**Database — Introduction (Part 1)**

**Why a database, What’s a database, and DBMS.**

**This is a long series of tutorials. We are going cover:**

* [**An introduction to databases (why, what, and DBMS)**](https://medium.com/omarelgabrys-blog/database-introduction-part-1-4844fada1fb0)**.**
* [**The fundamentals of the databases**](https://medium.com/omarelgabrys-blog/database-fundamentals-part-2-b841032243ac)**.**
* [**The database design process**](https://medium.com/omarelgabrys-blog/database-design-process-part-3-7b5fafc78774)**.**
* [**Normalization**](https://medium.com/omarelgabrys-blog/database-normalization-part-7-ef7225150c7f)**.**
* [**The structured query language (SQL)**](https://medium.com/omarelgabrys-blog/database-structured-query-language-part-8-230a1808ec96)**.**
* [**Indexing, Transactions & Stored Procedures**](https://medium.com/omarelgabrys-blog/database-indexing-and-transactions-part-9-a24781d429f8)**.**
* [**The different types of databases**](https://medium.com/omarelgabrys-blog/database-database-options-part-10-380c6e4467d0)**.**

Before jumping into the databases, and the features they provide, we instead will start by why we would need a database, because they are designed to solve problems. *“So, what are these problems?”*…

**Why Do We Need A Database?**

When you have some data, and you want to store this data somewhere. This data could be anything. It could be about customers, products, employees, orders, …etc. This data could be in text format, numeric, dates, document files, images, audio, or video.

Maybe if you have data about the customers in your company, the first thing that comes to mind is to open a spreadsheet. Then you start writing whatever data you want to store.

It could be the customer name, id, position, and so on. You may add as many as customers, delete any of them later, or even modify them. And that’s it!.

**Graphical user interface, application, table, Excel

Description automatically generated**

Probably this is what comes into your mind when you hear the term “database”.

Now, we have a kind of data, and you stored them in spreadsheets in a way that satisfies your need. That might be OK, because just having data is not a good enough reason to need a database, and it’s not the problem. The problem is what comes next, and there’s a lot of potential problems.

What if you have a bunch of data, maybe 10,000 customers, “*Are you going to scroll down in the spreadsheet to get the 9999 customers?!*”, What if the security was a concern, “*Do you care about if someone else got access to your data?*”, What if you accidentally put redundant information, “*Is it fine to have duplicate information along with the spreadsheet?”.*

This takes us to the next question, “*When do we actually need a database?*”. Consider the following potential problems:

**Size**

You may have thousands or millions of rows of customers, or any piece of information.

**Accuracy**

*“Do you care if someone entered incorrect data?”.* If yes, nothing could actually prevent me from typing incorrect data into a spreadsheet.

**Security**

If the data is sensitive, and you need to restrict access to the data; It doesn’t need to be shared with everyone. In addition, *“Do you need to know who made every change at every point?”*.

**Redundancy *(ihtiyac fazlasi olma, gereginden fazla*)**

If the redundant data (having multiple copies of the same data) will lead to conflict, you would need to have only non-repeated unique data.

**Importance**

*“What if you had a disconnect or a crash, and you lost your data?”.*You’ve probably felt that pain before. And it’s unacceptable to lose important data like orders of a customer, allergies of a patient, flight bookings, …etc.

**Overwriting**

How about having more than one person overwriting the same data at the same time. How about 10 at the same time or 100 people at the same time?. You’ll end up with everybody overwriting everybody else’s changes.

**—**

If you are saying “Yes” to one of these problems, or all of them and more besides, to keep the data reliable (guvenilir), secured, and maintainable. So, you need to have a database. That’s what we are going to discuss here.

**What’s A Database?**

**A picture containing glass, table

Description automatically generated**

Probably this is what comes into your mind when you hear the term “database”.

It’s a structured system to put your data in that imposes rules upon that data, and the rules are yours because the importance of these problems changes based on your needs. Maybe your problem is the *size*, while someone else has a smaller amount of data where the *sensitivity*is a high concern.

It’s the things you can’t see that is going on in the background; the security, the enforced integrity of the data, the ability to get to it fast and get to it reliably, the robustness; serving lots of people at the same time and even correctly survive crashes and hardware issues without corrupting the data.

And that’s what we need to do here; understand how to describe our structure and define those rules, so all these invisible things will actually happen.

**Database Management System (DBMS)**

We often mistakenly say our database is *Oracle, MySQL, SQL Server, MongoDB*. But, they aren’t databases, they are database management systems (DBMS).

The DBMS is the software that would be installed on your personal computer or on a server, then you would use it to manage one or more databases.

The database has your actual data and the rules about that data, while the DBMS is the program that surrounds and manages your actual data, and it enforces the rules you specified on your data. The rules for example could be the type of the data, like integer or string, or the relationship between them.

**A picture containing graphical user interface

Description automatically generated**

**Database Management System (DBMS)**

In practice, it’s very common to have multiple databases. The database that deals with your order and customer information might be completely independent of your database that deals with human resource information. And in many organizations, you don’t just have multiple databases but multiple DBMS. Sometimes it’s because one DBMS is better at something than the other.

**There are different DBMS, and they are categorized under:**

* **Relational Database Management Systems**
* **Hierarchical Database Systems**
* **Network Database Systems**
* **Object-Oriented Database Systems**
* **NoSQL Database Systems**

**We are going to focus on relational database management systems (RDBMS). And here’s *Why? …***

* **They are the most commonly used ones.**
* **The principles we are going to discuss here are usable across all of them.**
* **If you know you are going to jump into NoSQL databases, most of the introductions assume you already understand relational database concepts and will use these concepts to explain what’s offered by NoSQL databases.**

***RDBMS are like Oracle, MySQL, SQL Server, SQLite, DB2, …etc.***

**Wrapping Up**

Now you understand why would we need a database, what’s a database and the difference between a database and DBMS. This will take us to the first step in understanding [**the database fundamentals in the next tutorial**](https://medium.com/omarelgabrys-blog/database-fundamentals-part-2-b841032243ac)**.**

**Database — Fundamentals (Part 2)**

The first step towards understanding the databases (specifically relational databases) is understanding the basic features.

Learning the fundamentals is always the most critical part —[kidsatthecreek](http://kidsatthecreek.com/wp-content/uploads/2009/07/Fotolia_54221145_Subscription_Monthly_M.jpg" \t "_blank)

*Wish you already came along the last part*[*Database — Introduction (Part 1)*](https://medium.com/omarelgabrys-blog/database-introduction-part-1-4844fada1fb0)

**Database & Tables**

**Database**

A database is a collection, or a set of tables. Each table has a formalized repeating list of data about one specific piece of information. For example a table for customers, students, orders, products, and so on. Visually, it’s often shown like a spreadsheet.

**Tables**

The table is the most basic building of a database. It’s the place where you will put your data, define their data type, and also their relationship with the other tables. It consists of rows and columns.

*You may hear the term “entity” instead of table, but we’ll use tables for simplicity.*

**Rows & Columns**

Within each table, every single **row**represents one single student, customer, order, or employee. But each of these rows is not free form. You must apply structure to this data.

So, you must say what every row is made of, and you do this by defining the **columns**in that table. And each column describes one piece of data. It gives it a **name**like *name*, *id*, *email*, *date of birth*, and a **type**, perhaps, a *text*, or a *date*, or a *number*.

Now, every row must follow that same structure, following that same format. It’s not allowed to deviate from the way that the columns are set up. And by defining these columns, we’re imposing rules on the data, and the DBMS won’t let us break them.

**In a nutshell,** columns define what’s the data that should be in the table, while the rows hold the actual values that you are going to *retrieve*, *insert*, *update*, and *delete*.

*You may hear the term “tuple” instead of rows, and also you may hear the term “attribute” instead of column.*

Before heading into the next topic, there’s something to be mentioned about columns, and that’s there are 3 types of columns:

**1. Simple**

It’s just a single value.

**2. Composite**

A value that’s composed of some other values. For example, you may have a name that’s composed of *first name, middle name,*and *last name.*

*Any composite attribute will be decomposed into separate simple attributes.*

**3. Multi-valued**

Multiple values for a single column. For example, the color of the car may be *black* and *red*.

*Multi-valued attributes will be extracted in another table. This will be discussed later in*[*Database Mapping*](https://medium.com/omarelgabrys-blog/af029e93cc1f)*.*

**Primary Key**

Now, if you have a long list of rows, it’s essential to have something that uniquely identifies each row, and that’s called the “primary key”. A primary key is a column of unique values for each row.

You may have more than one student with the same name, but you can’t have more than one student with the same primary key. And if you tried to insert a duplicate value, this will be disallowed by the DBMS.

*Usually you will see a primary key column called “id” of integer values.*

The primary key is either naturally exists or generated by the DBMS. It means, for example, by default every customer has a unique *SSN*, assigned by the company. So, the *SSN* uniquely identifies every customer.

But, you may have a table that holds information about some products, and they don’t have ids by nature. So, you will ask the DBMS to generate a new unique column, like *product id* and you may want to mark it as “auto-increment”.

*Auto-increment columns allows a unique number to be generated when a new record is inserted into a table by increment the value by 1 for each new record.*

Primary keys are very important, not only to uniquely identify the rows, but also we are going to use them to connect between the tables and form relationships.

**One-to-Many Relationship**

Most of your tables will be naturally be connected, so we need to have a relationship between them. You’re not trying to invent relationships that don’t exist, you’re trying to describe what’s already there.

An example of a one-to-many relationship could be, a customer can place more than one order, but, an order is only placed by one and only one customer. You can’t have an order that’s placed by more than one customer.

The customer’s information exists in a table, and the order information also exists in another table, but they have a relationship. So, “*How can we define a one-to-many relationship?*”.

Do you remember when we said primary keys are used to connect between tables? So, here we will add a new column to the orders table (many side) called “foreign key”. This column has the primary key values of customers.

Table

Description automatically generated

One-to-Many Relationship

The values of the foreign key column could be redundant because a customer can place more than one order. And a customer can have one, or more or even nothing number of orders.

The benefit of defining this relationship is, it allows us to ask some questions like: “*What are the orders placed by a customer whose first name is “Smith”?*”, or go the other way, “*Who is the customer who placed the order of id 1012?*”.

**Many-to-Many Relationship**

You will be using a one-to-many relationship a lot between your tables. But, there is another way to relate tables together. What if you have a student, and each student can be enrolled in one or more courses, and at the same time, each course can have one or more students enrolled in it.

So, it’s no longer a one-to-many, it’s many from both sides. And, as usual, the primary keys are the way to connect the tables. But, this time we will create a new table. It’s usually called a “junction” or “linking” table.

This table exists only to connect the two tables, it has two foreign keys, one points to the primary key of the students table, and the second one points to the primary key of the courses table.

And the two foreign keys together will form the primary key of the new table.

Table

Description automatically generated

Many-to-Many Relationship

Now, using this relationship, we can know the courses of a particular student, or go the other way and get the students who are enrolled in a specific course.

*Officially, there is a third kind, a****one-to-one relationship****that is possible, but it’s not common.*

*If you think about it, if one row in one table is pointing to one and only one row and another table, well, you might as well just combine those tables so it’s one row in both places. This will be discussed later in*[*Database Mapping*](https://medium.com/omarelgabrys-blog/af029e93cc1f)*.*

These types of relationships (one-to-many, many-to-many, or one-to-one) are usually called “Cardinality Constraints”. They are constraints that specify the maximum participation between tables.

There are some other constraints like the “Existence Dependency Constraint” (also called “Participation Constraint”). They specify the minimum participation; zero (optional participation), or one or more (mandatory participation).

*A relationship is when we say an employee “works on” a department, but, the constraints is when we say, one employee can work in many departments.*

**Structured Query Language (SQL)**

It’s the language that’s used to *create*, *read*, *update*or *delete*data (falls under the acronym *CRUD)*, and also to define the databases themselves.

It’s not a programming language, it’s considered as a declarative query language. You just need to say what you want, and you let the DBMS handle how that’s actually done for you.

On the other hand, in the programming language, you would have to do write the steps to do this; maybe some loops, and check to see if this is the required data or not.

We are going to cover [SQL](https://medium.com/omarelgabrys-blog/230a1808ec96) in later tutorials, the syntax is very easy to understand, but, most DBMS have their own different implementation of the SQL language, although basic SQL knowledge works with all of them.

**Wrapping Up**

We’ve covered the basic fundamentals of a relational database. Now, we are going to take this further and walk through the steps needed to create your actual database. Next is the “[Database Design Process](https://medium.com/omarelgabrys-blog/database-design-process-part-3-7b5fafc78774)”.

**Database — Design Process (Part 3)**

We’ll walk through the steps to design and create a database.

*Wish you already came along the last part*[*Database — Fundamentals (Part 2)*](https://medium.com/omarelgabrys-blog/database-fundamentals-part-2-b841032243ac)

**Database Design Process**

The processes here aren’t the same as the agile model or iterative approach. They are defined steps to end up having a fully defined database, with its constraints, and structure.

There is no place for many changes because they are going to cost you a lot. So, you need to be specific and take things step by step. And, here’s the steps:

**1. Requirements Gathering**

Understanding what you want to do, and what you have is essential before you can dive into designing a database. We’ll look at the steps in this article.

**2. Conceptual Design** (kavramsal tasarim)

We specify the entities, columns, and their relationship. We may use an entity relationship (ER) diagram to visualize the database.

**The output is:**A conceptual schema (described using a conceptual data model like ER model).

**3. Logical Design**

It’s concerned about data model mapping; mapping a conceptual schema (like ER model) into logical schema to provide a much detailed description.

**The output is:** A logical schema (described using a logical data model specific to the DBMS like relational model).

**4. Physical Design**

It describes the details of how data is stored. You start by defining (already modeled) tables, how the data is stored, define relationships, … in DBMS.

This requires dealing with the DBMS, and could involve (icermek) SQL.

**The output is:** An internal (physical) schema (described using a physical data model).

*A database****schema****is a description of a database structure, data types, and the constraints (kisitlayici, sorlayici) on the database.*

**Requirements Gathering**

The first step in whether you are building a mobile application, desktop, or any kind of software is to gather the requirements about what’s actually needed. And, here are the steps:

*Take a look at*[*Requirements Engineering*](https://medium.com/omarelgabrys-blog/requirements-engineering-introduction-part-1-6d49001526d3)*in Software Engineering.*

**1. Why? and is it feasible** (uygulanabilir, olasi, muhtemel)**?**

So, first, you study “*Why we need a database?*”, and if it’s feasible or not. You need to check if it could be implemented under the current budget, under the current technical skills of your team, within the defined schedule, And if it does contribute (katkida bulunmak) to the whole organization's objectives or not.

**2. Collecting the requirements**

If the answer was Yes!, then you start collecting the requirements. You can do this by having interviews with all the stakeholders; anyone who will use the system.

And try to get any kind of existing data, any kind of spreadsheets, this will help you to in building the database.

And if there is an existing (var olan, mevcut) database, then ask, what’s wrong with it? what’s right with it? These questions will help to avoid any of the current problems.

**3. Group related requirements**

Now, you may need to group related requirements together, resolve any conflicts (catisma, celiski) , or any ambiguity (anlam bozuklugu, anlasmazlik) by negotiation with the customers.

**4. Specify & Verify the requirements**

Finally, you specify the requirements, or in other words, you write the requirements in a document, and validate (dogrulamak, gecerli kilmak) the requirements with the customer to make sure that everything is on the right track (yol, yorunge).

**Wrapping Up**

This was a quick tutorial before getting into the first step in the Database Design; “[Conceptual Design](https://medium.com/omarelgabrys-blog/database-database-modeling-conceptual-design-part-4-645545a74a4b)”.

**Database — Database Design: Conceptual Design (Part 4)**

The conceptual design provides a high-level description that’s close to the way many users perceive data.

*Wish you already came along the last part*[*Database — Design Process (Part 3)*](https://medium.com/omarelgabrys-blog/database-design-process-part-3-7b5fafc78774)

We’ll start by identifying the tables, their columns, and their relationships with other tables.

**Identifying The Tables and Relationships**

**Identify Tables**

As we’ve mentioned earlier, a database consists of tables, each table has related information.

You start identify the potential objects from your requirements. Objects could be anything; customers, students, courses, orders, employee, department, project, and so on. Objects also could be things that doesn’t exist in the real word, things like the comments in your blog, the posts, categories, and so on.

These objects will be your tables. But, you need first to refine your chosen objects. “*How can we do that?*”, here you are:

* Remove any duplicates, you may find same objects with different names, but they actually do the same thing.
* You may need to combine some objects or even splitting them into some other objects.
* You may identify an attribute(column) as an object instead. An attribute is a piece of info that’s related to a specific object.

**⚠ Conflict: Working In An OO Language**

Take care if already are doing a similar kind of process for your application, and you’re working in an object oriented language, because you often ask similar kinds of questions.

Sure, there is some crossover here, for example, in a sales application, we are likely to have both an order table defined in my database and an order class defined in my application, but they are different.

In the relational database, we are concerned about the data that needs to be saved. Our focus is not on things like methods and behavior, or inheritance, polymorphism, like it might be in an object oriented language.

So, we’re concerned here just about our tables, and their relationships.

**Identify Relationships**

Next is to identify the relationships between those tables. You have a customer *places*orders, or an employee *works in* a department.

You can use the symbol “1-M” to denote one-to-many relationship, and “M-N” for many-to-many relationship, and also “1-1” for a one-to-one relationship.

*Remember? These types of relationship (one-to-many, many-to-many, or one-to-one) are constrains that specify the maximum participation between tables.*

**Identifying The Columns and Their Data Types**

After identifying the tables in our application, now it’s time to determine what’s the data needs to be in each table.

A column, or an attribute defines every piece of information. It defines the **name**and the **type**for each piece of data in a table. For example, an employee table may have *name*, *address*, *salary*, …etc.

Then, determine the **data type** for each column. Is it text or character data, or is it numeric? Is it a date, a time, or even binary data like an image, or a piece of audio or video?.

There are many data types, and this depends on the DBMS you are going to use, but, when you get into using a specific DBMS, you’ll just have a summary of all the available data types until you get to know them.

*Data types in a database are not the same as in a programming language.*

**Length**

Columns like *name*, and *address,*we need not to just specify it’s data type, but also specify if it should be with a fixed length, or variable length.

**Null**

Another option is to specify whether the column should always have a value, or not. Maybe it’s fine if we didn’t insert the address of a specific employee, but, it’s impossible to insert an employee without a*name*.

In this situation, you can choose either the column should be *NULL*in case there is no values inserted, or*NOT NULL* so it must have a value.

*Usually the DBMS will insert the default value in case you didn’t insert any value to a column that’s defined as NOT NULL.*

**Default Values**

Do you want a default value to be inserted for a specific column in case you didn’t. For example, an employee could have a default value set to “0.0” in the *salary*column in case we didn’t insert it.

**Other Options …**

We might add as well some other constraints. For example, we may force a column value to match a specific pattern, like matching an email address or a phone number or a credit card number.

*DBMS wants to know these specifics so it can be efficient about storing and indexing them, and enforce your rules on those columns, that your data will stay valid and consistent.*

**— Primary Keys**

If you are following along from the previous tutorials, you should know what’s the primary key. Now, you choose which column will be the primary key. You may have a primary key that’s naturally exists as we discussed before, or ask the DBMS to generate it for you.

It’s typically an integer number. You can also have this primary key as “auto-increment”; means every time you insert a row, the primary key will be generated and incremented automatically, hence no need to insert it.

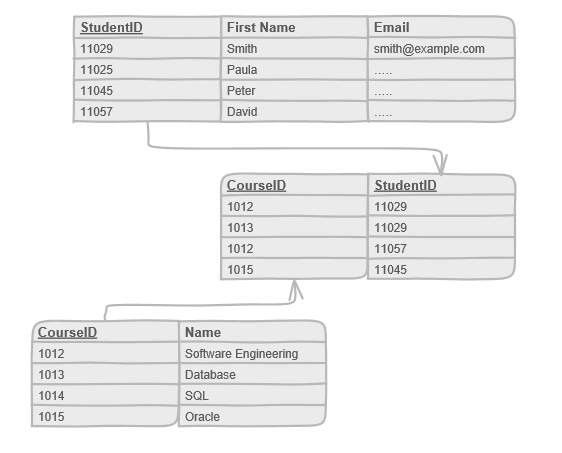
*Sometimes it’s more convenient to generate a primary key (system-generated), like an employee id* *even if you do have a piece of real data that uniquely identifies a row, like a social security number … Why?*,*because this is a number that could be passed around in emails or printed on an ID card, unlike a sensitive piece of data, like a social security number.*

**Composite Primary Keys**

Another option for the primary keys is that you can have more than one column together they form a primary key. This is when one value does not uniquely identify a row, but two (or more) values do.

You have already seen this in many-to-many relationship, when we created a “linking table”. It has two foreign keys from the participating tables, and the two foreign keys; c*ourse id*and s*tudent id*aretogether form a primary key together.

It means, a student can have more than one course, and a course can be taken by one or more student, but there can’t be a row with the same *student id*, and the same *course id*.



Many-To-Many-Relationship

*Now, it’s true that it might sometimes be more useful or even more convenient to generate a primary key column (by the DBMS) anyway.*

**Database, Table & Column Names**

Now you might be asking, “*What’s the best common way used to name my tables and columns?*”. Actually it depends on you, some people prefer Pascal case; upper case first letter with singular nouns for database, tables, and columns.

And some other prefer the lower case letters with underscores (plural nouns) for tables, and lower case letters with underscores (singular nouns) for database and columns.

The thing is, just choose one convenient way for you, and stick to it.

**Graphical Tools**

All you really need to model a database, at least initially, is pencil and paper, although there are some tools you can use, like *Visio*from Microsoft.

The tables, columns, and their relationships could be sketched using “Entity Relationship Diagram (ERD)”, which will be covered in the next tutorial.

**Wrapping Up**

After identifying your table, columns, data types, relationships, and primary keys. This takes us to the next step in the database design, which is the “Logical Design”.

But, before diving into it, we are going to give a quick tutorial about the “[Entity Relationship Diagram (ERD)](https://medium.com/omarelgabrys-blog/database-modeling-entity-relationship-diagram-part-5-352c5a8859e5)”.

**Database — Modeling : Entity Relationship Diagram (ERD) (Part 5)**

A common approach to sketch the entities and their relationships.

**Let’s take an example of a company database**

Diagram

Description automatically generated

ERD of a company database — From [Fundamentals of Database Systems](https://www.amazon.com/Fundamentals-Database-Systems-Ramez-Elmasri/dp/0136086209) by Ramez Elmasri, [Lecture Slides](https://www.cs.purdue.edu/homes/ake/cs348/index6.html)

*Wish you already came along the last part*[*Database — Database Design: Conceptual Design (Part 4)*](https://medium.com/omarelgabrys-blog/database-database-modeling-conceptual-design-part-4-645545a74a4b)

**Entity Relationship Diagram**

An entity relationship model, also called an entity-relationship (ER) diagram, is a graphical representation of entities (which will become your tables) and their relationships to each other.

***Database modeling****is the process of creating a data model.*

**Entity**

A simple rectangular block represents a table.

Diagram

Description automatically generated with medium confidence

**Relationship**

It’s sketched using the diamond shape.

Diagram

Description automatically generated

Relationship

You can sketch the type of the relationship, whether one-to-many using “1-M”, or many-to-many using “M-N”, or one-to-one using “1–1”.

The lines connecting an entity with a relationship whether single or double line refers to another constrain (we talked about earlier) called “Existence Dependency Constraint” (also called “Participation Constraint”).

So, for example, if an employee **must** work for a department, this is sketched by double lines, and called “total or mandatory participation”. And if an employee **may or may not** manage a department, this is sketched by a single line and called “partial or optional participation”.

*A relationship itself can have attributes. In which table we will include these attributes will be discussed in*[*Database Mapping*](https://medium.com/omarelgabrys-blog/af029e93cc1f)*in the next tutorial.*

**— Recursive Relationship**

It’s a relationship with the same entity. For example, an employee may act as a *supervisor*for one or more employees, while an employee may be *supervised by*an employee (it’s a one-to-many relationship).

Diagram

Description automatically generated

Recursive Relationship

**Weak Entity**

A weak entity is simply an entity where its existence depends on another entity. You can’t logically have dependent(son, daughter, ..etc.) with the absence of the employee table. It’s sketched the same as a normal entity but with double lines.

Diagram

Description automatically generated

Weak Entity

**— Partial Key**

A weak entity has what’s called a “partial key”. It’s one or more attributes that uniquely identify a weak entity for a given owner entity. In our example, the dependent *name*is unique for every employee.

It’s sketched the same as a normal attribute, but, with a dashed underline.

*It’s called partial because it can’t be a primary key on it’s own, it needs another column, which is the foreign key of the owner entity. This will be discussed in*[*Database Mapping*](https://medium.com/omarelgabrys-blog/af029e93cc1f)*in the next tutorial.*

**— Weak Entity Relationship**

Also whenever there is a relationship between a weak entity and another entity, it’s sketched the same as the one above, but with double lines.

Diagram

Description automatically generated

Weak Entity Relationship

**Attribute**

An attribute or a column (simple) represents a piece of data in the table, like an address, salary, and date.

A close-up of a logo

Description automatically generated with medium confidence

Attribute

**Primary Key**

It’s sketched the same as a normal attribute, but, with an underline.



Primary Key

**Composite Attribute**

A value that is composed of some other values, for example, you may have name that’s composed of ( *first name, middle name,*and *last name*).

Diagram

Description automatically generated

Composite Attribute

*We don’t define one column for name (if it’s*[*composite*](https://medium.com/omarelgabrys-blog/database-fundamentals-part-2-b841032243ac#da00)*), instead we split it up into simple separate columns; first name, last name. One reason is because you may do something different with the last name(or first name), and you want to get to it by itself. This will be discussed in*[*Database Mapping*](https://medium.com/omarelgabrys-blog/af029e93cc1f)*in the next tutorial.*

**Multi-valued Attribute**

Multiple values for a single column, for example, the locations of a department may have more than one value at the same time.

Diagram

Description automatically generated

Multi-Valued Attribute

*Multi-valued attributes will be extracted in another table. This will be discussed later in*[*Database Mapping*](https://medium.com/omarelgabrys-blog/af029e93cc1f)*.*

**Derived Attribute**

When you have a column where its value could be calculated from another column. Thus, there is no need to insert its value every time. For example, we can know the number of employees working in a specific department by counting the number of rows.

A picture containing text, saw, tool

Description automatically generated

Derived Attribute

This will be done automatically by the DBMS, and it will lead to having consistent values rather than accidentally inserting a wrong value.

**Symbols Summary**

Here is a summary of all the symbols in the ERD.

Diagram

Description automatically generated

ERD symbols — From [Fundamentals of Database Systems](https://www.amazon.com/Fundamentals-Database-Systems-Ramez-Elmasri/dp/0136086209) by Ramez Elmasri, [Lecture Slides](https://www.cs.purdue.edu/homes/ake/cs348/index6.html)

**Wrapping Up**

Using the entity relationship diagram to sketch your database is a common way to visualize your tables, columns, and their relationship.

Now, It’s time to translate the entities, relationships, multi-valued attributes, into actual tables. Next is the “[Logical Design (Data Model Mapping)](https://medium.com/omarelgabrys-blog/database-modeling-logical-design-part-6-af029e93cc1f)”.

**Database — Design: Logical Design (Part 6)**

**The logical design is about mapping of entities, relationships, and multi-valued attributes into a logical schema.**

**Diagram

Description automatically generated**

**Result of mapping the company ER schema into a relational schema. — From**[**Fundamentals of Database Systems**](https://www.amazon.com/Fundamentals-Database-Systems-Ramez-Elmasri/dp/0136086209)**by Ramez Elmasri,**[**Lecture Slides**](https://www.cs.purdue.edu/homes/ake/cs348/index6.html)

***Wish you already came along the last part***[***Database — Modeling : Entity Relationship Diagram (ERD) (Part 5)***](https://medium.com/omarelgabrys-blog/database-modeling-entity-relationship-diagram-part-5-352c5a8859e5)

**We are going to walk through defined steps to map our conceptual schema (described by ER model) we have just created into logical schema (described by relational model).**

**When we sketched our ER diagram, we showed that some kind of relationship exists between our entities, but to get closer to actually building this in a database, we need to translate these relationships (along with others like multi-valued attributes) in our database.**

**All the relationships we have identified are going to be implemented in the database by either creating a new table, or by just creating a new column, or maybe there are some other options we are going to cover here.**

**So, here are the steps:**

**1. Mapping of Regular Entity Types**

**For each entity, create a table that includes all of its simple attributes. Then, choose the primary key, if it’s composite, then a set of simple attributes will together form the primary key.**

***The composite column will be decomposed into separate simple columns.***

**2. Mapping of Weak Entity Types**

**For each weak entity, create a table that includes all of it’s simple attributes. And include a foreign key points to the primary key of the owner entity, where the foreign key and partial key will be the primary key of the weak entity.**

***A***[***partial key***](https://medium.com/omarelgabrys-blog/database-modeling-entity-relationship-diagram-part-5-352c5a8859e5#29ef)***uniquely identify a weak entity for a given owner entity.***

**3. Mapping of 1:1 Relationship Types**

**There are three ways:**

1. **Foreign Key approach: Choose the primary key from either one of the entities, and make a foreign key in the other entity referencing the primary key of the first one.**
2. **Merged relation option: Merge both entities, since every row in any entity will have a corresponding row in the other entity.**
3. **Cross-reference or relationship relation option: Create a third table, that has two foreign keys from the primary keys of both entities (overkill!).**

**4. Mapping of 1:M Relationship Types**

**Considering the example we have been using, where each employee *works for* only one department, while a department can have more than one employee.**

**Now, in order to map this relationship, we add a foreign key in the employee table (many side), which in turn will point to the primary key of the department table (1 side).**

***Also include any simple attribute of the 1-M relationship in the many side.***

**Table

Description automatically generated**

**Mapping 1:M Relationship**

**Recursive Relationship**

**Don’t let the recursive relationship trick you!. You would treat it like any other relationship between one entity and another.**

**In our example, we had a one-to-many recursive relationship called *supervision*between the employee entity and itself.**

***In other cases, you may have a many-to-many recursive relationship, or something else.***

**The same thing goes here, just add a foreign key in the many side (which is the employee table), which points to the primary key of the 1 side (which is also the employee table).**

**Diagram

Description automatically generated**

**Mapping of one-to-many recursive relationship**

**Now, if an employee is *supervised by*another employee, *SUPERSSN*will be assigned to the *supervisor*employee’s id. Otherwise, it’s given null.**

**More About Foreign Keys**

* **Foreign key’s values must exist in the primary key at any time. That’s it, a foreign key can’t have a value that’s not in the primary key it refers to.**
* **Foreign key can have different values, and could be not unique.**
* **Foreign key may be primary key but, one foreign key can’t be, two foreign keys can; composite primary keys.**
* **Foreign key may have different name from primary key.**
* **Foreign key’s data type must match with data type of the primary key.**
* **Foreign key may contain *NULL*(based on rules defined by the user).**

**5. Mapping of M:N Relationship Types**

**A good example for this relationship, is every employee can *work on*one or more project, and every project can have one or more employee involved in this project. So, it’s many from both sides.**

**Now, in order to map this relationship, we create a new table, this table exists only to connect the employee and project tables. You start by adding two foreign keys, each one will point to a primary key of one of the two tables. The two foreign keys together will form the primary key of the new table.**

**There may be an employee called *Adam*who works on *Project A*, and *Project B*. And another employee who’s working on *Project A*. But, you can’t have duplicates; meaning, you can’t have two rows with the same employee, and the same project.**

**Table

Description automatically generated**

**Mapping of M:N Relationship**

**Simple Attributes of M:N Relationship**

**If there are simple attributes related to the M-N relationship, you add them into the new table created. As an example, here we have *Hours*, that is every employee who works on a specific project has a specific working hours.**

**6. Mapping of Multi-valued Attributes**

**Remember? A multi-valued attribute is a set of different values.**

**As an example, a department may have different locations, thus it will have different location values for each department.**

**So, we create a new table that has a foreign key points to the primary key of department, and another column represent the multi-valued attribute (one-to-many relationship). This is done for each multi-valued attribute.**

**The foreign key and the multi-valued attribute together will form the primary key of the new table.**

**Table

Description automatically generated**

**Mapping of Multi-valued Attributes**

***If the multi-valued attribute is a composite attribute; consists of more than one attribute. You decompose the composite attribute into simple ones.***

**7. Mapping of N-ary Relationship Types**

***“What if you have a relationship that connects more than two tables?*”. It’s almost the same solution as we did in M:N relationship.**

**Create a new table, that has the foreign keys from the primary keys of all participating entities, the foreign keys together will form the primary keys of the new entity. Also include any simple attribute of that relationship in the new table.**

**Relational Integrity Constraints**

**There are some constraints**[**we can define**](https://medium.com/omarelgabrys-blog/database-structured-query-language-part-8-230a1808ec96#90d7)**and they are enforced by the DBMS to keep your data valid and meaningful across all of your tables. There are three main constraints:**

1. **Unique Key Constraints: A unique column or a group of columns that uniquely identify each row can’t be duplicated (although they could be *null*).**
2. **Entity Integrity Constraints: A primary key uniquely identifies each row, can’t be duplicated, and can’t take *null*.**
3. **Referential Integrity Constraints: A foreign key … (see**[**above**](https://medium.com/omarelgabrys-blog/database-modeling-logical-design-part-6-af029e93cc1f#6d55)**).**

***Deleting rows from the M-side table in one-to-many relationship, or, deleting rows from the linking tables in many-to-may relationship doesn’t violate the referential integrity constraint.***

**Wrapping Up**

**What we are going to do next is to take what we have implemented further, organize the tables, make our database easier to work with and more reliable. This process is called “**[**Normalization**](https://medium.com/omarelgabrys-blog/database-normalization-part-7-ef7225150c7f)**”, and there are 3 steps to achieve it.**

**Database — Normalization (Part 7)**

The process of organizing your database through a set of defined steps.

*Wish you already came along the last part*[*Database — Design: Logical Design (Part 6)*](https://medium.com/omarelgabrys-blog/database-modeling-logical-design-part-6-af029e93cc1f)

This is a process where you take your database design, and you apply a set of formal criteria of rules called “Normal Forms”. And we step through them first normal form, second normal form, and third normal form.

You usually will end up creating a few new tables as part of the process. But, the end result of normalization is to make your database easier to edit, easier to maintain, and preform operations, remove any duplicates, & more reliable to work with.

**First Normal Form**

First normal form requires that every column should have one, and only one value; there is no multi-valued attribute, and there shouldn’t be repeating groups of data.

So, for example, if you have a customer, and this customer can have two emails; the main email and the alternative one. Now, you can just put two columns in your table, each column represent on of these emails (repeating group ☹).

Alternatively, you can have two rows for this customer, one with the main email, and the other row with the alternative email, while keeping the other columns’ values as they are (repeating group ☹).

But, “*What if you want to extend this, and add more emails?*”. We don’t want to have a table that has repeating group of data.

So, what are going to do is to extract the customer emails in a new table. This new table will have a foreign key points to the primary key of the original table, and another column for the email values. Both the foreign key and the email value are together form the primary key of the new table.

Table

Description automatically generated

First Normal Form

Now, we can say that our tables are in the first normal form, let’s make it in the second normal form.

**Second Normal Form**

The problem arise when you have a composite primary key in a table. The second normal form requires that all of the non-*primary*columns have to be dependent on the entire (composite) primary key.

An example could be, let’s say you have a table that stores books information. A book could have a name, author, and date released.

According to the requirements, we can have book that’s written by more than one author, but we can’t have more than one book with the same book name and the same author. So, both name and author form a composite primary key.

Table

Description automatically generated

A book table

Now, the date column relies only on the name of the book, it doesn’t care about who wrote this book, it cares about the name of the book. And if you noticed we’ve written the date “2011” more than one time with “Database Systems” book. This is not a good practice when you have duplicates.

So, what we are going to do is to create a separate table, with one-to-many relationship with the original table, it has the book name as a primary key, and all the dependent attributes (like date).

Now, the book name in the book table is a foreign key that points to the primary key of the new created table, and every non-primary column in the book table (if exist) depends on the whole composite primary key; name and author, and not one of them.

Table

Description automatically generated

Second Normal Form

Now, we can say that our tables are in the second normal form, let’s make it in the third one.

**Third Normal Form**

The third normal form is concerned with the non-key attribute that rely on another non-key attributes, and not the primary key.

An example for that could be with the course table, where every course has an name as a primary key, room number and capacity.

Table

Description automatically generated

Third Normal Form

Now, it’s obviously noted that the capacity depends on the room, it has nothing to do with the course name. This means we have a non-key attribute that depends on another non-key attribute.

Why this sounds like a problem? Well*, “What happens if somebody reached into this table, and they changed that room for a specific course, but they didn’t change the capacity?”.*

So, what we are going to do is the same, we will create a new table with one-to-many relationship with the original table, it has the room number as a primary key, and all the dependent attributes like the capacity.

Now, the room number in the course table is a foreign key that points to the primary key of the new created table.

Table

Description automatically generated

Third Normal Form

**Derived Attribute**

Another example of third normal form is when you have a (non-primary) [derived](https://medium.com/omarelgabrys-blog/database-modeling-entity-relationship-diagram-part-5-352c5a8859e5#b18b) attribute that depends on other non-primary attributes.

We would remove that derived column form this table, as we can figure it out when we need to. One of the main reasons for this is to prevent any conflicts, any inaccurate data to be inserted mistakenly.

Many database systems offer you the option of defining a computed or calculated column. It’s not actually stored in the database, it is a convenient read-only fiction. It’s value is automatically calculated based on the other columns in the table, and you may find that useful from time to time.

**Denormalization**

Sometimes you will hit the situation where you will break the normalization rules for performance improvements, because normalization is often involves splitting data into multiple tables.

If you are following along, you remember at the first normal form, when we explained the example of emails. *“What happens if we added two email columns to the customer table?.”*

Technically, this can be described as breaking first normal form. It’s a repeating group. But in practice, you may find it more convenient to just allow an *email1*and *email2*columns, because you’re sure there won’t be a flexible number of email addresses.

This will improve your performance, and save you from creating more tables.

**Wrapping Up**

Now, we can say we have an organized, more reliable database, and this will improve the consistency of your data. In the next tutorial, we are going to cover basics of “[Structured Query Language (SQL)](https://medium.com/omarelgabrys-blog/database-structured-query-language-part-8-230a1808ec96)”.