Table of Contents

Before we cover the syntax for creating tables and setting constraints, it is time to talk a bit more about relations and relationships Relations is the formal term for how data is structured; we visualize relations as tables. Relationships are the way that tables relate to each other and are essential to understanding how a database works. So we will first look at a few more definitions and considerations for relationships.

As we talk about the design and structure of tables, we are discussing metadata. Metadata is the data that refers to the structure of the database, such as the names of the tables and their attributes, the data types of the attributes, and the primary keys. It does not refer to the values (such as the last name of a customer is Jones); instead it refers to the attribute that we use to store that value (such as the last name can store up to 25 characters)..

1. Business Rules and Table Design

The ultimate design of a database and its tables comes from the business rules of the company. Business rules are statements about the way a company structures its data and controls its operations. For example, in our vets database there is a rule that each animal is the responsibility of a single client. This business rule dictates that each animal row in the animal table is associated with one client row and, therefore, cl_id is an attribute of the animals table.

Many poorly designed databases have poorly designed tables. Most problems arise from having tables that are too complex— that have too many attributes that do not belong to that table. This is often a result of thinking of the data as a manual table. Since manual tables for storing data do not have automatic look-up features, we tend to put all of the data into one big table so that we can find it. This same attitude is often found when people create spreadsheets. We might have a spreadsheet where each row for an animal includes the client id but also their name and address. We would not want to bring that design directly into a database table.

Suppose we had the following table for animals and clients.

An_id	Type	Name	dob	Cl id	Last Name	First Name	Address	City	State	Zip
11015	snake	Kenny	2005-10-23	4534	Montgomery	Wes	POB 345	Dayton	OH	43784
11029	bird		2005-10-01	4534	Montgomery	Wes	POB 345	Dayton	OH	43874

12035	bird	Mr Peanut	1995-02-28	3560	Monk	Theo	345 Post Street	New York	NY	10006
12038	cat	Gutsy	2007-04-29	3560	Monk	Theo	3405 Post Street	New York	NY	10006
15001	turtle	Big Mike	2008-02-02	25	Harris	Eddie	2 Marshall Ave	Big Rock	AR	71601
15341	turtle	Pete	2007-06-02	93	Wilson	Sam	123 Park Place	Big Rock	AR	71601
15002	turtle	George	2008-02-02	25	Harris	Eddie	2 Marshall Ave	Big Rock	AR	71601
16002	cat	Fritz	2009-05-25	25	Harris	Eddie	2 Marshall Ave	Big Rock	AR	71601
21002	snake	Edger	2002-10-02	25	Harris	Eddie	2 Marshall Ave	Big Rock	AR	71601
21004	snake	Gutsy	2001-05-12	25	Harris	Eddie	2 Marshall Ave	Big Rock	AR	71601
21005	cat	Koshka	2004-06-06	25	Harris	Eddie	2 Marshall Ave	Big Rock	AR	71601
21006	snake		1995-11-02	25	Harris	Eddie	2 Marshall Ave	Big Rock	AR	71601
				5686	Biederbecke	NULL	50 Phelan Ave	San Francisco	CA	94112
				5689	Biederbecke	NULL	50 Phelan Ave	San Francisco	CA	94112
				5698	Biederbecke	Sue	50 Phelan Ave	San Francisco	CA	94112

The spreadsheet approach leads to two problems:

- having rows of data with a lot of empty cells
- data redundancy, having the same data value stored more than once.

In the above table we have several rows which have no data values for the animal columns because these clients don't have any animals. But if we are storing our animal and client data in this table we must have rows for every client even if they do not have any animals.

We also are repeating a lot of values which leads to more problems- did you notice the two errors in the data in the table?

Data redundancy leads to problems. One of the first problems we tend to think of is that we are wasting storage space- we do not need to store client 25's name and address 7 times just because he has 7 animals. But storage space is pretty cheap. The more important problems are the logical ones.

- update anomalies where one copy of the data value is changed but other values of the same data item are not changed
- delete anomalies where deleting one data value results in too much data being deleted
- insert anomalies where the user cannot insert the data values they wish to store.

Suppose client 25 moves- how many places do we have to update that data? Are you certain that every application that updates data will update every one of those rows?

If animal 15341 dies and we want to remove its row, we might also remove the only row we have for client 93.

Suppose we need a rule that the animal id and type and dob could not be null. How do we enforce this rule and still allow the table to include clients with no animals?

These are problems that were a major consideration with traditional file processing and database systems were suppose to help solve these problems.

With a DBMS you can split the data into smaller, more tightly focused tables and then reassemble the big collections of data when needed through queries. The general problem in going from a big table to properly designed tables is to create a collection of related tables without losing any of the information contained in the original collection of data. Note that we have to preserve the table information-not just the table data. Table information includes facts imbedded in the table design.

At this point in the semester, these should seem natural- the client table includes the client name and address and each client gets one name and one address. And the animal table gets one row for each animal.

Cl id	Last Name	First Name	Address	City	State	Zip
25	Harris	Eddie	2 Marshall Ave	Big Rock	AR	71601
93	Wilson	Sam	123 Park Place	Big Rock	AR	71601
3560	Monk	Theo	3405 Post Street	New York	NY	10006
4534	Montgomery	Wes	POB 345	Dayton	OH	43784
5686	Biederbecke	NULL	50 Phelan Ave	San Francisco	CA	94112
5689	Biederbecke	NULL	50 Phelan Ave	San Francisco	CA	94112
5698	Biederbecke	Sue	50 Phelan Ave	San Francisco	CA	94112

An id	Type	Name	dob	Cl id
11015	snake	Kenny	2005-10-23	4534
11029	bird		2005-10-01	4534
12035	bird	Mr Peanut	1995-02-28	3560
12038	cat	Gutsy	2007-04-29	3560
15001	turtle	Big Mike	2008-02-02	25
15341	turtle	Pete	2007-06-02	93
15002	turtle	George	2008-02-02	25
16002	cat	Fritz	2009-05-25	25
21002	snake	Edger	2002-10-02	25
21004	snake	Gutsy	2001-05-12	25
21005	cat	Koshka	2004-06-06	25
21006	snake		1995-11-02	25

You might notice that often well designed tables have fewer columns.

If it were important to allow more than one address for a client, then we could split the client table into multiple tables. One table for the client name (each client gets one name) and a second related table for the client addresses. But those tables would also each need the client id to link back to the main client table. This requires understanding the business rules and policies reflected in the data collection.

Now some clients can have multiple addresses

Cl id	Last Name	First Name
25	Harris	Eddie
93	Wilson	Sam
3560	Monk	Theo
4534	Montgomery	Wes
5686	Biederbecke	NULL
5689	Biederbecke	NULL
5698	Biederbecke	Sue

Cl id	Address	City	State	Zip
25	2 Marshall Ave	Big Rock	AR	71601
25	2957 Grover Ave	Springfield	IL	61258

93	123 Park Place	Big Rock	AR	71601
93	56 Meadowland Ln	Springfield	NY	10027
3560	3405 Post Street	New York	NY	10006
4534	POB 345	Dayton	OH	43784
5686	50 Phelan Ave	San Francisco	CA	94112
5689	50 Phelan Ave	San Francisco	CA	94112
5698	50 Phelan Ave	San Francisco	CA	94112

2. Relationships and Referential Integrity

The following are rules and considerations that you would need to think about when you set up tables and constraints.

2.1. Type of relationships

Relationships are logically classified as one of the following: One-to-One, One-to-Many, and Many-to-Many

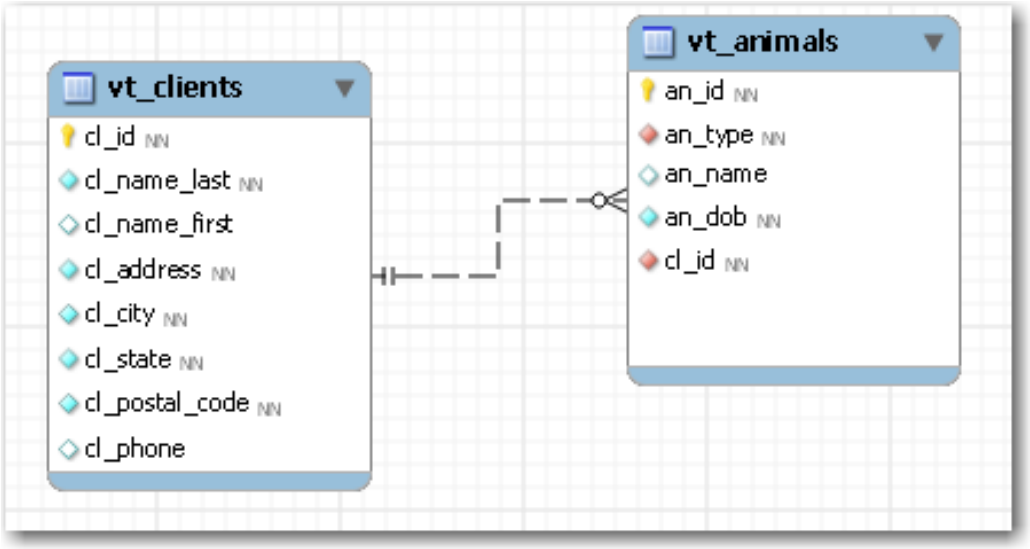
One-to-One- this means that a row in table A is related to at most one row in table B. Suppose we wanted to keep a photo of each member of our staff. Assuming we had a way to store a photo in our database, we might want to set up a second table vt_staffPhoto (staff_id, Photo) since a photo would take a lot of space and we would not access it very often. Keeping that data in another table makes our regular staff table more efficient to access. We might also decide that we want to keep some of the employee data confidential; we could store the confidential data in a second table and restrict access to the table to only a few users. We would include the staff id in the second table to link the two tables in a 1:1 relationship.

One-to-Many- this means that a row in table A can be related to 0, 1, or multiple rows in table B and each row in table B is related to one row in table A. This is the more common relationship. A client row in the clients table can be related to 0, 1, or multiple rows in the animals table and each row in the animals table can be related to 0, 1, or multiple rows in the exams table.

Many-to-Many – this means that a row in table A can be related to 0, 1, or multiple rows in table B and each row in table B is related to 0, 1, or multiple rows in table A. This is a relationship that we can think of as a logical relationship but we seldom see it in a database as most dbms do not support this directly. If we think about medications and exams- a medication can be used on multiple exams and an exam can include multiple medications. One of the purposes of the exam details table is to serve as a bridge to implement this many-to-many relationship.

2.2. Referential Integrity rule

We will look at these in terms of the vets set of tables. This is a diagram showing the clients table and the animals table and the relationship between them. The NN after some of the attributes such as cl_id and cl_name_last stands for "not nullable".



We have referential integrity set between the Clients table and the Animal table. The pk for the clients table is cl_id and since it is a pk it is not nullable. The pk for the animals table is an_id. The animals table also contains an attribute cl_id which is the fk to the client table. We can navigate through the relationship from the animal table back to the client table to find the name of the client for an animal.

Referential Integrity Rule: The database must not contain any non-null unmatched foreign keys. If there is a value in a foreign key attribute, then that value must match a value in the related attribute in the parent table.

> In terms of our tables this means that we cannot add a row to the animals table that has a value for the client id that does not match an existing client row. If we want to add a new animal for a new client, we have to add the client first.

The referential integrity rule does allow for null foreign keys. Your business rules might call for a non-null foreign key.

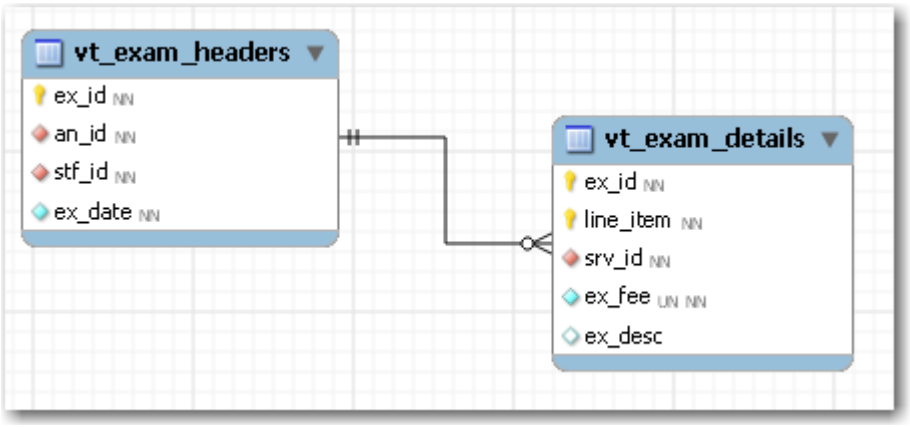
> In our animal table the cl_id field is not nullable. We are not allowed to have an animal row without a related client row. But that is not required by the referential integrity rule. It would be possible to set up the animals table with a nullable cl_id attribute. That would let us add animals with no associated client. This would make it difficult to know who to charge for an exam done for the animal- and the vet probably would not like this. What the referential integrity rule says is that an animal row cannot have a value for cl_id that is not a valid cl_id in the clients table- you can't just make up a client id for an animal.

You will not be able to set referential integrity if there already is data present in the tables that violates the rule, or if the attributes involved do not have the same data types.

2.3. Delete Consideration

One of the consequences of setting referential integrity for a relationship is that you must consider how the database should handle the deletion of a parent row if there are associated child rows.

This diagram shows the exams table and the exam details table and the relationship between them



We have an exam and an exam details table; exam_id is a foreign key in exam details table that refers to the primary key of the exam table. Suppose you decide to delete an exam row from the exam table.

If you could simply delete the exam row, you might have orphaned rows in the exam details table — exam details rows that contain a value for an exam row that no longer exists. Since this would violate referential integrity, the dbms will not allow this.

Logically there are four options that you could set for this relationship.

Option1: Restricted You cannot do that. You cannot delete an exam headers row if there are any rows for that exam in the exam details row. You would have to go to the exam details table and delete all relevant child rows first. This is generally the default setting.

Option 2: Cascade- Automatically delete all related exam details rows if an exam headers row is deleted.

Option 3: Nullifies- Keep the related exam details rows, but set their exam _id value to Null. This is the fk in the child table. To do this that attribute must be nullable.

Option 4: Default value- Keep the related exam details rows, but set their exam _id value to some default value for this situation. The default value must be one that exists in the parent table.

> Which of these options might we want to use for the relationship between exam headers and exam details? Starting with option 4- this would not make sense here. If we set the exam details fk column to a default exam id, this would mean that we would have treatments and fees charged that we cannot relate to an animal and therefore not to a client for billing purposes. We could not even tell what type of animal the treatment was done on. Option 3 is also not helpful for the same reason.
>
> The Cascade option might make sense. It would let us delete an exam and all of its details with one command- the command to delete the exam row. If we do not want to keep the exam header record, we probably don't want to keep the details. We tend to think of the exam header and its details as one item.
>
> The restricted option is always possible- it just makes us take more care when deleting exams.
>
> What about the relationship between animals and exams? We have the same option and pretty much the same decisions. If we delete an animal what do we want to do with the animal's exam records?

This brings us to a bit of a diversion- do we really want to delete data from our database? This might be a bit clearer if we think of an employee database. For each employee we probably have a lot of data stored- hiring date, payroll data, W2 data, and retirement data. If an employee quits, do we really want to throw away all of that employee's data? Actually a company has to maintain certain employee data in perpetuity so a cascade delete is not a good option. Normally records that we might want to delete are first copied to a set of archive tables that are maintained forever. Then the data in the active tables can be deleted – usually at a set schedule related to normal business cycles.

When might you want to use either the Nullifies or Default value rule for relationships? Suppose we had a table of products and a table of product categories. Each product is classified under a particular category. The product table has a foreign key to the product categories table. Now we decide that one of our categories no longer is useful-and we want to get rid of it but we don't yet know how we want to categorize those products. If the product table has a nullable category ID we could use the Nullifies option, or we could have a category id set up for uncategorized products and use that option.

What if you no longer stock a product? You might want to delete order details for that product when you delete the product row- in that case you might want cascade delete. But this could be confusing to a customer whose order just lost items with no explanation.

Another example might be a sales rep table and a customer table where each customer is assigned a sales rep. If a sales rep quits you might want to remove him from the sales rep table- but that does not mean that you want to cascade and remove the customers. Instead you would want the relationship to have either the nullifies property or the set default property and then later assign these customers new sales reps.

It may be necessary to set these options for several relationships for this to work properly. Suppose you really decide that is makes sense that if you delete a client in the vets database, that this should automatically remove that client's animals and their exam data. You would need to set the cascade option between clients and animals and between animals and exams and between exams and exam details. (And you would need to check with your boss and the database users if this makes business sense.)

2.4. Update Consideration

There is a similar situation if you want to change the value of a primary key if there are child rows for that parent row.

Example: Suppose you need to change the value for the exam_id in the exam headers table. What should happen to the related rows in the exam details table?

If you could simply change the primary key the exam headers row, you might have orphaned rows in the exam details table — exam detail rows that contain a value for an exam headers row that no longer exists. Since this would violate referential integrity, the dbms will not allow this. You have the same options of Update as you did for Delete: Restricted, Cascade, Nullifies, and Set Default.

If you change the value of the primary keys, you will create problems with historical rows that used the old pk values. If you do change primary key values, you should use new values that do not duplicate older existing values.

If you are doing a major redesign of a database, then you should consider keeping the old primary key attribute as a regular attribute in the table.

Allowing users to change a primary key value is generally not a good idea.

3. Normalization

Normalization is the formal process of decomposing tables that have design problems into well-structured relations. Good design is sometimes expressed as

- Storing one fact in one place
- Having all of the attributes in a table depend on the primary key, the whole key and nothing but the key.

3.1. Goals of normalization

Normalization helps you design table to

- preserve the integrity of the data
- identify a unique identifier for each record
- reduce redundancy. Redundancy is storing the same data value in more than one table. Redundancy leads to errors.

- minimize the space needed for the tables which also reduces the volume of data that must be transferred to the client.
- store the data you need without having to create dummy data

No design technique will guarantee that you have well designed tables. Having normalized the tables will help.

4. Some definitions

4.1. Functional Dependency

A functional dependency is a constraint between two attributes, A and B that states that for every legitimate value of attribute A there is one value (possibly null) for attribute B. For example, for every animal id there is one animal name and one animal type and one date of birth and one client id. This is sometimes written as

```
an_id → (an_name, an_type, an_dob, cl_id)
```

Relations are used to store functional dependencies.

4.2. Determinant

The attribute that functionally determines the other attribute is called the determinant. In the above, an_id is the determinant.

Candidate keys must functionally determine the other attributes in the relation (table). Candidate keys are determinants.

4.3. Anomalies

Errors in a table that are a result of a user attempts to modify the data are called anomalies.

Update Anomaly — having inconsistent data because the data was updated in one row but not in all rows. For example, in the spreadsheet style table for the animal and client data, the zip code for client 4534 has two different values.

Insertion Anomaly — the inability to add a new data to a table. For example, we cannot add a new client row to the table unless we have data for an animal for that client (or leave these as null values).

Deletion Anomaly —loss of data because we wanted to remove other data values. For example, if we want to remove the data for animal 15342, we could lose all information for client 93 (unless we leave several attributes with null values).

5. The normal forms

We'll first go through the definitions of the first three normal forms, then discuss some examples for these rules. These are formal definitions and not that easy to understand. The example can help.

Normal forms are the rules, standards, sets of conditions for the design of a table. These have been defined as the first, second, and third normal forms and a few more. This is a continuous set of rules. You first check that your table is in 1NF; then there are additional rules for the table to be in 2NF, and there are additional rules for the table to be in 3NF.

The tables we have been working with in the demos and assignments have been in 3NF

5.1. First normal form (1NF)

Data is stored in structures that we can think of as two dimensional tables. Each row in the table is unique- we do not have two rows with the same data values. Each row has the same number of attributes. Each attribute in a row stores a single value. Columns are not repeated in an attempt to handle multi-valued attributes.

5.2. Second normal form (2NF)

The table is in first normal form and there are no partial dependencies. Suppose we have a compound primary key. Then each non-key column in the table must depend on all parts of the PK.

This deals with a concept called functional dependency. Suppose we are working with the tables in the vets database and I say that I am looking at the exam details table. This table has a compound primary key of (exam_id, srv_id) In order to determine the ex_fee we need to know both the exam id and the srv id. We cannot say that if you tell me the exam id I can tell you the fee charged because I also need the service id. Also if you tell me the service id, I cannot tell you the fee charged because we can charge different fees for the same service on different exams. To tell you the fee charged I need to know both the exam id and the service id.

`(ex_id, srv_id)` → `(ex_fee)`

The rule for 2NF is that all non key attributes are functionally dependent on the entire pk. In order to determine is a table is in 2NF we need to know the pk for the table.

5.3. Third normal form (3NF)

The table is in second normal form and there are no transitive dependencies. The non key attributes must be independent of each other.

6. Examples of data that is not normalized

For this we will work with variations on the tables in vets database. The examples will focus on mistakes we might have made in the design and problems these might cause.

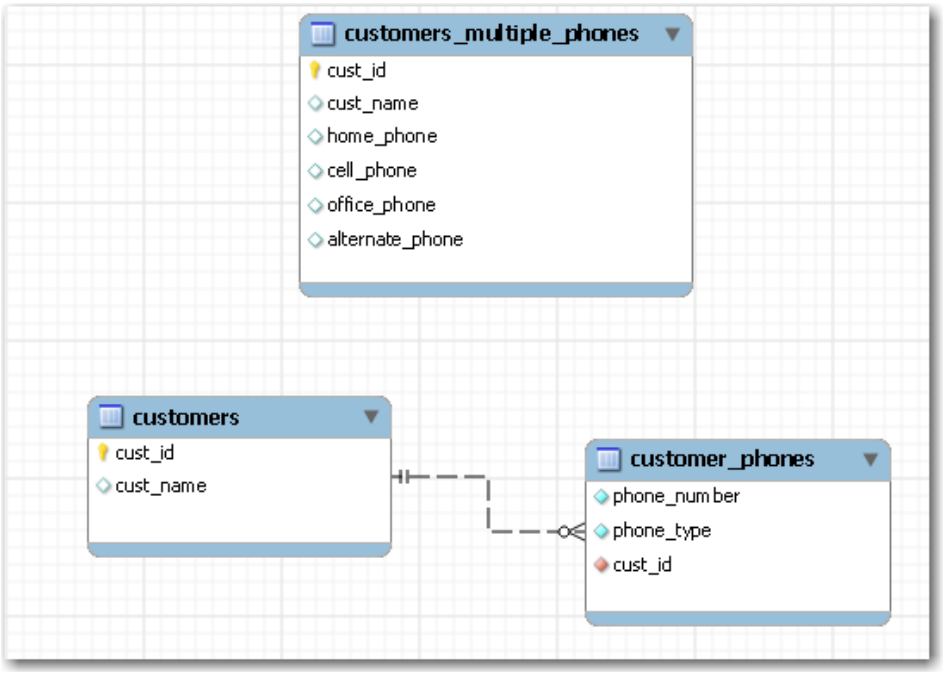
1NF: In the clients table we have a phone number column. We have now found that out clients have more than one phone number and want to give us both their home number and their office number. If the vet has to keep Fluffy overnight and make decisions about Fluffy's treatment, it might be important to be able to reach the client quickly. So we decide to add a new column to the client table and rename the existing column- we now have a column named home_phone and a column named office_phone. This might have worked in the past when people had one home phone and one job (M_F 9 to 5). This design might still work- but it would be harder to get a list of all of the phone numbers in a single column. And a query that tries to find out which client has a certain phone number has to query two columns. This table is now no longer in 1NF.

There are several bigger problems: (1) we cannot always distinguish between a home phone and an office phone- what do you do if someone works from home? (2) some people work multiple jobs and have more than one office phone; some people have more than one home phone. (3) how do we classify a cell phone?

Another approach that does not work well is to put all of the phone numbers into a single column separated by commas. And that would also take the table out of 1NF.

So if we are going to store multiple phone numbers for a customer, then we need another table- customer_phones. This table could have two columns: cust_id and cust_phone. The cust_id relates back to the customers table so we can identify a phone number as belonging to a specific customer and we can go from a customer to their phone numbers. We might decide to have more columns in that table- phone type- (home, office, cell, fax, weekendHome), maybe a column for the best time to call.

The relationship between customer and phone numbers now follows the same pattern as the relationship between clients and animals. A customer could have 0, 1, or more phone numbers. This is a one-to-many relationship. We implement it by making the cl_id in customer_phones be a fk to the customers table. We also avoid having a column in the customers table which is nullable (The vt.client.cl_phone attribute was a nullable column).



2NF: Suppose we had designed the exam headers and the exam details tables so that the exam date was in the exam details table. Perhaps we knew that we would need to write many queries about services that were done on specific dates. By putting the exam date in the exam details table we could avoid a join. If we did that the exam details table would not be in 2NF. The rule for 2NF was that we did not have partial dependencies. The PK of the exam details table is (ex_id, srv_id) but the exam date depends on the exam id only - it does not depend on the service id. From a practical point of view, putting the exam date in the exam detail table would be that we would have to repeat that data value in multiple rows- if an exam involved 5 treatments we would have to repeat the exam date five times. This is redundant data. There also is a problem that the exam date value might be entered with different values for different rows. This could be due to a data entry error or a change in the system time value while the rows are being entered. If the date needed to be corrected, then the app would need to correct all of the rows consistently. This could be handled at the app level- but it does make this harder.

3NF: Suppose we decide that it makes sense to store the staff name in the exam table- so we would have (ex_id, an_id, stf_id, stf_name_last, ex_date). That would save us a join when we wanted to display the details of who was responsible for an exam. But this is a transitive dependency. The pk is ex_id which determines the stf_id which determines the stf_name_last. So in a sense the ex_id determines the stf_name_last but the dependency goes through the stf_id. This design would also cause us to store redundant data and open up the possibility that we spell the person's last name in different ways in different rows. We would also have a problem when a staff person changed their name- we would have to update it in multiple tables.

7. Codds Rules

Dr Codd developed the relational model for databases in the late 1960s. When the relational database model caught on with software vendors, everyone wanted to call their database system a relational database. Dr Codd formulated a set of rules that a database must follow to be considered a relational database. Proprietary dbms generally do not follow all the rules.

1. **Relational capabilities:** The user must be able to manage the data in the database entirely through its relational capabilities without needing other tools. The DBMS might provide other tools but to be relational, those tools cannot be essential to the basic functionality of the database- such as creating tables and accessing the data.
2. **Information:** All data has to be represented in the same way- as rows and columns in one or more tables. This includes the metadata.
3. **Access**: Each piece of data can be identified and accessed by a combination of the table name, the column name and the primary key value. (This means that you do not use pointers or links to navigate a database. We can get to the last name of a customer by using the table name, the name of the attribute and the pk value for that customer.)
4. **Nulls**: The DBMS must have a consistent way to deal with null values. Null values are distinguished from numeric zeros or empty text strings or text strings of blanks.
5. **Catalog (data dictionary**): The user should be able to access catalog metadata on-line via the regular query facilities.
6. **Data sub-language**: There must be a single language that can be used interactively or embedded within programs. This language will allow manipulation of the database including data definition, view definition, data manipulation, integrity constraints, security and transaction processing. This language is not required to be SQL, but commonly is SQL
7. **Updateable views**: If it is theoretically possible to update a view, then the DBMS should allow the view to be updated. Some views cannot be updated.
8. **High-level update**: There must be a way to update or insert or delete a **set** of rows at one time.
9. **Physical data independence**: The end-user should not need to know the physical structure of the database. It should be possible to change the physical structure without requiring changes in the logical structure. The people who implement the physical database can use any of a variety of techniques- but the people manipulating the database via the database language don't need to worry about that.
10. **Logical data independence**: The end-user's view should be independent of the logical structure of the database.
11. **Integrity**: Integrity constraints are enforced by the DBMS and not by application programs. This provides a more consistent approach to integrity.
12. **Distribution independence:** Applications should work in a distributed database and should continue to work if the data distribution pattern is changed.
13. **Nonsubversion:** It should not be possible to subvert the integrity constraints. There should be no back door to the data