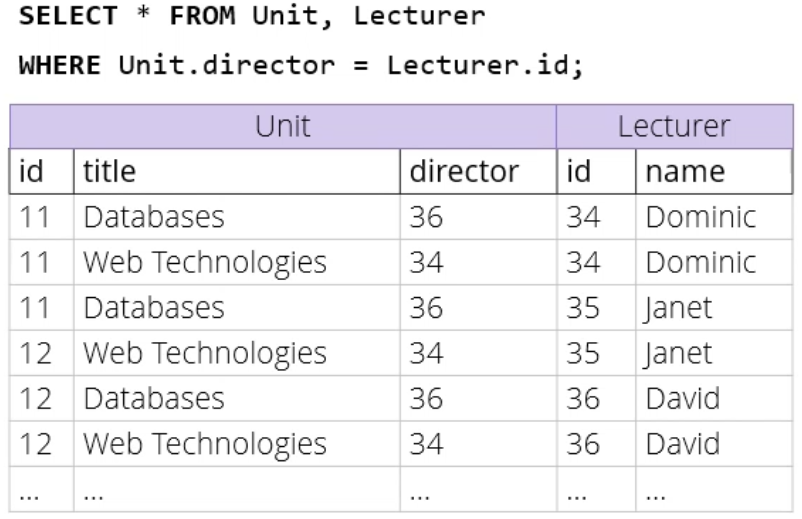
* Lets say we were trying to answer who teaches databases



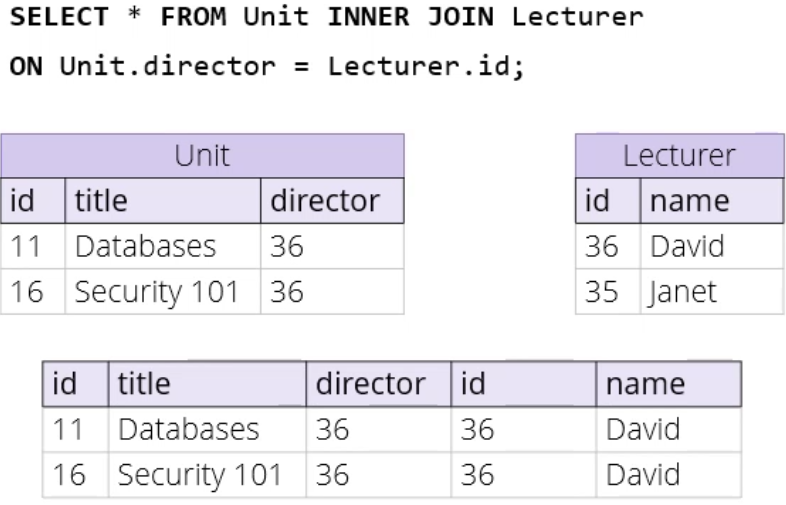
* But we want our database to be able to do this for us



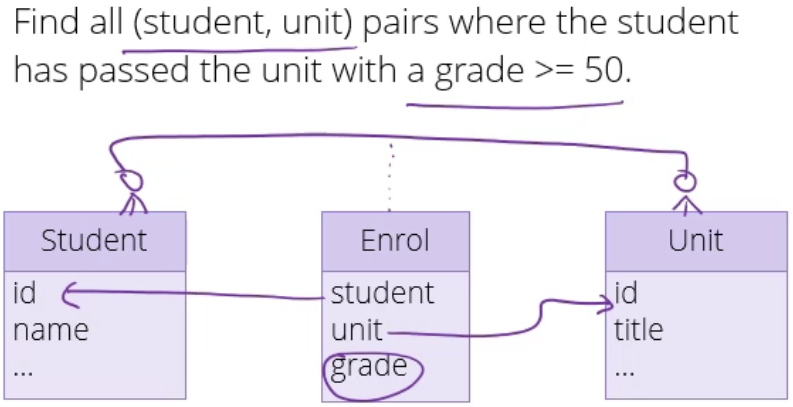
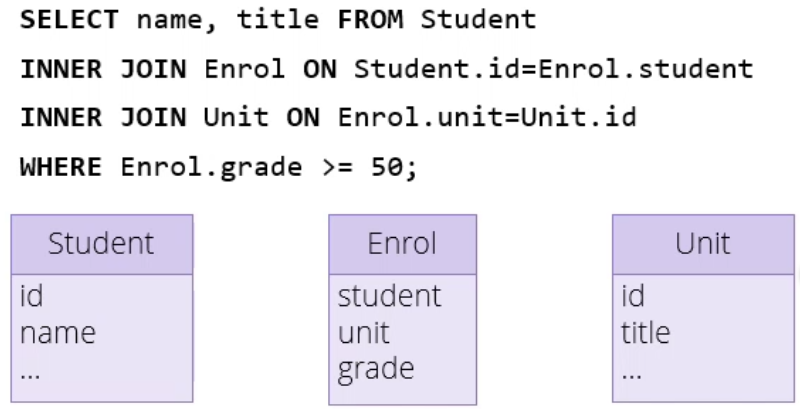
* The thing we want is an (inner) join
* SELECT always comes in three parts: SELECT FROM and WHERE
* In the example above, we’re selecting from the compound table Unit INNER JOIN Lecturer
* Effectively making a link from director to lecturer id
* In the FROM clause, you can give a comma separated list
* So in this example, it will give you every possible combination of Units and Lecturers, whether they make sense or not



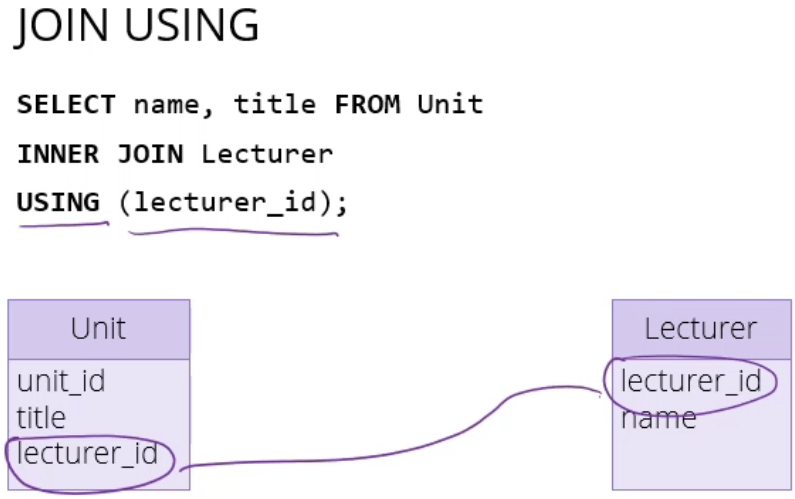
* If we add a WHERE clause, I cross out all the rows where it doesn’t match (have to use Lecturer.id to specify which id we mean)
  + Leaves 11 Web Technologies and 12 Databases
* Inner join does more or less that, but more efficiently
* The query I would actually use is:



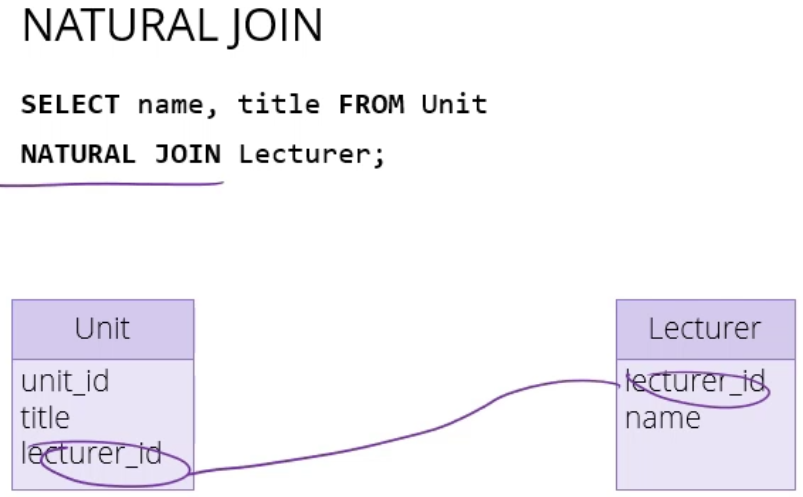
* Would join Unit.director and Lecturer.id
* Lets me read off the important information

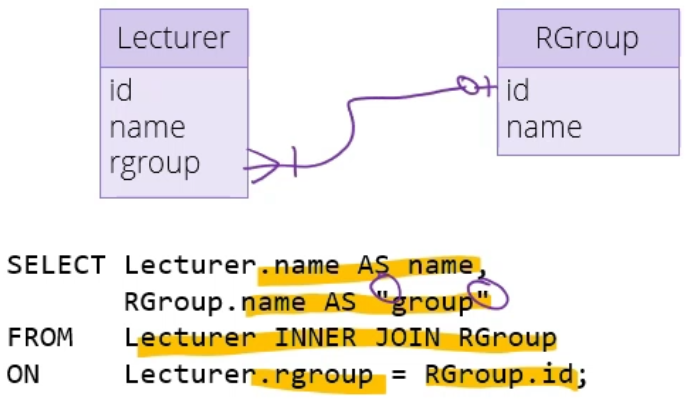
* Many-to-many between students and units (optional on both sides)
* Join implemented by the table Enrol that has two foreign keys – student to Student.id and unit to Unit.id
* Grade has to be on the join table as it isn’t a property of the student (they have different grades for different units) or the unit (different students have different grades for the same unit)
  + So grade has to sit on the join table as there is one grade per student per unit
* In the query, we are SELECTing – we want pairs of name and title
* We want only those pairs that are >= 50 (WHERE Enrol.grade >= 50; )
* FROM clause explains where we are selecting this from
* So the table starts off with Student, then we JOIN Enrol ON Student.id=Enrol.student
* Then we JOIN Unit ON Enrol.unit=Unit.id
* Some people always like to name their primary keys id and their foreign keys after the thing it points to
* So a foreign key in the Enrol table that points to the student table might be called student
* Another convention is to name every key (primary or foreign) with an \_id at the end
  + If you’re using this convention where things that point at each other also have the same name, there is a variant of INNER JOIN where instead of ON you say USING
  + USING takes a list of things so it’s in () and you give it a list of names
  + It automatically does all the joins between pairs of names you’ve given it in the list



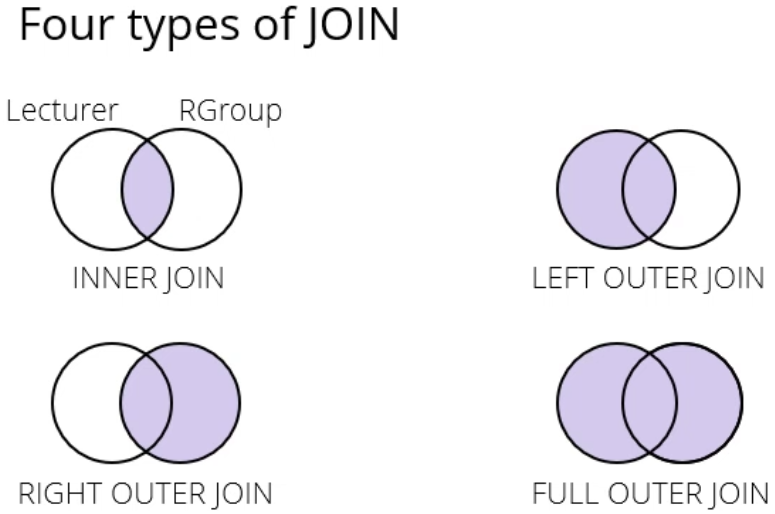
* + You can take this further and ask your database to automatically join any columns it finds with the same name
  + If you do that, you get a NATURAL JOIN
  + In practice, means join all combinations of columns that have the same name



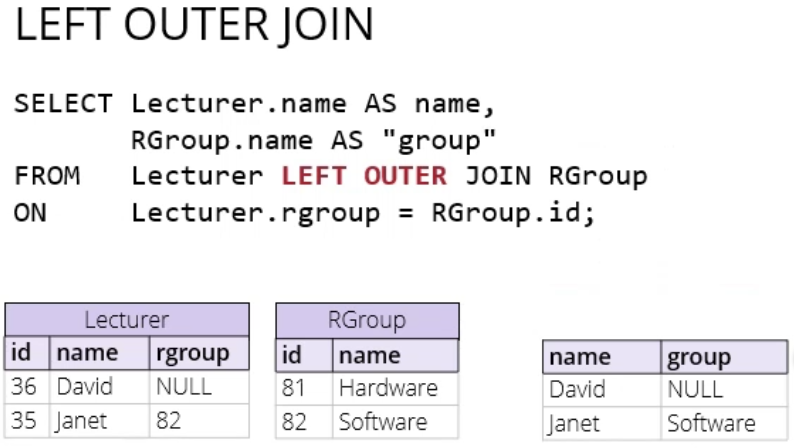
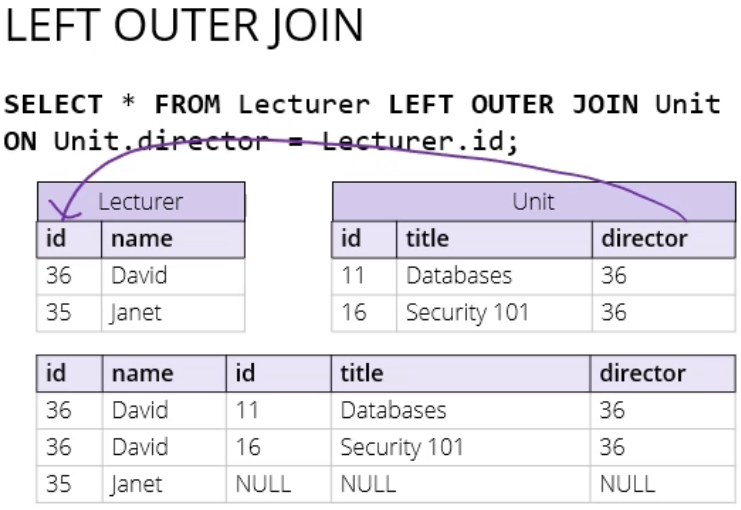
* Another advantage of NATURAL JOIN: columns that it’s joined only appear once or twice in the output



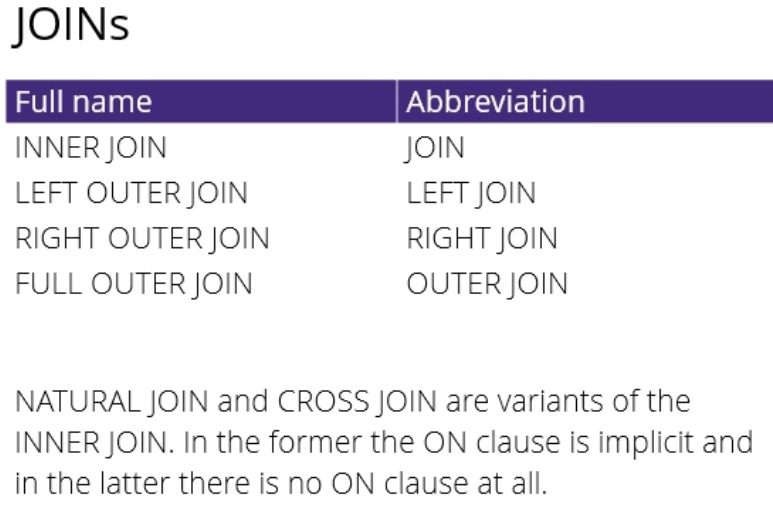
* Each group has at least one member, some lecturers are in research groups and some are not
* For the purpose of making a department website, we would like a list of lecturers and the groups they’re in
* First column of lecturers called name
* Second column called group (as name is already taken) (putting group in double quotes as it is an SQL keyword – using double quotes because it’s a column)
* Need to link it up so each group appears next to the lecturer correctly, so we do an INNER JOIN of Lecturer on RGroup ON Lecturer.rgroup (foreign key in the lecturer table) match up against the primary key in the group table
* Problem now is that if you have a lecturer that isn’t in a group, they won’t appear in this table at all
* There are more types of JOIN to allow for this



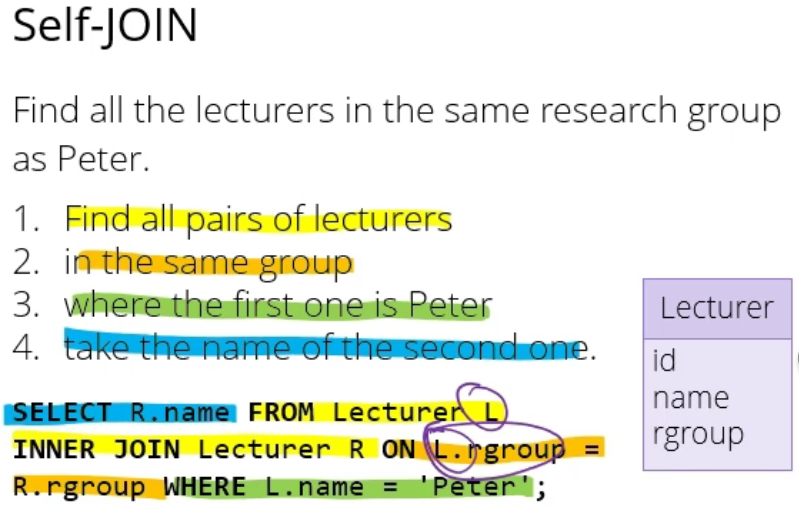
* If you do an INNER JOIN, then you will only get the lecturers that are connected to an RGroup and RGroups that have lecturers in them
* LEFT OUTER JOIN also includes the things on the left (what we want for our lecturers and groups scenario)

* Third meaning of NULL = you did an outer join, and these fields were filled with NULL because this row didn’t join on anything



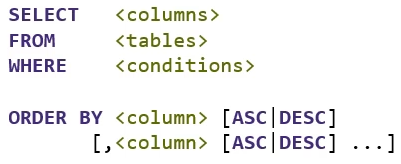
* NATURAL JOIN = inner join of all combination of all columns with the same name across tables
* CROSS JOIN = same thing as if you wrote FROM table1, table2 – does possible combinations ignoring relationships
* New scenario:
* Find all pairs of lecturers, then filter these lecturers so we only get those who are in the same group as each other
* Then out of these pairs, we only want the ones where the first one is Peter



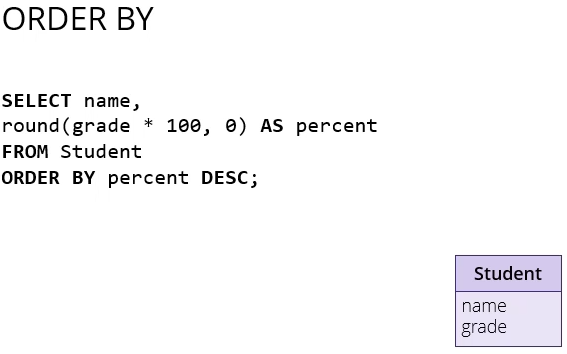
* So we now have a list of pairs where the lecturer on the left is Peter, and the lecturer on the right is in the same group as Peter
* Doesn’t matter if you do steps 2 and 3 in a different order – will be the same
* Then we just take the name of the second lecturer/the one on the right, and that gives you the lecturers in the same group as Peter
* You don’t read these queries in the same way that you write them, and the database doesn’t do this either
* FROM Lecturer L INNER JOIN Lecturer R gives you all possible combinations of two lecturers left and right
* Could do step 2 in a WHERE clause, but since we’re doing a join we may as well us ON
  + So ON L.rgroup = R.rgroup
  + Group of left gives you group of right
* Step 3 = WHERE L.name = ‘Peter’
* Step 4 = SELECT R.name (select name of the one on the right)
* In a Self-JOIN not actually using any foreign key relationships, database doesn’t mind
  + ON clause doesn’t have to refer to keys – can use anything
* Because we’re using two copies of the same table, we have to give both copies different aliases so that you can refer to them
  + E.g. alias first one to L then can refer to it as L.rgroup

**ORDER BY**

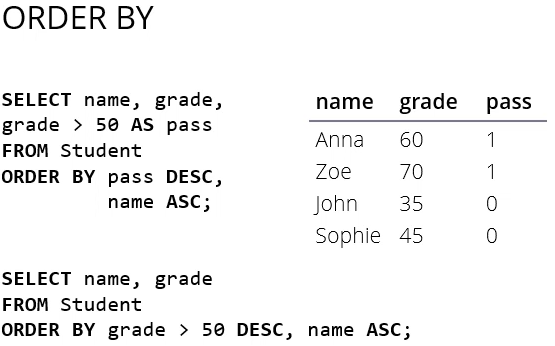
* When you do an SQL query, normally the database does the ordering of which way the results come back
* Typically if you’re just selecting from a single table, it will be something like the order in which they’re stored on disk in some kind of binary tree structure or something



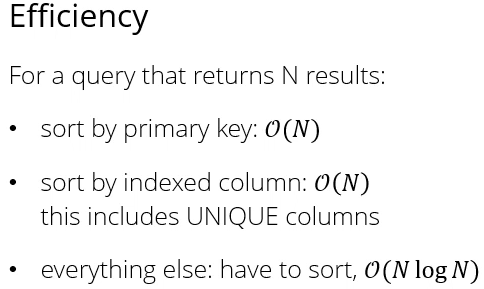
* If you want them in a particular order, you can do that by putting ORDER BY on the end of your SELECT statement
  + It takes a comma separated list of columns, and it will order them by the first
  + After each column name you can stick ASC or DESC for ascending or descending (ascending is default order)
* You can’t just order by columns, you can order by anything that produces a column



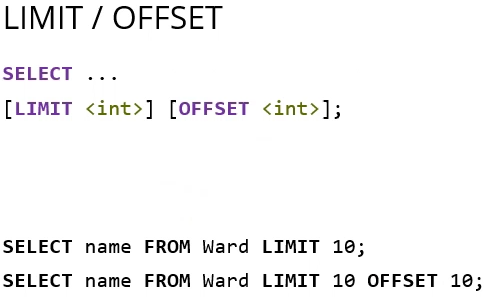
* So if you have a student table with names and grades, instead of just saying ORDER BY grade, if you want to round the grades to integers or percentages, you could say round(grade \* 100, 0), call that percent
  + Then ORDER BY percent descending
  + So the database will sort it by the percent column



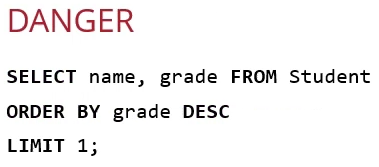
* Say you want a table of all the students above the pass mark of 50, you want to first show all the students who have passed in alphabetical order, then all the students who haven’t passed also in alphabetical order
* First make a column called pass and put true or false in there, or NULL if grade happens to be null
* Then order by pass descending so that the ones with 1 come first, then name ascending
* You can even do this if the pass column doesn’t appear in your table
* So first select all the names and grade (column) from Student table, then order by whether the grade is greater than 50 or not, even if that’s not a column in your table
* Always has a cost, but the database tries to make it as cheap as possible



* If you’re ordering something by a primary key, then that is typically related to how things are stored internally in the database, so you don’t have to pay the cost for a sort – the database can do that for you
  + Main cost is just printing or returning end results
* If you want to sort by columns that are indexed, its also big O
  + Indexes are a way to let your database work very quickly with particular column values – both searching for them and sorting them
  + One example that will always give you an index is if you have a UNIQUE constraint on a column, internally that’s implemented as an index
  + Whenever you try to insert or edit a value, the database will use the index to binary search to see if you’re about to produce a collision or not
  + So if you want to sort by it, you just use the index
* Anything else, the database actually has to sort it, so it will put all the results in a temporary table then sort that (in memory if it can) and will be N log N



* Another thing you can stick on the end of your query even after ORDER BY is LIMIT/OFFSET
* They are called “window functions”
  + They’re an example of functions that restrict which part of the output you see
* Limit says “after you’ve produced this many rows, stop” so you can use to peak in a table
* If you want to display the data in pages, you want OFFSET
* So in the two lines above, the first line gives you rows 1-10 and the second gives you 11-20



* Suppose you want to find the top student in a class, don’t use LIMIT 1 to use the top entry unless you’re guaranteed there’s going to be only one of them (e.g. might be 2 students with the same grade)

