**Lecture 134**

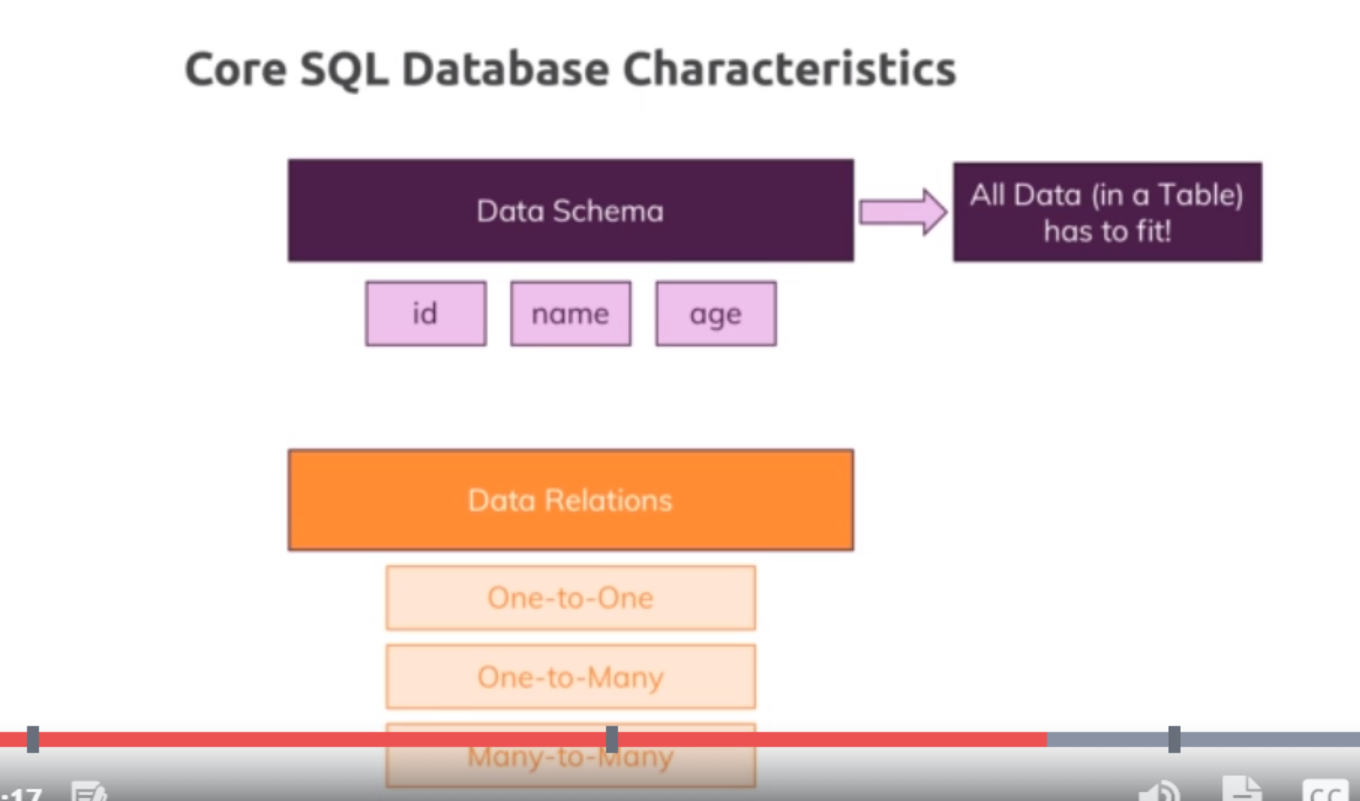
**Module Introduction**

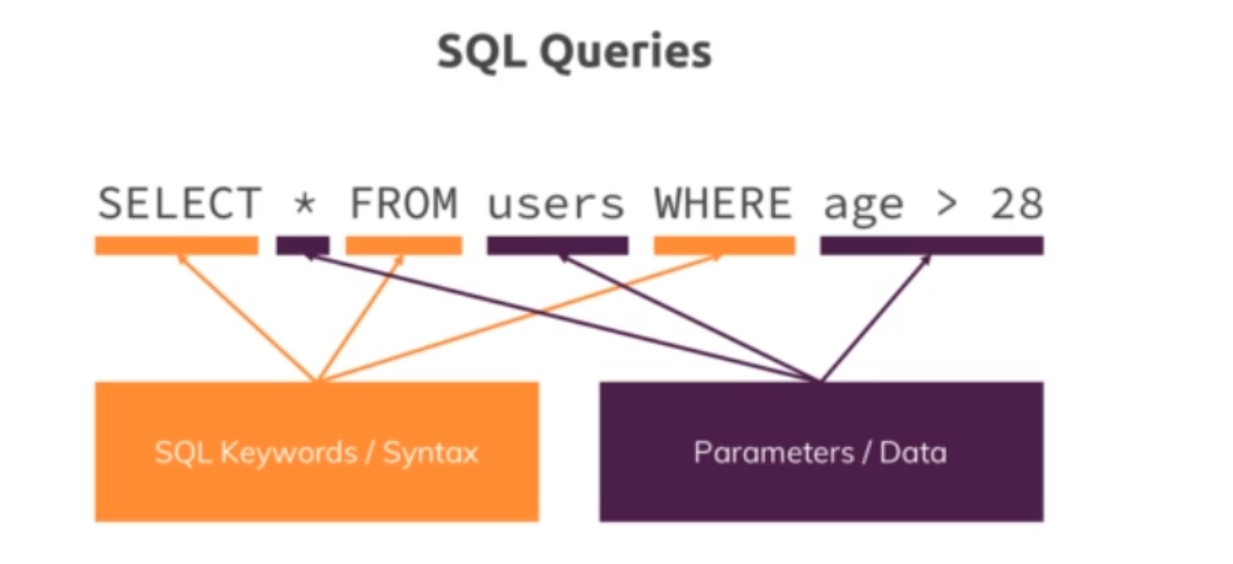
* Storing and retrieving data is slow when there is large amount of data. For that we have to use database.

**Lecture 135**

**Choosing a Database**

* Database is quicker than a file

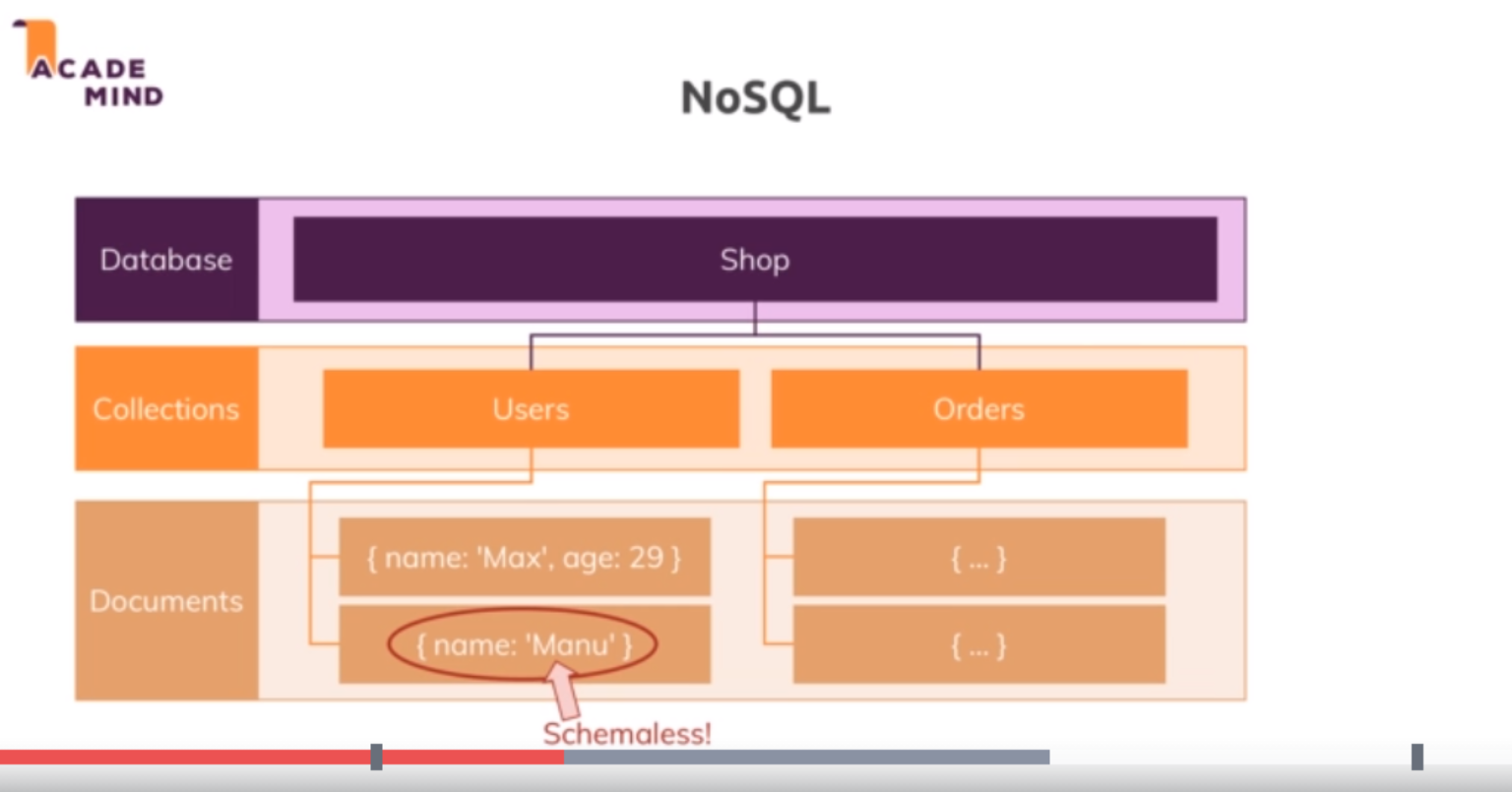


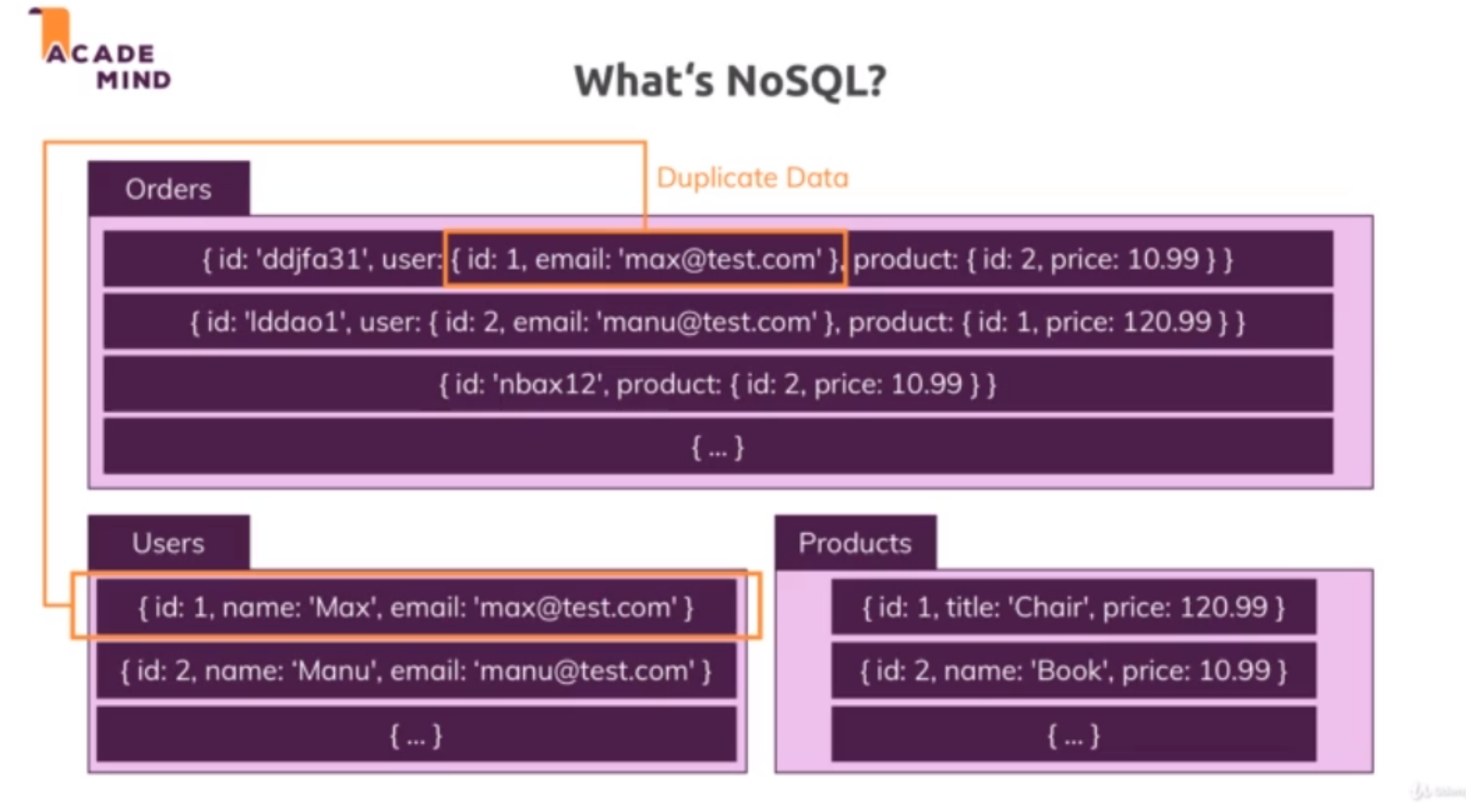
* . Our goal always is to store data and make it easily available or accessible so that we have an easy way of accessing our data and not just easy from a code perspective but of course efficient, it should be fast. That is why we use a database, it's simply quicker than accessing a file especially as the data in there grows, it also helps us with things like we don't have to read the entire file to just find one piece of information
* the core SQL database characteristics are that we have a strong data schema so that for each table, we clearly define how the data in there should look like, so which fields do we have, which type of data does each field store, is it a number, is it string, is it a text, is it a boolean? So that we have this strongly or strictly defined schema and all data in the table has to fit the schema for this table, this is really important, so this schema, this definition of how the data has to look like is one core thing in a SQL database. We also have relations between our data, that is another core characteristic of SQL based database, we relate our different tables with basically three important kinds of relations, one to one, one to many or many to many, this simply means that we can have two tables where each record fits one other record, a record might fit multiple other records or multiple records in table A can fit multiple records in table B.
* 

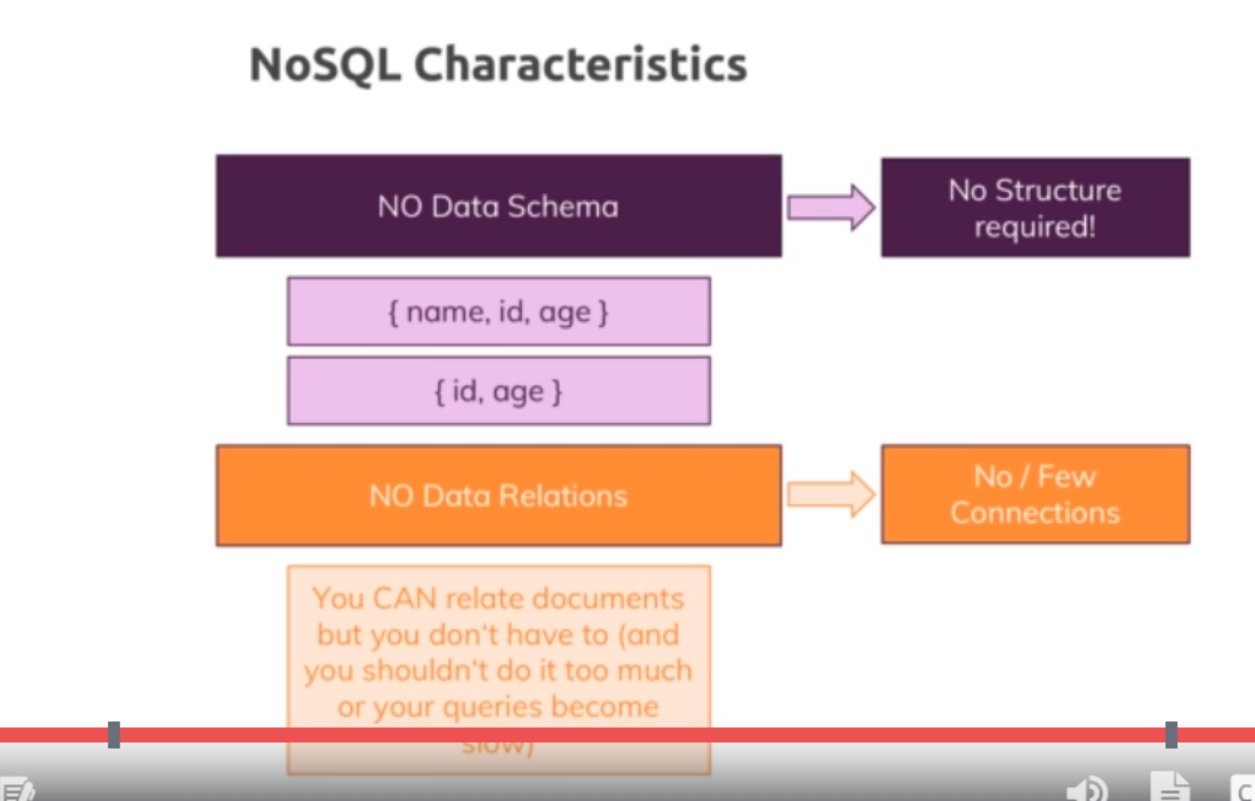
So this is a so-called query, we've got a couple of keywords there which are making up that SQL language, so the structured query language simply has these keywords and then we insert some parameters or some data we connect with these keywords, this is how SQL works.

**Lecture 136**

**NoSQL Introduction**



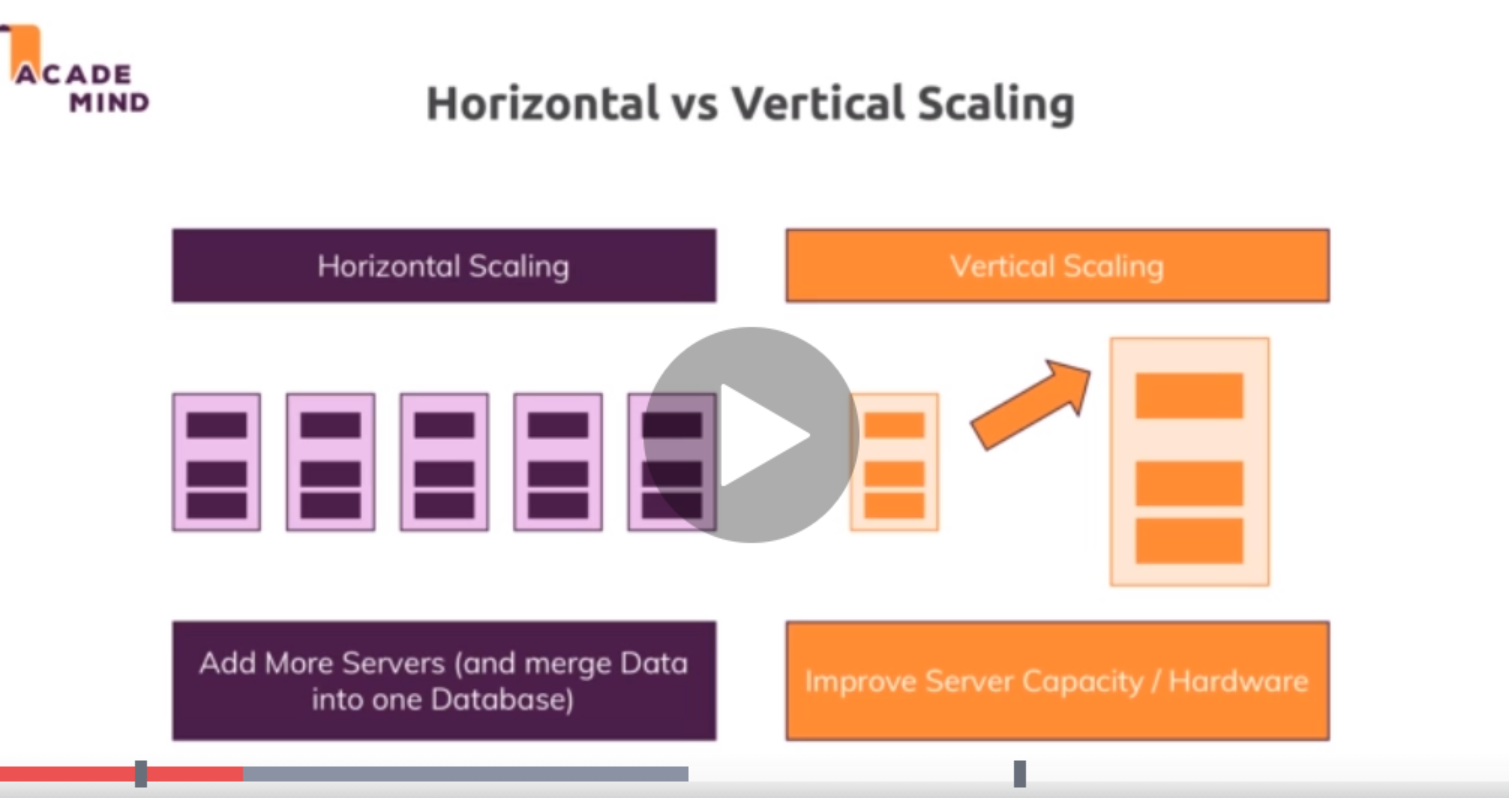
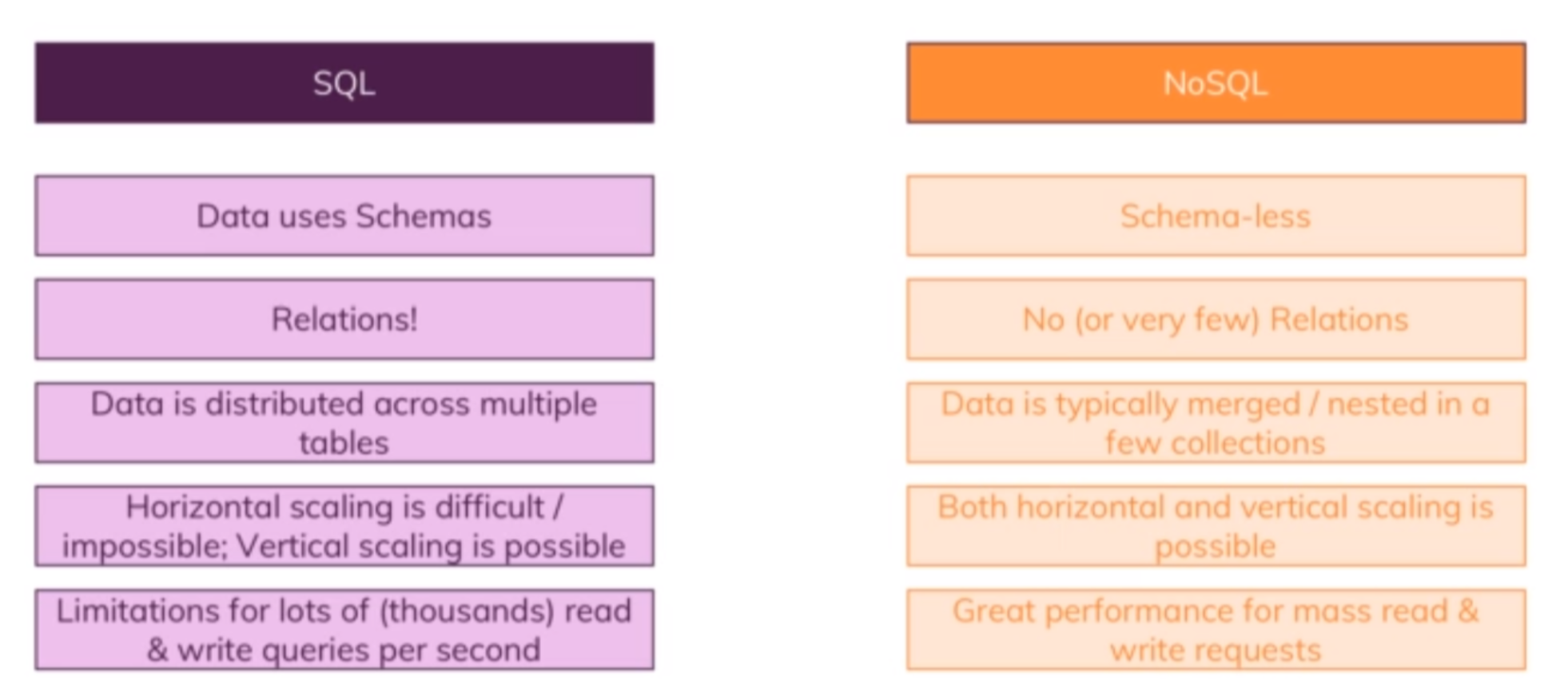




* in NoSQL we can still have a database.
* Now that's the same for SQL by the way, there we also have databases
* Now in NoSQL, tables are called collections but you can think of them as tables, so as the table equivalent but we call them collections in the NoSQL world.
* Now in a collection, we don't find records but so-called documents which look like this and since we're working with javascript in this course here, we of course can kind of see that this looks a bit like a javascript object, so documents are very close to how we describe data in Javascript.
* NoSQL doesn't have a strict schema. Here we got two documents in the same collection but the second document, Manuel here does not have an age and that is perfectly fine in NoSQL, you can store multiple documents with different structures in the same collection. Now often you of course still try to have kind of a similar structure but it's also not uncommon for some applications that you don't always have exactly the same fields available for the data you are storing in the database and that is ok in NoSQL, you can definitely store documents which are generally equal but where some fields might differ.
* One other thing is that in the NoSQL world, we got no real relations, instead we go for duplicate data. Now that simply means that if we have an orders collection here, we have a nested document, the user which also is stored as a separate document with more details maybe in the users collection and we don't connect that through some ID or behind the scenes setup relation, instead we simply duplicate data, to be precise the data we need in the orders collection here. That of course means that if that data changes, we have to update it in multiple places, if all these places need the latest update or the latest data change but that can be OK because this on the other hand gives us the huge advantage that if we ever retrieve data, we don't have to join multiple tables together which can lead to very long and difficult code and which can also impact performance, instead we can simply read the data from the orders collection and we probably got all the data we need to display on the orders page without having to reach out to other collections and therefore this can be done in a super fast way and that is one of the huge advantages of NoSQL, it can be very fast and efficient.
* Now we can relate documents in some way and this is possible and we will see how to work with connected data in the NoSQL module of this course but generally we have no or only a few connections and instead try to copy data and have a collection with documents that work on their own.
* we also got a difference between SQL and NoSQL regarding our scalability. So as our application grows and we need to store more add more data and access that data or work with it more frequently, we might need to scale our database servers and we can differentiate between horizontal and vertical scaling

**Lecture 137**

**Comparing SQL and NoSQL**

* <https://academind.com/tutorials/sql-vs-nosql/>
* 
* 
* Horizontal and vertical scaling are the two approaches we can use to scale our database. Now what do they mean? Well in horizontal scaling, we simply add more servers. and the advantage here of course is that we can do this infinitely. We can always buy new servers, be that on a cloud provider or in our own data center and connect them to our database and split our data across all these servers, of course this means that we also need some process that runs queries on all of them and merges them together intelligently, so this is generally something which is not that easy to do but this is of course a good way of scaling.
* Vertical scaling simply means that we make our existing server stronger by adding more CPU or memory or with something like that, especially with cloud providers, this is typically very easy, you simply choose another option from the dropdown, you pay more and you're done, the problem here is that you have some limit, you can't fit infinitely much CPU power into a single machine.
* So these are the two ways we can scale, how let's compare a SQL and NoSQL regarding that and in general. Now in general in SQL we use schemas, we also have relations, these are two core characteristics and data is typically distributed across many many tables which are then connected through relations. Now regarding the scaling, it's important that horizontal scaling often is very difficult or even impossible due to the way SQL works, you can of course add more servers but running them all on one shared data cloud so to say, one shared database is pretty difficult.
* Vertical scaling is easily possible, you can simply make your server stronger but adding more servers can be very hard or even impossible, so definitely not trivial. So this is a problem possibly if we have multiple or thousands of read and write queries per second, then maybe our SQL database especially if we do very complex joins between related tables can reach limits or can not be the best choice.
* NoSQL is schemaless and has only a few relations if at all, the data is typically not distributed across multiple collections but instead we work with merged or nested documents in an existing document, though we of course also have a couple of collections for the different features of our application typically. With NoSQL, horizontal scaling is easier, still something where you have to know what you do but there are cloud providers which do that for us so we don't have to know the ins and outs of that and in general, due to the way it works with less connections and so on, this is possible. And therefore we also get great performance for mass read and write requests and NoSQL can be very performant in an application with high throughput.
* Now this makes SQL look very bad but the full truth is that it always depends on the kind of data you are storing, if you are storing where the relations are really important and where you want to have a split up across tables and where you want to have strong schemas, SQL can be perfect, also not every part of your data is accessed multiple times per second. You can have parts of your application where you manage general data, let's say user data which does not change that often and therefore, SQL might be very good there. Other parts of the application, let's say orders or shopping carts that do change frequently could be stored with NoSQL and there, the relations might also not be that important because you can always put all the information that belongs to a shopping cart or to an order in one single document and even if you do for example store some user data there, you might not need to touch that document just because the user change his photo because you probably didn't store that along with the order anyways.

**Lecture 138**

**Setting up MySQL**

* <https://dev.mysql.com/doc/mysql-getting-started/en/#mysql-getting-started-installing>

**Lecture 139**

**Connecting our App to the SQL Database**

* Refer code 01-connecting-our-app
* Npm install l—save mysql2
* now there are two ways of connecting with a SQL database. One is that we set up one connection which we can then use to run queries and we should always close the connection once we're done with a query and the downside is that we need to re-execute the code to create the connection for every new query and there will be a lot of queries because we fetch data, we write data, we delete data, creating new connections all the time quickly becomes very inefficient both in our code and also regarding the connection to the database which is established and the performance this may cost.
* So a better way is to create a so-called connection pool and by the way, you can learn way more about this package, its options regarding how to set up connections and so on in the official docs for this tool, for this package and you find a link to those docs in the last lecture of this module. So to set up such a pool, I'll create a new constant pool, the name is up to you and I'll use that MySQL object and there I will call create pool and there you also see that create connection we could use.
* Now I don't want a single connection but a pool of connections which will allow us to always reach out to it whenever we have a query to run and then we get a new connection from that pool which manages multiple connections so that we can run multiple queries simultaneously because each query needs its own connection and once the query is done, the connection will be handed back into the pool and it's available again for a new query and the pool can then be finished when our application shuts down.

**Lecture 140**

**Basic SQL and Creating a Table**

* Refer code 01-connecting-our-app

**Lecture 141**

**Retrieving Data**

* Refer code 01-connecting-our-app

**Lecture 142**

**Fetching Products**

* Refer code 02-fetching-products-time-to-practice

**Lecture 144**

**Inserting Data into the database**

* Refer code 03-inserting-data.
* Now we're not done, this only defines where do we want to insert something, the what is missing, we now need the values keyword followed by brackets with the values. Now to safely insert values and not face the issue of SQL injection which is an attack pattern where users can insert special data into your input fields in your webpage that runs as SQL queries, we should use an approach where we just use question marks, one for each of the fields we insert data into separated with commas and then there is a second argument we pass to execute with the values that will be injected instead of these question marks, so the order of the elements we add here to this array is the order of arguments here. And we don't do this on our own because this MySQL package we're using here will then safely escape our input values to basically parse it for a hidden SQL commands and remove them, so now this is an extra security layer.

**Lecture 145**

**Fetching a single product with where condition**

* Refer code 04-fetching-a-single-product