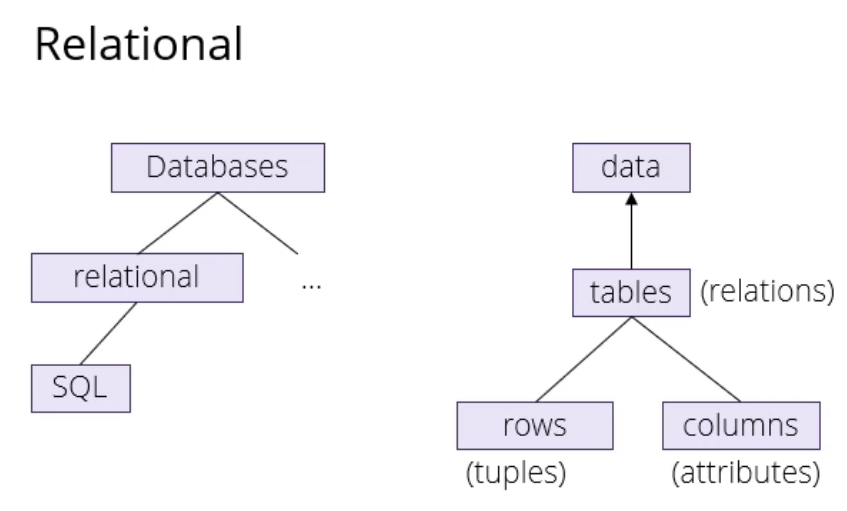
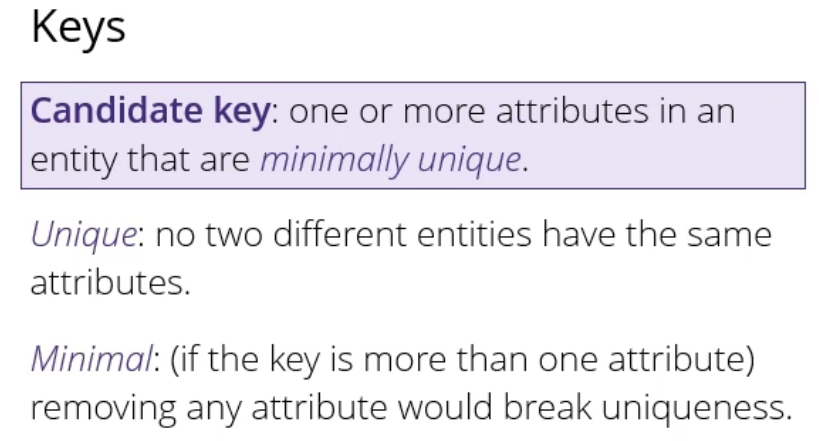
**Relational Modelling**

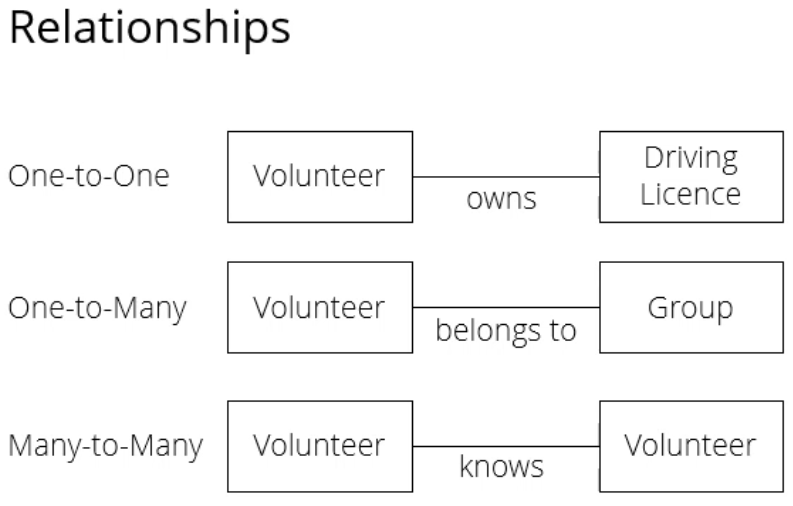
* Biggest distinction between relational and non-relational databases



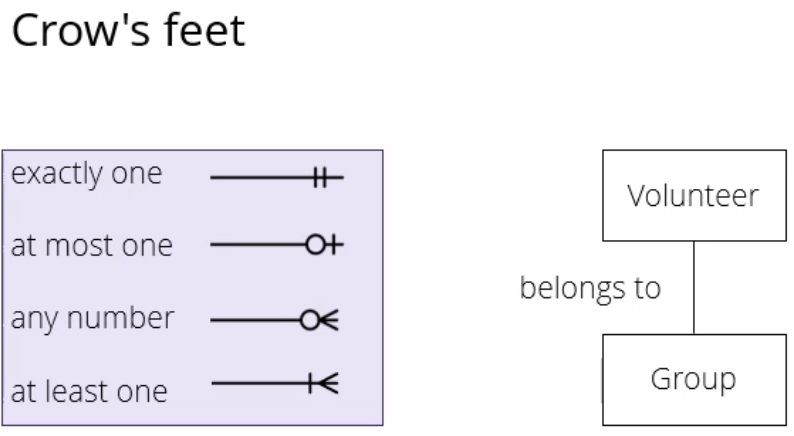
* Relational = probably using SQL to access it
* Non-relational = other languages grouped together under no-SQL
* In relational database, data stored in tables (relations) with rows (tuple) and columns (attributes)
* Point of a database is to capture information about the world
* Each row represents a fact
* Tuple = list of defined length
* Entity Relationship = ER modelling
* Relational databases came before OOP like Java, now when you’re designing a model you start with an OOP model and then represent that in the database – previously it was the other way around
  + Intersection between OOP and designing a database is how to you model facts about the world that you want to use in your application
  + For that, we use ER modelling
* Big difference between this and Java = Java classes are designed to represent data in memory, databases are designed to represent data on disk
* In a Java model you might have a Person class and an address class, and they might have attributes in the sense that a Person has an Address
  + In memory you store them as objects which are connected by things you aren’t meant to call pointers but that’s what they are
  + So you have a Person object with a way of referencing another Person object, and behind the scenes that will be implemented by some kind of pointers (which would be explicit in C)
* In databases you have tables, and tables have rows, and they’re completely disconnected in the way that when you have one Person there’s no automatic way to jump to the related Address – you have to tell the database to do that manually
* Database is designed to store all your data, even if it’s too much to fit in memory all at once
* Whereas your Java program assumes that the amount you want to work with (your classes and objects you have active at the moment) will all be in memory
* Design trade off how you represent these things – both make sense depending on what you do with data at the time
* Reason is that there’s a speed and size trade-off depending on how much data you want to store, and where you want to store it
* Generally speaking, registers, caches, RAM etc. (dynamic memory) will be comparatively small but fairly fast
  + Anything that fits inside the memory of a single machine is small data
  + Counterpart is big data – too big to fit in a single machine so need more than one machine or even a whole data centre to store it
* In between, you have things that fit on the hard disk of a single machine – not in memory (medium data)
  + Relational databases work here – too big to fit in RAM, but can fit on the disk of a single machine
  + Are more and more expanding capability into big data (e.g. cloud databases), but originally designed for medium data
* If your data fits in RAM, that’s where it belongs – you don’t need a relational database – and your process will be much faster
* Different notations to represent how you design a database model – here using Crow’s foot notation
* UML = Unified Modelling Language – might encounter in industry
  + Complicated and long (Crow’s foot much simpler)
* Start by talking to the people the database is for and finding out what they want from the model
* Then go over this information and pick out the most important nouns (entities)
* Entities are the things that will later become tables in our database
* Notation for entity = in a box
* Underneath entity noun you put attributes
  + Attributes = things about each type of noun you need to know
  + E.g. what kind of data do we need to collect about volunteers?
* Each box turns into a table, and each attribute turns into a column in that table
* Key = attribute/group of attributes we use to refer to something
  + E.g. name for a person



* Unique for example could be a phone number – no 2 people have the same attribute
* For each table, you have to choose one candidate key to become the primary key
* Point of primary key = way to refer to instances of this type in your database
* Surrogate key = ID column that has no meaning to your users
* Composite key = key with more than one attribute
* When you’re choosing a primary key, there are various forms of good and bad choices:
  + Good choices = things that are guaranteed to be unique
    - E.g. email, phone number, username, ID column
  + Bad choices = anything involving human names (much more complicated than you think)
* \* next to primary (key) in notation
* If entities are nouns, relationships are verbs
  + Connect boxes with lines and use verbs to describe them

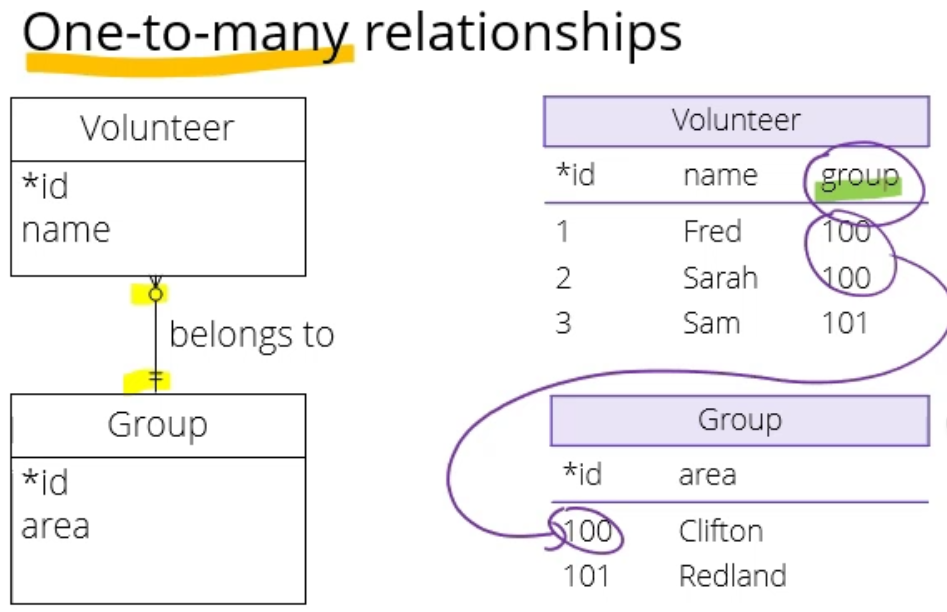
* Lines at the moment aren’t directed – not clear as to whether volunteer supports client – but usually clear from context
* Different kinds of relationships - 3 main kinds
  + One-to-one: only one entity on each side participates in the relationship
    - Each volunteer owns at most one driving license, each driving license is owned by at most volunteer
  + One-to-many: there can be many instances on one side participating in the relationship, but only one on the other side
    - E.g. each volunteer belongs to exactly one group, each group belongs to many volunteers
  + Many-to-many:
    - E.g. social network – each volunteer may know other volunteers etc.



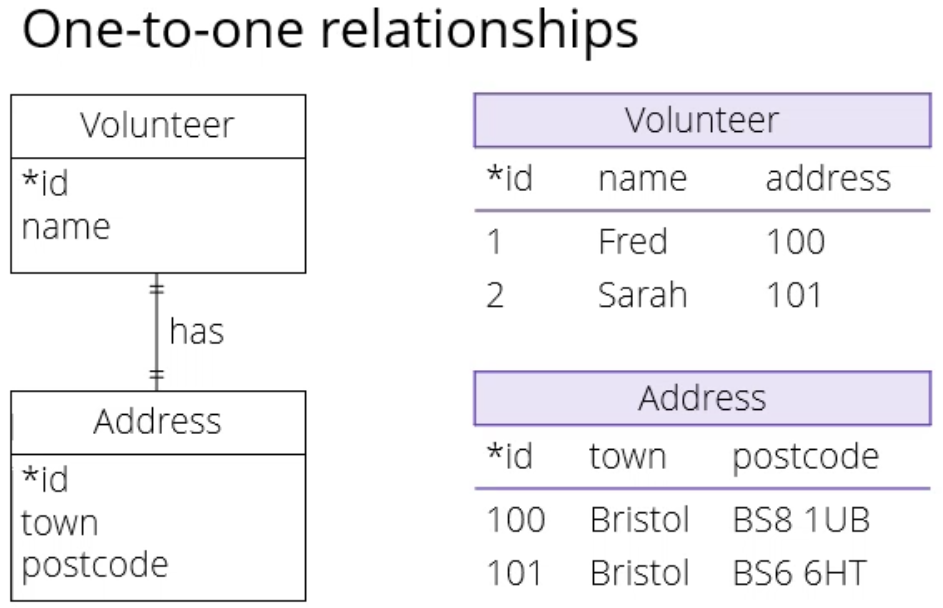
* Sometimes, you might want to have a separate table to model instances of the relationship itself
  + E.g. say a volunteer does tasks for a client, you might want to keep track of these tasks separately to see how long they took, what day they were carried out etc.
  + Called an associative entity
* When you make an associative entity, the feet split and turn around

**Turning diagrams into tables**

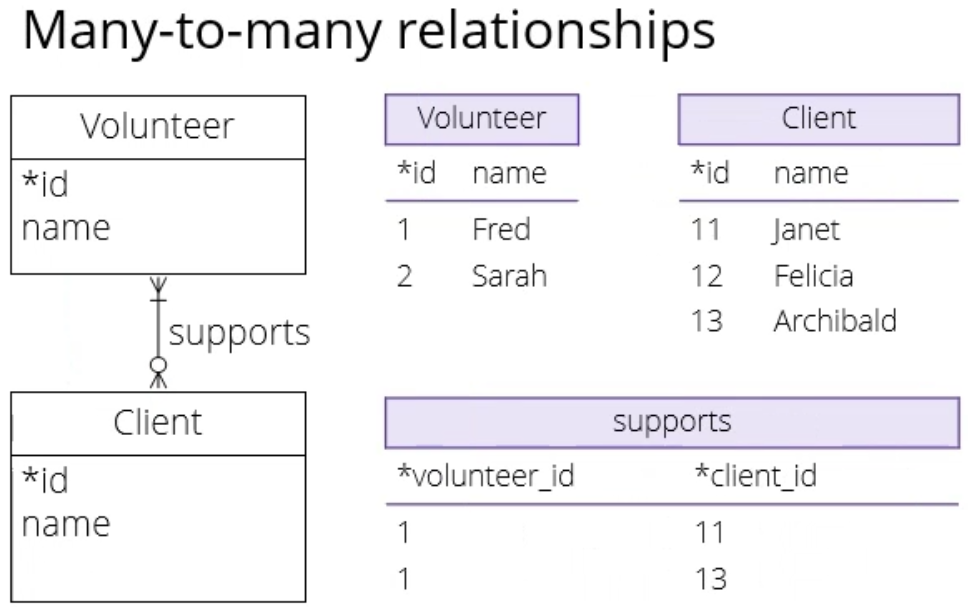
* Every entity in model will become table in database, every attribute will become a column
* Because every volunteer belongs to one group, we can include that as an attribute on the volunteer table
* The way you refer to an entity anywhere else in the database is by including a copy of the primary key of that entity
* The column of the key that refers to the primary key in another table, like group, is called a “foreign key”
  + Foreign key = a column in a table whose values must match values of a column in some other table
  + Foreign key constraints enforce referential integrity, which essentially say that if column value A refers to column value B, then column value B must exist
* In this example, you can’t put a foreign key on the group table because each group could have many members, so you need either lots of columns (even that would limit how many members a group could have), or you would have to do something like put a comma separated list of things in the group table
* If a volunteer belongs to exactly one group, then that’s the table where the foreign key goes



* Applying the same principles to one-to-one relationships, here’s an example where each volunteer has exactly one address and each address belongs to exactly one volunteer



* Design choice as to whether you want to make two separate tables or just one big one and include all info in it
* If you want two separate tables, it’s not clear as to where the foreign key has to go, and it could go either side depending on what you’re modelling
* If you have a belonging association, in the sense that an address belongs to a volunteer, typically you would put the foreign key on the volunteer side, but it would work equally well the other way around
* Mandatory & optional relationships
  + If you have a mandatory relationship, in the sense that a volunteer has to have exactly one address, the address column in the volunteer table would be declared not only as a foreign key, but also as not null, which enforces that every person has exactly one address
  + If a person were allowed to have 0 or 1 addresses, you would have the column in the same place but you would make the address column “nullable” so if someone doesn’t have an address, or hasn’t entered one yet, you just store null in that field



* In this scenario, each volunteer can support many clients and each client can be supported by many volunteers
* In this case, you can’t put the foreign key on the volunteer table because each volunteer can support many clients
* For the same reason it can’t go on the client table
* In this instance, you make a third table called the join table
  + Conceptually the same as an associative entity in modelling
  + Has the purpose of connecting to other entities
  + Each entry in this join table is going to refer to one association
* Any volunteer could be connected to any client etc.
* Could ask questions like which volunteers support three people, that would just be counting rows in the join table
* Because you have the foreign keys in the join table, there’s no way to enforce, for example, that every client is supported by at least one volunteer on the database level
  + In this case, you would have to enforce it in the application level
  + So when you create a client in the application, the application would have to make sure they are assigned at least one volunteer
* The database manages the associations, which you can think of as a graph between volunteers and clients, by rows in a join table