Week 5
Video Transcripts

Video 1(12.44) Basics of Databases (Part 1)

Today, we look at the basics of databases. This is not intended to be a substitute for a database class, but we need some basic knowledge of what the structure of a database looks like, and how we can use SQL to get data from a database. So, let's take a look at the basics of databases, and the first thing to understand is that in a computer program, the data that we use in the program is transient. When the program gets turned off or the computer gets switched off or the program ends or crashes, or whatever, all the data, that the program is using is lost, because it's transient. It exists only for that instant of time that the program is running or the computer is running, and then it's gone. But typically, when we have large storage of—large chunks of data, that we need to use for analysis, we need to store the data in a persistent fashion. That means that the data should be available forever, so to speak. So, the...the key understanding is that we–what we don't want is we or what we want to be able to do is to get data from sources that are persistent. And typically, a persistent database is one, that is—sits on a computer or a server or something, and has an organized collection of data. So, the key here is 'organized' and 'data'. It's data that's organized in some fashion, and there're many different kinds of databases, but the key is they're organized, and they sit permanently on the computer, and they are persistent. So, you can turn the computer off, turn it back on, and it's still over there, that's the idea there, right! Or, if it's on a server, you turn the server off and on, and it's still over there. And, there are many databases like this. And typically, the two types of databases that we are–we usually using on data analytics are either relational database or something called a NoSQL database. So, let's take a look at what these things are, we will in a minute, but these are the two basic forms of databases, that we want to consider. Who uses databases? Well, almost everyone does. If you take any application, you're going to be using a database. If you go on the web, you go to Amazon.com and you type in your favorite thing that you're searching for, perhaps the latest drone or whatever, then what Amazon does is it goes to its database, sends a query saying, "Find drones under 500 dollars" or whatever your query is, and returns a long list of drones, the things that satisfy that condition, and shows them to you. So, Amazon has a permanent repository of data that contains all the products that are available for sale, and you're essentially, when you send a request over the web, you're querying that database, that's what you're doing. So, almost everyone uses databases, and we–when we're interacting on the web, we're actually using a database whether we realize it or not. Well, most of the time anyway.

The two kinds of databases that are of interest are relational databases and NoSQL databases. And, relational databases are very straightforward kind of things, where the analogy with...with tables is very clear. In relational database, data is stored in tables, two-dimensional tables. So, they look exactly like this with column '1', column '2' etc, and row '1', row '2'etc. That's what a table looks like. And, that's what we see inside a relational database. The idea in a relational database is that if you have a database, then the tables are logically connected. That means that you don't have tables that are dissociated from everything else in the database. Typically, they're connected through a column. So, we might have a column here, like 'C1' for example, that is a key column, and that column is used in some other column as well, and then we can say row ‘1’is–row ‘1’contains data for whatever table this is, and we can find some other table that has the same value of 'C1', and then connect the two up. So, the idea here is that if you think of a relational database, you really have 'T1', 'T2', 'T3', and these are all linked somehow, okay, may...may not be all of them link to each other, for example, perhaps 'T2' and 'T3' are not linked, but you don't really want something that is like 'T4' over here, that is not connected to anything at all. Within the table itself, the columns are attribute values. So, it's like an Excel table. You can think of having multiple excel tables in a relational database, and the columns are Social Security Number, Name, Phone Number, and stuff like that. So, what we end up having is rows and columns, and we finally want to get data from this. So, we use a language called SQL for information retrieval, and that's really the key thing that you want to learn out of all this. How do we write SQL query is to get data from a relational database. And, the goal in a relational database is to do two things, to make sure that the database has very low redundancy, and is maximally consistent. Redundancy means that if in a database you have the same data repeated twice, or three times, or ten times, or 15 times, then that's redundant. So, if you have, for example— let's say, your name is John Smith. And you—in table one, there is 'John Smith', a row that corresponds to 'John Smith' and say your address, right! So, let's say you live in New York. And then, table two also has the same thing, 'John Smith', 'NY', and maybe some other information attached to that. This is redundancy. We have the same piece of information, 'NY', 'NY', and 'John Smith' and 'John Smith' repeated in the database, That's less than ideal because what'll happen if you move from New York to Paris, right! If you move to Paris, we have to find every instance of your address in a database and change it, and that's not so good. You forget it in one place, then what you're going to have is– like if you go here and we say, "Hey, we change 'NY' to say 'France'." Okay! And now, if I ask the database, "What is—where does John Smith live?" If the database goes to 'T1', and gets the address from there, it's going to say 'France'. And, if it goes to 'T2', and gets address from there, it's going to say 'New York'. And, what that means is that we have a database that is no longer consistent. Consistency means that the database should give you the same answer to the same query, irrespective of how that query is executed. In the case of our example here, if the query was executed by looking at 'T1', we get one answer, if the query was executed by looking at 'T2', we get a different answer. So, that's inconsistent. So, redundancy leads to inconsistencies and of course, to wasted space and all kinds of other things as well, not so nice things as well. And, the goal in a relational database is to make sure that we have minimal redundancy and maximal consistency in the database. The second kind of database, called a NoSQL database—and there's really not a single type, a NoSQL database is essentially a database that doesn't use sequel for SQL, that is—SQL, sequel are interchangeable terms, for accessing data, in other words, it's not a Relational database. So, NoSQL databases are typically used for very large data sets on the and–often on the Internet. And, the idea there is that you have a very large data set, and let's say you have a row of information about a person, that contains every possible piece of information known about that person, you know, what their web habits are, what they spend money on, all their credit card numbers, their addresses, their spouse or previous spouses, their children, their children's ages, their children's schools, you know, all kinds of stuff. So, you have all this data. In a relational database, typically that data will be spread out over multiple tables, and if you want to access all that information in one step, then you have to go to multiples tables and collect that information and then produce the result, and that can be slow. If, however, you know that your application is going to actually get all that information in one step— it's going to require that information, the complete information in one step, then maybe it makes sense to rather than storing it in a NoSQL database to store it in a–so sorry, rather than storing it in a relational database, in multiple tables, to store it in a single record of some form or the other, right!

A single data element. And then, you say, "Get me the data for John Smith", and the entire data there his history, the kids' history, the spouses, ex-spouses, parents, grandparents, grandchildren, whatever, can show up in one shot. So, that's the idea with this–with the...with the NoSQL database. Often that record is stored in the JSON format. We've already seen JSON formats. So, often that the format is either JSON, or something similar to that, but the idea here is that whatever makes the most amount of sense in this–in terms of getting the data as fast as possible, in other words, you want low latency. So, you have to know beforehand how you're going to query the database. If you don't know that, if you want generalized query, relational databases are better. But, if you have a specific form of query, then you need to think about, "Okay! I know this is what I want to get. So, I might...might store it in the format that brings it out very, very quickly." The second nice thing about it is it's very scalable, because if you want to add more records to the database, you just–you have–you can think of it as having one giant table with all the information. So, some new person comes along and you want to add all the information about them, you just tag it onto the bottom, and you can keep tagging on...tagging on at the bottom. And, in fact, if you run out of space, you put it on the cloud. You just get another machine, another machine, another machine, another disk, whatever, and you can go on forever. So, it's very scalable. And, the downside of course is that there's lots of redundancy. So, the same information may be stored in multiple places.

For example, if we have John Smith's entire history in one record, and we have– that includes his children's history, and if you have let's say–his...his kid is Jane Smith. So, Jane Smith's record is in a different record–information is in a different record, and we have all that information there is duplicated. We already have it with John Smith, but we need to duplicate with Jane Smith because if our query says, "Get me information on Jane Smith.", we don't want to have to go, and first find 'John Smith', and then get her data, That's the idea. So, there's a lot of redundancy built inside a NoSQL database, which is initial because we're talking about very large data sets. But, it's also less of an issue because computing power, the...the price of...of storage, all these things are actually getting lower as time goes by. So, since they're getting lower redundancy at least in storage terms is not a big deal. We still have the issue of inconsistencies, but we have to live with that, okay! So like I said, typically they're stored on a cloud, because that's where you have the scalability, and it doesn't use SQL, that's why they call it NoSQL. There may be some NoSQL databases that use SQL, I don't know of any. But the idea NoSQL really says, doesn't use SQL. And there's some very nice examples out there. There's MongoDB, which is a proprietary commercial database. There's Google BigTable that's by Google...Google, and you have to pay for using it, above a certain amount. There's Sparksee. There's Amazon DynamoDB. So, they are...they are...they are the possible venders or users of data, or...or possible database servers you can get that are NoSQL databases. We are not going to look at NoSQL databases in our class. And the easiest of these would be, if you really want to look it up, it's to look up MongoDB. MongoDB is a very simple database. It also uses a JSON format. We're already...already familiar with that. So, it's something really familiar for us. So, that should be really easy to use. Or, you can use Google BigTable or Amazon DynamoDB.

But just note that if you—beyond a certain threshold, a fairly small threshold, you're going to end up paying for it, which is not a very great thing. But MongoDB, you can get a free license for certain uses. So, it's not a bad thing to start with. So that's...that's up to you, if you want to do that. We're not going to do it in this class. We don't have the time for that. What we're going to do is we're going to focus on relational databases.

Video 2(10.09) Basics of Databases (Part 2)

So, let's take a look at relational databases. When you think of relational databases, we can think of three different aspects of our data. So, typically what happens is that we have something called a data model, and the data model is the structure of our database, the entities, and the relationships. The idea behind relational databases, like I said earlier, is to somehow come to a point, where your database has minimal redundancy and maximal consistency. Right! And, I say minimal and maximal because there— you can build a perfectly un—non-redundant database, I assume, but in reality, you know, you will sacrifice some querying speed for every gain that you make in reducing redundancy or increasing consistency. So, there's a tradeoff, right! So—but the goal is to minimize that. So, the first step really is to look at your problem, look at the data world that you're looking at, and build a data model, an abstract model that has nothing, to do with computers or nothing to do with storage, but just a model that represents the structure of your data. And, the idea in that is that you build two kinds of things. You outline what are the main entities, and how they are related. The other thing we talked about in relational databases is we said that the tables are...are related, that you have a linked, related set of tables, right! So, we want to get relationships between different entities. We'll take a look at that in a minute, what that means. The next thing is that once you have a data model, you convert that into what's called a relational model. And, the relational model is nothing other than the set of tables that your databases should contain. So, you have a data model that is an abstraction. You have a relational model that is the structure of tables that you want to finally store on your computer. And, before you actually go and store it, you do something, called normalization. And the process of normalization, which is a very well-structured process, so to speak, but is to take your data, your set of tables that we have, take our set of tables, our relations—tables and relations are interchangeable terms here, and reorganize them so that we have low redundancy and high consistency, right! So, we follow this process of normalization to do that. So, let's take a look at what all this stuff means. So, the...the data model is– there are many ways of constructing a data model, and we'll look at one approach, which is called an Entity-Relationship model. And, the Entity-Relationship model is actually a kind of very nice, a very simple, way of looking at your data. And, the idea is very simple. We are going to build a conceptual data model. So, it's a bunch of concepts.

That data model is going to capture semantic information about the world being modeled. Right! That's the goal here. And, it's going to do that through entities and the relationships between the entities. That's our idea here. So, what does that give us? Well, it tells us that the main components of our Entity-

Relationship model are two of them. They are entities and relationships. So, let's take an example. Let's say we are modelling a university system. In a university system, we have many, many things that we might want to....want to...want to model, but some of them are students. There are students in the

university. So, we have an entity, a...a real world thing called an—a student, right! We know that there are students So, we say, "Hey, we have 'students'." So let's say, they're one entity. We have 'professors', obviously, because the students are there to learn etcetera. We have 'courses'. That's what they take. Like this is a course, so it's one entity, right! And, we have rooms in a–not...not in a online university, of course, but in a...in a real...real world, face to face university, you have 'rooms' where the students and professors are going to meet and attend classes. Okay! that's the idea here. There are many other entities. We could have—add entities like buildings, departments. We could add entities like, furniture, you know, maybe cafes, dining halls, student dormitories, this–everything that has a real world equivalent, and it doesn't have to be physical. It could be an abstraction. But, some kind of real world equivalent is a potential entity, right, so we can—many such entities. But, let's say, we start with these four, right! There could be many more. And, the goal—roughly as you can—you should have figured out by now, is that we should be able to build a model that takes these four entities or these entities, whatever we've identified, and builds a network of relationships between these entities, and then we have what's called a relational—and a...a...a model of our data that will work in the relational world. For the relational world, we need a linked, networked model, right, relationships between the different entities. So, that's our entities.

Now, we take our—so, the first step is to identify the entities. So, once we have them, we take each entity, and see how it is related to the other entities. There may not be a relationship with everything, but there should be a relationship with at least one other entity. Otherwise, there's something wrong with your model. You don't really have things, that are completely disconnected from everything else inside a data model. So, for example, we have a....a 'student' entity and a 'course' entity, and we have

students who enroll inside courses. So, we can call this relationship an 'enrolled-in' relationship. Students are enrolled in courses, or one–a student is enrolled in a course. So, that's the relationship between students and courses. We have 'professors' and 'courses', and a professor 'teaches' a course. So, that's a pretty straightforward relationship. We have 'professors' and 'students', and a professor could be an advisor, an academic advisor, for a student. Typically, in a university setting every student has one academic advisor, right! So, that's our relationship between a professor, and a student that they're being advised by—the student being advised by the professor. And we could say, that a 'professor' has an office in a 'room', right! You can have a course being taught in a room, or you can have a professor who has an office in a room, right! I mean it's a–this is all semantic, and it really depends upon, how you view the world. So, maybe we could say the room is an attribute of a professor, right! So, let's take a look at attributes.

The–every entity and every relationship can have attached to it a set of attributes. What's an attribute? An attribute is a property of the entity or relationship. So, it's some–you can think of some data that– think of it as being some data element that is useful to record with that entity, and the key term is useful. It has to be–have meaning in our...in our world. So, for example, we have professors in a university, and there are many attributes we can...we can...we can attribute to them. So, for example, a professor has a 'name'. So, clearly we need to know the name of the professor that's useful to store in the...in the database. Possibly, a 'department' that they're attached to that's useful stored in the database. But, the professor also has a mother and a father. We don't care about that, right! That's not– that is an attribute of the person, who happens to be a professor, but it's not an attribute of the role that the person is playing as a professor. So, we want to look at the attributes that are meaningful in the context of our database. So, the mother, the father, of the professor are not meaningful in our database. The children might be, for example, the professor might be getting benefits from the university, and those benefits may be accruable by the students. Free...Free education in the university, maybe, healthcare, stuff like that. So, if we were looking at the professor as an employee of the university, then that would be useful, but we don't care who the mother or the father or the grandparents of...of the professor are. We don't even care where they were born, which city they were born in. You know, this is all information, that doesn't really matter to us, right! So, we only want to store the stuff that is useful in the context of our database. So, if you're looking only at the academic, if you're building a purely academic database and that's probably what we are doing here—this is purely an academic database, then we don't even care about children or anything that has to do with benefits, right! We only care about the academic role, and the academic role we're interested in where the professor's office is. We want...want to know maybe the department and...and things like that, and maybe the average ratings or history of ratings, any of those kinds of things. But, nothing really beyond that, right! So, only the academic data is what we care about. So, let's take our 'professor' attribute then. So, our 'professor'—'professor' relationship—entity then. So, 'professor' entity contains—we can think of having three attributes, the 'name' of the professor because that's important, the 'office' where the room is—I've...I've—and this is a nice example, I'll come to that in a second, and the 'department' that they belong to, okay! So here, what I've done actually is I've taken the office relationship here, this one, the office–the 'has-office' relationship, this one, and made that into an attribute over here, and that's perfectly reasonable, okay! You can do that. You can do that if it makes sense. So, an attribute can become a relationship, if it's important enough to have existence of its own. So, for example, if a...if a professor's office has– today professors have multiple offices, right, then you have a problem here, so office one, office two, office three, and we don't want to do that. Then, we might want to say, "Hey, office is a relationship rather than a entity, rather than an attribute." Okay! So, things like that. So, it depends on how we want to structure it. The point being that the way our data, our model, is going to represent our data depends upon the way, we want to use the data, right! So, there is a–there's no perfect way of recording a data model. That's the bottom line.

Video 3(8.20) Basics of Databases (Part 3)

So, we also have a student entity. So, the student entity here has say, name for student, the ID number, social security number, or some other identification number. Perhaps the major, you know and we can think of other things as well. And then we have a relationship 'teachers'. Professor teaches a course and perhaps in this case, we want to just have one attribute 'rating'. Typically, relationships will have attributes, should apply only to the relationship. And in a very general rule of thumb sort of way–you want to minimize, the number of attributes a relationship has. Okay, but there's no requirement like that. So, this becomes our, you know, the basics of entity relationship model. And we've got a bunch of entities, relationship, and attributes over here. And when you look at the relationships, the next thing you want to do is, you want to figure out what kind of relationship is there between two entities? So, for example, if you have a relationship called 'has-office', and in our university, let's assume that a professor has only one office, which is fairly typical in universities, right! So, in that case, what we are saying is, that if you take a professor 'John', then we're get one room for John. Let's say the room...room is 'SB42', whatever that is, something. So, John has an office in 'SB42'. So, this is the 'has-office' relationship, right! And what we are saying is, that since in our...in our data world, John can have only one office, and no professor, let's say in our data world, no professors share an office, every professor has their own office. Then what we are saying is, that John has, there can be only one such row in my database. One row that has John and 'SB42'. 'SB42' will not occur anywhere else, and John will not occur anywhere else. Okay, assuming that John was a unique name for a second; right! Let's make that assumption. So, then what we see is, that this relationship here is a 'one-one' relationship. That means one professor has one room. This is kind of an important thing because when we are talking about entities, and relationships and, you know, tables, and all that kind of thing, relationship model. We want to be able to uniquely identify each row in a table. So, we have a table that looks like this. John, 'SB42', Jill, 'AC24', right, etcetera. So, if a relationship, and this is our 'has-office', right! If, a relationship is 'one-one', that means that there can be only one John in this column, only one Jill in this column, and only one 'SB42' in this column, and only one 'AC24' in this column, In which case, what we're really saying is that either John or 'SB42' can be used to access the row, to uniquely identify the row. So, either of them can be a key for that table, right!

A key is a field, or a column, or a group of columns, that can uniquely identify a row in a table. That's the...the goal in a...in a... in a...in the key field. So, what we are saying is that John can be a key, or 'SB42' can be a key. Jill can be a key, or 'AC24' can be a key. In other words, either the name or the room can be keys to, can be uniquely used to get rows in this table. You could ask that table, "hey, give me all the data for 'SB42'," you get only John back. You could say, hey, give me all the data for Jill, you'll get only 'AC24' back. That's the point; right! You won't get more than one thing back. So, that's a 'one-one' relationship. You can have a 'one-many' relationship, or 'many-one relationship'. So, for example, we have this relationship between professors and students where a professor advises a student. So clearly, we have many more students in a university, than we have professors. If every student had an advisor, then you gone have more than one advisee for any given professor, right! So, what we here say is that one professor as advises many students and the reverse relationship will be different because every student will have only one advisor and typically that is the case. You don't have two academic advisors because then there be conflicting advice which is never a good idea, right! So, you'll have one advisor for each student. So what we end up with is a relationship, in the case of professor to students, which says– which is called 'one-many'. 'One-many', because one professor advises many students but one student is advised by only one professor. So, what we have is a relationship here between professors and students. It is something like this. Okay, one professor advising many students. But, the reverse is many students identified by one professor–advised by one professor. And the final thing of course is, we can have what's called a 'many-many' relationship. And the whole idea in this of course is, to build a key. So, what we do–what we are saying is in a sense is that, if, we have a table that contains professors and students, then we're going to have something like this, say, professor John, student Jill, professor John, student

Adam, professor John, student Qing, etcetera, right! So, what we have here is that we can't say, give me the student advised by John, because we're going to–we have...we have three students in our table advised by John. So, if you want a key for this, then we can either make the key, the student, because we could always ask the student Jill, who is advisor, right! That would be the key for that because that uniquely identifies every row in this table. and actually that is the key, so that becomes our key over there.

Finally, the third case is 'many-many', in 'many-many' relationships, you have many on one side and many on the other side and the clearest example in our setup is, we have many students belong in one course and each student can take many courses. So, we have a student, oops! A student-course relationship, that says, many students enter a course, and many courses are in a student's academic requirement, whatever, right! So, they...they...they enrolled in many courses. So in now in this case, we have a...a, our key is going to be slightly different. Because, if you look at this thing here and we...we have a table here that has say, student one course 'C2', student one, 'C3', student two, 'C1', then, if you want to uniquely identify rows in this table, we need to look at both columns together. So we have a key which is called a 'composite key'. It requires us to know both the student as well as the course to uniquely identify rows in our table. That's the goal with that. So, these three kinds of relationships and it's good to bear in mind that they exist. It's not, you know, completely necessary for us, to understand it totally. But we should know that these are the kind of relationships, that do exist over there. So once again, reviewing this stuff, we have an 'Entity-Relationship model'. We have figured out the entities and the relationships. Now, what we are going to do is, we want to look at, how do we diagrammatically represent this...this data, because obviously, a picture is worth a thousand words.

Video 4(9.54) Normalization (Part 1)

So, that becomes our ER model, and once we have that, we've got our relation model set up. So, we end up with a bunch of tables that look something like this, right! So, we have here a 'Student' table that has Social Security Number, First Name, Last Name, emails—I've expanded on the attributes just to make it look slightly different. The students have multiple phones. Everybody has lots of phones, right! So, we have for example—Roberto Perez has two telephone numbers, lives or comes from San Francisco, you know, all that kind of stuff, has two email addresses—people might have multiple email addresses, and all that kind of stuff. We have courses, and for each course, we've got a course number, and then data about the course, and the course data that I put over here, again, added a few attributes, includes a room, and the number of seats in that room. Okay, so we have here the room and the number, of seats in that room. So, this tells us that room '1127' has '60' seats. If you're alert, you've already notice that I've, sort of, rigged this a little bit, because this data is redundant, and we can see it has severe inconsistency issues. So, for example, if I accidentally change this to '70', and I ask the question, "How many seats are there in room '1127'?", then depending on whether I start from here in searching or I start from the back, I'm going to get a different answer, that's inconsistency, and we can see that's coming out of the redundancy that's inbuilt in this stuff here. And finally, we have a relationship table, which is 'Enrolls-in', and the relationship table has borrowed the– because that's a many-many relationship, has borrowed the key of the 'Student' table, and the key of the 'Course' table. And then, it has, you know, some other data of its own, the First Name, Last Name of the student, and their grade, okay, right! So, that becomes our 'Enrolls-in' table. Again, this is not meant to be a good database, because I'm–I have sort of rigged it a little bit to... to make it–make our next step a little bit more meaningful. But, this is what is–it's not unreasonable, right, to have a bunch of tables, that look like this, and this comes straight off our ER model. So, that's the goal here, and once we've done that, then, you know, we can say, "Hey, we've got our relation model. We have a bunch of tables, and SQL is going to work now in answering queries on these tables." So, if necessary, we can use that. So, that's where we are right now. Once we've got the relational model, we want to move on and normalize it. So, what does this normalization stuff? normalization is the process of reorganizing the database to reduce redundancies and increase integrity that means increase the consistency of the data, okay, make it more consistent. So, the idea of normalization is it actually ends up making the queries a little bit more efficient and more consistent, which is kind of nice, all right, because it's where we want to be in this kind of world here. But, more importantly, what normalization does is it addresses three different types of anomalies that, you know, typically give rise to redundancies and inconsistencies in a database. So, these anomalies are anomalies in the sense they are—you can think of them as features of the database that will cause a problem when you are trying to insert data or change something, or remove something from database. That's why they are called anomalies because they...they cause problems. So, the problems occur when you're inserting new data into the database, or when you're updating, changing a value in the database, or when you are deleting a piece of data, from the–complete data item from the database, and the...the whole purpose of normalization is to ensure, that these anomalies are, at least the probability of having these problems, is greatly reduced.

That is...that is the goal of this. It's a...it's a very complicated field, and typically it would require perhaps, you know, normalization alone would require a half semester, of coursework. So, what we're going to do here is do a very simple philosophical view of what normalization aims to achieve and leave it at that. So, the first anomaly that we want to look at is the insertion anomaly, and this occurs when you want to add something to the database, but there is no place to add it. So, you want to add some new information, a new entity, or a new thing to the database, and you can't add it anywhere, like, you look at your table structure, there's no such–no place at all to add it. So, let's take a look at an example. So, let's say we have a table here called 'Professor', and the 'Professor' table has three columns, the name of the professor, the office where they sit, and their department, okay. So here, we've got this— these three things here, and we got professors—this got a little bit messed up, but let's say, there's actually a name over there, professor Lee, professor Wu, and two other professors. And, their departments are over here. And, that information is there. Now, what happens is if the university decides that it needs a Physics department so it says, "All right, let me add a Physics department to my database." But initially, we don't have any faculty members, the...the senate, university senate has sat down and, you know, had a meeting, and said, all right, from now on we have a Physics department, and we're going to start offering Physics degrees.

We're going to start admitting students for a Bachelor's in Physics, starting in the fall of this year, right! So now, we have a decision to do that. We have the–have to set up the process of admitting students, and doing all kinds of things. But the only place in our database, where a department is mentioned, is in this table, called 'Professor'. If there's no professor in the Physics department, we can't have a Physics department, in our database. There's no place for it, we can't add it to our setup, because you just can't have a nothing, a nothing and 'Physics'. You need values over here, and you need a value over here, there's no professor, so we can't add it, right! That's as simple as that. So, this is what's called an insertion anomaly, because we have and...and, you know, conceptually what's happening is, that we have this real world thing called Physics, right, or real world thing called department or major, or something, right, that is out there. But, we haven't modeled it in our database as an entity. We modeled it as an attribute of another entity, right! We modeled it as an attribute, of the entity 'Professor', So, though it's...it's an entity, which apparently now we can see has some importance of its own, we treat it as an attribute, so that's our problem, so, there's no way to add it. So, this is an insertion anomaly because now, you know, the senate has said Physics, and the Computer Science—Computer IT department or whatever is sitting down and saying, "Hey, where do I put Physics? I can't see anywhere to put Physics." This is a problem, right! So, they got to solve that, and this is fairly easily solvable. What you can do is you can take out—or what you would need to do is, you need to take the 'Department', remember, I said that the reason we're having the problem is, because 'Department' is really an entity of some importance, and we got it as an attribute. So, what we should do is make it an entity. So, we make it an entity over here, and we have now Industrial Engineering and Operations Research, Mechanical Engineering, Computer Science, and we add a new department called 'Physics', right! And, we give them a code as a key, which is not necessary, but typically, when you have long strings of this sort, it's a good idea to have a simple key attached to that because, you're going to repeat this in other places. So, rather than having to put Industrial Engineering and Operations Research here, we can just use IEOR, and that's the idea here. So, now we've got this. So, now we have—we can add 'Physics', because we just added to the 'Department', nothing happens to 'Professor'. We don't have a new professor in that department. There's no—there no one teaching in that area, so there's nobody there, so, we don't need to add it there. We can just add it over here, and then, if there's any other administrative stuff that needs to be done, it will follow from this, right! Therefore, example, our brochures—the...the school bulletins or brochures, they list departments using this table, departments, they can just use this table, and say we have a Physics Department, so what, if you don't have any Professors, but it's there. So, we can either make a...q new relationship called, say 'Belongs-to', and create yet another table that has, you know, each Professor name say, 'Micha', 'IEOR' etcetera, or we could simplify matters by just including the 'Department', as another column here, even though, it's not typically— no longer an attribute. But since, it's a— this table is only going to contain just one column, and it's not...it's not a perfect solution, but it's, for ease of access, it makes sense to say, "Hey, we'll just collapse the department right into this column over here, into this column inside our 'Professor' table." And, our 'Professor' table now contains the department code, for each department, and we can access the name of the department by following, that code over there, okay. So, this really is a...a...a sort of, you can think of it, as a synonym for a relationship, but we just collapsed it into a single table. So, that's the way to deal with insertion anomalies, and you can—you, what you're really doing is, you are saying that, "Hey, I have an entity, that is modeled as an attribute so, I need take it out, make it a separate entity, and build relationships between that entity and other things." Okay, that's the goal there.

Video 5(10.02) Normalization (Part 2)

Second kind of anomaly is an update anomaly, and this occurs when there's a change to the value of an attribute. But, that change has to be made in multiple places. We've already seen this. This is the issue of redundancy. So, you have the same attribute having a multiple– occurring in multiple locations and with the same value, and when you change that value, you need to go, and change it in multiple places. It was never a great idea. So, if do you that, then you're going to have, possibly have redundant data, and inconsistencies in the way your queries are respond–answered. So, let's take an example. So, we've already seen this, but let's take it—look at it again. The—we have a 'Course' table over here. The 'Course' table contains the course number, course number, the name, the room, the seats.

So, what we're doing here is we are taking the room and the seats, this stuff over here, and what we're really modelling here, if you think about it, is a relationship between the room and the number So, if you think of the...the number '60' in that–in our example, the '60' is not an attribute of the Course 'c4'. t's an attribute–it's not an attribute of this, it's an attribute of this. So, we have inside here—what's in database term is called a dependency, what we are saying is that the number of seats, depends on the...the room number. It doesn't depend on the course, because the course can be switched, to a different room, right! So, that's the...the...the reason why we have this repeating—redundant field in our...in our table, right! Because we really have a dependency, that is, between two things that are not really what the table is about, right? That's...that's the thing.

The table is not about the room and the seats, it's about the courses. So, what...what happens, of course, here is that if we change this '60' to '70', and we asked the question, "How many seats are there in room '1127'?", we're gone have a problem, because if we–if our algorithm that looks at the table and—you know, we don't know—we don't have control over the way SQL actually is executing the...the tables, the queries, if it looks at the...the query by searching top down, then it's going to say '70'. On the other hand, if it looks at the query bottom up, it's going to say '60'. We get two different answers for the same question. Not a good idea, okay! So, how do we deal with this? Well, pretty straightforward, we know that our problem is occurring, because our number of seats depends on the room, not on the course, right! The dependency is going this way, from the seats—depends on this, and does not depend on this. Therefore, what do we do? We take this out and make this into a separate table. So now, we have a table that has room number, the number of the room, and the number of seats in the room etcetera, and we are all set. Because now, if we want to change the number of seats, all we need to do, is go in here and replace the '60' by '70', and there's no other place, where the number of seats is listed. So, we are all set with that. So, this is a straightforward way of doing it. But we can see, again, the...the reason that this happened was because there was a dependency between one column, in this case, in our example here, between the seats column, and something that wasn't, really what the table was about, right! And, in a more formal sense, we're saying there's a dependency between seats and something room, in this case, that is not a key for the table. The key for this table is course number, it's not the room. In fact, the room, you can see occurs numerous times '1127', '1127', '1127'. So, it's not a key, and we have a dependency between seats and something that's not a key. That's the formal way of saying it, but in a more, you know, informal sense, what we're really saying is that we have here, seats that depends on something that is not what this table is about. So, that's why we're getting this problem. So, that's update anomalies. And then, the third kind of anomaly is what's called a, 'deletion anomaly'.

This occurs when deleting something from the database results in some other, possibly important fact, being deleted. So, that you take something out of the database, and something else that is important also goes away, right! And, we can—actually if you think back to our tables, we see many such examples. So, if we do this what will happen is that we end up losing data. Just a simple example. If you go back here, and I delete information about, like say, if I delete course 'c3' from this table, not only we—do we lose information about Course 'c3' but, we also lose this relationship between the room and the number of seats. Once I deleted that if I have a question like, "How many seats there in room '331'?" I can't answer it, right! It's not in my database anymore. So that's, you know, a really bad thing. You don't lose information. So going back to our course over here, our example, over here. Let's say, we have our table, which has four columns—four attributes and...and the course number, the name of the course, the room number the course is in, and the enrolment in the course, okay. So, the enrollment in the course means the number of students who are enrolled. So, let's take a look at the first, course number one. There are only '6' students enrolled, okay. Data analytics in Python or data, whatever, right! There are only '6' students enrolled in that class. So, the university decides it's not cost effective, to hold a class with only '6' students. So, let's fire the professor and close this class down. So, that class gets closed down, and what that means is then, this row vanishes from our table. But, not only are we losing the enrollment or...or that kind of information with it, but we're also using other information, which is not exactly over here, but should be, is things like the 'course description', you know, what is the long description of the course, what are the ratings of the professor, maybe they didn't fire the professor but they kept him on but his or her ratings are useful information, so that's also gone, you know, maybe their over here, we had columns that said 'ratings', 'description', okay, previous enrollments, you know, etcetera, right, all kinds of stuff, that's all gone. So, we...we lose a lot of information when we delete an entity or a thing from our database, an item from our database. And, the question is, are we structuring our database so that we don't actually lose the information that we don't want to lose.

We only lose information that we don't care about. So, in this case, it will be again, fairly straightforward, because what we are really doing, if we look at our table here, is we—our course table contains in this structure, and this is a really artificial structure so forgive me for that, but in this structure what it contains is information about the current occurrence of a course. If you don't offer a course this semester, that course doesn't exist, which is not really a reasonable thing to do. So, what we need to do is we need to say, "Hey, we need to take out the listing of courses that exist separately and courses that are currently offered, and keep them separate." Okay, we don't want them to be in the same table. They're linked by the course number, so that's our link, okay! So, the course number links these two tables together.

But, all information that needs to persist is contained in this table, and only current enrollment information and the current room number is contained in this table. That's the idea here, okay! So, that would may take care of the...take care of the deletion anomaly. So, what we've done is we've taken 'Courses' and 'Offered courses' in separated out these two tables, and made them into independent tables, so that, this is the persistent table, and it will contain the information forever. And, this table is our current enrolment or current offerings table, and it contains only the current information. Now, if we decide to scrap this class, we can do it without losing any meaningful information. Okay, that's the goal there. So, if we can remove all these anomalies, then we are in a position to say, that we have what's called, a normalized database. And, the database normalization is the process of removing these things. And, again, we are not going to database theory over here, but just to be familiar with the terminology. We say that a relational database can be in any one of these forms, first normal form, second normal forms, third normal form, Boyce-Codd normal form, and fourth normal form, and there's actually another form as well, but let's stick with this. And generally, if we are free in our– if we gone there this process of removing anomalies, if we are free of these three kinds of anomalies, then this– then the database is said to be in '3NF' or third normal form. Meanwhile, we're not going to study the normal forms here. It's a–you know, reasonably comprehensive piece of literature. But, you should be aware they exist, and when you're designing a database of your own, you should think about these issues, whether, you know, how do we get rid of these three things, insertion, updation, and deletion anomalies. And, if you can think about this and get rid of them, then you're...you're in...in pretty good shape.