# **Database Design**

## **Objectives**

1.1 Design tables for storing data:

Identify entities, rows/records, columns/fields

1.2 Identify the appropriate primary key:

Primary key, composite/compound key

1.3 Choose data types to meet requirements:

Definition and importance of data types; how data types affect storage requirements; data types for storing text, numbers, dates & times, and Boolean values

1.4 Design relationships between tables:

How to establish relationships using primary and foreign keys, entity relationship diagrams (ERDs), referential integrity

1.5 Normalize a Database:

Reasons for normalization, how to normalize a database to third normal form (3NF)

1.6 Identify data protection measures:

Backups, restore, principle of least privilege, GRANT, WITH GRANT OPTION, REVOKE, purpose of roles

## **Reading Materials**

SQL Primer - An Accelerated Introduction to SQL Basics: Chapters 1, 2, 3, 7 & 16

### Chapter 1: An Introduction to SQL

Overview

Modern society is driven by data. Whether it is at a personal level, like a notebook containing scribbled notes; or at a countrywide level like Census data, it has permeated all our workflows. There is always a growing need to efficiently store and organize it so that meaningful information can be extracted out of raw data.

A **database** is nothing but a collection of organized data. It doesn't have to be in a digital format to be called a database. A telephone directory is a good example, which stores data about people and organizations with a contact number. A to-do list is also a rudimentary form of a database. With ever-larger amounts of data being collected about even the most mundane of processes, digital databases have become increasingly important since their inception in the 1960s.

Software that is used to manage a digital database is called a ***Database Management System (DBMS)*.** When you hear someone talking about *PostgreSQL* or *MySQL*, they are referring to a DBMS. A database is what is created when you use the DBMS software to store data about topics that make sense to you or your organization. For example, your company may use *PostgreSQL* to store inventory information about cellular phones – the product that you sell. In this case, you have created an inventory *database* using *PostgreSQL* as your DBMS.

The Relational Model and SQL

Data comes in myriad shapes and sizes, and every context generates data in a different way. The data generated by a bank keeping a record of account balances is different from keeping track of members of a family tree. But for a DBMS to provide uniform data management and reporting capabilities, we must adhere to a data organization structure or *data model*.

The most prevalent database organizational model is the *Relational Model*, developed by Dr. E. F. Codd in his groundbreaking research paper – *A Relational Model of Data for Large Shared Data Banks* in 1970. In this model, the data to be stored is organized in a tabular format with rows and columns. Each row inside a table represents a distinct record with the column headings specifying the corresponding type of data stored. This is not unlike a spreadsheet where the first row can be thought of as column headings and the subsequent rows storing the actual data.

A database would typically consist of more than one table, each with different column headings. There may be certain columns that are common between tables, but this is a topic we will approach later in the book.

**W**hat does the word *relational* in relational database mean?

It is a common misconception that the word relational implies a relationship between the tables. A relation is a mathematical term that is roughly equivalent to a table itself. When used in conjunction with the word database, we mean to say that this particular system arranges data in a tabular fashion.

A possible origin of this misconception might have been the *set relation* command in *dBase*, a DBMS from the 1980s. That command indeed was used to create linkages between tables, but it has nothing to do with relational theory.

SQL stands for **Structured Query Language**, and it is the de facto standard for interacting with relational databases. Almost all database management systems you'll come across will have an SQL implementation. SQL was standardized by the American National Standards Institute (ANSI) in 1986 and has undergone many revisions, most notably in 1992 and 1999. However, all DBMS's do not strictly adhere to the standard defined but rather remove some features and add others to provide a unique feature set. Nonetheless, the standardization process has been helpful in giving a uniform direction to the vendors in terms of their database interaction language.

While SQL is a computer language, it is not like the other programming languages that you may have heard of like Python or C. Such programming languages are generic in nature, suitable for a wide variety of tasks from programming basic calculating systems to advanced simulation models. SQL is a special purpose query language meant for interacting with relational databases. It has no use other than this context.

This does not mean that it is the only database query language to exist. In the 1980s, another language called *QUEL* from *Ingres* was fairly popular, but the standardization effort around SQL cemented its position. In recent years, we have seen a large number of non-relational databases being developed under the umbrella term of *NoSQL*. Most of their query languages, however, bear some resemblance to SQL even though their data model varies significantly from the relational model: <https://www.seas.upenn.edu/~zives/03f/cis550/codd.pdf>

Advantages of using SQL

* It is *standardized* – no matter which relational database you choose, it will have an SQL query interpreter built in. The sheer popularity of SQL makes it worth everyone's time who interacts with a data system.
* It has a reasonable English-like syntax. None of the painstaking detail of programming languages like C or Java have to be specified when using SQL. It is concise, easy to understand, and easy to write database queries with. It is *declarative* in nature, meaning you only have to declare what you want to achieve rather than going over the steps to achieve the results.
* It allows a uniform way to query and administer a relational database. Many of the database administration commands are standard SQL commands making the transfer of skills much easier.
* It is *mature* – SQL has been around for over 35 years. While many new features have been added to it, the core of SQL has largely been unchanged. You can derive a lot of utility knowing a few basic SQL concepts and commands, and they will serve you well into the future.

SQL Commands Classification

SQL is a language for interacting with databases. It consists of a number of commands with further options to allow you to carry out your operations with a database. While DBMS's differ in the command subset they provide, usually you would find the classifications below.

* **Data Definition Language (DDL)**: *CREATE TABLE, ALTER TABLE, DROP TABLE, etc*. These commands allow you to create or modify your database structure.
* **Data Manipulation Language (DML)**: *INSERT, UPDATE, DELETE*. These commands are used to manipulate data stored inside your database.
* **Data Query Language (DQL):** *SELECT*. Used for querying or selecting a subset of data from a database.
* **Data Control Language (DCL):** *GRANT, REVOKE, etc*. Used for controlling access to data within a database, commonly used for granting user privileges.
* **Transaction Control Commands:** *COMMIT, ROLLBACK, etc*. Used for managing groups of statements as a unit of work.

Besides these, your database management system may give you other sets of commands to work more efficiently or to provide extra features. But it is safe to say that the ones above would be present in almost all DBMS's you encounter.

Explaining Tables

A *table* in a relational database is nothing but a two-dimensional matrix of data where the columns describe the type of data, and the row contains the actual data to be stored. Have a look at [Table 1-1](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=635813830&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#) to get a sense of the visualization of a table in a database:

| ID | Language | Author | Year |
| --- | --- | --- | --- |
| 1 | Fortran | Backus | 1955 |
| 2 | Lisp | McCarthy | 1958 |
| 3 | Cobol | Hopper | 1959 |

The above table stores data about programming languages. It consists of four columns (id, language, author, and year) and three rows. The formal term for a column in a database is a *field* and a row is known as a *record*.

**Note:** The example tables in this book primarily deal with programming languages, their authors, and the year they were created. We could have used database query languages, but they are far fewer in number. Our computer hardware and technologies have changed quite a bit since the 1950s and 1960s, but the early programming languages from that era still have a lasting impact on the programming languages of today. *Lisp* – imagined by John McCarthy in 1958 is still alive in the form of *Common Lisp, Scheme*, and *Ciojure*. Even *Fortran* still sees regular use in scientific computing.

There are two things of note in the example table. The first one is that the *id* field effectively tells you nothing about the programming language by itself, other than its sequential position in the table. The second is that though we can understand the fields by looking at their names, we have not formally assigned a data type to them, that is, we have not restricted (not yet anyways) whether a field should contain alphabets, or numbers, or a combination of both.

The ***id*** field here serves the purpose of a ***primary key*** in the table. **It makes each record in the table unique**, and its advantages will become clearer in chapters to come. But for now consider this, what if a language creator made two languages in the same year; we would have a difficult time narrowing down on the records. An *id* field usually serves as a good primary key since it's guaranteed to be unique, but usage of other fields for this purpose is not restricted.

A key concept of tables is that they are conceptual in nature and may not have any bearing upon the actual files where the data is stored. When users create a spreadsheet, they associate a file name with the spreadsheet and place it somewhere on their disk. But relational databases hide all these details from the user. The physical storage of a table on the disk might be to a single file, or to many files, or even have a relationship of storing many tables in a single file. It is the responsibility of your DBMS to provide a way to read and write to tables: <http://jmc.stanford.edu/articles/recursive/recursive.pdf>

Data Types in SQL

Just like programming languages, SQL also has ***data types* to define the kind of data that will be stored in its fields.** In the table given above, we can see that the fields *language* and *author* must store English language characters. The *id* and *year* fields both store whole numbers.

The commonly used data types you will encounter in subsequent chapters are shown in [Table 1-2](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=176507341&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#).

Character types char, varchar

Integer values integer, smallint

Decimal numbers numeric, decimal

Date data type date

A string of characters is usually stored in either ***char*** or ***varchar***. The former reserves as much space as you want when you specify the field, but if the value you store in it is shorter, the remaining space is wasted. A ***varchar***, however, stands for a **varying character and will occupy the exact length of the string**, **nothing wasted.** There is, however, a maximum limit to how long a string value you can assign to such a field, and that is specified during the field definition itself:

char(12)

varchar(12)

If you store the value 'McCarthy' that is eight characters long, the *char* will store it but waste four characters. The *varchar* will store it as exactly eight characters but the whole dynamism comes at a cost of speed. Nonetheless, the speed difference is small enough that for most scenarios you would see the varying character data type being used.

In case of number values, we get a split across two major classes – *integer* for storing whole numbers and *numeric* for storing number values with a decimal point in them. The ranges and limits of the values being stored in them vary with your choice of DBMS. However, a good rule of thumb to follow is to use the smallest data type that will suffice for the present and foreseeable future of your application.

For example, if I were storing student roll numbers, using a ***smallint*** would suit just fine. In most implementations, this data type allows a maximum value of **32767**, a number I mostly expect to be much greater than the number of students in any class.

Decimal point numbers are trickier to specify. We use the *numeric* data type to fix how large the number could be and how many numbers can occur after the decimal point.

numeric(precision, scale)

numeric(5, 2)

The total number of digits is specified by the *precision* and the number of digits after the decimal point is represented by *scale*. So in the example given, we would be able to store a number like 999.99 but not any further.

Since data types still vary from a DBMS implementation to another, I suggest you keep your DBMS manual handy. Each implementation gives you many other types to work with, but for our learning purposes, the ones above should suffice.

### Chapter 2: Getting Your Database Ready

### **Overview**

The best way to learn SQL is to practice writing the commands on a real relational database management system. In this book SQL is taught using either one of the following systems: **PostgreSQL** or **SQLite**. The reasons for choosing these DBMS systems are simple – they are free and open source, with availability on most major platforms. PostgreSQL is a full-features enterprise class database management system with a great community. SQLite is a small but robust system that is especially suited for learning purposes. Choose the latter if you are not comfortable with software installations.

However, any relational database product that you can get your hands on should serve you just fine. In some cases, you may already have access to one in your organization, but be sure to ask for permissions to use it for learning purposes. There might be minor incompatibilities between different vendors, so if you choose something else to practice on while reading this book, it would be a good idea to keep the database vendor's user manual handy.

Since this text deals largely with teaching SQL in a product-independent manner, rather than the teaching of a specific DBMS system, details with respect to installation and specific operations of the product will be kept to a minimum. Emphasis is instead placed on a few specific steps that will help you to get working on writing SQL as fast as possible.

### **Using PostgreSQL**

The latest version of PostgreSQL as of writing this book was 9.6. You don't absolutely need the latest version; in fact I use version 9.5 in this text.

You can download the latest version of PostgreSQL from [*https://www.postgresql.org/download/*](https://www.postgresql.org/download/) for your platform. For the fastest and easiest installation, I would recommend you choose your platform from the *Binary Packages* list. Pre-built binaries mean that you can simply download and install PostgreSQL like any other software using a graphical step-by-step installer.

After choosing your platform, you might still get multiple ways to perform an installation. I'd recommend choosing the graphical installer version from third-party vendors like *BigSQL* or *EnterpriseDB*. I had chosen the *EnterpriseDB* installer for my *Fedora Linux* machine, and a friendly installation procedure popped up when I ran the downloaded file, asking for details like the installation directory ([Figure 2-1](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=465911416&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

Some other details will also be asked for, most importantly the port number and password. The default value of 5432 for the port number should suffice. At the end of installation, you would have user named 'postgres' on your system and a working database installation.

You can quickly verify that everything went well using *psql*, which is a command-line utility to interact with your PostgreSQL installation. I am capturing the command and output from my system below ([Listing 2-1](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=465911416&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

[~]$ /opt/PostgreSQL/9.5/bin/psql -U postgres

psql.bin (9.5.8)

Type "help" for help.

postgres=#

If you get a similar output, you are ready to start using your PostgreSQL installation. If not, I'm afraid you will have to do some digging on your own. You can also choose SQLite, which is discussed in the next section and has a much easier installation procedure.

### **Using SQLite**

If installing PostgreSQL seems like a daunting task, you are in luck. There is a very credible, free alternative database for you to practice on. It is called **SQLite** and its creator D. Richard Hipp has generously licensed it in the public domain. You can download it from the project page at: <https://www.sqlite.org/download.html>

Like the previous section, what you are looking for to get the fastest start is *precompiled binaries* corresponding to your operating system. SQLite is tiny; on most platforms its core engine is less than one megabyte!

If you are using Microsoft Windows, you are looking for the section titled "Precompiled Binaries for Windows." Download the SQLite DLL zip archive, named like *sqlite-dll-win32-x86-xxxxxxx.zip*, which contains SQLite but not a way to interact with it. For that you must download the SQLite shell, named like *sqlite-tools-win32-x86-xxxxxxx.zip*, which will allow us to create and query SQLite databases through the command line.

Extract both these archives into the same directory and you are done installing SQLite. Your folder should now contain at least three files:

* *sqlite3.dll*
* *sqlite3.def*
* *sqlite3.exe*

The last one launches the command shell used to interact with SQLite databases.

If you are on a Linux or MacOS X system, chances are high that you already have SQLite installed. To test this, you can attempt launching the SQLite shell *sqlite3* ([Listing 2-2](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=567817314&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

[~]$ sqlite3

SQLite version 3.13.0 2016-05-18 10:57:30

Enter ".help" for usage hints.

Connected to a transient in-memory database.

Use ".open FILENAME" to reopen on a persistent database.

sqlite>

If you get the output as above, you have everything you need to run SQLite. Alternatively, if you get an error message, it means you have to install it yourself. You can either use the similar precompiled binary method for your platform or you could use the system installer.

For systems like Red Hat Enterprise Linux, Scientific Linux, and CentOS, you can use *yum* to install SQLite.

# yum install sqlite

On a Fedora Linux system, you have to use *dnf* as below.

# dnf install sqlite

If you happen to use a Debian- or Ubuntu-based system, you can achieve the same result with the following.

$ sudo apt-get install sqlite3

Once the installation is done, you can verify the installation by launching the SQLite shell as before.

### **Creating Your Own Database**

Database management systems like PostgreSQL allow you to create multiple databases. For practice purposes, it's advisable to create your own database, so that you are free to perform any operations on it.

Most database systems differ in the way they provide database creation facilities. PostgreSQL achieves the same by providing you multiple ways to do this, including through the *pgAdmin III* graphical utility. However, for didactic purposes, we will instead use a command operation to create our database. Open up the *psql* shell and enter the command as below ([Listing 2-3](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=534361772&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

CREATE DATABASE testdb;

The command **CREATE DATABASE** is used to create a database that will serve as a holding envelope for your tables. In the example and output shown above, we created a database called *testdb* for our use. The login user you used while connecting with psql, in most cases *postgres*, is now the owner of this database and has full control of entities within it. This is analogous to creating a file in an operating system where the creator gets full access control rights and may choose to give other users and groups specific rights.

The SQL standard by definition allows commands and keywords to be written in a case-insensitive manner. In this book we will use uppercase letters while writing them in statements, which is a widely accepted practice.

Oddly enough, the SQL standard doesn't include the CREATE DATABASE command. In the 1992 standard, aptly named SQL-92, there is a CREATE SCHEMA command that was close to the former but not exactly similar. In modern times, however, databases like MySQL treat the two commands as synonyms of each other.

If you are using SQLite, fire up the command shell, and you will be greeted with a text printing version information ([Listing 2-4](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=534361772&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)). This is exactly the same message we saw in the previous section when we were verifying our SQLite installation.

SQLite version 3.13.0 2016-05-18 10:57:30

Enter ".help" for usage hints.

Connected to a transient in-memory database.

Use ".open FILENAME" to reopen on a persistent database.

sqlite>

Here we enter our .open command to both create a SQLite database or open it in case it already exists.

sqlite> .open testdb

On a Linux system, you could also simply write the database name after the command of the SQLite shell like below, and you would be able to open the said database.

sqlite3 testdb

Interestingly, this invocation would not result in creation of *testdb*, it would simply open it if it exists. If you don't perform any other operation and close the shell (Ctrl-D), there would be no *testdb* file on your machine on Linux. On a Windows system, you would get an empty file with a length of 0 bytes.

### 

### **Table Creation**

We have already explored the concept of a table in a relational model. It is now time to create one using a standard SQL command – CREATE TABLE ([Listing 2-5](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=563000304&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

CREATE TABLE <Table\_Name>

(<Field 1> <Data Type>,

<Field 2> <Data Type>,

\. \. \.

<Field N> <Data Type>);

This is the simplest valid statement that will create a table for you, devoid of any extra options. We'll further this with clauses and constraints as we go along, but for now let us use this general syntax to actually create the table of programming languages we introduced in [Chapter 1](http://viewer.books24x7.com/assetviewer.aspx?bkid=142634&destid=21#21) ([Listing 2-6](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=563000304&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

CREATE TABLE proglang\_tbl (

id INTEGER,

language VARCHAR(20),

author VARCHAR(25),

year INTEGER);

We have to key this command in PostgreSQL on the psql shell. Notice that when we launch the shell, the last line where our cursor waits looks as below:

postgres=#

This actually means that we are connected to the database named *postgres*, which is something the PostgreSQL installation uses internally for management purposes. We have already created our very own database. Let's switch to that before creating our tables using *\c* ([Listing 2-7](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=563000304&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

postgres=# \c testdb

You are now connected to database "testdb" as user "postgres".

testdb=#

Notice that the text on the last line has changed, indicating the current database we are connected to. Now you can key in the table creation statement given in [Listing 2-6](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=563000304&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#), and if you don't miss any of the important punctuation or misspell the keywords written in uppercase, your table would be created and the shell would reply simply with:

CREATE TABLE

testdb=#

A non-successful command would yield an error with a helpful explanation. To see this in action, let's run the exact same table creation command again. The shell would now respond:

ERROR: relation "proglang\_tbl" already exists

The statement by itself is simple enough since it resembles the general syntax of [Listing 2-5](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=563000304&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#). It is interesting to note the data types chosen for the fields. Both *id* and *year* are specified as integers for simplicity, even though there are better alternatives. The *language* field is given a space of 20 characters to store the name of the programming language while the *author* field can hold 25 characters for the creator's name.

The semicolon at the last position is the delimiter for SQL statements, and it marks the end of a statement.

If you are using SQLite, the statement remains exactly the same as [Listing 2-6](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=563000304&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#). The only difference being that since an SQLite database is a particular file and you open it when opening the SQLite shell, there is no switching of database required ([Listing 2-8](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=563000304&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

[~]$ sqlite3 testdb

SQLite version 3.13.0 2016-05-18 10:57:30

Enter ".help" for usage hints.

sqlite> CREATE TABLE proglang\_tbl (

...> id INTEGER,

...> language VARCHAR(20),

...> author VARCHAR(25),

...> year INTEGER);

sqlite>

Notice that there is no successful operation message. If all goes well, SQLite shell simply moves on. Just for some experimentation, if we try to create the same table again, we get an error saying:

Error: table proglang\_tbl already exists

which is again a helpful and somewhat friendlier error message.

### **Inserting Data**

The table we have just created is empty so our task now becomes insertion of some sample data inside it. To populate this data in the form of rows, we use the DML command INSERT, whose general syntax is given below ([Listing 2-9](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=295995493&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

INSERT INTO <Table Name>

VALUES ('Value1', 'Value2', ...);

Fitting some sample values into this general syntax is simple enough, provided we keep in mind the structure of the table we are trying to insert the row in. For populating the proglang\_tbl with rows like we saw in [Chapter 1](http://viewer.books24x7.com/assetviewer.aspx?bkid=142634&destid=21#21), we would have to use three INSERT statements as below ([Listing 2-10](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=295995493&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

INSERT INTO proglang\_tbl

VALUES (1, 'Fortran', 'Backus', 1955);

INSERT INTO proglang\_tbl

VALUES (2, 'Lisp', 'McCarthy', 1958);

INSERT INTO proglang\_tbl

VALUES (3, 'Cobol', 'Hopper', 1959);

If you do not receive any errors from psql or sqlite3 (or the SQL interface for your chosen DBMS), then you have managed to successfully insert three rows of data into your table. Notice how we've carefully kept the ordering of the fields in the same sequence as we used for creating our table. This strict ordering limitation can be removed, and we will see how to achieve that later on.

If you ran these three statements in *psql*, at the end of each executed statement, you would receive a message like:

INSERT 0 1

indicating a success.

### **Writing Your First Query**

Let us now turn our attention to writing a simple query to check the results of our previous operations in which we created a table and inserted three rows of data into it. For this, we would use a Data Query Language (DQL) command called **SELECT**.

A *query* is simply a SQL statement that allows you to retrieve a useful subset of data contained within your database. You might have noticed the INSERT and CREATE TABLE commands were referred to as statements, but a fetching operation with SELECT falls under the query category.

Most of your day-to-day operations in an SQL environment would involve queries, since you'd be creating the database structure once (modifying it only on a need basis) and inserting rows only when new data is available. While a typical SELECT query is fairly complex with many clauses, we will begin our journey by writing down a query just to verify the contents of our table. The general syntax of a simple query is given below ([Listing 2-11](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=965672068&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT <Selection> FROM <Table Name>;

Transforming this into our result verification query is a simple task ([Listing 2-12](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=965672068&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)). We already know the table we wish to query – *proglang\_tbl* and for our selection we would use \* (star), which will select all rows and fields from the table.

SELECT \* FROM proglang\_tbl;

The output of this query would be all the (3) rows displayed in a matrix format just as we intended. If you are running this through psql, you would get an output similar to the one given below.

testdb=# select \* from proglang\_tbl;

id | language | author | year

----+----------+----------+------

1 | Fortran | Backus | 1955

2 | Lisp | McCarthy | 1958

3 | Cobol | Hopper | 1959

(3 rows)

testdb=#

The output from SQLite would be slightly messier at first, but let's fix that one step at a time ([Listing 2-13](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=965672068&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

sqlite> select \* from proglang\_tbl;

1|Fortran|Backus|1955

2|Lisp|McCarthy|1958

3|Cobol|Hopper|1959

sqlite>

Clearly not the cleanest output, but setting a few options would fix that. The first of them is called:

.mode column

and this would output a neatly spaces resultset rather than the squashed one we saw before ([Listing 2-14](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=965672068&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

sqlite> .mode column

sqlite> select \* from proglang\_tbl;

1 Fortran Backus 1955

2 Lisp McCarthy 1958

3 Cobol Hopper 1959

Ah, much better! But there is still a little room for improvement here. We see that column headers are still missing from the output and having them would be advantageous. So we turn on the headers option and the result starts looking pretty neat ([Listing 2-15](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=965672068&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

sqlite> .headers on

sqlite> select \* from proglang\_tbl;

id language author year

---------- ---------- ---------- ----------

1 Fortran Backus 1955

2 Lisp McCarthy 1958

3 Cobol Hopper 1959

I recommend that you keep these options turned on for your learning sessions. The output becomes much easier to verify at a glance.

### Chapter 3: The Benefits of Constraints

### **Overview**

Relational databases are well into their fourth decade of dominance as a data storage and organization mechanism. A large part of this success is owed to the flexibility of the data model. It is easy to visualize all kinds of data fitting into a neat tabular structure with predefined columns.

The flexibility also extends to querying – while creating and populating tables, little restriction is placed upon what you can query from a table. You might try to generate completely new insights from a table you hadn't thought of before. To enable all of this, relational databases expect a certain amount of discipline and thought being put upfront when designing your tables. Neat tables with well-defined data types are essential for success, and certain rules help you keep on this path of good database design.

A **constraint** is a rule that you apply or abide by while doing SQL operations. They are useful in cases where you wish to make the data inside your database more meaningful and/or structured.

### 

### **The Null Constraint**

Consider the example of the programming languages table – every programming language that has been created must have an author (whether a single person, a couple, or committee). Similarly it should have a year when it was introduced, be it the year it first appeared as a research paper or the year a working compiler for it was written. In such cases, it makes sense to create your table in such a way that certain fields do not accept a *NULL* (empty) value.

A null value does not mean 0 (zero) or an empty string like “''. Think of it as either empty or undefined. If you haven't captured someone's age while populating a table, you can't assume their age to be 0. This might have serious implications if someone was using this data for statistical analysis. Putting a null value there makes much more sense.

We now modify our previous CREATE TABLE statement so that we can apply the NULL constraint to some fields ([Listing 3-1](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=675245373&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

CREATE TABLE proglang\_tblcopy (

id INTEGER NOT NULL,

language VARCHAR(20) NOT NULL,

author VARCHAR(25) NOT NULL,

year INTEGER NOT NULL,

standard VARCHAR(10) NULL);

In this table, we only allow the *standard* field to have a null value. Every other field ends with the option NOT NULL, which specifies that this field must necessarily have a value. All fields by default are nullable in most database management systems, so you have to specify a non-nullable field. Writing the word NULL to specify a nullable field is optional.

If we try to insert a row into this table with a *NULL* value in one of the non-nullable fields like *year*, we expect an error message to be thrown at us. In both SQLite and PostgreSQL, we represent a null value with the literal null noting the lack of any quotation marks that generally enclose strings ([Listings 3-2](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=675245373&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#), [3-3](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=675245373&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)). Null is not a string value and writing 'null' makes an actual string of length 4 and is decidedly non-null. Other database management systems might represent null values in a different way, so check your manual for such details.

sqlite> INSERT INTO proglang\_tblcopy

VALUES (1, 'Fortran', 'Backus', null, 'ANSI');

Error: NOT NULL constraint failed: proglang\_tblcopy.year

testdb=# INSERT INTO proglang\_tblcopy

VALUES (1, 'Fortran', 'Backus', null, 'ANSI');

ERROR: null value in column "year" violates not-null

constraint

DETAIL: Failing row contains (1, Fortran, Backus, null, ANSI).

We see in this case that we have achieved our objective of creating a table in which the field's id, language, author, and year cannot be empty for any row, but the new field standard can take empty values. We now go about trying to insert new rows into this table using an alternative INSERT syntax.

### 

### **Selective Fields INSERT**

From our last encounter with the INSERT statement, we saw that we had to specify the data to be inserted in the same order as specified during the creation of the table in question. We now look at another variation that will allow us to overcome this limitation and handle inserting rows with embedded NULL values in their fields by not specifying them at all ([Listing 3-4](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=891931010&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)). While this approach may seem verbose initially, its advantages quickly outweigh any statement length-related concerns.

INSERT INTO <Table\_Name>

(<Field Name 1>,

<Field Name 2>,

...

<Field Name N>)

VALUES

(<Value For Field 1>,

<Value For Field 2>,

...

<Value For Field N>);

Since we specify the field order in the statement itself, we are free to reorder the values sequence in the same statement, thus removing the first limitation. Also, if we wish to enter an empty (NULL) value in any of the fields for a record, it is easy to do so by simply not including the field's name in the first part of the statement. The statement would run fine without specifying any fields you wish to omit, provided they do not have a NOT NULL constraint attached to them.

We now write some *INSERT* statements for the *proglang\_tblcopy* table, in which we try to insert some languages that have not been standardized by any organizations and some which have been ([Listing 3-5](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=891931010&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

INSERT INTO proglang\_tblcopy

(id, language, author, year, standard)

VALUES

(1, 'Prolog', 'Colmerauer', '1972', 'ISO');

INSERT INTO proglang\_tblcopy

(id, language, author, year)

VALUES

(2, 'Perl', 'Wall', '1987');

INSERT INTO proglang\_tblcopy

(id, year, standard, language, author)

VALUES

(3, '1964', 'ANSI', 'APL', 'Iverson');

When you run this through your SQL interface, three new rows would be inserted into the table. Notice the ordering of the third row; it is not the same sequence we used to create the table. Also, since Perl (row id 2) has not been standardized by an international body yet, so we do not specify the field name itself while doing the INSERT operation. This ensures that the *standard* field for the row is populated with null.

To verify the results of these statements ([Table 3-1](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=891931010&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)) and to make sure that the correct data went into the correct fields, we run a simple query as before.

SELECT \* FROM proglang\_tblcopy;

| id | language | author | year | standard |
| --- | --- | --- | --- | --- |
| 1 | Prolog | Colmerauer | 1972 | ISO |
| 2 | Perl | Wall | 1987 |  |
| 3 | APL | Iverson | 1964 | ANSI |

Nulls are often shown by SQL interfaces by a blank space or a question mark (?) or sometimes even the word 'null' or '(null)'. Each implementation is free to choose its representation since it is not standardized among vendors.

### 

### **Check Constraints**

Data must be meaningful for someone to derive insights from it. A great advantage of relational databases is that they enable good structuring of data, proper data type-based storage, and null value rules. *Check constraints* go a step even further by providing validation of what values are allowed in a particular field.

They allow you to provide a logical expression against which inserted values are tested and subsequently accepted or rejected. For example, suppose we wish to ensure that in our programming languages table, no language creation year could be less than or equal to 1950 ([Listing 3-6](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=745191340&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)). This would stop someone from entering values like 190 for the year, which makes sense unless we wish to capture programming languages created during the Roman Empire!

CREATE TABLE proglang\_constraints (

id INTEGER NOT NULL,

language VARCHAR(20) NOT NULL,

author VARCHAR(25) NOT NULL,

year INTEGER NOT NULL

CHECK (year > 1950),

standard VARCHAR(10) NULL);

Note the full definition of the year field that defines the check constraint after the NOT NULL constraint. The logical expression we are testing against is year > 1950, which disallows any row containing a year value less than 1951. Let's try entering such a row to test the hypotheses ([Listing 3-7](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=745191340&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

testdb=# INSERT INTO proglang\_constraints

(id, language, author, year)

VALUES

(1, 'Short Code', 'Mauchly', 1949);

ERROR: new row for relation "proglang\_constraints" violates

check constraint "proglang\_constraints\_year\_check"

DETAIL: Failing row contains (1, Short Code, Mauchly, 1949,

null).

While an underused feature, check constraints are extremely useful. A lot of application software code is written with the purpose of validating data to be inserted, an area where check constraints can help immensely.

### 

### **Primary Key Constraint**

The mathematical concept behind the relational data model was Set theory. This area of discrete maths deals with unordered bag of values that can be uniquely identified, that is, contains no duplicates. For a table, a value is a record of data and a key column for each record is the perfect way to identify it.

A *primary key* is used to make each record unique in at least one way by forcing a field to have a unique value. They do not have to be restricted to only one field; a combination of them can also be defined as a primary key for a table. One must think carefully about the logical implications of choosing a field or a combination of them as a primary key.

Often the best primary key candidates are not our instinctive identifiers for a collection. If you were storing data about people, their names are something we identify them with in real-life scenarios. But what would happen in the unforgiving world of primary keys if two people were named 'David Childs'?

In our programming languages table, the *id* field is a good choice for applying the primary key constraint. We will now modify our CREATE TABLE statement to incorporate this ([Listing 3-8](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=781754031&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

CREATE TABLE proglang\_tbltmp (

id INTEGER NOT NULL PRIMARY KEY,

language VARCHAR(20) NOT NULL,

author VARCHAR(25) NOT NULL,

year INTEGER NOT NULL,

standard VARCHAR(10) NULL);

ID fields are usually chosen as primary fields. Note that in this particular table, the *language* field would have also worked, since a language name is unique. However, if we have a table that describes people, we should try to find a logically unique field like their SSN number or employee ID number.

Even though the concept of a primary key seems to be natural and necessary, most database implementations don't really enforce it. This includes the two databases used for examples in this book – *PostgreSQL* and *SQLite*. You are free to create a table without any primary keys (like we did before this section came along) and insert exactly duplicated data again and again. Not an ideal scenario but allowed nonetheless if you are so inclined.

Let us add some duplicated rows in our *proglang\_tblcopy* table that we were working with in the beginning of the chapter ([Listing 3-9](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=781754031&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

INSERT INTO proglang\_tblcopy

(id, language, author, year)

VALUES

(2, 'Perl', 'Wall', '1987');

INSERT INTO proglang\_tblcopy

(id, language, author, year)

VALUES

(2, 'Perl', 'Wall', '1987');

Note that we already had three unique rows in the table to which we added two duplicated ones. The execution of the INSERT statements was silent, indicating success. Let's verify the contents of the table now ([Table 3-2](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=781754031&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT \* FROM proglang\_tblcopy;

| id | language | author | year | standard |
| --- | --- | --- | --- | --- |
| 1 | Prolog | Colmerauer | 1972 | ISO |
| 2 | Perl | Wall | 1987 |  |
| 3 | APL | Iverson | 1964 | ANSI |
| 2 | Perl | Wall | 1987 |  |
| 2 | Perl | Wall | 1987 |  |

If we try to add duplicate records in our table containing the primary key constraint – proglang\_tbltmp, we promptly get an error thrown at us ([Listing 3-10](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=781754031&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

INSERT INTO proglang\_tbltmp

(id, language, author, year)

VALUES

(2, 'Perl', 'Wall', '1987');

ERROR: duplicate key value violates unique constraint

"proglang\_tbltmp\_pkey"

DETAIL: Key (id)=(2) already exists.

### **Unique Key Constraints**

A *unique key* like a primary key is also used to make each record inside a table unique. Once you have defined the primary key of a table, any other fields you wish to enforce as unique is done through this constraint. Well thought-out uniqueness constraints go a long way in ensuring that the data inside the table is sane.

For example, in our database it now makes sense to have a unique key constraint on the *language* field ([Listing 3-11](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=886729136&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)). This would ensure none of the records would duplicate information about the same programming language even if the *id* field was non-matching.

CREATE TABLE proglang\_tbluk (

id INTEGER NOT NULL PRIMARY KEY,

language VARCHAR(20) NOT NULL UNIQUE,

author VARCHAR(25) NOT NULL,

year INTEGER NOT NULL,

standard VARCHAR(10) NULL);

We will now try to insert two rows about the language Prolog cleverly changing the id field to test out our unique constraint ([Listing 3-12](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=886729136&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

testdb=# INSERT INTO proglang\_tbluk

(id, language, author, year, standard)

VALUES

(1, 'Prolog', 'Colmerauer', 1972, 'ISO');

INSERT 0 1

testdb=# INSERT INTO proglang\_tbluk

(id, language, author, year, standard)

VALUES

(2, 'Prolog', 'Colmerauer', 1972, 'ISO');

ERROR: duplicate key value violates unique constraint

"proglang\_tbluk\_language\_key"

DETAIL: Key (language)=(Prolog) already exists.

Note that we write the word UNIQUE in front of the field and omit the KEY in the table creation command. You can have as many fields with unique constraints as you wish.

We will revisit the unique and primary key constraints again in [Chapter 15](http://viewer.books24x7.com/assetviewer.aspx?bkid=142634&destid=447#447) when we discuss indexing.

### 

### **Differences Between a Primary Key and a Unique Key**

You might have noticed that the two constraints discussed above are similar in their purpose. However, there are a couple of differences between them.

1. A primary key field cannot take on a NULL value, whereas a field with a unique constraint can. However, there can be only one such record since each value must be unique due to the very definition of the constraint.
2. You are allowed to define only one primary key constraint for a table, but you can apply the unique constraint to as many fields as you like.

This is a favorite interview question for any job that deals with SQL as far as my experience goes. It is not too unfair considering the importance of these constraints to a logical data model. Just remember to think of a primary key as a NOT NULL UNIQUE constraint.

A primary key ensures a logical way to differentiate between rows of a table. It is the bare minimum criterion for a differentiated record. Unique constraints are usually added as additional rules to ensure data sanity while keeping the business or domain rules in mind. It's not necessary to have them, but they act as gatekeepers to allow only good data through.

### Chapter 7: Organizing Your Data

### **Overview**

Since this is a text meant to teach SQL to people unfamiliar with it, our data has been very simplistic. The number of fields you'd wish to store in your database would be a larger value than the five-column table we saw in earlier chapters. Also, some assumptions were made intrinsically on the kind of data we will store in the table. But this is not always the case in real life.

In reality the data we encounter will be complex, even redundant. This is where the study of data modeling techniques and database design come in. While it is advised that the reader refer to a more comprehensive treatise on this subject, nonetheless we will try to study some good relational database design principles since the study would come in handy while learning SQL statements for multiple tables.

### 

### **Normalization**

Let us suppose we have a database of employees in a fictional institution as given below ([Table 7-1](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=983067076&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)). If the database structure has not been modeled but has been extracted from a raw collection of information available, *redundancy* is expected.

| employee\_id | name | skill | manager\_id | location |
| --- | --- | --- | --- | --- |
| 1 | Socrates | Philosophy | (null) | Greece |
| 2 | Plato | Writing | 1 | Greece |
| 3 | Aristotle | Science | 2 | Greece |
| 4 | Descartes | Philosophy | (null) | France |
| 4 | Descartes | Philosophy | (null) | Netherlands |

We can see that *Descartes* has two rows because he spent his life in both France and the Netherlands. Doesn't seem very elegant, does it? Now if at a later point we decide that we wish to classify him with a different skill, we would have to update both his rows since they should contain an identical (primary) skill.

Wouldn't it be saner to have a separate table for skills and somehow allow the records that share the same skill to refer to this table? This way if we wish to reflect that both Socrates and Descartes were thinkers in *Western Philosophy*, renaming the skill record in the second table would do the trick.

This process of breaking down a raw database into logical tables and removing redundancies is called *Normalization*. There are even levels of normalization called *normal forms* that dictate how to achieve the desired design.

There are five accepted normal forms that serious database administrators and developers are familiar with. They range from *first normal form* 1NF to *fifth normal form* 5NF. These forms are progressive in nature, meaning that a design in 3NF is also 1NF and 2NF compliant. Since the origin of these forms are based on academic research, the working developers usually restrict themselves to 3NF or 4NF in most cases. Again, we advise the reader to refer to a more comprehensive text dealing with database design and normalization. We will do mere lip service in exploring these vast fields.

For now, let's turn to our programming languages data to see the need for normalization playing out.

### 

### **Atomicity**

In the programming language examples that we've seen, our assumption has always been that a language has a single author. But there are countless languages where multiple people contributed to the core design and should rightfully be acknowledged in our table. How would we go about making such a record? Let us take the case of BASIC, which was designed by John Kemeny and Thomas Kurtz. The easiest option to add this new record into the table is to fit both authors in the *author* field ([Table 7-2](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=965203576&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

| id | language | author | year | standard |
| --- | --- | --- | --- | --- |
| 1 | Prolog | Colmerauer | 1972 | ISO |
| 2 | Perl | Wall | 1987 | (null) |
| 3 | APL | Iverson | 1964 | ANSI |
| 4 | TcI | Ousterhout | 1988 | (null) |
| 5 | BASIC | Kemeny, Kurtz | 1964 | ANSI |

You can immediately see that it would be difficult to write a query to retrieve this record based on the *author* field. If the data were written as "Kemeny, Kurtz" or "Kurtz, Kemeny" or even "Kemeny & Kurtz," it would be extremely difficult to put the right string in the WHERE conditional clause of the query. After all, it is possible that the person who inserted the data is not the same as the one querying it.

The correct solution is to redesign the table structure to make all field values *atomic*. Atomicity of values means that every intersection of a row and column must contain a single, indivisible value. If in your current design you have some fields containing non-atomic values, you need to start thinking of changing your table structures.

### 

### **Repeating Groups**

Another simple (but ultimately wrong) approach that comes to mind is to split the *author* field into two parts – *author1* and *author2*. If a language has only one author, the *author2* field would contain a null value ([Table 7-3](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=716055717&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)). Can you spot the problem that will arise from this design decision?

| id | language | author1 | author2 | year | standard |
| --- | --- | --- | --- | --- | --- |
| 1 | Prolog | Colmerauer | (null) | 1972 | ISO |
| 2 | Perl | Wall | (null) | 1987 | (null) |
| 3 | APL | Iverson | (null) | 1964 | ANSI |
| 4 | TcI | Ousterhout | (null) | 1988 | (null) |
| 5 | BASIC | Kemeny | Kurtz | 1964 | ANSI |

This imposes an artificial constraint on how many authors a language can have. It seems to work fine for a couple of them, but what if a programming language was designed by a committee of a dozen or more people, and we did want to include all of them in the credits? At the database design time, how do we fix the number of authors we wish to support?

This kind of design is referred to as a *repeating group* and must be actively avoided. This also has an ugly effect of having too many null values in some of the fields, a first sign of bad database design.

### 

### **Splitting the Table**

Our first stab at table design lumps the languages and authors together. It is natural to think that way because our understanding of the data at first glance views all the fields as a logical whole. All the data to us belongs to the programming languages, the entity being described.

But as we have seen above, the authors of the languages seem to be a distinct entity in our data. We have not even begun to capture multiple languages by the same author, and already we feel a pressing need to distinguish between languages and authors as entities.

The correct design to remove the problems listed above is to *split* the table into two – one holding the author details ([Table 7-5](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=648715240&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)) and one detailing the language details ([Table 7-4](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=648715240&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

| id | language | year | standard |
| --- | --- | --- | --- |
| 1 | Prolog | 1972 | ISO |
| 2 | Perl | 1987 | (null) |
| 3 | APL | 1964 | ANSI |
| 4 | TcI | 1988 | (null) |
| 5 | BASIC | 1964 | ANSI |

| author\_id | author1 | language\_id |
| --- | --- | --- |
| 1 | Colmerauer | 1 |
| 2 | Wall | 2 |
| 3 | Iverson | 4 |
| 4 | Ousterhout | 3 |
| 5 | Kemeny | 5 |
| 6 | Kurtz | 5 |

Once you have removed the non-atomicity of fields and repeating groups along with assigning unique id's to your tables, your table structure is now in the first normal form (1NF). The author table's language\_id field, which refers to the id field of the language table, is called a foreign key constraint ([Listing 7-1](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=648715240&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

CREATE TABLE newlang\_tbl

(id INTEGER NOT NULL PRIMARY KEY,

language VARCHAR(20) NOT NULL,

year INTEGER NOT NULL,

standard VARCHAR(10) NULL);

CREATE TABLE authors\_tbl

(author\_id INTEGER NOT NULL,

author VARCHAR(25) NOT NULL,

language\_id INTEGER REFERENCES newlang\_tbl(id));

Notice that in the author's table we've made a foreign key constraint by making the language\_id field reference the id field of the new programming languages table using the keyword REFERENCES. You can only create a foreign key reference as a primary or unique key; otherwise during the constraint creation time we would receive an error similar to the following.

ERROR: there is no unique constraint matching given keys for

referenced table "newlang\_tbl"

Since we have created a reference to the language\_id, inserting a row in the author's table that does not yet have a language entry would also result in an error, called a Referential Integrity violation ([Listing 7-2](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=648715240&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

INSERT INTO authors\_tbl

(author\_id, author, language\_id)

VALUES

(5, 'Kemeny', 5)

ERROR: insert or update on table "authors\_tbl" violates

foreign key constraint "authors\_tbl\_language\_id\_fkey"

DETAIL: Key (language\_id)=(5) is not present in table

"newlang\_tbl".

However, when done sequentially, that is, the language first and then its corresponding entry in the author table, everything works out ([Listing 7-3](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=648715240&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

INSERT INTO newlang\_tbl

(id, language, year, standard)

VALUES

(5, 'BASIC', 1964, 'ANSI');

INSERT INTO authors\_tbl

(author\_id, author, language\_id)

VALUES

(5, 'Kemeny', 5);

### **Referential Integrity in SQLite**

If you tried to run [Listing 7-2](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=648715240&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#) in SQLite, you wouldn't get an error back despite there being no *language\_id 5* in the *newlang\_tbl*. SQLite, by default, turns off referential integrity checking for backward compatibility reasons. TO turn it on for your database, run the following pragma in its command shell.

PRAGMA foreign\_keys = ON;

If you now violate referential integrity, we would get a familiar error message.

Error: FOREIGN KEY constraint failed error

Referential integrity is a key benefit of good relational database design. Since it applies to related entities, it ensures that the values of these remain in sync. In our example above, this constraint makes sure that we never have an author's data whose created programming language is not captured in the languages table.

When designing databases to solve a business problem, deciding how referential integrity comes into play is a big decision. This is done mainly in discussion with domain experts who understand the business logic of the entities data you are trying to capture.

The other statements to get fully populated tables are given below ([Listing 7-4](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=648715240&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

INSERT INTO newlang\_tbl

(id, language, year, standard)

VALUES

(1, 'Prolog', 1972, 'ISO');

INSERT INTO newlang\_tbl

(id, language, year)

VALUES

(2, 'Perl', 1987);

INSERT INTO newlang\_tbl

(id, language, year, standard)

VALUES

(3, 'APL', 1964, 'ANSI');

INSERT INTO newlang\_tbl

(id, language, year)

VALUES

(4, 'Tcl', 1988);

INSERT INTO authors\_tbl

(author\_id, author, language\_id)

VALUES (6, 'Kurtz', 5);

INSERT INTO authors\_tbl

(author\_id, author, language\_id)

VALUES (1, 'Colmerauer', 1);

INSERT INTO authors\_tbl

(author\_id, author, language\_id)

VALUES (2, 'Wall', 2);

INSERT INTO authors\_tbl

(author\_id, author, language\_id)

VALUES (3, 'Ousterhout', 4);

INSERT INTO authors\_tbl

(author\_id, author, language\_id)

VALUES (4, 'Iverson', 3);

### **The Pursuit of Normalization**

The man who created the Relational Model and in turn normalization – Dr. Codd – was an academic genius. While he was working at IBM at the time, decidedly non-academia, the whole thing has a whiff of mathematical purity about it.

But the cost of removing redundancies in data is speed. While there may be many advanced levels of normal forms, 1NF-5NF and the lesser-known Boyce-Codd Normal Form, we must not be too pedantic about pursuing the higher normal form. Common sense must prevail in the head of the database designer.

In some cases, denormalization does have the benefit of faster access. Indeed, many in-vogue NoSQL database systems tout redundant data storage as a feature. This obviously comes at the cost of consistency of truth. But then again, when our seemingly random clicks on the Web are captured and analyzed to decide our most suitable insurance provider, perhaps a loss of truth is acceptable.

Nevertheless if 40 years of dominance is anything to go by, unless you have run into a very special case, Codd (and the relational model) is always right.

### Chapter 16: Access Control Statements

### **Overview**

Due to their ease of use, extensive feature sets, and reliability, SQL-based relational database management systems have become the golden source of truth for enterprises everywhere. When large companies think of storing critical data, the only question in their minds is which relational DBMS vendor and not what kind of data store.

When such important data is being stored in the system and when the DBMS becomes the central data store across the organization, some level of access control is absolutely essential.

Access control refers to permissions within a software system. When you log onto a server or sometimes even your own computer, you have been given permission to access the resources of the system. Often when you wish to install new software on your machine, you require *root* or *administrator* privileges. This is the operating system's access control mechanisms at work.

Relational databases understandably also have very powerful access control mechanisms. While most systems vary widely in how they provide access control, almost all vendors do provide the Data Control Language (DCL) SQL commands of GRANT and REVOKE.

**Access Control in SQLite**

Permissions truly come into play in multi-user systems, that is, when multiple users have access to a system but not equally. This is typically the case in client-server database systems like PostgreSQL, Sybase ASE, etc.

SQLite is a single file-based system typically used in scenarios where we need to embed a simple database within tight constraints or within an application. It is not truly meant for multi-user access though there are provisions in it to allow it to some degree. This does not mean that it is not a capable RDBMS. I personally think the world would be better off using SQLite in half the cases where more expensive and resource-hungry systems were put, but that is a discussion for another time.

SQLite being single file-based relies on the operating systems to grant or restrict access to its data file. Consequently, it does not implement any GRANT or REVOKE commands. The rest of the chapter focuses on access control mechanisms using PostgreSQL examples.

**Creating New Users in PostgreSQL**

For running the examples in the book, we have been using the user postgres. This has served us well as a catch-all account with all rights and permissions. Just to recap, we used to start our psql session by specifying the username in the -U option as below ([Listing 16-1](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=451124043&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

/opt/PostgreSQL/9.5/bin/psql -U postgres

But this does not accurately reflect the real-world setup. Usually you would have your own user account, which will have lesser privileges than the administrator account. This is safer both for you and the database administrators knowing that one account being compromised does not affect everything in the system.

Let us go about creating a new user in PostgreSQL through *psql* ([Listing 16-2](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=451124043&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)), and then we'll go on to specifying the rights that the particular user gets.

postgres=# CREATE USER primer with PASSWORD 'hunter2';

CREATE ROLE

postgres=#

When you execute this command, a new user by the name primer is created with the password hunter2 and the console displays the message CREATE ROLE. Since the session was created with the user postgres and the text before the =# is still the same, we can see that the CREATE USER command does not switch to the new user directly but continues the same session with postgres.

**The Legend of hunter2**

Before social networks were popular, Internet Relay Chat (IRC) ruled the instant messaging landscape. People connected to an IRC server and joined one or more chat rooms called channels and talked with like-minded people.

If a conversation was particularly funny, people would post it to a site Bash.org. Around 2004, somebody posted a funny exchange between two users where the first user convinces the other one that when they type their real password in IRC, everyone else sees only \*\*\*\*\*\*'s. This was of course not true, but funny nonetheless. Whether the conversation truly happened can also not be verified, but the password in question was 'hunter2'.

You can read the Bash.org entry here: <http://www.bash.org/?244321>

We will now try to verify whether the user was indeed created ([Listing 16-3](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=451124043&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)). Like before, psql gives us a short command for this – \du.

**postgres=# \du**

**List of roles**

**Role name | Attributes |**

**Member of**

**-----------+------------------------------------------------+**

**postgres | Superuser, Create role, Create DB, Replication,|**

**| Bypass RLS |**

**{}**

**primer | |**

**{}**

We see that indeed our user has been created, though the list of its attributes is empty. Don't worry, we will get to that in a bit.

If you want to verify the same information without using the *psql* metacommand, we can query the inbuilt database catalog information ([Listing 16-4](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=451124043&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT usename,

usesysid,

usecreatedb,

usesuper FROM pg\_user;

| usename | usesysid | usecreatedb | usesuper |
| --- | --- | --- | --- |
| postgres | 10 | t | t |
| primer | 41095 | f | f |

**Grant Privileges to Users**

Let us try to open a psql session using this newly created user. We will pass the value primer to the -U option of psql ([Listing 16-5](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=119649672&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

**/opt/PostgreSQL/9.5/bin/psql -U primer**

**Password for user primer:**

**psql.bin: FATAL: database "primer" does not exist**

Along with the user, it also tried to open the default database for the new user whose name was assumed to be the same as the username. We'll remedy this by explicitly stating that we want to operate on testdb ([Listing 16-6](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=119649672&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

/opt/PostgreSQL/9.5/bin/psql -U primer -d testdb

Password for user primer:

psql.bin (9.5.8)

Type "help" for help.

testdb=> SELECT \* FROM proglang\_tbl;

ERROR: permission denied for relation proglang\_tbl

Logging into the testdb database worked. But when we ran a basic query on one of the tables, it immediately gave us a permission denied error. This seems logical in retrospect since we haven't granted any special access to *primer*. We don't want any new user to immediately gain access to our meticulously created tables.

We will use the GRANT statement to give specific privileges to our newly created user. The general syntax of GRANT is given below ([Listing 16-7](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=119649672&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

GRANT <privilege> ON <table name> TO <user>

The most obvious privilege we wish to give the primer user is the ability to query proglang\_tbl. This is equivalent to giving the user a read-only access to the particular table. We run this statement as the superuser postgres who will bestow privileges to other users ([Listing 16-8](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=119649672&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

GRANT SELECT ON proglang\_tbl TO primer;

Now we can exit the psql session as postgres and reopen testdb as the user primer. We will attempt to first query and then update a row in the table just to see whether the GRANT statement worked as advertised ([Listing 16-9](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=119649672&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

testdb=> SELECT count(\*) FROM proglang\_tbl;

count

-------

9

(1 row)

testdb=> UPDATE proglang\_tbl SET year=1982 WHERE author='Ross';

ERROR: permission denied for relation proglang\_tbl

The query worked fine but the row update did not, so everything is working as expected. As you might have guessed, we would need to GRANT the UPDATE privilege too for the second statement to work.

SELECT and UPDATE are not the only privileges available for finegrained access control. You can specify other privileges like INSERT and DELETE too. Finally, there is an ALL privilege that grants all the available privileges on that particular database object to the user specified. If you wish to specify multiple privileges in one go, you can specify them like a list ([Listing 16-10](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=119649672&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

GRANT SELECT, UPDATE, INSERT ON proglang\_tbl TO primer;

Granting privileges is usually done when the users who wish to access the table or database object in question are not the ones who created it. If a user has created a table, they get all privileges on it by default.

There are other privileges in most DBMS systems out there than the four basic ones we covered. However their use is usually of interest to the database administrators rather than query users. Feel free to refer to your DB manual to know more about the supported privileges.

**Revoking Privileges**

The REVOKE command is the exact opposite of GRANT. It allows you to remove privileges from a user for a database object. Its general syntax is similar to GRANT with the exception that it uses FROM instead of TO ([Listing 16-11](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=409485713&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

REVOKE <privilege> ON <table name> FROM <user>

While discussing views in [Chapter 14](http://viewer.books24x7.com/assetviewer.aspx?bkid=142634&destid=417#417), we had mentioned how views help in data security by providing a virtual table containing only the fields you want to show to others. But this plan would be foiled if the users could query the base table too. Using REVOKE here is a good idea. We'd allow users to query the view but not the underlying table ([Listing 16-12](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=409485713&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)). This way we ensure that usability is not hampered while still being able to keep all kinds of fields together that make sense on a data-modeling level.

testdb=# GRANT SELECT ON language\_decade TO primer;

testdb=# REVOKE SELECT ON proglang\_tbl FROM primer;

We are running these commands using the superuser postgres. Let us now log in as the user primer and see how these statements have affected our privileges ([Listing 16-13](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=409485713&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

SELECT \* FROM proglang\_tbl;

ERROR: permission denied for relation proglang\_tbl

SELECT \* FROM language\_decade WHERE decade='The 1950s';

language | decade

----------+-----------

Jovial | The 1950s

APT | The 1950s

(2 rows)

We notice that the user can query the view but not the base table. This is in fact a very common access control workflow in large databases. Right after the data definition process, different views are created on the basis of how we expect the data to be queried, and then base table privileges are revoked for interactive querying.

While we can specify a list of users in GRANT and REVOKE, we cannot reasonably expect the list of users of a database system to remain the same over time. Most DBMS software provides a keyword PUBLIC to refer to all current and future users of a system. This can be used with our access control statements to minimize the need for routine access administration ([Listing 16-14](https://viewer.books24x7.com/assetviewer.aspx?bookid=142634&chunkid=409485713&resumebookmarkid=a025dd84-1f41-ee11-aa72-005056b54d63#)).

testdb=# GRANT ALL ON proglang\_tbl TO PUBLIC;

testdb=# REVOKE DELETE ON proglang\_tbl FROM PUBLIC;

What these two statements in succession would achieve is to first open up the proglang\_tbl for everyone and then remove only the DELETE privilege. The other privileges like INSERT, UPDATE, etc., would be available to all users of the system without us having to list them one by one. If a new user is created, these access control levels would be applicable to them too.

**Database Development for Dummies: Chapter 4**

### **Chapter 4: The Entity-Relationship Model**

### **In This Chapter**

* Understanding the structure of an E-R model
* Sorting out entities, attributes, identifiers, and relationships
* Taking a look at E-R diagrams
* Refining the E-R model
* Examining some E-R examples

After you have a users’ data model that all stakeholders can support, you need to put that model into a formal structure that you can map directly into a relational model and then implement as a database. This chapter describes the most popular such formal structure, the entity-relationship model.

### **Technical Stuff**

In 1976, six years after E. F. Codd’s seminal paper on relational database theory appeared, Peter Chen published a paper in the ACM Transactions on Database Systems that described the entity-relationship (E-R) model. The E-R model was significant because it could be directly translated into a relational model that met Codd’s criteria for a true relational design. At that time, relational databases were still a theoretical discipline. The first commercial relational database management system had not yet reached the market. The practical applicability of the E-R model gave added incentive to software companies to take the financial risk inherent in developing the first relational DBMS products.

### **Exploring the Structure of the E-R Model**

The E-R model can be used to represent a wide variety of systems that people want to track in some detailed manner. The system could be physical, such as the space shuttle or the human genome, or it could be conceptual, such as the financial records of a large corporation. One of the strengths of the E-R model is that even the largest and most complex systems can be modeled using just four elements:

* Entities
* Attributes
* Identifiers
* Relationships

### **Entities**

Although you might at first think that an entity is a highly advanced intelligence from another galaxy, or perhaps a creature from a higher dimension, it is actually a pretty mundane concept. In the context of the E-R model, an [*entity*](http://viewer.books24x7.com/assetviewer.aspx?bkid=12764&destid=786#786) is merely something that the user can identify and wants to keep track of. Examples might be EMPLOYEE Santos McKinney, SALESPERSON Joshua Flores, CUSTOMER 1732, and SALES\_ORDER 314159.

Entities of a given type are grouped together into *entity classes*. Thus, EMPLOYEE is an entity class, and Santos McKinney is an *instance* of the EMPLOYEE entity class, as shown in [Figures 4-1](https://viewer.books24x7.com/assetviewer.aspx?bookid=12764&chunkid=265738970&resumebookmarkid=13b44b1d-6043-ee11-aa72-005056b54d63#) and 4-2.

| EMPLOYEE |  |
| --- | --- |
| EmpID | 97209 |
| FirstName | Santos |
| LastName | McKinney |
| JobTitle | Database Developer |
| Exempt/NonExempt | E |
| HireDate | 01/03/2000 |
| Extension | 267 |
| Email | smckinney@acme.com |
| Department | Molecular Biology |

### **Attributes**

There are some things about an entity instance that an organization wants to track and others that it probably does not. For instance, the company that employs Santos McKinney will doubtless want to record his first and last names, and perhaps his hire date, but not his shoe size.

The *attributes* of an entity are the characteristics of that entity that are of interest to the user. Thus, FirstName, LastName, and HireDate would likely be attributes of the EMPLOYEE entity class, but ShoeSize would not. In most organizations, management does not care about the shoe size of its employees.

### **Identifiers**

In a database, it is pretty important that you be able to distinguish one entity instance from another. For example, it is bad form to send a bill to a CUSTOMER for an item that the customer did not order. It is even worse to fail to bill a CUSTOMER who did order, and then took delivery on your company’s top-of-the-line luxury yacht.

So, you must have some way to identify the individual instances of an entity class. Logically enough, the E-R model does this with attributes or combinations of attributes called *identifiers*:

* A unique identifier will identify one and only one instance of an entity class.
* A non-unique identifier will identify a set of instances that share some common characteristic or group of characteristics.
* An identifier that is composed of two or more attributes is called a composite identifier.

Often, a single attribute will serve as a unique identifier for an entity class. At other times, however, no single attribute will narrow things down to that extent. In such cases, other attributes, making up a composite identifier, can be added until the combination of attributes is enough to uniquely identify every instance in the class. For example, a Social Security Number is a unique identifier of a resident of the United States. A person’s last name is probably a non-unique identifier of a resident of the United States, particularly if that last name happens to be Smith, Rodriguez, Lee, or Nguyen. A composite identifier, such as the combination of a person’s first name, last name, street, city, and state is probably enough to uniquely identify a person, unless perhaps a father and son have the same name and live at the same address.

### 

### **Relationships**

Entities are associated with other entities through *relationships*. *Relationship classes* are associations among entity classes. *Relationship instances* are associations among entity instances.

Relationships can exist in varying *degrees*. A degree 2 relationship is one between two entity classes or two entity instances. A degree 3 relationship relates three entity classes or entity instances to each other. An example of a degree 2 relationship would be that between a basketball team and its players. Each team has a set of associated players, none of whom are associated with any other team.

[Figure 4-3](https://viewer.books24x7.com/assetviewer.aspx?bookid=12764&chunkid=265738970&resumebookmarkid=13b44b1d-6043-ee11-aa72-005056b54d63#) shows a diagrammatic representation of the relationship.

**Team > Team-Player < Player**

An example of a degree 3 relationship would be that among a composer, a lyricist, and the songs they jointly create. Each song is created by the combined efforts of a composer and a lyricist. [Figure 4-4](https://viewer.books24x7.com/assetviewer.aspx?bookid=12764&chunkid=265738970&resumebookmarkid=13b44b1d-6043-ee11-aa72-005056b54d63#) shows a diagrammatic representation of the relationship.

**Composer > Composition < Lyricist**

**I**

**Song**

Degree 2 relationships, also called binary relationships, are very common and are simpler than relationships of higher degree. Happily, most problems of practical significance to businesses and other organizations can be modeled using binary relationships. Degree 3 relationships can often be reduced multiple degree 2 relationships, which can then be handled more easily.

Remember, three kinds of binary relationships exist:

* **A one-to-one (1:1) relationship** relates one instance of one entity class to one instance of a second entity class.
* **A one-to-many (1:N) relationship** relates one instance of one entity class to multiple instances of a second entity class.
* **A many-to-many (N:M) relationship** relates multiple instances of one entity class to multiple instances of a second entity class.

[Figure 4-5](https://viewer.books24x7.com/assetviewer.aspx?bookid=12764&chunkid=265738970&resumebookmarkid=13b44b1d-6043-ee11-aa72-005056b54d63#) shows a diagrammatic representation of an example of a **1:1 relationship** between a driver and a driver’s license. Every driver has one and only one license, and every license belongs to one and only one driver.

**Driver > Driver-Licence < Licence**

[Figure 4-6](https://viewer.books24x7.com/assetviewer.aspx?bookid=12764&chunkid=265738970&resumebookmarkid=13b44b1d-6043-ee11-aa72-005056b54d63#) shows an example of a **1:N relationship** between a driver and the traffic tickets he has received. A driver may receive multiple traffic tickets for multiple infractions of the traffic laws, but each ticket is written for one and only one driver.

**Driver > Driver-Ticket(s) < Ticket**

[Figure 4-7](https://viewer.books24x7.com/assetviewer.aspx?bookid=12764&chunkid=265738970&resumebookmarkid=13b44b1d-6043-ee11-aa72-005056b54d63#) shows an example of a **N:M relationship** between a driver and the routes that she might take to get to work. A driver may take any one of several routes to get to work. Similarly, each route may be taken by multiple drivers.

**Driver > Driver-Route < Route**

### **Creating Entity-Relationship Diagrams**

[Figures 4-5](http://viewer.books24x7.com/assetviewer.aspx?bkid=12764&destid=131#131), 4-6, and 4-7 are examples of entity-relationship diagrams. Entities are represented as named rectangular boxes, and relationships as lines connecting the boxes. The relationship lines contain symbols that represent the maximum and minimum cardinality of the relationship.

### **Maximum Cardinality**

The diamond at the center of the relationship lines in [Figures 4-5](http://viewer.books24x7.com/assetviewer.aspx?bkid=12764&destid=131#131), 4-6, and 4-7 shows the [maximum cardinality](http://viewer.books24x7.com/assetviewer.aspx?bkid=12764&destid=795#795) — that is, the maximum number of instances of an entity on each side of a relationship. For example, the 1:1 in the diamond in [Figure 4-5](http://viewer.books24x7.com/assetviewer.aspx?bkid=12764&destid=131#131) means that a single instance of the entity on the left is related to a single instance of the entity on the right. Similarly, the 1:N in the diamond in [Figure 4-6](http://viewer.books24x7.com/assetviewer.aspx?bkid=12764&destid=132#132) means that a single instance of the entity on the left is related to multiple instances of the entity on the right. The N:M in the diamond in [Figure 4-7](http://viewer.books24x7.com/assetviewer.aspx?bkid=12764&destid=133#133) means that multiple instances of the entity on the left are related to multiple instances of the entity on the right, and the maximum number of instances on the left is not necessarily equal to the maximum number of instances on the right.

### **Minimum Cardinality**

Just as maximum cardinality is the maximum number of instances of an entity on each side of a relationship, [*minimum cardinality*](http://viewer.books24x7.com/assetviewer.aspx?bkid=12764&destid=798#798) is the minimum number of instances of an entity on each side of a relationship. In some cases, the minimum number of instances of an entity might be zero. For example, in the DRIVER-LICENSE example shown in [Figure 4-5](http://viewer.books24x7.com/assetviewer.aspx?bkid=12764&destid=131#131), it is possible that a DRIVER might have his LICENSE revoked. If so, it would be possible for an instance of the DRIVER entity class to have no corresponding instance in the LICENSE entity class. In such a case, the minimum cardinality of LICENSE would be zero. On the other hand, every instance of the LICENSE entity class corresponds to an instance of the DRIVER entity class. So the minimum cardinality of DRIVER would be one.

[Figure 4-8](https://viewer.books24x7.com/assetviewer.aspx?bookid=12764&chunkid=620159512&resumebookmarkid=13b44b1d-6043-ee11-aa72-005056b54d63#) shows the same relationship as [Figure 4-5](http://viewer.books24x7.com/assetviewer.aspx?bkid=12764&destid=131#131) but with minimum cardinality denoted. The oval on the LICENSE side of the relationship means that the minimum cardinality of LICENSE is zero. The slash on the DRIVER side means that the minimum cardinality of DRIVER is one.

**Driver > Driver-Licence < Licence**

No general rules exist for determining minimum cardinality. Cardinality is strictly determined by how the users view the system of interest. In other words, the users’ data model determines cardinality. Consider the case where a DRIVER moves to a small island near Tahiti, where there are no cars. He is no longer a DRIVER, but his license remains in the LICENSE entity class until it expires. In this case, the minimum cardinality of DRIVER would be zero.

Make sure that your model reflects nuances such as this. The only way to get it right is to question the users carefully on how they view their reality. [Figure 4-9](https://viewer.books24x7.com/assetviewer.aspx?bookid=12764&chunkid=620159512&resumebookmarkid=13b44b1d-6043-ee11-aa72-005056b54d63#) shows the same relationship as [Figure 4-8](https://viewer.books24x7.com/assetviewer.aspx?bookid=12764&chunkid=620159512&resumebookmarkid=13b44b1d-6043-ee11-aa72-005056b54d63#), but with a different minimum cardinality value, based on a different users’ data model.

**Driver > Driver-Licence < Licence**

In most cases, the minimum cardinality of an entity will be either zero or one. It is zero if the existence of at least one instance of the entity is *not* required, and it is one if the existence of at least one instance *is* required. In some cases, however, the minimum cardinality for an entity may be more than one. For example, a baseball team must have at least nine players to engage in a game of baseball. Any group of athletes that consists of less than nine players, is thus not a baseball team. You could argue that the minimum cardinality of a baseball team is nine. You could also argue otherwise. It all depends on the users’ data model.

The difference between a minimum cardinality of one and a minimum cardinality of zero is the difference between “must exist” and “need not exist.” This is a pretty big difference. On the other hand, the difference between a minimum cardinality of one and a minimum cardinality of nine is merely a matter of degree. In both cases, something exists. In most systems that you model, you only need to distinguish between those things that must exist and those that need not. The exact value of the minimum number that exists is usually not of major concern as long as that minimum number is not zero.

### **Refining the E-R Model**

Many real world systems can be modeled using entities, attributes, identifiers, and relationships, incorporating the notions of maximum and minimum cardinality. However, for many other systems, those elements alone cannot be combined in such a way as to adequately match the users’ model of the system. For those cases, the E-R model must be extended in any of several ways to provide a better match. A major extension is the recognition that not all entities are alike. There can be different kinds of entities.

### **Strong Entities & Weak Entities**

In previous sections of this chapter, I talk about entities as if they were roughly equivalent to each other, at least conceptually. For example, a retail database might have a CUSTOMER entity class, a SALESPERSON entity class, and a PRODUCT entity class. These three classes might all be related to each other through a SALES\_ORDER entity class. [Figure 4-10](https://viewer.books24x7.com/assetviewer.aspx?bookid=12764&chunkid=302313285&resumebookmarkid=13b44b1d-6043-ee11-aa72-005056b54d63#) shows one possible E-R diagram for such a system.

**Customer > Sales\_Order < Salesperson**

**I**

**Product**

Business rules may vary from one company to another, but one reasonable set might comprise the following rules:

* A CUSTOMER can exist who has not yet bought anything.
* A SALESPERSON can exist who has not yet sold anything.
* A PRODUCT can exist which has not yet been sold.
* A SALES\_ORDER can exist without a CUSTOMER, SALESPERSON, or PRODUCT.

The first three rules seem reasonable, based on how the user views the business. You might consider anyone on your mailing list to be a customer, whether or not they have bought anything yet. You would certainly consider a newly hired salesperson to be a salesperson, even if she had not sold anything yet. Products sitting on your shelves, but as yet unsold, are still products.

However, the fourth rule might require some explaining. Perhaps someone walks into the business, buys a special service for cash, and then walks out. There is no opportunity to enter this person into the CUSTOMER table, the special service is not listed in the PRODUCT table, and the technician who performed the service is not listed in the SALESPERSON table.

Nevertheless, a SALES\_ORDER is generated to record the sale. So, in this four-entity E-R model, none of the four entities depend on any of the others for its existence. They all can exist independently of the other entities.

Now consider the DRIVER-LICENSE relationship that I mention in previous sections of this chapter. A driver can exist without a driver’s license, but a driver’s license cannot exist without a driver. A driver’s license is existence-dependent on a driver. An entity that is existence-dependent on another entity is called a [*weak entity*](http://viewer.books24x7.com/assetviewer.aspx?bkid=12764&destid=817#817). Any entity that is not a weak entity is called a *strong entity*. Thus, LICENSE is a weak entity, and DRIVER is a strong entity.

You denote weak entities by rectangles with rounded corners. The diamond showing the maximum cardinality of a relationship between a weak entity and its associated strong entity also has rounded corners. [Figure 4-11](https://viewer.books24x7.com/assetviewer.aspx?bookid=12764&chunkid=302313285&resumebookmarkid=13b44b1d-6043-ee11-aa72-005056b54d63#) shows the DRIVER-LICENSE relationship, with LICENSE displayed as a weak entity.

**Driver > Driver-Licence < Licence**

### **ID-Dependent Entities**

A special class of weak entity depends on a strong entity not only for its existence, but also for its very identity. Consider the case of a seat on an airline flight. A seat cannot exist unless it is installed in an airliner. Thus, the identifier 10-B by itself does not specify a seat. However, seat 10-B on today’s flight 1372 from San Jose to San Diego does specify a seat adequately. In this case, seat 10-B is a weak entity that depends on the strong entity flight 1372 for its existence. It also depends on flight 1372 for its identity, because the identifier for the flight is part of the identifier for the seat. SEAT is ID-dependent on FLIGHT. [Figure 4-12](https://viewer.books24x7.com/assetviewer.aspx?bookid=12764&chunkid=302313285&resumebookmarkid=13b44b1d-6043-ee11-aa72-005056b54d63#) shows the SEAT-FLIGHT relationship.

**Flight > Flight-Seat < Seat**

### **Supertype & Subtype Entities**

An entity class might include instances that share some characteristics with all other instances, but have other characteristics that they do not share with all other instances. Consider a college community. It will have students, faculty members, and staff. Students are working toward a degree. Faculty members teach classes and do research. Staff employees, such as the president and the janitors, provide other services besides teaching and research.

All members of the community have some common characteristics, such as name, home address, telephone number, and e-mail address. Other characteristics are unique to each group. Students would have a grade point average, but faculty and staff would not. Faculty would have an academic rank, but students and staff would not. Staff would have a job category, but students and faculty would not.

In this example, STUDENT, FACULTY, and STAFF are all subtypes of the supertype COMMUNITY.

An important feature of supertype/subtype relationships is the notion of *inheritance*. The subtypes in such a relationship inherit all the attributes of the supertype entity. The inherited attributes combine with the attributes that are exclusively held by the subtype to provide a total description of the subtype entity.

Thus, all members of the COMMUNITY have the attributes Name, HomeAddress, TelephoneNumber, and EmailAddress. Only members of the STUDENT subtype would also have the GradePointAverage attribute. Only members of the FACULTY subtype would have the AcademicRank attribute, and only members of the STAFF subtype would have the JobCategory attribute. [Figure 4-13](https://viewer.books24x7.com/assetviewer.aspx?bookid=12764&chunkid=302313285&resumebookmarkid=13b44b1d-6043-ee11-aa72-005056b54d63#) is an E-R diagram of a supertype-subtype relationship.

**Community**

**I**

**Student - Faculty - Staff**

| **COMMUNITY** | **STUDENT** | **FACULTY** | **STAFF** |
| --- | --- | --- | --- |
| **Name** | **GradePointAverage** | **AcademicRank** | **JobCategory** |
| **HomeAddress** | **Advisor** | **Department** | **JobTitle** |
| **TelephoneNumber** | **Class** | **PhoneExtension** | **PhoneExtension** |
| **EmailAddress** |  |  |  |

In [Figure 4-13](https://viewer.books24x7.com/assetviewer.aspx?bookid=12764&chunkid=302313285&resumebookmarkid=13b44b1d-6043-ee11-aa72-005056b54d63#), the **ε** next to the relationship lines shows that STUDENT, FACULTY, and STAFF are subtypes of COMMUNITY. The curved line indicates that every member of COMMUNITY must be a member of one of the three subtypes, and cannot be a member of more than one subtype.

A supertype member does not necessarily have to belong to a subtype, nor must subtypes be mutually exclusive. Consider American Major League Baseball players. They all share certain characteristics, including name, height, weight, position played, as well as such offensive statistics as at-bats, hits, runs, stolen bases, and runs batted in. Every player could at least potentially be called upon to go to the plate and try to get a hit, although weak-hitting pitchers and pinch runners may never be called upon to do so.

Weak-hitting pitchers are rarely called upon to hit because in the American League a manager may use a designated hitter to hit for any player in the lineup. This is usually the pitcher. Pitchers in general are notoriously weak hitters because they must spend so much time concentrating on pitching, they are not able to practice hitting as much as the other players.

Players who are not pitchers, but play other defensive positions, such as first base, shortstop, or center field, are called position players. Position players have defensive statistics, including putouts, assists, and errors. Pitchers share those characteristics but have additional ones such as innings pitched, strikeouts, walks, wins, losses, and saves.

All players have offensive statistics, but designated hitters, pinch hitters, and pinch runners that never play the field would not have defensive statistics. Position players do have defensive statistics. Pitchers have defensive statistics and also have pitching statistics. Pitchers and position players are subtypes of the supertype baseball player.

[Figure 4-14](https://viewer.books24x7.com/assetviewer.aspx?bookid=12764&chunkid=302313285&resumebookmarkid=13b44b1d-6043-ee11-aa72-005056b54d63#) shows an E-R diagram for baseball players. PLAYER is the supertype that includes the PITCHER and POSITION-PLAYER subtypes. Designated hitters do not fall into either of these categories, but are still members of the PLAYER supertype. Also, it is possible that a player who is normally a position player might be called upon to pitch in an extra-innings game after all the regular pitchers have been used. In this case, the subtypes are not mutually exclusive.

**Player**

**I**

**Pitcher - Position-Player**

The *m* next to the curved line means that a member of PLAYER may belong to from zero to many subtypes. A designated hitter would belong to zero.

A position player who also occasionally pitched would belong to multiple subtypes.

### **Capturing Business Rules**

In previous sections of this chapter, I make occasional references to business rules. Business rules are formal statements about how an organization does business. For example, one university may have a rule that a person may not simultaneously be a student and a member of the faculty. Another university may allow faculty to take classes for credit, thus making them students.

Business rules vary from one organization to the next, even though the organizations seem otherwise to be virtually identical. Even two branch offices of the same company may differ in their employee relations policies, for example. You must interview the actual users at the location where your program will be used and ask them detailed questions about the way they conduct their business. From their responses, you can infer the appropriate business rules.

To ensure that your database application accurately reflects the way your clients conduct their business, you must capture all the important business rules. You should interview people representing every group that has a stake in the application, and ask every question you can think of, to maximize the accuracy of your user’s model.

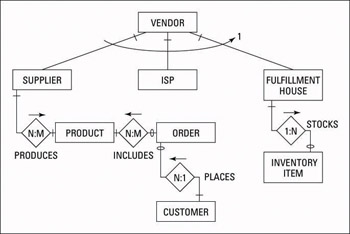
### **It’s Time to Look at Some E-R Examples**

One of the better ways to understand a fairly complex concept is to carefully examine one or two examples, so that you can relate theory to actual practice. In this section, I show you two examples of usage of the E-R model — one fairly simple, and the second more complex. After you understand these examples, you should be able to construct E-R models of your own.

### **A Fairly Simple Example**

Consider a small dot-com business named Mistress Treasure that sells women’s intimate apparel from its Web site. The business displays its products and takes credit card orders on the site. There is no “brick and mortar” store. Fulfillment is outsourced to a fulfillment house, which receives and warehouses products from vendors, and then, upon receiving orders from Mistress Treasure, ships orders to customers.

The Web site front end consists of HTML pages that include text descriptions and graphic images of products, as well as a virtual shopping cart and forms for capturing customer and payment information. The Web site back end contains a database that stores customer, payment, inventory, and shipment status information. [Figure 4-15](https://viewer.books24x7.com/assetviewer.aspx?bookid=12764&chunkid=159974850&resumebookmarkid=13b44b1d-6043-ee11-aa72-005056b54d63#) shows an E-R diagram of the Mistress Treasure system. It is a very small system for a very small “boutique” business.



Note that VENDOR is a supertype of the SUPPLIER, ISP, and FULFILLMENT\_ HOUSE subtypes. Shared attributes of all vendors belong to the VENDOR entity, while attributes that are unique to each subtype belong to either the SUPPLIER, ISP, or FULFILLMENT\_HOUSE entities.

A many-to-many relationship exists between SUPPLIER and PRODUCT, as well as between PRODUCT and ORDER. [Chapter 7](http://viewer.books24x7.com/assetviewer.aspx?bkid=12764&destid=275#275) explores the problems that such relationships can cause, as well as techniques for overcoming those problems. For now, you just want to create an E-R model that matches the users’ data model as closely as possible.

Other relationships in the model are one-to-many. The model has no one-to-one relationships. Often, a one-to-one relationship between two entities means they are actually two aspects of the same entity. In such cases, you should combine the attributes of both into a single entity. In other cases, however, keeping the two entities separate makes sense. For example, if some attributes are public information and others are confidential, assign the public attributes to one entity and the confidential attributes to the second entity.

#### **A more complex example**

Most real-life applications are more complex than the Mistress Treasure example that I describe in the preceding section. Consider the Sawbones Placebo Clinic, which treats hypochondriacs with a variety of therapies. By interviewing the clinic manager, Pat Answer, you find that she wants to keep records on all employees, and also track the treatment of patients by clinic staff. She tells you what her mental model of the clinic is, and what information she wants to capture for later review and analysis. You listen carefully to her description, either writing it down or recording it for later transcription.

Sawbones employs doctors, nurses, medical technologists, and medical assistants. The company extends certain benefits to employees and to their dependents. Doctors, nurses, and medical technologists all must be licensed by a recognized licensing authority. Medical assistants may be certified, but need not be, to be employed in a hospital or clinic.

Typically, a patient will see a doctor, who will examine the patient and then order one or more tests. A medical assistant or nurse will take samples of the patient’s blood, urine, and so on, and take the samples to the laboratory. In the lab, a medical technologist will perform the tests that the doctor ordered on these samples, usually called specimens. The medical technologist then sends the test results to the primary care physician who ordered the tests, as well as perhaps one or more consulting physicians. Based on the test results, the doctor, in consultation with the consulting physicians, makes a diagnosis of the patient’s condition and prescribes a course of treatment. One or more nurses then administer the prescribed treatment to the patient.

To build a database that meets the client’s needs, using the tool of the Entity-Relationship diagram, the first step is to identify those important items that should be considered entities. By carefully listening to Ms. Answer and asking for clarification frequently, you come up with the following list of candidate entities:

Employee

Manager

Doctor ( physician)

Nurse

Medical technologist

Medical assistant

Benefits

Dependents

Patients

Doctor’s license

Nurse’s license

Medical technologist’s license

Medical assistant’s certificate

Examination

Test order

Test

Test result

Consultation

Diagnosis

Prescription

Treatment

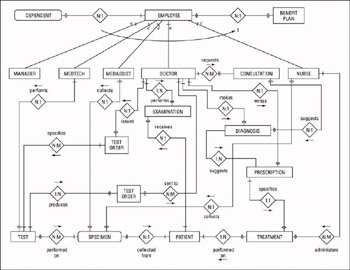
This list is a first cut at an entity list. You build the list by extracting the major nouns that Pat Answer used in her description of the business. Some of these nouns definitely represent entities and others do not. After listing them all, go over the list carefully and eliminate those items that are perhaps attributes of entities, as well as those that are not relevant to the client’s goals for your database solution.

Pat has told you that she wants to keep track of employees, and mentioned several different types of employees. She also made a point of mentioning benefits and the fact that employee dependents receive some of those benefits. From this information, you conclude that EMPLOYEE, MANAGER, DOCTOR, NURSE, MEDTECH, and MEDASSIST are entities, while MANAGER, DOCTOR, NURSE, MEDTECH, and MEDASSIST are subtypes of the supertype EMPLOYEE. You also assume that DEPENDENT is an entity that you will be concerned about. You find upon asking, that the company provides only one benefit plan that applies to all employees equally. You conclude that BENEFITPLAN should also be an entity.

Because doctors, nurses, and medical technologists must possess current licenses in order to legally practice their professions, it is important to keep track of license status for these professionals. However, because each professional can have one and only one license for its job category, you assume that these licenses are attributes of their respective professional’s entity. The same would be true for medical assistant certification.

PATIENT clearly deserves to be an entity, as does EXAMINATION. These are things that questions might arise about later. The examination is the first step in a chain of events, each represented by an entity. The TESTORDER is an entity, as are TEST and TESTRESULT. You may want to record the physicians that took part in the CONSULTATION that arrived at a DIAGNOSIS and issued a PRESCRIPTION. Finally, the results of the TREATMENT should be recorded.

Knowing the entities, you can start thinking about how they relate to each other and what the minimum and maximum cardinalities of those relationships should be. [Figure 4-16](https://viewer.books24x7.com/assetviewer.aspx?bookid=12764&chunkid=159974850&resumebookmarkid=13b44b1d-6043-ee11-aa72-005056b54d63#) shows one possible E-R diagram that represents my understanding of the users’ model. Other interpretations are possible, and could be equally valid.

****

This diagram reflects certain things you know about the system. You can state these things in a series of sentences:

* An EMPLOYEE can have from zero to many DEPENDENTs, but a DEPENDENT must be associated with one and only one EMPLOYEE.
* An EMPLOYEE could be a MANAGER, MEDTECH, MEDASSIST, DOCTOR, or NURSE.
* A DOCTOR can perform many EXAMINATIONs, but each EXAMINATION is performed on one and only one PATIENT.
* A DOCTOR can issue many TESTORDERs, and each TESTORDER can specify multiple TESTs.
* A MEDASSIST or a NURSE can collect multiple SPECIMENs from a PATIENT, but each SPECIMEN is from one and only one PATIENT.
* A MEDTECH can perform multiple TESTs on a SPECIMEN, and each TEST might be performed on many SPECIMENs.
* A TEST can produce multiple TESTRESULTs, but each TESTRESULT is associated with one and only one TEST.
* A TESTRESULT is sent to one or more DOCTORs.
* A DOCTOR may request a CONSULTATION with one or more other DOCTORs.
* A CONSULTATION may suggest a DIAGNOSIS.
* A DOCTOR may make a DIAGNOSIS of a PATIENT, based on one or more TESTRESULTs and possibly one or more CONSULTATIONs.
* A DIAGNOSIS could suggest multiple possible PRESCRIPTIONs.
* A DOCTOR writes many PRESCRIPTIONs for the TREATMENT of PATIENTs.
* Each PRESCRIPTION specifies one and only one TREATMENT for each PATIENT.
* One or more NURSEs administer a TREATMENT to a PATIENT.

-----------------------------------------------------------------------------------------

## Domain

**Pre-Assessment**

Q1: Data Storage (ID 27793) = V

Within a database, what is data stored in?

Entities

Q2: Primary Key Field (ID 27794) = V

What is a primary key field?

A field that ensures every record in a table is unique

Q3: Date Format Fields (ID 27795) = V

Which data field type would be used to store a value of 0 or 1?

Bit

Q4: Text Fields (ID 27796) = V

Which two data field types are text fields?

Varchar & Nvarchar

Q5: Smalldatetime Fields (ID 27797) = V

Which two statements about a smalldatetime field are true?

Has a date range from 1900 to 2079

Time can be calculated to the second but no further

Q6: One-to-One Relationships (ID 27798) = V

What is a characteristic of one-to-one relationships?

They use primary keys from two tables

Q7: ERD Creation (ID 27799) = V

In the database design and building process, when should an entity-relationship diagram (ERD) be created?

Before a database is built.

Q8: Orphaned Records (ID 27800) = V

What ensures that orphaned records do not occur?

Referential Integrity

Q9: Normalization (ID 27801) = V

Which statement best defines normalization?

The process of ensuring that an entity in a database has a solid design and that the data in the entity can work well with other entities

Q10: Backing Up a Database (ID 27802) = V

It is important to make sure a database is backed up regularly, so that the data can be \_\_\_ if it is accidentally deleted or changed.

Restored

**Workbook & Support Files**

*Databases\_Support\_Files\_Student folder in Downloads*

## Videos

### Primary Key Rules (8m)

### Understanding Data Types (8m)

### Benefits of Normal Forms (9m)

### Understanding the Atomic Value (5m)

### Privileges and Access Rights (7m)

## Exercises

### Normalization

### ERD (Entity Relationship Diagrams)

## Quiz

Q1: For each statement, select the correct answer from the dropdown menu:

An \_\_\_ is a graphical representation of entities used in the design of a database.

**Entity Relationship Diagram (ERD)**

An \_\_\_ is an object or concept about which data is stored in a database.

**Entity**

A \_\_\_ is how the data is shared between entities in a database.

**Relationship**

Q2: Match the memory storage requirement to the database field type, selecting the correct answer from the dropdown menu:

2 times n bytes

**Nvarchar**

8 bytes

**Datetime**

4 bytes

**Smallmoney**

Stores the number of characters entered plus one more byte to store the length

**Varchar**

Q3: Match the data field type to its number range, selecting the correct answer from the dropdown menu:

0 to 255

**Tinyint**

-32,768 to 32,767

**Smallint**

-922 trillion to 922 trillion

**Money**

-214,000 to 214,000

**Smallmoney**

-2 billion to 2 billion

**Integer**

Q4: For each statement below, fill in the blanks using the dropdown menu:

\_\_\_ is a command used to provide access or privileges on a database object (such as a table/entity) to a user or a group of users.

**GRANT**

\_\_\_ conveys a privilege or role to a user with the right to grant the same privileges or role to other users.

**WITH GRANT OPTION**

\_\_\_ is a command that removes user access rights or privileges to a database object (such as a table/entity).

**REVOKE**

\_\_\_ is a collection of privileges or access rights. One can grant or revoke privileges to one of these.

**Role**

Q5: Fill in the blanks using the dropdown menus:

A \_\_\_ is used to link two tables together.

**Foreign Key**

It is a column, or set of columns in the child table, which references a \_\_\_ column or set of columns, in the parent table.

**Primary Key**

Q6: Fill in the blank using the dropdown menu:

The \_\_\_ data type has a lower range of 1753-01-01 and an upper range of 9999-12-31.

It can define a date and time with fractional seconds (1/1000th of a second)

**Datetime**

Q7: What do primary keys usually consist of?

Multiple Rows

A single column

Multiple Columns

A single row

**A single column**

Q8: Which database term is described below?

It is a data record within a table (also known as a tuple).

It represents a single, implicitly structured data item in a table.

A field

A row

An entity

A column

**A row**

Q9: Which type of database backup is described below?

A \_\_\_ backup is a cumulative backup of all files changed since the last backup.

Differential

Full

**Differential**

Q10: Which of the following data types takes up the most memory?

Tinyint

Money

Int

Date

**Money**

Q11: Select the correct answer to fill in the gap:

A bit field is usually used as a \_\_\_ field.

Numeric

Text

Boolean

Date

**Boolean**

Q12: Which step in the normalization process is described below?

Any attributes that are not directly dependent on the primary key (they are more dependent on a non-primary key column in the table) are moved to a new table.

They are referred to as ‘transitive dependencies’.

3NF

0NF

1NF

2NF

**3NF**

Q13: Which step in the normalization process is described below?

In each table, the non-primary key columns are fully dependent on the whole of the composite primary key (not just on a part of the composite primary key). These are referred to as ‘partial dependencies’.

0NF

2NF

1NF

3NF

**2NF**

Q14: Which step in the normalization process is described below?

Each table must have a primary key

Eliminate repeating groups

2NF

0NF

1NF

3NF

**1NF**

Q15: What is the principle of ‘least privilege’?

A principle that states that people should have access to all permissions.

A principle that states that people should have access to all permissions, unless there is no need for them to have access to a permission.

A principle that states that people should only have the permission they need to do their job.

A principle that states that people should only have the permissions they request.

**A principle that states that people should only have the permission they need to do their job.**

Q16: Refer to the image below; what is the displayed graphic called?



Entity Resolution Diagram (ERD)

Entity Relationship Diagram (ERD)

Entry Relationship Diagram (ERD)

Entry Resolution Diagram (ERD)

**Entity Relationship Diagram (ERD)**

Q17: What database design concept is described below?

In a database for a public library, imagine that there are two tables. One table stores data about borrowers (Borrowers). The other table stores data about book fines on borrowed books (Fines). The tables share a relationship using the BorrowerID field, which is a primary key in the Borrowers table and a foreign key in the Fines table. A record in the Borrowers table cannot be deleted if related records exist in the Fines table.

Database backups

1NF

The principle of least privilege

Referential integrity

Database indexing

**Referential integrity**

Q18: Which database design feature is being described below?

This key is a combination of two or more columns in a table that when combined guarantees the uniqueness of records in a table. The columns individually do not guarantee the uniqueness of the records but together, they do.

An entity

A foreign key made up of one column

A composite primary key

A primary key made up of one column

**A composite primary key**

Q19: Which type of relational database model is described below?

In this model, a single record in the parent table relates to multiple records in the child table. For example, in a database created for a public library, the borrowers' contact information is stored in a Borrowers table. Data on book fines issued to members is stored in a Fines table. The Borrowers table is the parent table and the Fines table is the child table. The common field will be the BorrowerID (pk) included in the Fines table.

None of the other stated answers are correct

One-to-Many

One-to-One

Many-to-Many

**One-to-Many**

Q20: Which type of relational database model is described below?

You are creating a database to store employee data. You wish to make sure that sensitive data (ex. salary) is kept separate from non-sensitive data (ex. job title). You decide to create two tables, one to store the non-sensitive data and one to store sensitive data. The two tables will be related using a (unique) EmployeeID field.

None of the other stated answers are correct

One-to-Many

Many-to-Many

One-to-One

**One-to-One**

Q21: Which type of relational database model is described below?

You are creating a database for a Pizza delivery shop. You propose to store data about sales invoices in a table named Invoices and data about the different pizzas in a Pizza table. Any given pizza can appear in many sales invoices and any given sales invoice can contain many pizzas.

To solve this dilemma, you decide to create a third table, 'a Junction table' that is the child of the aforementioned tables. Into this Junction table, you put the primary keys of the aforementioned tables along with the detail data for the invoice, such as invoice number, invoice date etc.

None of the other stated answers are correct

One-to-Many

Many-to-Many

One-to-One

**Many-to-Many**

Q22: Which data field type stores data as either a zero or one?

Varchar

Bit

Datetime

Date

**Bit**

Q23: What database term is described below?

A \_\_\_\_\_\_\_\_ is a set of data values, of the same data type, in a table. It is also referred to as a column or an attribute.

Row

ERD

Entity

Database Field

**Database Field**

Q24: Which data field type allows for Unicode characters?

Nvarchar

Bit

Varchar

Datetime

**Nvarchar**

Q25: If you need to restore a database from a differential backup, you first need to restore the last full backup of the database and then the most recent differential backup of the same database.

Is this statement True or False?

**True**