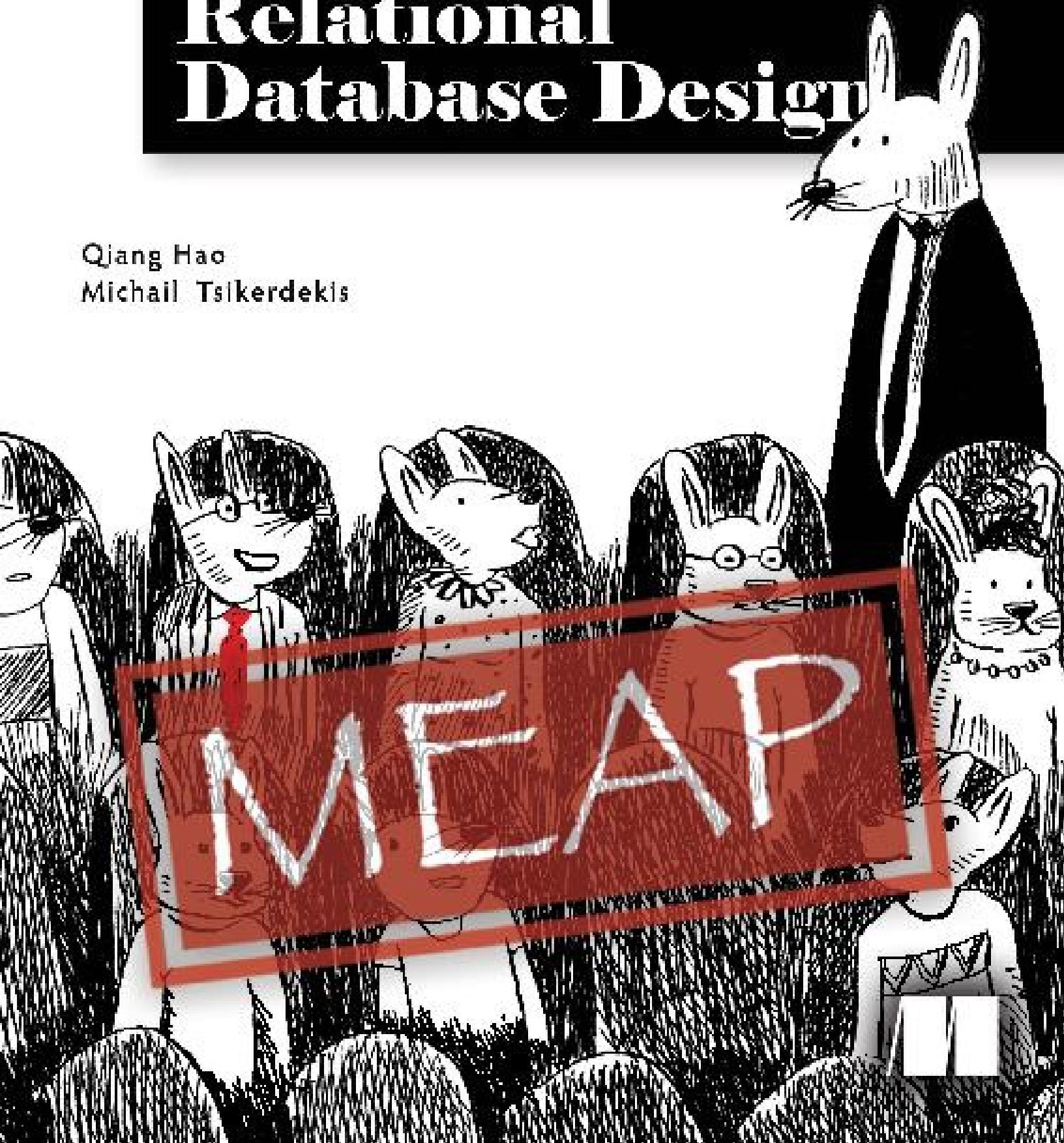


**grokking**

# Relational Database Design

Qiang Hao  
Michail Tsikerdekis



**grokking**

# **Relational Database Design**

Qiang Hao  
Michail Tsikerdakis



# Grokking Relational Database Design

1. [welcome](#)
2. [1\\_Introduction\\_to\\_databases\\_and\\_SQL](#)
3. [2\\_Related\\_tables\\_and\\_more\\_SQL](#)
4. [3\\_Overview\\_of\\_database\\_design](#)
5. [4\\_Entities\\_and\\_attributes](#)
6. [5\\_Relationships](#)

# welcome

Dear reader:

Thanks for purchasing the MEAP of *Grokking Relational Database Design*.

Relational database design is relevant to a broad array of fields where data management and information storage are critical, such as information technology, software development, data science, and business intelligence. If you study or work in such a field, and you seek personal growth or career advancement, it is crucial for you to have a solid understanding of relational database design. After all, relational databases continue to be the standard approach for data storage in many industries. Understanding relational database design can serve as a foundation for you to learn other related concepts and systems, such as data warehousing, data mining, and NoSQL databases.

We have been teaching database design and management for years, and we understand the barrier to learning this topic deeply. We appreciate the opportunity from Manning to write a book on relational database design, and want to make this book intuitive, informative, and, most importantly, appropriate for anyone who wants to learn a bit about this topic. As a result, we write this book with no assumption of your prior knowledge in programming or databases. If you have a high school education, you can read this book and understand how to design relational databases from scratch.

We divided this book into three parts. In Part I, you will learn the essentials of Structured Query Language (SQL) and its use to query and create tables and databases; you will also get an overview of the process of database design.

In Part II, you will learn the key steps in database design by designing a database for an online market from scratch. Some of the key steps include identification of entities and attributes, development of entity-relationship diagrams, and database normalization.

In Part III, you will get a glimpse into using generative AI to facilitate database design by designing a database for a spacecraft manufacturer. You will evaluate the proposals and answers from ChatGPT given specific design requirements, and explore how to use tools like ChatGPT effectively.

Your feedback is invaluable in improving this book, and we hope that you will share your thoughts and comments in the [liveBook Discussion forum](#). We appreciate knowing where we can make improvements and increase your understanding of the material.

—Dr. Qiang Hao and Dr. Michael Tsikerdeks

#### **In this book**

[welcome](#) [1 Introduction to databases and SQL](#) [2 Related tables and more SQL](#) [3 Overview of database design](#) [4 Entities and attributes](#) [5 Relationships](#)

# 1 Introduction to databases and SQL

## In this chapter

- You get a foundation for the rest of the book.
- You learn the basics of relational databases.
- You peek into database design.
- You write your first SQL query and learn more about the basics of SQL.

### What you need to know

As you read this chapter, you will find some code snippets. If you want to execute those code snippets or see what changes need to be made to the code for different RDBMS, you will need to check the GitHub repository that accompanies this book (<https://github.com/Neo-Hao/grokking-relational-database-design>). You can find the scripts for this chapter in the chapter\_01 folder and follow the instructions in the README.md file to run the scripts.

## 1.1 Overview

Database design is a critical yet easily neglected step in software development. Nearly every application requires data storage and management to some extent, but not every application has a well-designed database. If you design a database without knowing the principles of effective database design, your application may suffer from a series of issues you weren't expecting, such as disorganized data or queries taking too long and too many resources to run. Such issues can lead to bugs and a bad user experience.



In contrast, effective database design can serve as a solid foundation for effective software development. Effective database design makes sure that the data of an application is well organized and structured, which in turn supports efficient data querying and manipulation that contributes to solid applications and superior user experience. As such, regardless of where you are in your journey of learning programming and software development, it is essential to learn how to design databases effectively, and possibly also learn how to talk to non-tech people without making their eyes glaze over with boredom.

This book covers how to design databases, and assumes no prior knowledge in databases or programming. By the end of this book, you will have a good understanding of how to design relational databases from scratch. We aim to help you achieve this goal via down-to-earth definitions and explanations, rich examples, and active learning practice.

This chapter aims to introduce relational databases, define a set of terms that you will see in the next few chapters, and cover the basics of *Structured Query Language (SQL)*. SQL (pronounced as *Sequel*) is the programming language used to manage data in relational databases, and some understanding of SQL is essential for you to have a robust understanding of database design.

## 1.2 Relational databases

Once upon a time, a small company used spreadsheets to store all its customer data. At first, everything seemed to be running smoothly, and the company was able to access and update the data as needed. As time went on, the company grew and acquired more customers, and the Excel spreadsheets became increasingly difficult to manage. There were duplicates and inconsistencies in the data, and the spreadsheets became so large and unwieldy that they took a long time to load and update.

One day, the company received a call from a customer who had been charged twice for a single purchase. When the company tried to access the customer's data in the spreadsheet to investigate the issue, they found that the data had been corrupted and was no longer accessible. As more and more customers began to report similar issues, the company learned the hard way that spreadsheets are a bad choice for storing customer data. The company eventually invested in a relational database system that could handle the scale of their data and ensure the integrity of their records.

If you have a very small amount of data with a simple structure to store, a spreadsheet can get the job done and you don't need a database at all. However, as the data complexity and volume increase, you may find a spreadsheet unwieldy. When you need to apply access control to the data, maintain its consistency, integrity and scalability, as well as conduct routing data analysis, you will absolutely need a database.

*Relational databases* have been and still are the default technology for storing and accessing data when scale, data consistency, and data integrity are all required. In this section, you will learn some fundamental concepts in relational databases, such as tables, entities, and relational database management systems.

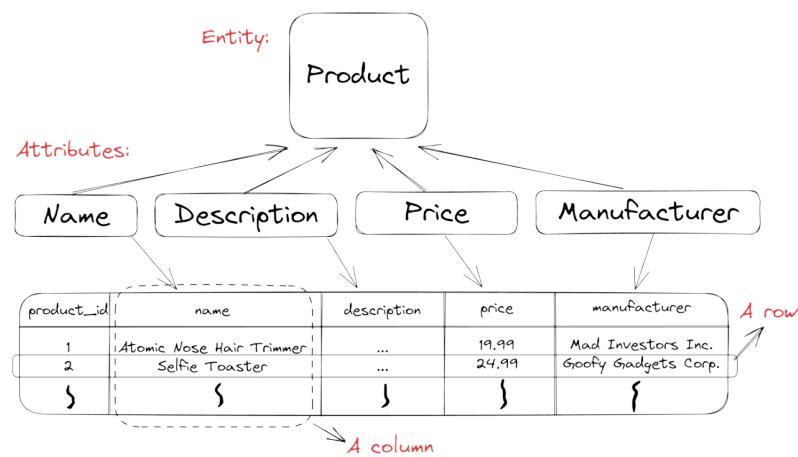
### 1.2.1 Tables, entities, and primary keys

A *relational database* is a collection of tables that store data. A *table* is like a spreadsheet, which you are likely familiar with. Just like a spreadsheet, the data in a table is organized into rows and columns. A table can be used to

represent an *entity* or a relationship between entities, where each row represents a single data record of an entity, and each column represents an *attribute* of that entity.



What is an entity? An entity is an object or concept that can be described by many attributes. For example, imagine that we are running an online store called The Sci-Fi Collective which sells sci-fi products (e.g., a time machine that only takes you back five minutes, just in case you forgot your keys). Products sold by our online store are an entity, and they can be described by at least four attributes, including name, description, price, and manufacturer. When we map products to a table in the database supporting the online store of The Sci-Fi Collective, the four attributes will be mapped to four individual columns, and each product will be represented as a row in this table.



In addition to the four columns, you may notice that we added another column `product_id` in the above table. All values in the `product_id` column are unique, and can be used to identify an individual row. We call the `product_id` the *primary key* of the product table. Interesting fact about

primary keys per table: much like in the popular Highlander series, there can be only one. You can find a much deeper discussion on primary keys later in Chapter 3.

It is not uncommon that one spreadsheet is used to store the information of multiple entities. You may be asking: Can we do the same with tables in a relational database? For example, if we decide to store the information of customers and the products in one table for The Sci-Fi Collective, the table will look like the following:

product_id	name	price	manufacturer	customer_id	customer_name	customer_email	quantity
1	Atomic Nose ...	19.99	Mad Inventors Inc.	a1	Bob	bob@gmail.com	5
2	Selfie Toaster	24.99	Goofy Gadgets Corp.	b2	Dave	dave@outlook.com	15
3	Cat-Poop Coffee	29.99	Absurd Accessories	a1	Bob	bob@gmail.com	2
...	...	...	...	...	...	...	...
9	The Infinite ...	9.99	Silly Supplies Co.	j8	John	john@123.net	1
10	The Neuralyzer	33.55	Silly Supplies Co.	p9	Katy	katy@123.net	2

This is a typical poorly designed table. Beyond the data redundancy that you can easily spot, such a design can cause many unexpected problems. For example, if a customer's information only appears in one row, when we want to delete a product, we will have to delete the customer in the same row from our database. This issue is known as *delete anomaly*. For another example, from time to time, we need to insert a product into this table that has not been bought by any customers. However, the table requires valid customer information to be provided whenever a new row is added. The contradicting requirements will leave us in awkward situation in which we can't add any new products at all. This issue is known as *insert anomaly*.

As you can see, bad database design can lead to issues that negatively impact software quality. To avoid such issues, it is necessary to master the basic principles and best practice of database design.

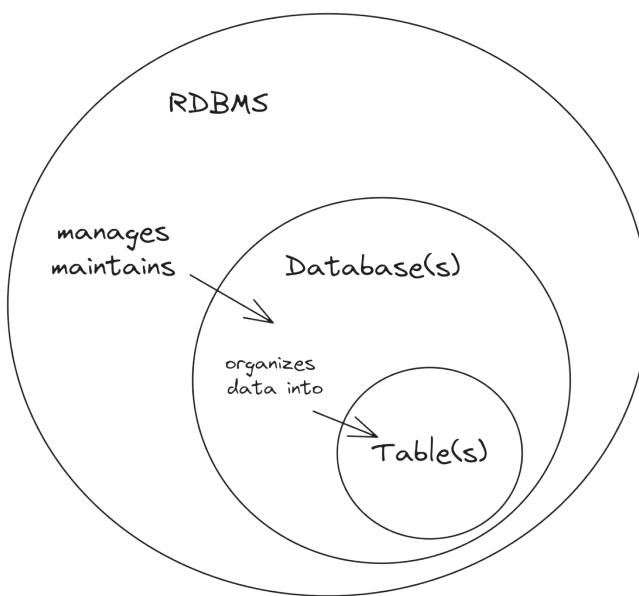
## 1.2.2 Relational database management systems and Structured Query Language

Relational databases and tables rely on the help from *relational database*

*management systems* (RDBMS) to physically store and manage the data. Edgar Codd at IBM developed the first RDBMS in the 1970s.



What is an RDBMS? An RDBMS is software that interacts with the underlying hardware and operating system to physically store and manage data in relational databases. Additionally, an RDBMS provides tools to create, modify and query databases along with other important functionalities (e.g., security controls). You might be familiar with some of the commonly used RDBMS such as SQLite, MySQL, MariaDB, and PostgreSQL. When you need to deploy a database that you designed, you will need to interact with one of the available RDBMS on the market.



One of the most notable tools supported by nearly all RDBMS is SQL. SQL is a programming language that you can use to create, modify, and query data stored in tables in an RDBMS. Although different RDBMS vendors may

implement their own variations and extensions, SQL has been standardized over the years. As such, the consistency of SQL among different RDBMS is high, and the variations don't matter that much in the context of this book.

Of course, this being primarily a database design book, SQL may seem less important. Database design doesn't necessarily require you to use SQL. Some RDBMS come with graphical tools that can generate SQL scripts that automatically create databases and tables based on your design. However, some understanding of SQL can make it easier for you to learn database design, especially when it comes to structural or design issues such as data integrity, optimization, and scalability. After all, SQL is a standardized language used by most RDBMS. Knowing SQL will allow you to rely less on graphical tools and work with different types of RDBMS. As a result, we will cover the basics of SQL in this chapter and Chapter 2.

## 1.3 Your first SQL query

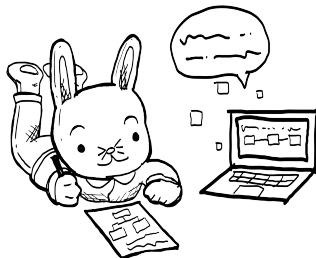
You will learn SQL by executing your first SQL query in this section. We will use the same example that you saw in the last section, the database of The Sci-Fi Collective (because who doesn't like Sci-Fi stuff?). The database contains many tables, but the product table is all you need to focus on for now. The product table looks like the following:

product_id	name	description	price	manufacturer
1	Atomic Nose Hair Trimmer	...	19.99	Mad Inventors Inc.
2	Selfie Toaster	...	24.99	Goofy Gadgets Corp.
3	Cat-Poop Coffee	...	29.99	Absurd Accessories
...	...	...	...	...
9	The Infinite Improbability Generator	...	9.99	Silly Supplies Co.
10	The Neuralyzer	...	33.55	Silly Supplies Co.

You will first load a prepared SQL script to generate a database and this table. We have prepared SQL scripts that generate the above table with data in different RDBMS, which you can find in our GitHub repository

(<https://github.com/Neo-Hao/grokking-relational-database-design>). You can follow the instructions of the README.md file in the chapter\_01 folder to execute the prepared script in your preferred RDBMS or tool.

For example, the easiest approach is to use SQLite Online:



1. Clone or download our GitHub repository (<https://github.com/Neo-Hao/grokking-relational-database-design>).
2. Navigate to SQLite Online (<https://sqliteonline.com>).
3. Click **Import** and load the script named sqlite\_online.sql from the downloaded or cloned GitHub repository.
4. Click **Okay**.

After that, you will be ready to query the product table. You can type the following query (as a whole) into the code editor on SQLite Online and then click **Run**:

```
SELECT name  
  
FROM product  
WHERE price > 20;
```

So, what does this query do? The “price > 20” may be a dead giveaway. The query retrieves the names of products whose prices are higher than 20. We know that there are 10 rows representing 10 products in the product table, and 5 products (e.g., Atomic Nose Hair Trimmer) sell at a price below 20, so the names of the other 5 products are returned. Your results should look like this:

name
Selfie Toaster
Cat-Poop Coffee
Inflatable Briefcase
Lightsabers
The Neuralyzer

You may notice that this SQL query shares a lot of similarity with plain English. That is because SQL is special. You see, most programming languages are imperative. Coding with an imperative language, such as Java or Python, requires you to specify both what to do and how to do it. However, SQL is declarative, which means that you only need to specify what to do. As far as the steps require to carry out the task, that's for SQL to figure out. Specifying what you want instead of how to get it done is more natural to human beings, and that's why SQL resembles the English language.

SQL is like English with a lot less small talk. However, you don't have all the same freedom in word choices when it comes to putting together a SQL query. You must use a set of SQL clauses (also known as statements) and follow some rules. In your first query, you used the following three clauses:

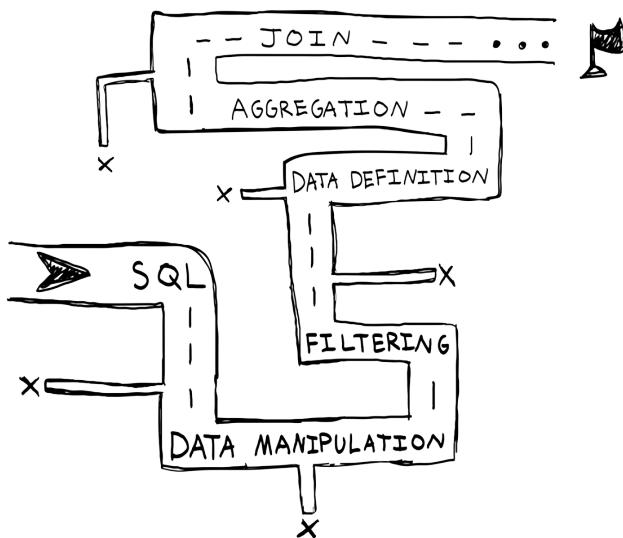
- **SELECT:** The `SELECT` clause allows you to specify the columns you want to retrieve from a table. In your first query, you only asked for the `name` column; thus, the `SELECT` statement was `SELECT name`
- **FROM:** The `FROM` clause is used to specify the source you want to retrieve data from, such as one or more tables. In your first query, you only asked for data from the `product` table; thus, the `FROM` clause was `FROM product`
- **WHERE:** The `WHERE` clause allows you to specify conditions with which to filter the data retrieved by the `SELECT` clause. In your first query, you only want the names of those products whose prices are higher than 20; thus, the query was `SELECT name FROM product WHERE price > 20;`

When you finish a SQL query, you should use a semicolon (`;`) to indicate its

end. The semicolon tells the RDBMS that this is the end of a SQL query, and anything coming after that is a new query.

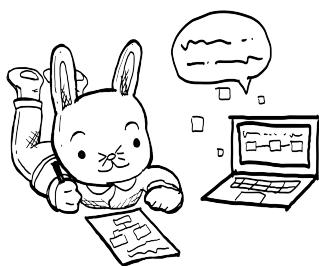
## 1.4 The basics of SQL queries

Our preferred approach for learning SQL is to grasp the most important clauses, and learn the rest only when necessary. Although there are many SQL clauses, they are not equally important. The most important ones can help you build a solid foundation, as well as construct a map in your mind that can guide your future learning.



Therefore, instead of trying to cover every SQL clause, we will cover only the ones that are essential or critical to your future learning. In this chapter, you will learn important clauses that can be used to query a single table.

### 1.4.1 Filtering



Filtering is a common data retrieval task. Whenever you need only a subset

of data that meets some criteria, you need the help from the `WHERE` clause to filter the data.

From your first SQL query, you know that the `WHERE` clause is followed by the criteria that you want to use to filter the data. For example, the query below retrieves the name and description of the products whose prices are lower than 30 from the product table:

```
SELECT name, description  
FROM product  
WHERE price < 30;
```

When you want to retrieve more than one column, you can list all of them after the `SELECT` keyword, and separate them with commas.

What if we only want to retrieve the products that come from a specific manufacturer, such as `Mad Inventors Inc.`? We can achieve this via the following query:

```
SELECT name  
FROM product  
WHERE manufacturer = 'Mad Inventors Inc.';
```

This query will yield the following result:

name
Atomic Nose Hair Trimmer
The Mind Probe
Lightsabers

In the above query, the operator that checks equality is a single equal sign (`=`). Additionally, you may notice that the manufacturer name is wrapped in single quotes (`' '`), which indicates a string data type. Are there different data types in SQL? Yes. SQL data can be broadly divided into six categories, including:

- Numeric data types (e.g., `INT`)

- String data types (e.g., TEXT)
- Date or Time data types (e.g., DATE)
- Unicode character string data types (e.g., VARCHAR)
- Binary data types (e.g., BINARY)
- Miscellaneous data types (e.g., XML)

In the product table, the data type of the manufacturer column is string. In contrast, the price column is numeric.

Now you know how to filter both numeric and string data, you can even create one filter that combines the two criteria using logical operators. The AND and OR are the two most frequently used logical operators. The AND operator means the same as in plain English; the same can be said for OR. For example, we can combine the two individual criteria using AND:

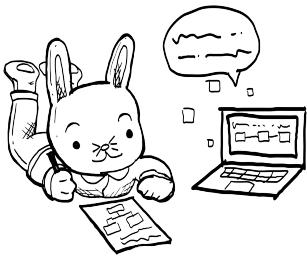
```
SELECT *
FROM product
WHERE price < 30 AND
manufacturer = 'Mad Inventors Inc.';
```

This query will yield the following result:

product_id	name	description	price	manufacturer
1	Atomic Nose Hair Trimmer	...	19.99	Mad Inventors Inc.
6	The Mind Probe	...	19.99	Mad Inventors Inc.
7	Lightsabers	...	25	Mad Inventors Inc.

Different from prior queries, this query will retrieve every column from the product table. The star (\*) following the SELECT keyword indicates all columns. The combination of the two filtering criteria will retrieve all columns of only the products that are manufactured by Mad Inventors Inc. and have a price below 30.

## 1.4.2 Aggregation



Aggregation, as an important task in SQL, involves performing calculations on a set of rows to produce a single result. By aggregating data, you can gain insights into trends and patterns in the data that might not be visible at the individual record level. The most frequently used aggregate functions include:

- `COUNT()`: counts the number of rows
- `SUM()`: calculates the sum of values in a numeric column
- `AVG()`: calculates the average value in a numeric column
- `MAX()`: finds the maximum value in a column
- `MIN()`: finds the minimum value in a column

When it comes to formulating a SQL query that involves aggregation, the aggregate function should be placed in the `SELECT` statement. For example, we can count the number of rows in the `product` table:

```
SELECT COUNT(*) FROM product;
```

This query will yield the following result:

COUNT(*)
10

You may notice that the column name is the verbatim of the aggregate function command. If you are dealing with a different RDBMS rather than SQLite, the column name might be `COUNT()` or something else. If you don't like the default column name, you can provide a more readable one using an alias via the `AS` clause. For example, you can calculate the average price of all products that are sold in our store, and use `avg_price` as the column name:

```
SELECT AVG(price) AS avg_price
```

```
FROM product  
WHERE manufacturer = 'Mad Inventors Inc.';
```

This query will yield the following result in which the column name is avg\_price, and the only value is the average of the prices of all products in this table:

avg_price
21.66

In both examples, you applied aggregate functions to all rows in a table. You can also apply aggregate functions to multiple groups of rows in a table. Sometimes, you need to group the data by one or more columns and analyze the grouped data. Grouping data can be achieved via the GROUP BY clause, which is commonly used in combination with aggregate functions. The GROUP BY clause is always followed by one or more attribute names separated by commas. For example, you can count the number of products per manufacturer:

```
SELECT COUNT(*) AS product_count, manufacturer  
FROM product  
GROUP BY manufacturer;
```

This query will yield the following result:

product_count	manufacturer
2	Absurd Accessories
1	Goofy Gadgets Corp.
3	Mad Inventors Inc.
3	Silly Supplies Co.
1	Wacky Wares Ltd.

For another example, you can calculate the average price of products per manufacturer:

```
SELECT AVG(price) AS avg_price, manufacturer  
FROM product  
GROUP BY manufacturer;
```

This query will yield the following result:

avg_price	manufacturer
22.5449...	Absurd Accessories
24.99	Goofy Gadgets Corp.
21.659...	Mad Inventors Inc.
17.843...	Silly Supplies Co.
39.99	Wacky Wares Ltd.

When you use aggregate functions with the `GROUP BY` clause, you need to include the attributes following the `GROUP BY` clause in the `SELECT` statement. Otherwise, the results may not make much sense. For example, the query below groups the data by the `manufacturer` column but didn't include it in the `SELECT` statement:

```
--comment: will yield something difficult to interpret:
```

```
SELECT COUNT(*) AS product_count  
FROM product  
GROUP BY manufacturer;
```

The result will become much harder to chew because you will only see a column of numbers, but have no idea what the numbers stand for:

product_count
2
1
3
3
1

For another example, the query below calculates the average product price

per manufacturer but didn't include the manufacturer column in the SELECT statement:

```
--comment: will yield something difficult to interpret:  
SELECT AVG(price) AS avg_price  
FROM product  
GROUP BY manufacturer;
```

Same as the last example, the result is difficult to interpret because you will only see a column of decimals, but have no idea what they stand for:

avg_price
22.5449...
24.99
21.659...
17.843...
39.99

More importantly, you should exclude any columns that are not in the GROUP BY clause from the SELECT statement. For example, the following query attempts to count the number of products per manufacturer. However, the name column appearing in the SELECT statement is NOT in the GROUP BY clause:

```
-- comment: will either lead to an error  
-- comment: or yield a misleading result  
SELECT COUNT(*) AS product_count, manufacturer, name  
FROM product  
GROUP BY manufacturer;
```

A query like this will either lead to errors or yield a misleading result depending on the RDBMS you use. For example, PostgreSQL will make the following complaint:



ERROR: column "product.name" must appear in the GROUP BY clause or...

For another example, SQLite will yield a misleading result without complaints:

product_count	manufacturer	name
2	Absurd Accessories	Cat-Poop Coffee
1	Goofy Gadgets Corp.	Selfie Toaster
3	Mad Inventors Inc.	Atomic Nose Hair Trimmer
3	Silly Supplies Co.	Unicorn Hair Polish
1	Wacky Wares Ltd.	Inflatable Briefcase

If you check all the 10 products in the product table, you will see that there is only one atomic nose hair trimmer instead of three. However, since the query doesn't know how to deal with the name column, it will simply show the name value in the first row it encounters per group.

For another example, the following query attempts to calculate the average price of products per manufacturer. However, the product\_id column appearing in the SELECT statement is not in the GROUP BY clause:

```
-- comment: will either lead to an error
-- comment: or yield a misleading result
SELECT product_id, AVG(price) AS avg_price, manufacturer
FROM product
GROUP BY manufacturer;
```

Depending on the RDBMS you use, you may either get an error or a misleading result. For example, PostgreSQL will make the following

complaint:



ERROR: column "product.product\_id" must appear in the GROUP BY clause

For another example, SQLite will yield a misleading result without complaints:

product_id	avg_price	manufacturer
3	22.5449...	Absurd Accessories
2	24.99	Goofy Gadgets Corp.
1	21.659...	Mad Inventors Inc.
5	17.843...	Silly Supplies Co.
4	39.99	Wacky Wares Ltd.

If you check the 3<sup>rd</sup> row in the product table, you will see its price is 29.99 instead of 22.5449... The above aggregated result is obviously wrong. However, since this query doesn't know how to deal with the product\_id column, it will simply show the first product\_id value it encounters per manufacturer group.



In summary, when you use aggregate functions with the `GROUP BY` clause, you need to be careful with what attributes to include in the `SELECT` statement. The `SELECT` statement should contain only those non-aggregate attributes showing up in the `GROUP BY` clause. Non-aggregate attributes refer to those attributes that are not involved in aggregation.

### 1.4.3 Table and data management

You just worked with SQL on a table that we gave you, but sometimes you will need to make your own tables, manage such tables and data stored in them.

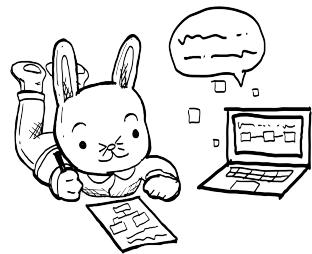


Table and data management are another type of important tasks in SQL. The SQL commands dedicated to such tasks are commonly known as *data definition language* (DDL). In contrast, the SQL clauses and statements you saw in prior sections are known as *data manipulation language*.

Understanding some DDL is particularly useful for database design (and for impressing all your friends at parties with your exciting conversations about data organization).

In this section, you will learn three common table management tasks, including creating, altering and deleting a table. You will also learn how to add data to a newly created table.

## Create tables and add data to tables



You will learn how to create a table and add data to it from the script that we prepared for this chapter. The script aims to create the product table for the database supporting The Sci-Fi Collective, and populate it with a set of sample data. The product table is created using the following command:

```
-- comment: works for MySQL, MariaDB, and PostgreSQL

-- comment: see the code repo for other RDBMS
CREATE TABLE product (
    product_id INT PRIMARY KEY,
    name TEXT NOT NULL,
    description TEXT NOT NULL,
    price DECIMAL(5, 2) NOT NULL,
    manufacturer TEXT NOT NULL
);
```

When executing the prepared script, you have successfully created the product table in your database. If you try to run the above SQL snippet again in the same database, you will get a complaint telling you that the product table has existed. As you have guessed, table names need to be unique in a database.

There are two questions that we need to answer about the command that

creates the product table:

1. What is the general syntax for creating a table?
2. What do the different keywords do in this query example?



**What is the general syntax for creating a table?** To create a table, you need the help of the CREATE TABLE command. The syntax for the CREATE TABLE command is as following:

```
CREATE TABLE table_name (
    column1_name datatype [optional_parameters],
    column2_name datatype [optional_parameters],
    ...
    columnN_name datatype [optional_parameters],
    PRIMARY KEY (columnX_name)
);
```

The definition of the primary key can also be a part of the definition of a column, like what you saw from the product table example. Different from other optional parameters, the primary key is required in every table.

**What do the different keywords do in this query example?** First, we specified the table name to be product, and defined five columns:

- product\_id: a numeric data type (INT)
- name: a string data type (TEXT)
- description: a string data type (TEXT)
- price: a numeric data type (DECIMAL)
- manufacturer: a string data type (TEXT)

`INT` indicates integer, and `TEXT` indicates string. The only data type worth explaining here is probably `DECIMAL`. `DECIMAL`, as a numeric data type, accepts two parameters: the first one defines the total number of digits, and the second one defines the number of digits to the right of the decimal point. For example, we use `DECIMAL(5, 2)` to define the `price` attribute to allow five digits in total, and two digits to the right of the decimal point.

In addition to the data types, you may see that we specified every attribute to be `NOT NULL`. In SQL, a `NULL` value represents an unknown value. Just like when you're trying to remember someone's name but it's on the tip of your tongue - it's either missing or unknown. Allowing `NULL` values for attributes may lead SQL to have unexpected behaviors. For example, when you add 10 and a `NULL` value, you will end up with a `NULL` value – the sum of an unknown value and 10 is still unknown. When you do calculations on `NULL` values, the results may all end up as `NULL`.

Lastly, the `PRIMARY KEY` definition was used to specify which attribute we want to use as the primary key for this table. The attribute name needs to be placed in the parentheses following the `PRIMARY KEY` definition.

When the product table is created in a database, it is ready for you to add data to it. To add data to a table, you need help from the `INSERT INTO` command. The `INSERT INTO` command allows you to insert one or more rows of data into a table. Here's its basic syntax:

```
INSERT INTO table_name (column1, column2, ...)
VALUES (value1, value2, ...);
```

From the same script, you can find an example of adding data to the product table. You can insert a single row into the table:

```
-- comment: works for MySQL, MariaDB, and PostgreSQL

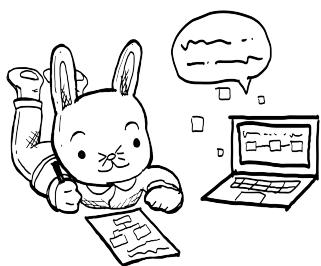
-- comment: see the code repo for other RDBMS
INSERT INTO product (product_id, name, description, price, manufa
VALUES (
  1,
  'Atomic Nose Hair Trimmer',
  'Trim your nose hairs... of an atomic clock!',
  19.99,
```

```
'Mad Inventors Inc.'  
);
```

Or you can insert multiple rows of data into the table:

```
-- comment: works for MySQL, MariaDB, and PostgreSQL  
-- comment: see the code repo for other RDBMS  
INSERT INTO product (product_id, name, description, price, manufa  
VALUES  
(  
  2,  
  'Selfie Toaster',  
  'Get your face on... with our selfie toaster',  
  24.99,  
  'Goofy Gadgets Corp.'  
,  
(  
  3,  
  'Cat-Poop Coffee',  
  'The only coffee... the finest cat poop ...',  
  29.99,  
  'Absurd Accessories'  
);
```

## Alter and drop tables



From time to time, you may need to alter or drop an existing table because, let's face it, sometimes you just need to rearrange the furniture in your data house.

There are many ways to alter a table, such as adding a column, modifying the data type of a column, or renaming the entire table. You can rely on the help of the `ALTER TABLE` command to perform all these tasks. For example, if you want to add another column representing serial numbers to the `product` table, you can use the following query:

```
ALTER TABLE product  
ADD serial_number INT;
```

When the above query gets executed, a new column named `serial_number` will be added to this table, and its data type is integer. When you realize that integer is not the best data type for serial numbers, you may update its data type to string via the following query:

```
ALTER TABLE product  
ALTER COLUMN serial_number TEXT;
```

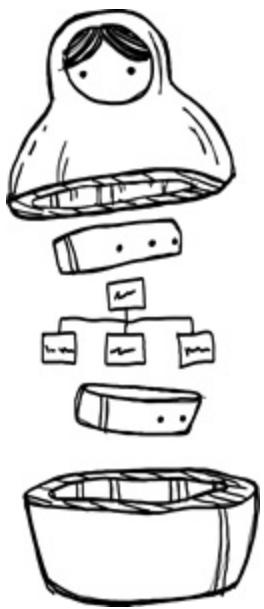
In contrast to many possible ways to alter a table, there's only one way to drop a table. To do so, you need the help of the `DROP TABLE` command, followed by the name of the table that you want to drop. For example, if we intend to drop the `product` table, we can use the following query:

```
DROP TABLE product;
```

You need to be careful when using the `DROP TABLE` command, because it will permanently delete the table and all its data.

This section doesn't aim to be an exhaustive list of every command to alter or drop a table. If you want to know more, please check out the SQL manual or your target RDBMS. With that said, you have taken an important step forward towards mastering databases and database design. The few things that you've achieved in this chapter will propel your learning throughout the rest of the book. Small choices that cascade to a design masterpiece perhaps worthy of being displayed in a tech gala one day, should they ever become a reality.

## 1.5 Summary



- A relational database is a collection of tables that store data.
- A table is used to represent an entity or a relationship between entities in a database.
- An entity is an object or concept that can be described by many attributes.
- An RMDBS is software that interacts with the underlying hardware and operating system to physically store and manage data in relational databases.
- Some of the important tasks in querying databases include filtering, data aggregation, table management and data management. SQL clauses and statements deal with such tasks are known as data manipulation language.
- Filtering data requires help from at least three SQL clauses, `SELECT`, `FROM`, and `WHERE`.
- Data aggregation functions are often used in combination with the `GROUP BY` clause.
- SQL commands are used to manage tables are known as data definition language. Table management typically involves three commands, including `CREATE TABLE`, `ALTER TABLE`, and `DELETE TABLE`.
- You can insert a single row or multiple rows of data into a table via the `INSERT INTO ... VALUE ...` statement.

# 2 Related tables and more SQL

## In this chapter

- You learn what related tables are and how to query them.
- You revisit table and data management.
- You explore how to learn more SQL on your own.

### What you need to know

To understand the SQL covered in this chapter, you need to know what databases and tables are, as well as the basics of SQL queries. If not, you may start by reading chapter 1.

As you read this chapter, you will find more code snippets. If you want to execute those code snippets or see what changes need to be made to the code for different RDBMS, you will need to check the GitHub repository that accompanies this book (<https://github.com/Neo-Hao/grokking-relational-database-design>). You can find the scripts for this chapter in the chapter\_02 folder and follow the instructions in the README.md file to run the scripts.

For example, the easiest approach is to use SQLite Online:

1. Clone or download our GitHub repository (<https://github.com/Neo-Hao/grokking-relational-database-design>).
2. Locate the scripts in the chapter\_02 folder
3. Visit SQLite Online (<https://sqliteonline.com>).
4. Click **Import** to load the script named sqlite\_online.sql.
5. Click **Okay**.

## 2.1 Overview

In Chapters 1, you learned the basics of SQL and how to query or create a single table. In this chapter, you will continue learning a bit more of SQL by querying and creating related tables. In the end, you will peek into how to pick up more SQL keywords by yourself in the future.

## 2.2 Related tables

In this section, you will learn how to work with two or more tables that are related to each other. Such tables are known as *related tables*.



What are related tables? Related tables are tables in a database that are connected to each other by one or more common columns. For example, our online store The Sci-Fi Collective allows customers to create user accounts, shop online, and leave reviews for the products they bought. The Sci-Fi Collective is supported by a database composed of multiple tables. Among these tables, there are two that are related to each other, the product and review table. The product table represents the products we sell, and the review table represents the reviews customers leave for the products they bought. The two tables share a common column (i.e., `product_id`). This relationship between the two tables is summarized below:

The diagram illustrates two database tables: **product** and **review**.

**product Table:**

product_id	name	price	manufacturer
1	Atomic Nose ...	19.99	Mad Inventors Inc.
2	Selfie Toaster	24.99	Goofy Gadgets Corp.
3	Cat-Poop Coffee	29.99	Absurd Accessories
...	...	...	...

**review Table:**

review_id	product_id	review_text	date
1	3	Great product ...	2023-01-01 12:30:00
2	5	...best thing ...	2023-01-02 13:45:00
3	2	...not recommend	2023-01-03 14:15:00
...	...	...	...

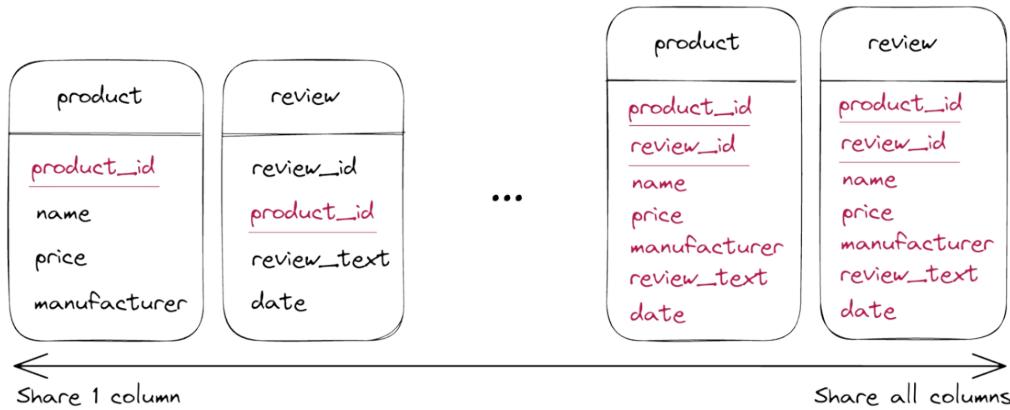
The `product_id` column is shared by both the `product` and `review` tables. In the `product` table, the `product_id` column is the primary key. In the `review` table, the `product_id` column is known as the *foreign key*, which is used to refer to the primary key of another table.

In other words, the value in the `product_id` column helps connect a row in the `product` table and a row in the `review` table. For example, in the `product` table, the row with a `product_id` 3 records a product named Cat-Poop Coffee; In the `review` table, the row with a `review_id` 1 records a positive review for Cat-Poop Coffee. How would you know that review is for Cat-Poop Coffee? They share the same `product_id` value.

### 2.2.1 Number(s) of shared columns

As curious as you are, you may wonder if the `product` and `review` tables can share a few more common columns. That's a great question on database design. A more generalized question would be whether two related tables should share more columns beyond the primary/foreign key? The answer is NO.

To simplify our discussion, let's first look at the scenario where two related tables use single columns as the primary keys, just like the `product` and `review` tables. Theoretically speaking, the number of common columns of the two related tables can range from only the primary/foreign key to all. If we use the `product` and `review` tables as an example, this spectrum can be summarized below:



If two related tables share every column, it is obvious that one of the two tables is redundant, thus unnecessary. If you choose to delete one of them, you will find yourself dealing with a similar issue you saw in Chapter 1 – using one table to represent two entities, which will lead to insertion or delete anomalies. Okay, making two tables share every column is apparently a bad idea of database design.

How about making them share only a few columns, such as one or two beyond the primary/foreign key? Still a bad idea. First, you would still have redundancy, even if it's less serious than having a redundant table. Second, you will set a trap for yourself when you need to update data in such tables. If you forget one table and only update data in the other table, you will end up with inconsistent data. For example, suppose we decide to make the `product` and `review` tables share one more column, `manufacturer`, beyond the primary/foreign key. When we need to update the manufacturer of a product, we need to make sure to update both the `product` and `review` tables.

Otherwise, we will end up with new manufacturer data in one table but old data in the other. Such an issue is known as an *update anomaly*:



Two tables are shown:

<u>product_id</u>	<u>name</u>	<u>price</u>	<u>manufacturer</u>
1	Atomic Nose ...	19.99	Mad Inventors Inc.
2	Selfie Toaster	24.99	Goofy Gadgets Corp.
3	Cat-Poop Coffee	29.99	Absurd Accessories
...	...	...	...

Handwritten annotations:

- "updated to:" next to "Absurd Accessories"
- "Mad Inventors Inc." in a box next to "Absurd Accessories"
- "conflict with" next to "Mad Inventors Inc."

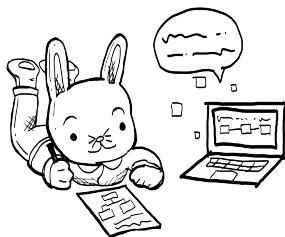
<u>review_id</u>	<u>product_id</u>	<u>manufacturer</u>	<u>review_text</u>	<u>date</u>
1	3	Absurd Accessories	Great product ...	2023-...
2	5	Silly Supplies Co.	...best thing ...	2023-...
3	2	Goofy Gadgets Corp.	...not recommend	2023-...
...	...	...	...	...

Handwritten annotations:

- An arrow points from the manufacturer column in the first table to the manufacturer column in the second table.

As you can see, given two related tables, they shouldn't share more columns beyond the primary/foreign key(s).

## 2.2.2 Join data from related tables



From time to time, you will need to join data from related tables. For example, if you want to know about how each product on The Sci-Fi Collective is reviewed, you will need to join at least the product names from the product table and the corresponding product reviews from the review table, and make sure that data from both tables are properly joined. In SQL,

we can write the query that does this job as follows:

```
SELECT name, review_text  
FROM product  
JOIN review  
ON product.product_id = review.product_id;
```

This query will yield the following result:

name	review_text
Cat-Poop Coffee	Great product ...
Unicorn Horn Polish	This is the best ...
Selfie Toaster	Not worth the money ...
...	...

There are quite a few things that we need to explain about this query example:

1. What is the general syntax for retrieving data from related tables?
2. What exactly does the JOIN...ON... clause do?
3. What is the dot and how do we use the dot notation as is seen in product.product\_id or review.product\_id?

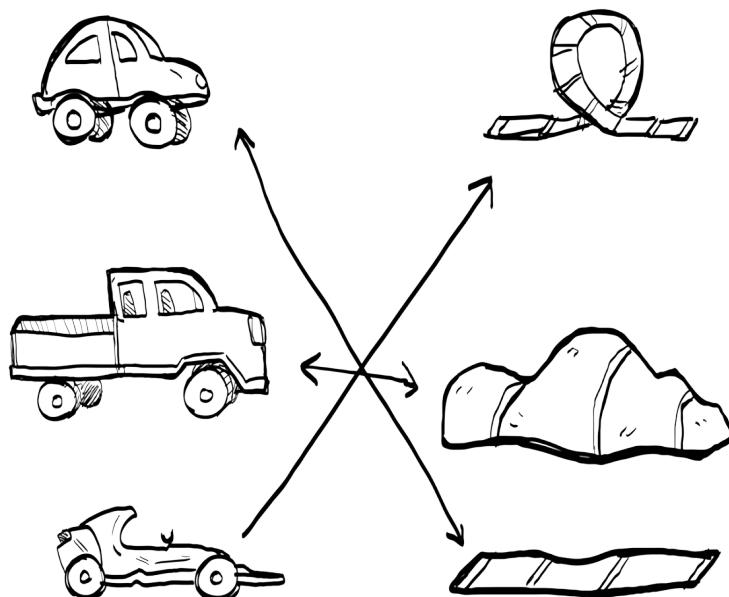


What is the general syntax for retrieving data from related tables? The general syntax is as follows:

```
SELECT column1, column2, ...  
FROM table1  
JOIN table2
```

```
ON table1.column = table2.column;
```

What exactly does the JOIN...ON clause do? Imagine you have two toy boxes, one with cars and the other with racetracks. You want to play with both together, so you pour all the toys from both boxes onto the floor. But now you need to find which cars would work on what types of racetracks. That's what the JOIN clause does.



In detail, the JOIN clause takes two tables (toy boxes), and pours all the rows (toys) onto the floor, then it looks for matching values in the shared column (like the match between a car and a racetrack) between the two tables. If it finds a match, it puts the corresponding rows (cars and trucks that go together) together in a new table (your play area).



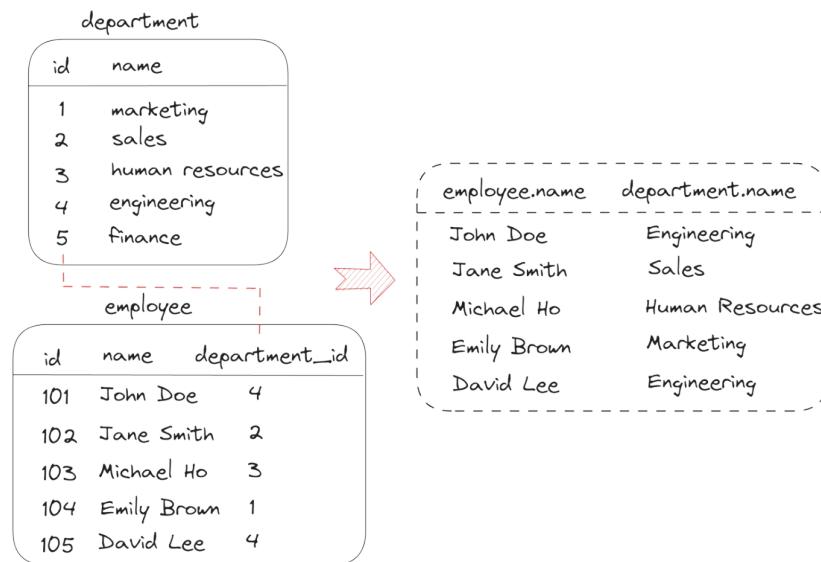
What is the dot and how do we use the dot notation as is seen in product.product\_id or review.product\_id? The dot notation is an SQL

syntax used to separate parts of a name. For example, `product.product_id` is used to refer to the `product_id` column in the `product` table. For another example, `product.name` can be used to refer to the `name` column in the `product` table. The dot notation is especially handy when you query related tables because it helps you to be specific on the columns with the same name, such as `product.product_id` and `review.product_id`. This makes it clear which column and which table you are referring to, and prevents any confusion from only using the column name without specifying the table, like `product_id`.

If two tables that you want to join happen to have more than one column that share the same names, you may want to be specific in the `SELECT` statement to avoid confusions. Otherwise, you might end up with a result set that looks like a mixed-up game of Scrabble played by a mischievous toddler. For example, imagine you have two tables, `employee` and `department`, and you want to join them to get the names of the departments that each employee belongs to. Both tables have a column named `name`, so you need to use the dot notation to specify which `name` column you want to select:

```
SELECT employee.name, department.name  
FROM employee  
JOIN department  
ON employee.department_id = department.id;
```

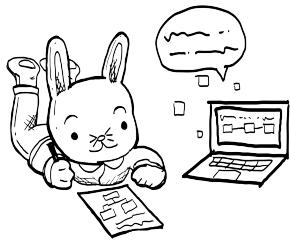
This query will work its way as is demonstrated below:



Depending on the RDBMS you use, you may either see the column names the same as above, or two exactly the same column names (i.e., name). To avoid this confusion, you can rename the columns with an alias via the AS clause:

```
SELECT employees.name AS employee_name,
       departments.name AS department_name
  FROM employees
 JOIN departments
    ON employees.department_id = departments.id;
```

## Types of JOINS



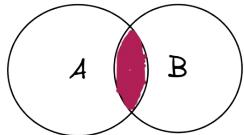
Now you know enough about the basics of the `JOIN` clause, let's dive a bit deeper into joins by discussing its variations. It is a good time for you to refresh your memory on the general syntax of joining tables:

```
SELECT column1, column2, ...
  FROM table1
 JOIN table2
    ON table1.column = table2.column;
```

The `JOIN` keyword can be replaced by the following keywords, which may lead to different query results:

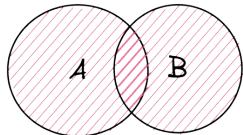
- `INNER JOIN`: returns only the rows that have matching values in both tables; `INNER JOIN` is the same as `JOIN`.
- `LEFT JOIN`: returns all the rows from the left table (table1) and the matching rows from the right table (table2).
- `RIGHT JOIN`: returns all the rows from the right table (table2) and the matching rows from the left table (table1).
- `FULL OUTER JOIN`: returns all the rows from both tables, including the non-matching rows.

The relationship among the left table, right table, and the returned results are summarized as follows:



```
SELECT *
FROM A
INNER JOIN B
ON A.A_id = B.A_id
```

```
SELECT *
FROM A
FULL OUTER JOIN B
ON A.A_id = B.A_id
```



```
SELECT *
FROM A
LEFT JOIN B
ON A.A_id = B.A_id
```

```
SELECT *
FROM A
RIGHT JOIN B
ON A.A_id = B.A_id
```

It is worth noting that LEFT JOIN, RIGHT JOIN, and FULL OUTER JOIN may lead to query results with NULL values. One side effect of getting NULL values in the result is that you need to handle them carefully. NULL values can cause errors if you try to perform calculations or comparison (e.g.,  $10 + \text{NULL}$  will lead to NULL).

For example, our product table contains some products are newly added to our store, and have not been reviewed by any users. When we perform a LEFT JOIN between the product and review tables, we will end up with some rows that have NULL values in the columns from the review table. The LEFT JOIN query would be as follows:

```
SELECT name, review_text
FROM product
LEFT JOIN review
ON product.product_id = review.product_id;
```

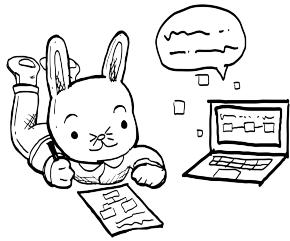
This query will yield the following results:

name	review_text
Atomic Nose Hair Trimmer	Decent product for ...
Atomic Nose Hair Trimmer	Would not recommend ...
...	...
The Mind Probe	NULL
The Sonic Screwdriver	NULL
...	...

As you can see, every match between the product and review table is included. A product can be reviewed for more than once, and show up in the result table as multiple rows, like Atomic Nose Hair Trimmer. Additionally, if a product was not reviewed, it will still be included in the result table, like The Mind Probe whose review\_text value is NULL.

Although non-INNER joins may lead to results with NULL values, they can still be useful when you need to retrieve data from related tables, even if there may not be a match for some of the rows in one of the tables.

## WHERE vs JOIN



As curious as you are, you may be tempted to try joining two tables using the WHERE clause. You learned how to use the WHERE clause to filter a subset of data from a table that meets some criteria. If you know that it is possible to list multiple tables in the FROM statement, you may put together the following query to join the product and review tables we saw earlier:

```
SELECT name, review_text
FROM product, review
WHERE product.product_id = review.product_id;
```

Would this work and yield the same result as the example we saw earlier? Yes. The above query will work fine to join the two tables and yield the same result as the query using the `JOIN` clause:

```
SELECT employee.name, department.name
FROM employee
JOIN department
ON employee.department_id = department.id;
```

However, whenever you need to query related tables, `JOIN` is the preferred method rather than `WHERE` in general for many reasons, such as:

- Readability: Explicit `JOIN` make the query's intention clearer and easier to understand.
- Maintainability: Explicit `JOIN` are less prone to errors and more straightforward to modify or debug.
- Optimization: When you use a `WHERE` clause to query two related tables, the query essentially asks for a *cross join* between the two tables, which is more difficult for most RDBMS to optimize than an explicit `JOIN`.



#### **Cross Join: A wild dance where everyone twirls with everyone else.**

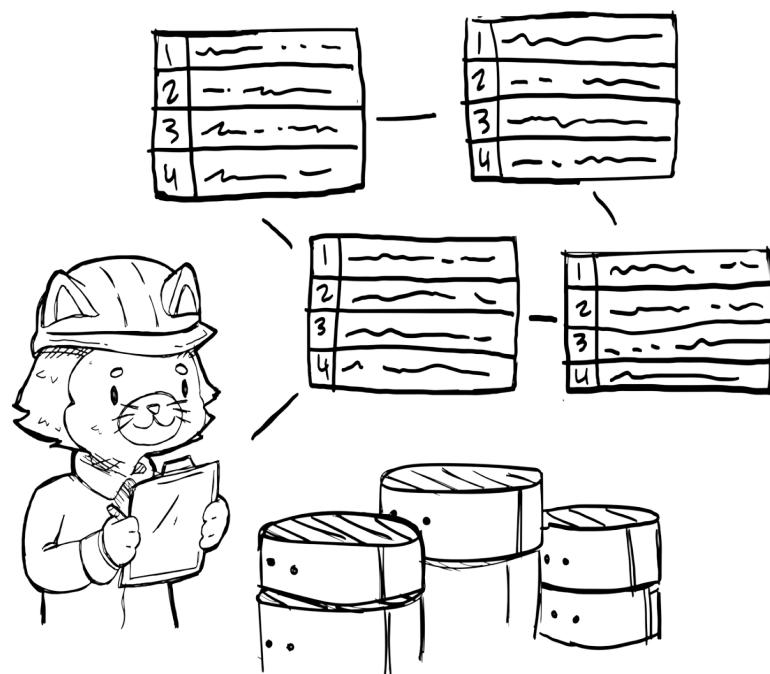
A cross join in SQL is an operation that combines every row from one table with every row from another table. It generates all possible combinations of rows between the two tables, resulting in a larger result.

If the `FROM` clause in your query is followed by two or more tables, your query will perform a cross join between those tables. For example, `FROM product, review` means that every row in the `product` table will be paired with every row in the `review` table.

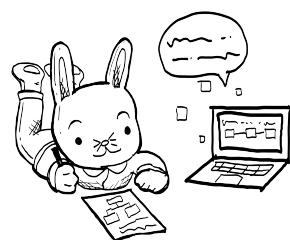
Cross joins may require scanning the involved tables separately, and demand a large amount of memory from RDBMS.

## 2.3 Revisit table and data management

You will explore how to manage related tables and their data in this section. In Chapter 1, you learned how to manage a table (and its data) that is not associated with any other tables. In the previous section, you worked with a pair of related tables that we gave you. You will now apply the concepts and techniques you learned in Chapter 1 and the previous section to expand your knowledge in table and data management of related tables.



### 2.3.1 Manage related tables



You will learn how to create related tables from the script that we prepared

for this chapter. The script aims to create two related tables, product and review, for the database supporting The Sci-Fi Collective, and populate them with a set of sample data.

The product table is created the same as in Chapter 1:

```
-- comment: works for MySQL, MariaDB, and PostgreSQL

-- comment: see the code repo for other RDBMS
CREATE TABLE product (
    product_id INT PRIMARY KEY,
    name TEXT NOT NULL,
    description TEXT NOT NULL,
    price DECIMAL(5, 2) NOT NULL,
    manufacturer TEXT NOT NULL
);
```

The review table is created as follows:

```
-- comment: works for MySQL and MariaDB

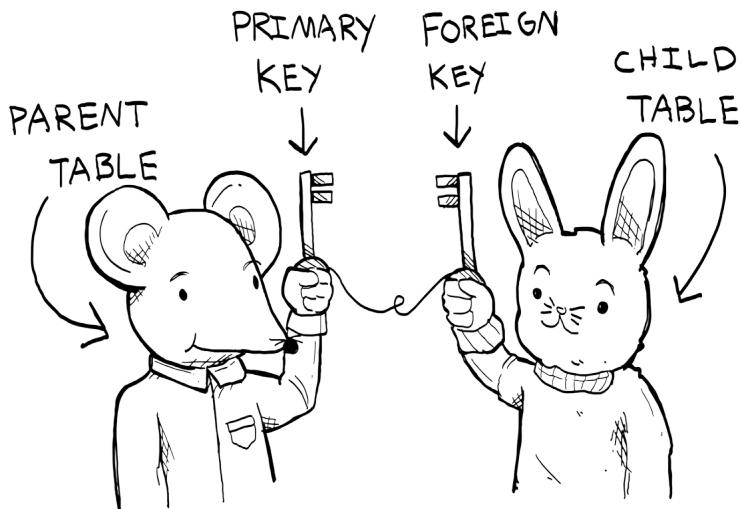
-- comment: see the code repo for other RDBMS
CREATE TABLE review (
    review_id INT PRIMARY KEY,
    product_id INT NOT NULL,
    review_text TEXT NOT NULL,
    datetime DATETIME NOT NULL
        DEFAULT CURRENT_TIMESTAMP,
    CONSTRAINT fk_product_review
        FOREIGN KEY (product_id)
        REFERENCES product (product_id)
);
```

There are two questions that we need to answer about this query example:

1. What is the general syntax for creating two tables that have a relationship?
2. What does the CONSTRAINT...FOREIGN KEY...REFERENCES... clause do?



What is the general syntax for creating two tables that have a relationship? As you know, the shared column `product_id` is the primary key in the `product` table and the foreign key in the `review` table. Given a pair of two related tables, we call the table like `product` the *parent table* because it uses the shared column as the primary key. We call the table like `review` the *child table* because it holds the foreign key.



As you can see from the command that creates the `product` table, the syntax of creating a parent table is the same as creating a table that is not related to other tables.

To create a child table, you need to specify the foreign key that references the primary key in the parent table. Of course, you still need to define the columns, data types, and its primary key. The general syntax of creating a child table is as follows:

```
CREATE TABLE child_table_name (
```

```

column1 datatype1 [NOT NULL],
column2 datatype2 [NOT NULL],
...
foreign_key_column datatype,
CONSTRAINT fk_parent_child
    FOREIGN KEY (foreign_key_column) REFERENCES
        parent_table_name(parent_table_primary_key)
);

```

Or you can create the two tables independently, and add the foreign key constraint to the child table afterward:

```

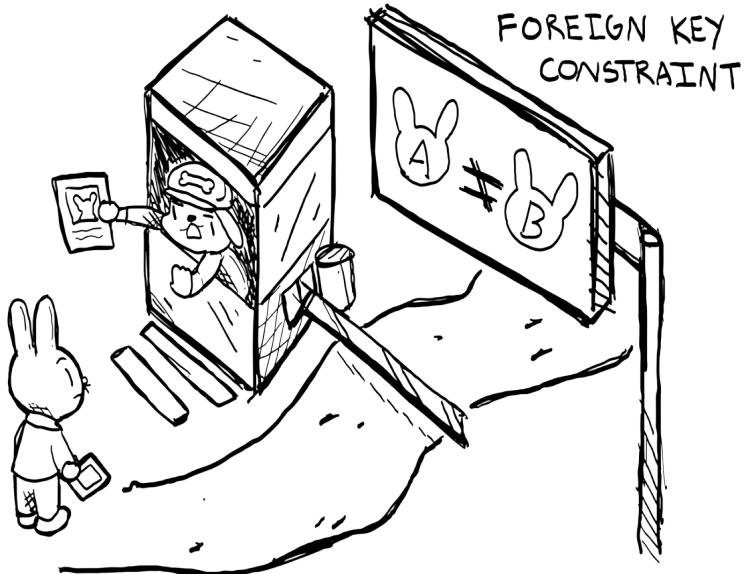
-- comment: assuming the parent and child tables
-- comment: have been created
ALTER TABLE child_table_name
    ADD CONSTRAINT fk_parent_child
        FOREIGN KEY (foreign_key_column) REFERENCES
            parent_table_name(parent_table_primary_key);

```



**What does the CONSTRAINT...FOREIGN KEY...REFERENCES... clause do?** In short, the clause creates a foreign key constraint, which serves as a link between two related tables. The constraint is two-folded:

- The constraint ensures that the foreign key column in the child table references only valid primary key values in the parent table.
- The constraint ensures that the update or deletion of rows in the parent table doesn't violate the consistency between two related tables.



We refer to these two aspects as *referential integrity*. The clause CONSTRAINT...FOREIGN KEY...REFERENCES... enforces referential integrity between two related tables.

If you take a closer look at the CONSTRAINT...FOREIGN KEY...REFERENCES... clause, you can divide it to two parts:

Part 1: name the constraint

CONSTRAINT fk\_parent\_child

  FOREIGN KEY (foreign\_key\_column)

  REFERENCES parent\_table\_name(parent\_table\_primary\_key)

Part 2: create the foreign key constraint

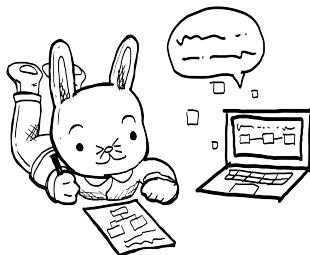
The FOREIGN KEY...REFERENCES... statement creates the foreign key constraint that enforces referential integrity. The CONSTRAINT... clause allows you to name this constraint. When you create a foreign key constraint, you don't necessarily need to name it, but naming it will make it easily accessible whenever you need to modify such a constraint in the future. For example, we named the foreign key constraint fk\_product\_review in the review table. If we ever need to drop this constraint, we can access the constraint via this name:

```
ALTER TABLE review  
DROP CONSTRAINT fk_product_review;
```

If you don't name a constraint yourself, the RDBMS will name it

automatically using its default naming convention. Although the automatically picked name can be retrieved, this name and the default naming convention vary from one RDBMS to another. To avoid this hassle, you are recommended to always name constraints (as you should always name pets too). After all, dropping or disabling a constraint can be a common task whenever you need to modify your database design.

### 2.3.2 Manage data in related tables



The syntax for adding, updating and deleting data in related tables remains the same as you saw in Chapter 1. For example, the script that we prepared for this chapter added a set of data to both the `product` and `review` tables:

```
-- comment: works for MySQL, MariaDB, and PostgreSQL
-- comment: see the code repo for other RDBMS
-- comment: add data to the product table
INSERT INTO product (product_id, name,
                      description, price, manufacturer)
VALUES (
    1,
    'Atomic Nose Hair Trimmer',
    'Trim your nose hairs ... an atomic clock!',
    19.99,
    'Mad Inventors Inc.'
),
...;

-- comment: add data to the review table
INSERT INTO review (review_id, product_id,
                    review_text, datetime)
VALUES (
    1,
    3,
    'Great product, would definitely recommend!',
    '2022-01-01 12:30:00'
),
```

...;

However, what makes data management for related tables different is the foreign key constraint. In the last subsection, you learned that the foreign key constraint enforces referential integrity on a pair of two related tables:

- The foreign key constraint ensures that the foreign key column in the child table references only valid primary key values in the parent table.
- The foreign key constraint ensures that the deletion of rows in the parent table doesn't violate the consistency between two related tables.

When you try to add data to the child table, the newly added data needs to be consistent with the existing data in the parent table. Otherwise, the RDBMS will complain. For example, you are trying to add a new row of review data to the `review` table, but the `product_id` value (i.e., 3000) in this row can't be found in the `product` table. You put together a command as follows:

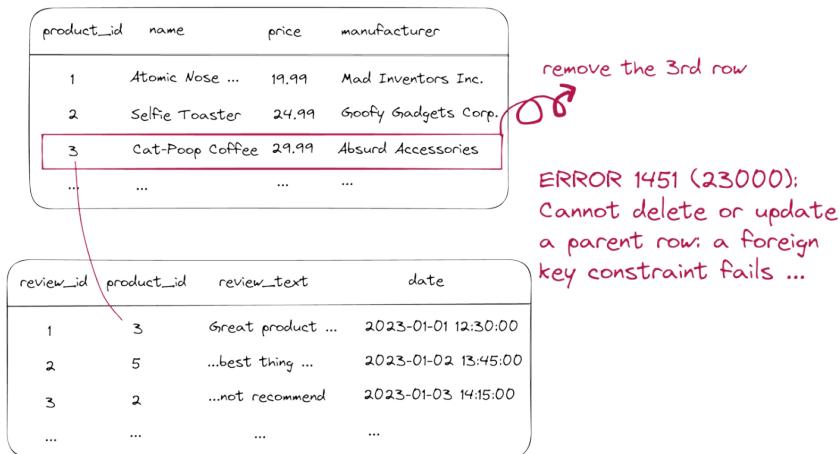
```
-- comment: works for MySQL, MariaDB, and PostgreSQL
-- comment: see the code repo for other RDBMS
INSERT INTO review (review_id, product_id,
                     review_text, datetime)
VALUES (
    1,
    3000,
    'Great product!',
    '2023-05-01 12:30:00'
);
```

When you execute this command, your RDBMS will give you an error message similar to the following:



ERROR 1452 (23000): Cannot add or update a child row: a foreign k

Likewise, when you alter or delete data from the parent table, the altering or deletion shouldn't lead to orphan data records in the child table. Otherwise, the RDBMS will complain. For example, you are trying to delete a row of product data from the product table, but this product data is referenced in the review table. If this row is to be deleted, it will create some orphan review data in the review table. Fortunately, the foreign key constraint will stop this from happening, and you will get an error message similar to the following:



The diagram illustrates a database schema with two tables: 'product' and 'review'.

**product Table:**

product_id	name	price	manufacturer
1	Atomic Nose ...	19.99	Mad Inventors Inc.
2	Selfie Toaster	24.99	Goofy Gadgets Corp.
3	Cat-Poop Coffee	29.99	Absurd Accessories
...	...	...	...

**review Table:**

review_id	product_id	review_text	date
1	3	Great product ...	2023-01-01 12:30:00
2	5	...best thing ...	2023-01-02 13:45:00
3	2	...not recommend	2023-01-03 14:15:00
...	...	...	...

Annotations in red text:

- "remove the 3rd row" with a red arrow pointing to the third row in the 'product' table.
- "ERROR 1451 (23000): Cannot delete or update a parent row: a foreign key constraint fails ..." with a red arrow pointing to the error message.

The ultimate form of deleting data records from the parent table would be to drop the table entirely. If this is ever allowed, all the data in the child table will become orphan data. Fortunately, the foreign key constraint will stop this from happening, and you will get an error message similar to the following:



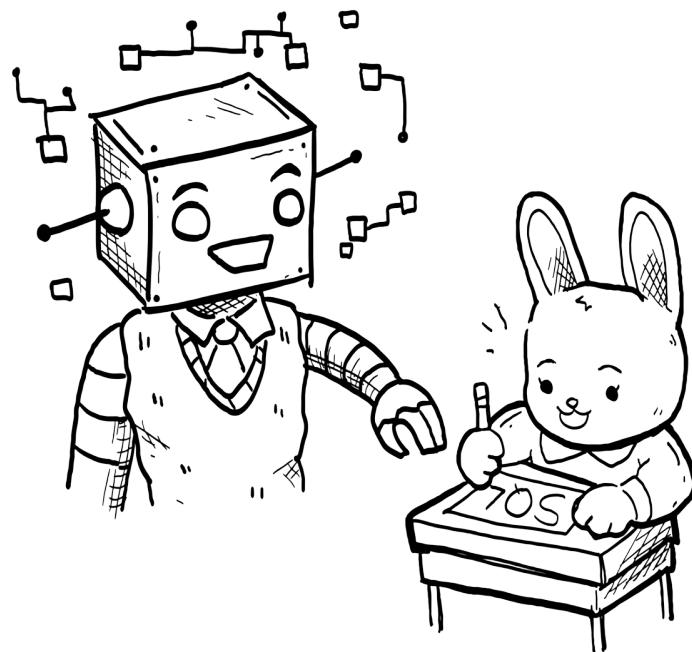
```
ERROR: cannot drop table ... because other objects depend on it
DETAIL: constraint... on table...depends on table...
```

If you are still in the process of perfecting your database design, and need to

modify the table structure, you can refer to chapter 1 in terms of how to modify an individual table, or the last subsection in terms of how to alter the foreign key constraint. In an extreme scenario, you may need to drop a pair of two related tables with some sample data. In such a case, you should always save off the data first, and drop the child table first before dropping the parent table. This order is important, because dropping the child table first will also eliminate the foreign key constraint. Otherwise, you will see the above error message because the foreign key constraint will work hard to enforce the referential integrity.

## 2.4 Learn more SQL on your own

You will explore how to approach learning more SQL on your own in this section. The journey to SQL mastery is a long one, but at least you'll have plenty of time to contemplate all the ways you misspelled `SELECT` or messed up with `NULL` along the way. However, if you have read Chapter 1 and this chapter up to this point, you have already equipped yourself with the essentials that you need for database design. Such knowledge will serve as a solid foundation for you to learn more SQL on your own.



In this brave new age, mastering SQL on your own is significantly easier than just one or two years ago due to the rise of generative AI tools such as

ChatGPT and Bard. Please make no mistake: we are not advocating relying on ChatGPT solely for learning SQL. If the generative AI tools, like ChaGPT, are well used, you will be able to learn new knowledge much faster and efficiently, and we want to demonstrate how you can use such tools to facilitate your learning of SQL.

Of course, some aspects of learning SQL remain the same as in the past — You will need the help from a quality book to learn SQL systematically, especially the conceptual knowledge such as indexing and the query order of execution. This step can't be easily replaced by asking questions of ChatGPT. There are many quality SQL books on the market, and here two examples:

- [Learn SQL in a Month of Lunches by Jeff Iannucci](#)
- [Learning SQL: Generate, Manipulate, and Retrieve Data by Alan Beaulieu](#)

Some other aspects of learning SQL can be very different and more efficient nowadays, especially when you have learning questions or when you want to know how a SQL keyword is used. The biggest advantage of generative AI tools, such as ChatGPT, is that they can provide relatively reliable just-in-time help to common learning questions. Without such tools, you will need to go through an iterative process to get your questions answered, which involves articulating a search query, searching online, and filtering the returned search results. This process, especially filtering all the search results, can be daunting and increase your cognitive load dramatically.

The wide availability of generative AI tools can help you avoid this iterative process. For example, if you want to know how to use the `sum` aggregate function, you can ask the following questions to ChatGPT:

- Question: How do I use the `sum` aggregate function in SQL?  
The answer is typically composed of the general syntax and a concrete example, which is good enough and even better than a lot of tutorials written by human beings:
- A sample answer from ChatGPT:  
“In SQL, the `sum` function is used to calculate the total of a set of values in a column of a table. Here's the basic syntax:

```
SELECT SUM(column_name) FROM table_name;
```

For example, let's say we have a table called orders with the following columns: id, customer\_name, product\_name, and price. If we wanted to calculate the total revenue from all orders, we could use the following query:

```
SELECT SUM(price) FROM orders;
```

This would return a single value representing the total sum of all prices in the price column of the orders table.”

If you still find the answer difficult to chew, you can ask ChatGPT to be more patient and explain in detail:

- Question: How do I use the `sum` aggregate function in SQL? Explain with more examples and as if I am five.  
You will find ChatGPT can answer this better than most tutors:
- A sample answer from ChatGPT:  
“Sure! Let's say you have a table with some numbers in it, like this:

ID	Value
1	2
2	4
3	6
4	8

Each row has a number in the Value column. You can think of these numbers like pieces of candy. The ID column is like a label that helps us tell the difference between the pieces of candy.

Now, let's say you want to know how many pieces of candy you have in total. You could count each piece of candy one by one, but that would take a long time. Instead, you can use the `SUM` function in SQL to add up all the pieces of candy for you! Here's what the SQL query would look like:

```
SELECT SUM(Value) FROM my_table;
```

When you run this query, the computer will add up all the numbers in the

Value column for you. In this case, the total is 20. You can think of this as having 20 pieces of candy in total! Here's another example. Let's say you have a table with some prices in it, like this:

ID	Price
1	1.99
2	2.99
3	3.99
4	4.99

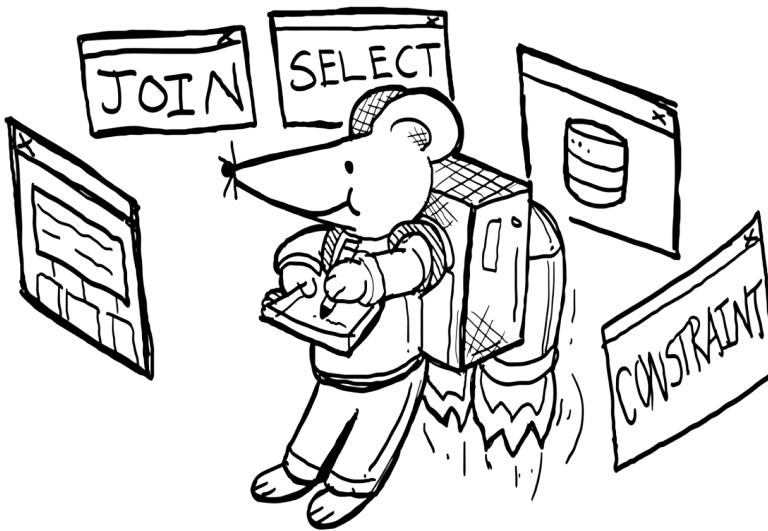
Each row has a price in the Price column. You can think of these prices like the cost of buying things, like toys or candy. If you want to know how much money you would need to buy all these things, you can use the SUM function in SQL. Here's what the query would look like:

```
SELECT SUM(Price) FROM my_table;
```

When you run this query, the computer will add up all the prices in the Price column for you. In this case, the total is 13.96. You can think of this as how much money you would need to buy all these things!"

One of the most significant benefits of Generative AI tools in SQL learning is that they can provide relatively reliable just-in-time help. For common questions, you don't necessarily need to ask the perfect question to receive immediate answers that are informative and well-structured. What's more, in contrast to fine tuning search engine queries, it's easier to tune your questions to get better answers. Using generative AI tools well can make your learning of SQL much more efficient.

## 2.5 Summary



- Related tables are tables in a database that are connected to each other by a common column.
- In general, it is a bad idea to have related tables share more than one column.
- To query two related tables, the general syntax is as follows:

```
SELECT column1, column2, ...
FROM table1
JOIN table2
ON table1.column = table2.column;
```

- The most common join types include `INNER JOIN`, `LEFT JOIN`, `RIGHT JOIN`, and `OUTER JOIN`. The `INNER JOIN` is the same as `JOIN`, returning only the rows that have matching values in two tables. The other three types may return non-matching values from one or two tables.
- In general, the `JOIN` clause is preferred to the `WHERE` clause when querying related tables.
- Given a pair of two related tables, the table that uses the shared column as the primary key is typically known as the parent table. The table that holds the foreign key is typically known as the child table.
- The parent table can be created in the same way as creating a table that is not related to other tables.
- The general syntax of creating a child table is as follows:

```
CREATE TABLE child_table_name (
    column1 datatype1 [NOT NULL],
    column2 datatype2 [NOT NULL],
```

```
    ...  
    foreign_key_column datatype,  
CONSTRAINT fk_parent_child  
    FOREIGN KEY (foreign_key_column)  
REFERENCES  
    parent_table_name(  
        parent_table_primary_key  
    )  
);
```

- The foreign key constraint enforces referential integrity in related tables. The referential integrity applies to both table and data management in related tables.
- In general, it is a good idea to name the foreign key constraint when creating related tables.
- Taking advantage of generative AI tools, like ChatGPT, can make your learning of SQL more efficient.

# 3 Overview of database design

## In this chapter

- You learn about the goals of database design.
- You get an overview of the database design process.
- You jump start with the requirement analysis.

### 3.1 Overview

In Chapters 1 and 2, you learned the basics of relational databases and SQL. Starting from this chapter, you will embark on your journey of learning database design by designing a database from scratch for the online store The Sci-Fi Collective. In this chapter, you will get an overview of the goals and process of database design. After that, you will jump start the requirement analysis for The Sci-Fi Collective.

### 3.2 Goals of database design

The overall goal of database design is to deliver a well-structured and efficient database that meets the requirements of the users and organizations. Beyond meeting the requirements from users and organizations, a successful database design typically meets five common goals, including data consistency and integrity, maintainability and ease of use, performance and optimization, data security, scalability and flexibility. In this section, you will take a peek at these common goals to understand better what you should aim to achieve in database design. Let's get started.

#### 3.2.1 Data consistency and integrity

Data consistency and integrity are about defining appropriate data types, constraints, and relationships among entities to ensure that data remains consistent across tables in the database, data redundancy is minimized, and

anomalies are avoided.

In the last two chapters, you peeked into some poor designs that lead to data redundancy and learned about three types of different anomalies.



#### **Insert, update, and delete anomaly**

An insert anomaly happens when it is not possible to add a new record to a database without adding unrelated data.

An update anomaly happens when modifying data results in inconsistencies within the data.

A delete anomaly happens when removing data leads to unintentional loss of information.

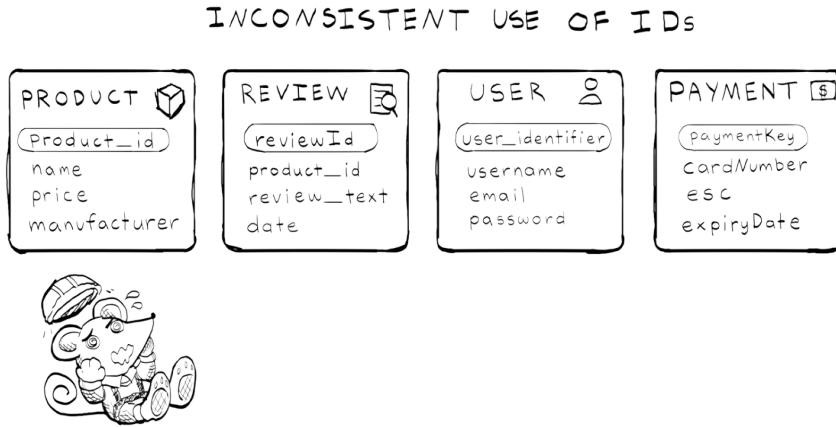
In short, data consistency is about taking measures to ensure that those issues don't happen.

#### **3.2.2 Maintainability and ease of use**

A well-designed database should be intuitive to use and easy to maintain for people who use it, including database administrators, data analysts, and developers of web or mobile applications.

You can take a lot of measures to increase the maintainability or the ease of use of a database. For example, following a consistent naming convention is a small thing to do when you design a database, but it can save a lot of time when it comes to using or maintaining a database. If developers who use a

database have to spend time figuring out if and where id, Id, key or identifier was used, the database is hardly intuitive to use, let alone easy to maintain. Just think about if you have to maintain a database with the following tables:



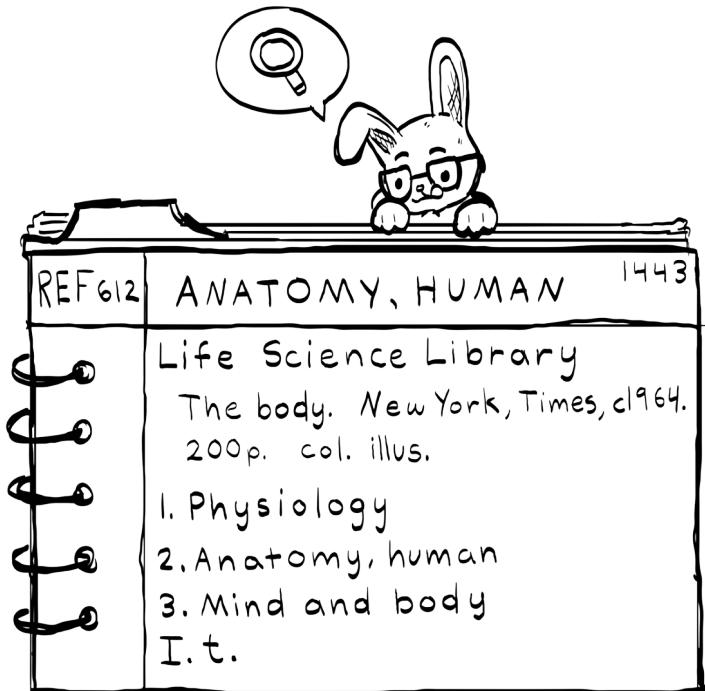
### 3.2.3 Performance and optimization

A well-designed database should be able to optimize query performance and reduce the response time. An efficient database can help save the running costs, and boost the performance of the applications it supports, which in turn will enhance the user experience.



You can take a lot of measures to optimize the performance of a database. For example, the data in the review table you saw in Chapter 2 often needs to be sorted, because the table and its database support the online store of The Sci-Fi Collective, and the potential customers often want to see the latest reviews of the products they browse. You can index the date column in the review table to speed up the sorting operation. What is *indexing*? Think of the data in the review table as a library of books. Indexing is the process of

creating an index card catalog that lists every book alphabetically along with its location. When it comes to sorting, you can simply use the “index card catalog” to locate every book and put it in its sorted position.

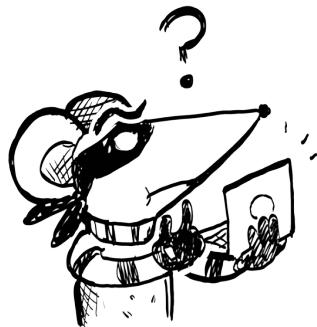
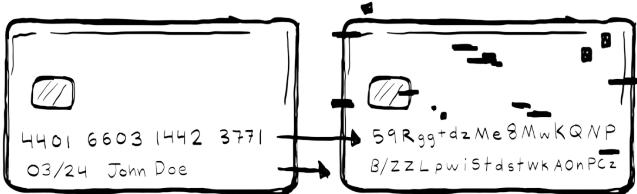
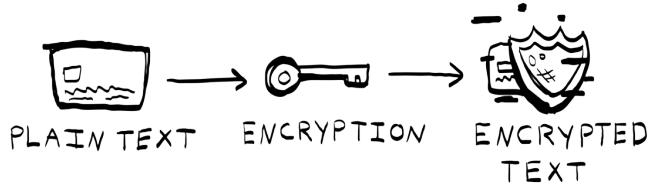


Remember card catalogs?

### 3.2.4 Data security

A well-designed database should have robust security measures in place. In other words, a well-designed database needs to prevent unauthorized access, insertion, modification, or deletion. Even if such issues happen, the sensitive data should still be well-protected and easy to recover.

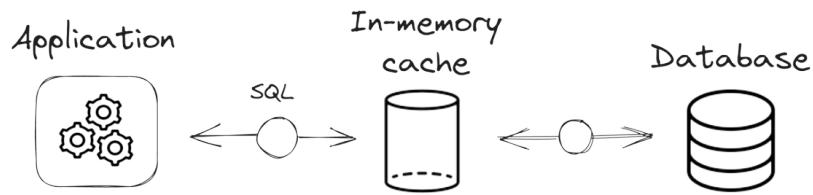
You can take a lot of measures to safeguard the data. For example, if you ever need to store payment method information in your database, you should store only the encrypted payment method information. Storing customers’ payment method information in plain text is a bad idea. If an evil hacker gains access to the database, the hacker will know everyone’s credit card number. In contrast, encryption helps protect sensitive information even in the worst-case scenario.



### 3.2.5 Scalability and flexibility

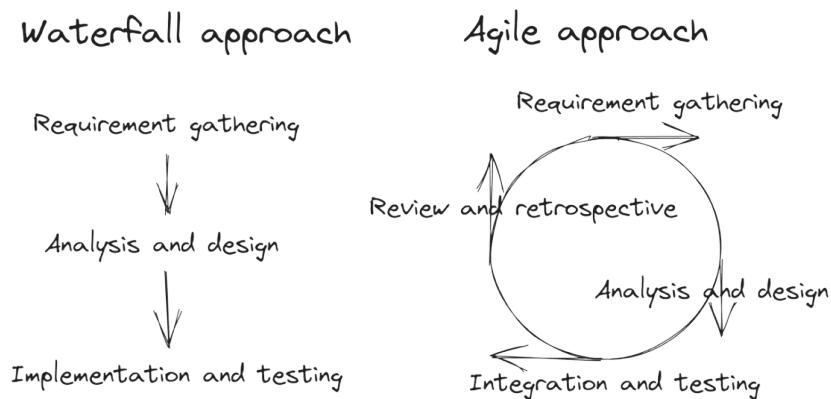
A well-designed and efficient database should be able to accommodate future growth and changing requirements, without sacrificing performance (a.k.a., trying to have your cake and eat it too).

You can take various measures to enhance the scalability and flexibility of your database design. For instance, when designing your database schema, separating tables can make them smaller, which in turn can speed up data lookups. Additionally, you can implement a cache mechanism for frequently accessed data, like the product information in The Sci-Fi Collective's database. Caching involves storing frequently accessed data in fast-access memory, such as RAM, which can significantly improve database performance and responsiveness, particularly as the data size grows. Popular caching systems like Redis and Memcached can be utilized to implement this caching mechanism.



### 3.3 Overview of the design process

In this section, we review the overall database design process by covering the key phases and steps in database design. Some approaches to database design emphasize a sequential and well-defined process, as seen in the *waterfall* approach, where each phase must be completed before moving to the next, while others, like the agile approach, focus on an iterative and flexible approach, allowing for adjustments as the project unfolds. Despite the differences, all database design approaches share the same key phases, including requirement gathering, analysis and design, and *implementation/integration and testing*.

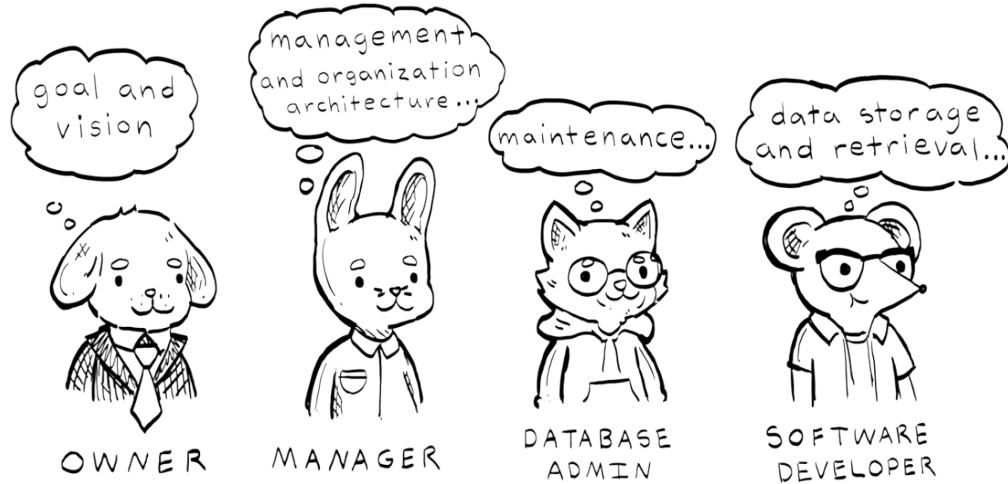


#### 3.3.1 Requirement gathering

Requirement gathering refers to gathering information about the database via different ways, such as talking to all the people involved in or will be using the database, studying existing databases (if there are any), or examining other relevant aspects of information management.

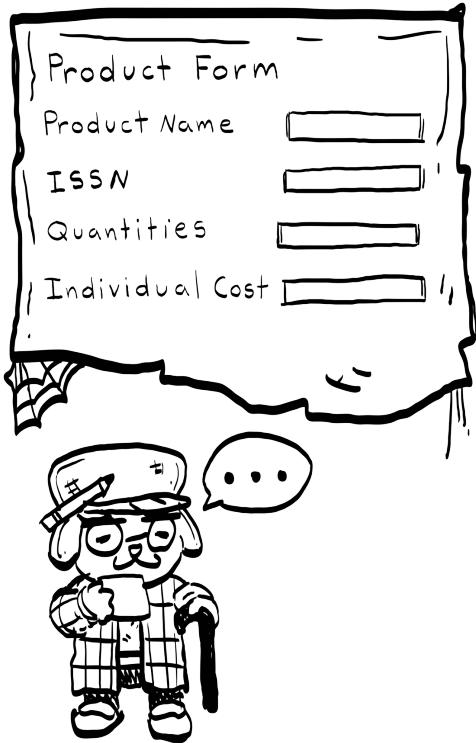
When it comes to talking to all the people involved in using the database, you will need to organize meetings, ask good questions, and have conversations with different groups. For example, to build the database for the Sci-Fi

Collective, you would talk to the shop owners and managers, future database administrators, and software developers who will build its web and mobile applications to find out what kind of information they need to keep track of.



If there are any existing databases used by a legacy application, you will need to study the legacy application and the databases carefully. Figuring out the gap between the current expectation and the old databases is critical to a successful database design. For example, the Sci-Fi Collective has a legacy online store. After you study its database and talk to all the stakeholders, you will see that the old database doesn't support tracking inventory numbers of in-stock products, which sometimes leads to a scenario where customers buy products when the Sci-Fi Collective no longer has any in stock. The web app developers of the Sci-Fi Collective hope the new database to track the inventory number, so that the web app can let customers know when a particular product goes out of stock in a timely manner.

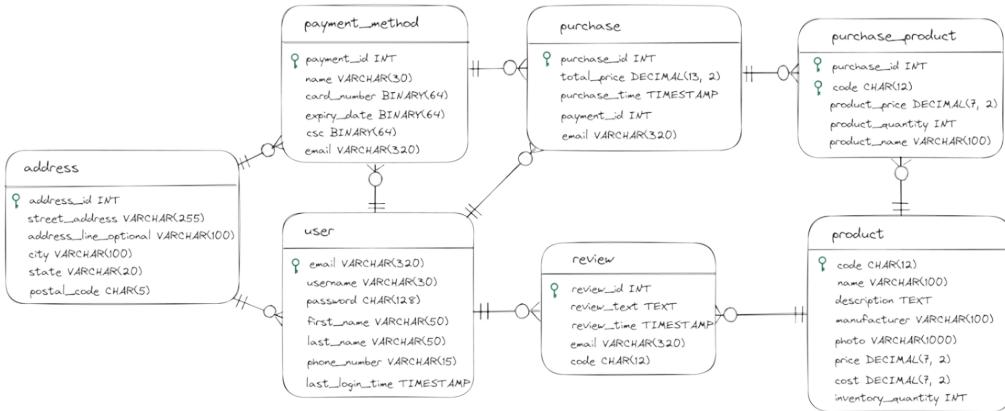
Any information management tasks within the organization that stakeholders expect to be a part of the database you design are also relevant. For example, the purchasing manager of the Sci-Fi Collective used to manage the inventory information using a spreadsheet and a paper form. To make this aspect a part of the database, you will need to study the paper form, the spreadsheet, and the management flow using these two tools.



### 3.3.2 Analysis and design

The analysis and design phase is about carefully thinking through all the requirements and coming up with a solid plan for how the database will be structured and how it will work. In the end, you will create a detailed blueprint of the database. Some of the key steps in this phase include data modeling and normalization.

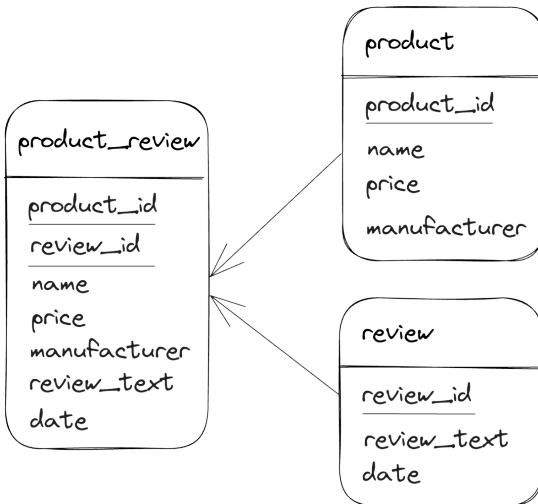
Data modeling aims to create a conceptual design that shows how the different parts of a database fit together and how they relate to each other. The conceptual design is typically represented as a visual diagram called an entity-relationship (E-R) diagram. For example, an E-R diagram for the database of the Sci-fi Collective could look as follows:



We know that this diagram contains symbols and syntax that you may not understand yet. You will learn them and how to model data using E-R diagrams in Chapter 5.

Normalization comes after when the E-R diagram is established.

Normalization is about minimizing redundancy by breaking down a table representing more than one entity into smaller, logical units and organizing them into separate tables. For example, someone designed a `product_review` table to hold the data of both the products and their reviews. As you saw in Chapter 2, storing information of more than one entity in the same table can lead to redundancy and anomalies. Such a table could be “normalized” and broken into two tables, `product` and `review`.



You will learn more about normalization in Chapter 6, and how to determine if a table needs to be normalized.

### **3.3.3 Implementation/Integration and testing**

The implementation/integration and testing phase is about building and validating the database based on the blueprint you made in the design and analysis phase. If the waterfall approach is used, the database will be implemented all at once; if the agile approach is used, a part of the database will be implemented and integrated into what has been implemented. After that, you will test the database to ensure that it functions correctly, performs well, and meets the intended requirements.

During implementation, you will create the tables, define the columns and their data types, establish relationships between tables, apply any constraints or rules specified in your design blueprint, and determine which columns to index to optimize query performance. You learned how to create a single table or related tables using SQL in Chapters 1 and 2, and that knowledge can be very useful for this step.

Once the database has been implemented, you want to test it before putting it to use. Typically, the testing of a database needs to validate at least three different aspects:

- **Functionality:** You need to check if the database performs the expected tasks correctly, such as creating, updating, and deleting a data entry.
- **Performance:** You need to check how well the database handles large amounts of data or heavy usage.
- **Security:** You need to verify that the database has appropriate security measures in place to protect the data. For example, in the database of The Sci-Fi Collective, the transaction data and purchase history should only be accessible to the authenticated user.

Of course, you should identify and fix any bugs discovered during testing. Just as in the implementation phase, the knowledge of SQL queries you gained from Chapters 1 and 2 will be very useful for testing. You will learn more about the details of this phase in Chapter 6 and 7.

## **3.4 Requirement gathering**

In the last section, you had an overview of the process of database design. Starting from this section, you will explore and learn the key phases in database design by designing a database for the online store of The Sci-Fi Collective from scratch. Working on a project from scratch will give you hands-on experience and expose you to detailed knowledge components that you would otherwise not pick up.

In this section, you will learn more about the first key phase in database design - requirement gathering. Requirement gathering is about gathering information from different venues, such as interviewing all the people involved in using the database, studying existing databases (if there are any), or examining other relevant aspects of information management. Because requirement gathering is an art rather than science, leveraging advice and insights from experts or veterans in this trade can help make your life much easier.

### **3.4.1 The goals of the database**

As you take on any requirement gathering tasks, you should keep in mind that a critical question needs to be answered based on all the information to collect – What are the goals of the database? Every database is created for some specific purposes, whether it is to handle the daily transactions of a business, or to manage the information of an organization.

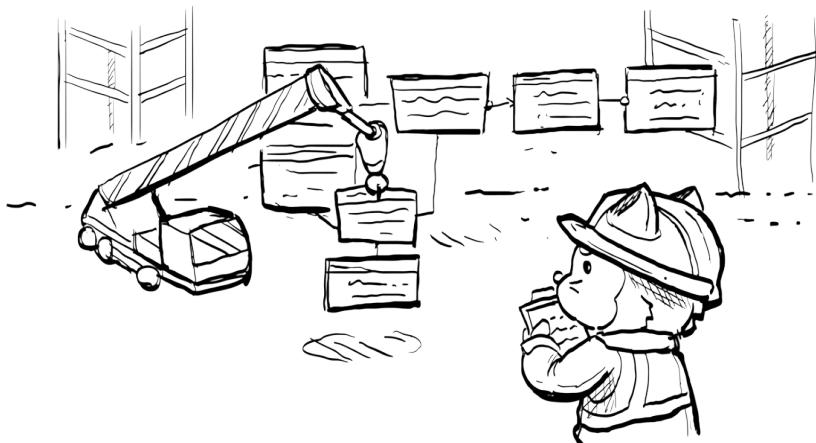
You need to identify the goal of the database clearly because it will be used to make important decisions in database design. It may sometimes take longer than expected to come to a full understanding of the goal of the database, and you definitely need to be ready for that. For example, to have a good understanding of the goal of the database for The Sci-Fi Collective, you will need to interview the owner, managers, staff, and software developers. You can summarize the goals of this database as follows:

- The database of The Sci-Fi Collective is to maintain the information of products, such as their inventory and reviews, and information of users, such as their payment information and purchases, as well as the transaction information linking users and products.

### **3.4.2 Existing databases and other aspects of information management**



Sometimes there might be an existing database that you can refer to for your work. If so, you should resist the urge to base your new design on the structure of the existing database. There is a good reason why the organization/business decided to hire you to design a database from scratch instead of modifying the old database themselves. Although the existing database can contain valuable information in terms of what entities and attributes are needed to structure some tables, you need to be careful with potential design errors in the existing database, and recognize that it will take the same amount of, or more, effort to figure out the current demands for the new database, and how they are different from those of the existing database. How do you figure out the current demands? Interviews. How do you figure out the gap between the existing database and the current demands? Interviews with more questions.



When it comes to other aspects of information management, you will typically find that many people discovered ingenious ways to use word processors, spreadsheets, or paper forms to collect and manage data effectively. If such data management needs to be a part of the new database you design, you may want to do at least two things. First, you want a subject matter expert to walk you through how the data is managed via demonstration and stepwise explanation, because it is usually difficult to grasp such data management through only interviews. Second, you should ask for data samples whenever possible. The data samples will play an important role in helping you verify the attributes and data types in the next phase of database design.

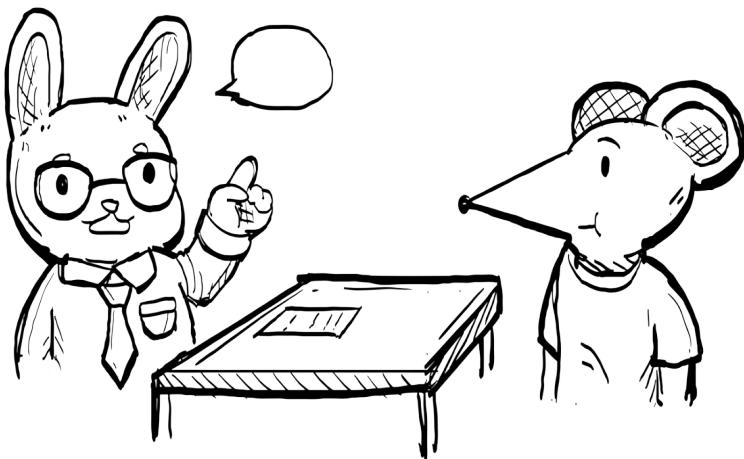
Let's use an example to illustrate the above two points. The purchasing manager of The Sci-Fi Collective currently manages all the inventory data in a spreadsheet, and the database you design will eventually replace the spreadsheet to manage the inventory data. Instead of just talking to this manager and the purchasing team, you want them to walk you through the whole process of purchasing a batch of products, such as ordering, data entry of a new product, update and removal of a product record that's already in the inventory. The process can be complicated, so a demonstration is the best whenever possible. More importantly, you should ask for a copy of the authentic data, which should be anonymized ideally. The data will help clarify many issues that the demonstration can't, such as the exact number of attributes and data types.

	A	B	C	D	E	F	G
1	Product name	Manufacturer	Description	Cost	Quantities	MSRP	
2	Neuro-Enhancer	CyberDyne Corp	Neural implant ...	\$499.99	10	\$799.99	
3	Quantum Flux...	HyperTech Labs	Revolutionary ....	\$999.99	2	\$2999.99	
4	Plasma Blaster	NovaArmory	Portable hand ...	\$299.99	15	\$599.99	
5	Nanotech Gel	BioSolutions Inc	Advanced nano...	\$79.99	50	\$149.99	
...	...	...	...	...	...	...	...

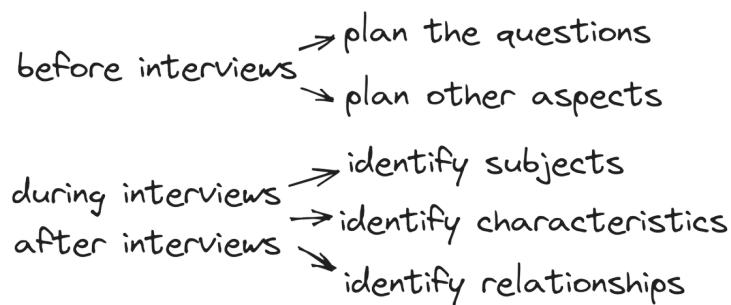
### 3.4.3 Interviews

Interviews are the most important tasks in requirement gathering. Before interviews, you need to carefully plan the interview questions and other aspects of interviews depending on the project and interview participants.

During and after the interviews, you need to identify subjects, characteristics, and relationships among subjects.



These three pieces of information will be critical for you to sail through the next phase in database design. Relationship is self-explanatory, but what are subjects and characteristics? Subjects stand for people, places, things, events, or anything similar. Characteristics refer to the features of subjects.



## Prepare for interviews



Before interviews, you need to carefully plan the interview questions and other aspects of interviews. What questions should you prepare? It really depends on the project as well as whom you are interviewing. To simplify the

problem, you may want to group the people you interview. Although the number of groups can vary, a business or organization typically has at least three groups of people that you should consider interviewing, including the stakeholders, subject matter experts, and IT/technical staff.



For example, The Sci-Fi Collective happens to be such a typical business – we have two owners and one manager, two Minotaurs responsible for tasks such as data entry and customer service, and three elves working as software developers. Below you can find some sample interview questions for each group:



- Stakeholders:
  - What is the primary purpose of the database and what specific goals do you want to achieve with it?
  - What are the key features or functionalities that you expect the

- database to support?
- Are there any specific reporting or analytics needs that the database should support?
- Subject matter experts:
  - How do you currently manage and organize your data? What challenges or limitations do you face?
  - Can you walk me through the typical workflow or process you follow when working with data?
  - What specific information or data elements are most critical to your work?
- IT/technical staff:
  - What are the main tasks or activities you perform that involve data storage or retrieval?
  - What reports or outputs do you typically generate from the data? What information do these reports provide?
  - Are there any specific calculations or computations you perform on the data?

Of course, you should prepare more questions for each group yourself. What is the guiding principle for preparing interview questions? A good question should help gather relevant information on what data the database should store, how the data should be stored, and what constraints should be put in place. However, if there's no time constraint, a bad question is always better than no questions at all. If you are totally new to this, you can always ask for some example questions from ChatGPT. For example, you may ask ChatGPT to provide some sample questions using a prompt as follows:



- When you need to design a database, you need to conduct interviews with stakeholders. What questions are typically asked during such interviews?

Beyond interview questions, you need to prepare many other things

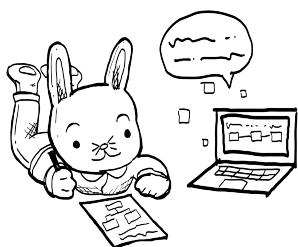
beforehand. Here are some key questions that you need to ask yourself during the preparation:

- How do you want to record the interview? Taking notes, or recording the conversation? If the latter, do you need to gain permission from the organization?
- Where do you want to conduct the interviews?
- Do you need to invite more than one group to the same interview? If so, who had better not be invited to the same interview? Should there be a limit on number of interviewees?

Of course, you will need to ask yourself more questions specific to the project and participants of the interviews. The guiding principle of any preparation is to make the interviews productive and informative.

### **Identify subjects, characteristics, and relationships**

During and after the interviews, you need to identify subjects, characteristics, and relationships among subjects. The identification of subjects, characteristics, and relationships among subjects is critical for you to sail through the next database design phase, analysis and design. In case you are wondering why, one of the tasks you will take on during the next design phase is to map the subjects, characteristics, and relationships to entities, attributes, and relationships among entities.



To identify subjects and characteristics during the interview or from the record of the interview, you can look for nouns in the responses to your questions. How do you differentiate subjects from characteristics? Typically, if you can build a sentence with two nouns in which one “has” or “owns” the other, the one that is possessed is the characteristic, and the other is the subject. For example, you can put “user” and “password” in the following sentence:

- A user has a password.

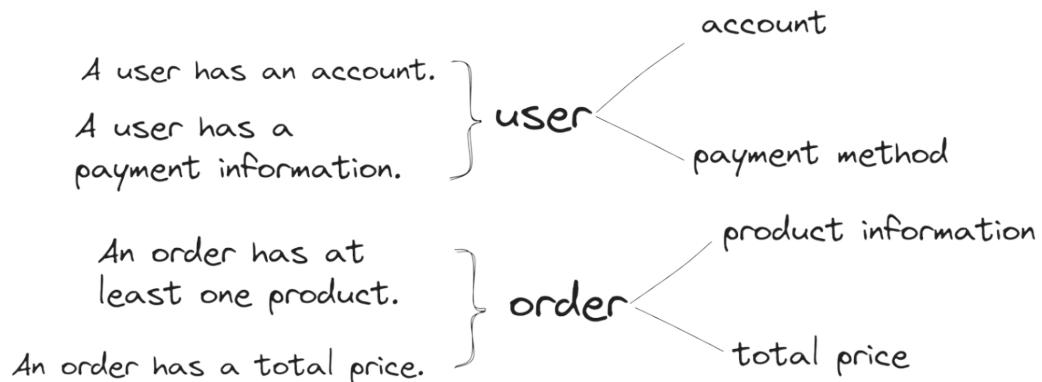
The password is possessed by the user, so it is a characteristic of a user, whereas the user is a subject. You will need to perform similar deductions on the interview conversations. For example, you may ask the following question of an IT staff member working for The Sci-Fi Collective:

- What are the main tasks or activities you perform that involve data storage or retrieval?

The participant may give you a response as follows:

- As a software developer, *I am mainly responsible for building and maintaining the transaction system of the online store. When a user makes an order, the transaction system is supposed to retrieve the user's account and payment method, generate an order with all the information for the ordered products, calculate the total price, bill the payment method, and generate an invoice.*

This is a very good response. The participant answered your question well with a lot of useful details. You can track all the nouns and see if they can be used in sentences with the verb “has” or “owns” to identify the subjects and characteristics:



As you can see, it is not hard to deduce at least two subjects, user and order from this response. A user may have two characteristics, including the account and payment method. An order may have two characteristics, product information and total price. Additionally, there are another two subjects, invoice and product, which don't have any characteristics.

Typically, you need to ask follow-up questions to get all the characteristics of

a given subject. For example, you may have a follow-up conversation with the participant as follows:

- What information does a complete order have?
  - Well, the same as any other online store: the prices and quantities of all the products bought by a customer. That's it. An order is not complicated.
- What about the total price? Is the total price a part of the order?
  - Sort of. Yes. The total price is calculated based on the unit price and quantities of all the bought products.
  - What about the date and time when an order was put into the system? *Is that a necessary piece of information for an order?*
  - Yes, yes. That's absolutely a necessary piece.

The answers such as “the same as any other online store” and “Sort of” are vague, and often require you to follow up and ask clarification questions. Luckily, the participant provided enough details after such vague answers. Based on the answers, you can update the characteristics of the order subject:



To wrap up the discussion on subjects and characteristics, you should always ask for sample data if possible. For example, when you have a good understanding of the subjects and characteristics discussed by the participant, you could follow up with a question as follows:

- Can you provide me some sample data for products, orders, invoices and users? Anything will help.

After identifying the subjects and characteristics, you will be ready to ask about relationships among subjects. Your focus should be the relationship between every two subjects. For example, you can ask a follow-up question about the relationship between users and orders as follows:

- How are orders related to users? Can a user make multiple orders? Can multiple users contribute to one order?

The participant may give you a response as follows:

- A user can of course make as many orders as he or she likes.  
However, our system doesn't support multiple users contributing to one order.

For another example, you can ask a follow-up question about the relationship between products and orders as follows:

- How are orders related to products? Can an order have more than one product? Can a product show up in more than one order?

The participant may give you a response as follows:

- An order can have more than one product. Vice versa, a product can show up in different orders.

You don't necessarily need to do any analysis on these responses except for recording them well.

So far, you have walked through all the necessary steps in an interview. When you finish interviewing all the groups of The Sci-Fi Collective, you will be able to identify a set of subjects and characteristics associated with each subject:

user	product	payment method
* username	* code	* name
* email	* name	* credit card number
* password	* description	* expiry date
* first name	* manufacturer	* csc
* last name	* photo	* billing address
* phone number	* price	
* address	* cost	
review	* inventory quantity	
* review text		
* date and time		

order
* total price
* product price
* product quantity
* date and time

Additionally, you will be able to identify the following relationships:

- A user can make multiple orders. An order can be made by only one user.
- A user can review multiple products as long as the user bought those products. A product can be reviewed by multiple users.
- A user can maintain multiple payment methods. A payment method can be associated with only one user.
- An order can have more than one product. A product can show up in multiple orders.

With the above information, you are ready to start the next phase in your journey of designing a database for The Sci-Fi Collective.

### 3.5 Summary



- The overall goal of database design is to deliver a well-structured and efficient database.
- A series of key subgoals of database design include data consistency and integrity, maintainability and ease of use, performance and optimization, data security, scalability, and flexibility.
- All database design approaches share the same key phases, including

requirement gathering, analysis and design, and implementation/integration and testing.

- Requirement gathering is gathering information about the database via different ways, such as talking to all the people involved in using the database, studying existing databases, or examining other relevant aspects of information management.
- The analysis and design phase involves carefully thinking through all the requirements and coming up with a solid plan for how the database will be structured and how it will work.
- The implementation/integration and testing phase is about building and validating the database based on the blueprint you made in the design and analysis phase.
- Interviews are the most important tasks in requirement gathering. Before interviews, you need to carefully plan the interview questions. During and after the interviews, you need to identify subjects, characteristics, and relationships among subjects.

# 4 Entities and attributes

## In this chapter

- You start the design and analysis phase in database design.
- You learn about keys and common data types in databases.
- You design your first few entities, identifying their attributes, primary keys, and data types of attributes.

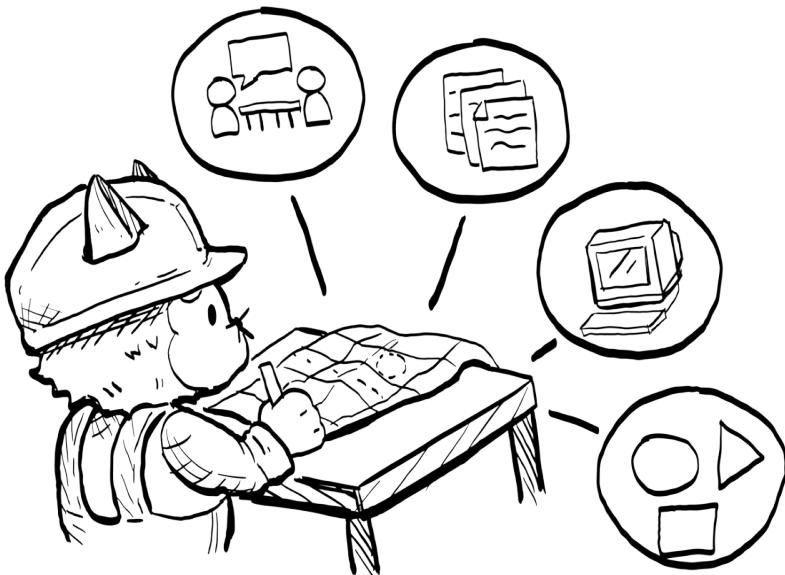
### What you need to know

You can find the database design covered in this chapter (so far it's only entities and attributes) implemented in tools commonly used by practitioners, such as dbdiagram.io, and MySQL Workbench in the GitHub repository (<https://github.com/Neo-Hao/grokking-relational-database-design>). You can navigate to the chapter\_04 folder, and follow the instructions in README.md to load the database design into corresponding tools.

The data types covered in this chapter are applicable to most RDBMS, such as MySQL, MariaDB, and PostgreSQL. If you are using another RDBMS such as SQL Server or SQLite, you need to make small modifications to the design by replacing certain data types with their equivalences specific to an RDBMS. Such information can be found in README.md in the chapter\_04 folder.

## 4.1 Overview

In Chapter 3, you walked through an overview of the database design process, and went over the requirements gathering phase for the online store of The Sci-Fi Collective.



Starting from this chapter, you will move to the next database design phase, design and analysis. The first step of design and analysis is data modeling, which aims to generate an entity-relationship (E-R) diagram. In this chapter, you will focus on designing all the required entities for the database of The Sci-Fi Collective online store. By doing so, you will learn about entities, attributes, keys, and data types.

## 4.2 Entities and attributes

In this section, you will focus on turning the subjects and characteristics you deduced from requirements gathering into entities and attributes. The subjects and characteristics for the online store of The Sci-fi Collective are shown below:

user	product	payment method
* username	* code	* name
* email	* name	* credit card number
* password	* description	* expiry date
* first name	* manufacturer	* csc
* last name	* photo	* billing address
* phone number	* price	
* address	* cost	
review	* inventory quantity	
* review text		
* date and time		

order
* total price
* product price
* product quantity
* date and time

In Chapter 1, you were introduced to entities and attributes. It is a good time to refresh your memory of on those concepts – An entity is a distinct object or concept that can be described by many attributes.

A subject and its characteristics may seem ready to be turned into an entity with attributes with little to no changes. A subject is simply an entity, and a characteristic is simply an attribute. However, you need to put a little bit more thoughts into how to name the entities and attributes.

First, you need to pick between singular and plural names in terms of naming your entities and attributes. More importantly, you need to stick to your choice when naming all the entities and attributes. Sticking to a consistent naming convention can help avoid typos and other errors in query writing and database maintenance. In this book, we will stick to the singular naming convention.



### **Singular vs. Plural - To 'S' or Not to 'S'?**

Edgar Codd at IBM developed the first relational database management system in the 1970s. In his database, he used singular names for entities (e.g., employee). Others followed his lead. Singular names are best when used with primary entities (e.g., single employee table). Singular approaches have their root in Object Oriented Programming (OOP), where a class translates as an entity that contains several objects of the same class. On the other hand, plural names are more natural as table titles. For example, the employees table contains records of employees. Of course, the same likelihood of confusion and errors exists with plural table names. As such, our best recommendation is to aim for consistent use of either convention.

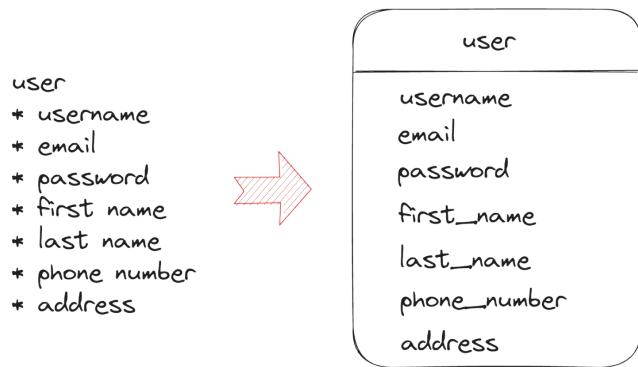
Second, you need to pick a naming convention for the attributes and stick to

it. Commonly used naming conventions include:

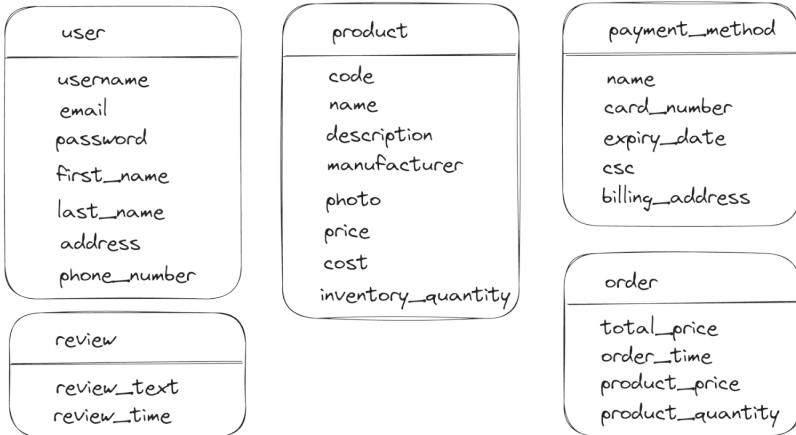
- Snake case – e.g., `first_name`
- Camel case – e.g., `firstName`
- Pascal case – e.g., `FirstName`
- Upper case – e.g., `FIRST_NAME`
- Hungarian notation: a special case that includes the data type (often abbreviated) as a part of the name – e.g., `strFirstName`
- Semantic naming: a special case that includes the purpose of a variable as a part of the name – e.g., `customerName`

Among these naming conventions, the snake and camel cases are equally popular, and the Pascal case comes the next. At the end of the day, you need to make a choice based on your preference and the unique software requirements. In this book, we will stick to the snake case.

With these two naming decisions made, you can easily map subjects/characteristics to entities/attributes. For example, the user subject and its characteristics can be mapped to the `user` entity and its attributes:

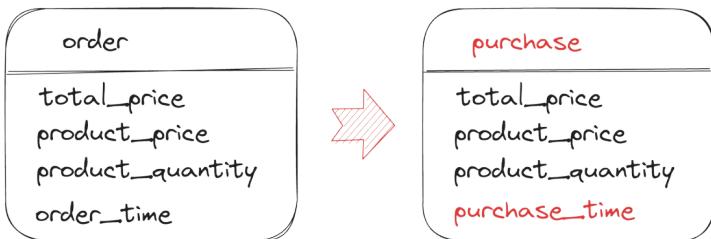


Following the two naming conventions, we will convert all the subjects/characteristics for the online store of The Sci-fi Collective to the following entities/attributes:



Beyond the naming conventions, you need to check two more things in terms of column names:

- whether you have super long names whose lengths surpass the upper limit. Many RDBMS have upper limits on column name lengths (e.g., MariaDB – 64 characters). If you have such a column name, you should shorten it.
- whether you used any reserved SQL keywords as your entity or attribute names. Using reserved SQL keywords as names can lead to many issues, such as syntax errors in SQL query execution and maintainability issues. For example, if you used `SELECT` to name an entity, the database system may not understand whether you are referring to the keyword `SELECT` or the entity with that name. Therefore, if you find yourself using any reserved keywords as names, you should replace them. The reserved SQL keywords may vary from one database system to another. You can find a list of keywords in their documentation (e.g., [MySQL documentation](#)). All database systems share a set of common keywords, such as `SELECT`, `ORDER`, `INSERT`, `GROUP`, `JOIN`, etc. In our case, we used the reserved SQL keyword `ORDER` to name one of our entities, so we need to replace it with a different word with a similar meaning, such as `purchase`.





## Reserved keywords in SQL

The reserved keywords in SQL have specific meanings and functions. They are used to define, manipulate, and control the database structure and data. The list of reserved keywords can vary slightly depending on the RDBMS. However, the most common keywords are shared by all RDBMS.

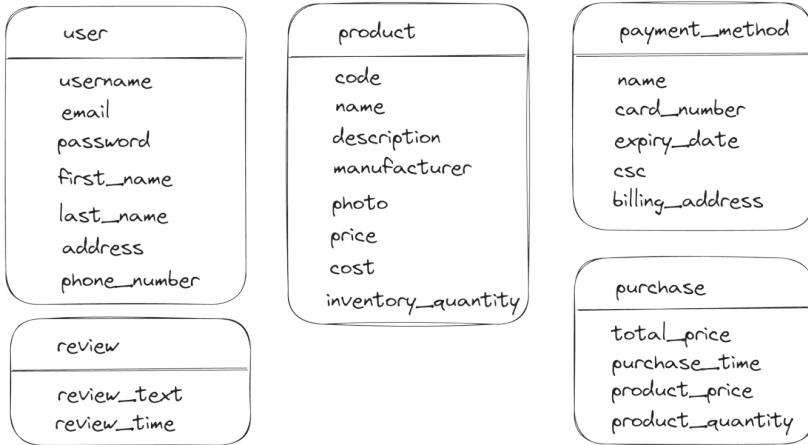
If you just want to know the common reserved keywords, the fastest approach is probably to ask a generative AI tool, like ChatGPT:

- What are the common reserved keywords in SQL?

If you need to know the reserved keywords of a specific RDBMS, asking a generative AI tool can still be a good start. For example, to know the reserved keywords in MySQL, you can ask ChatGPT the following questions:

- What are the reserved keywords in MySQL?
- Where are the reserved keywords of MySQL listed in its manual?

When you finish all the above steps, the entities of the Sci-Fi Collective online store will look like the follows:



## 4.3 Keys

Now that you have converted the subjects and characteristics you deduced in Chapter 2 to entities and attributes, you will focus further on your design of entities and attributes by identifying primary keys in this section.

In Chapter 1 you learned about primary keys. Now is a good time to refresh your memory on this concept. A primary key refers to one or more attributes that can be used to identify an individual data record. The values of primary keys are unique. Given a table, there can be only one primary key.

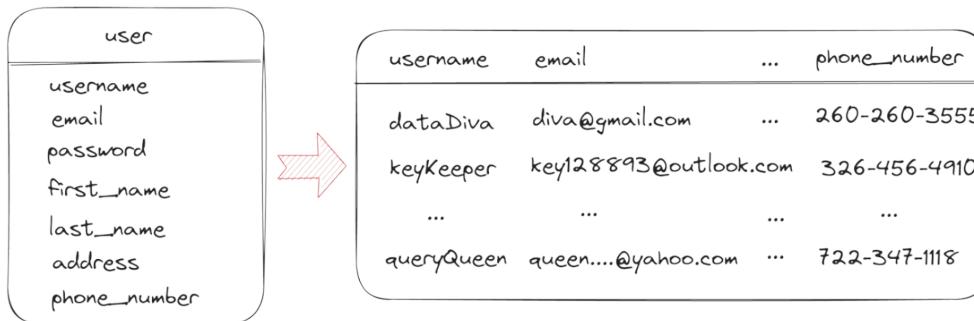


Why does an entity need a primary key? The entity (set) will eventually be converted to a table in a database. The primary key helps identify each row in a table uniquely, enforce data integrity by preventing duplication or inconsistency, establish relationships between tables in a relational database, and more. All in all, identifying primary keys is a very important step in completing your database design.

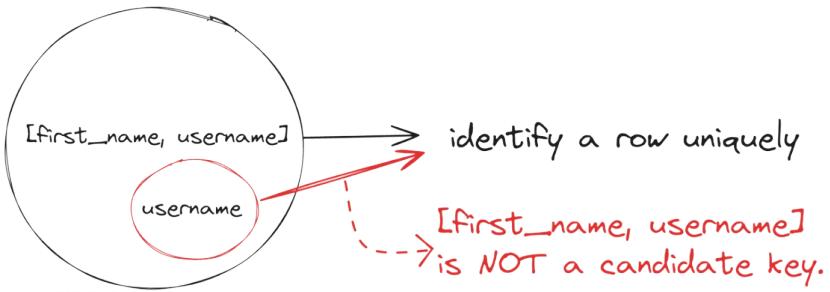
The guiding principle in identifying a primary key is simple – Pick the best candidate key as the primary key; If there are no good candidate keys available, you will create a new attribute that serves as the primary key. We will explain what candidate keys are and demonstrate this principle using two examples, starting from the user entity.

The user entity contains 7 attributes. When you put the gathered requirements and sample data side by side, you can easily spot the attributes that should have unique values and can prevent duplications of user data:

- **username:** Two users cannot have accounts with the same username. Different users can't choose the same username.
- **email:** Emails can be used for verification and account recovery, which requires them to be unique. An email address can't be used to register two different users.
- **phone\_number:** Phone numbers can be used for verification and account recovery, which requires them be unique. Different users can't register the same phone number.



First things first: what is a candidate key? The three attributes are three different candidate keys for the user table. A *candidate key* is the smallest possible combination of attributes that can uniquely identify a row in a table. “Smallest” means that no subpart of a candidate key can uniquely identify a row. For example, the combination of `username` and `first_name` can identify a row uniquely. However, `username` as a subpart of this combination can also uniquely identify a row. As such, the combination of `username` and `first_name` is not a candidate key. On the other hand, `username` is a candidate key.



Is there a systematic approach to identifying candidate keys? Yes. You will start by examining each attribute to see if it can uniquely identify a row. If you can't find such an attribute, you will start combining columns to see if they can uniquely identify a row. In our case, the user table contains three different candidate keys: `username`, `email`, and `phone_number`. To pick one primary key among the three choices, you must compare their qualities.

	(username)	(email)	(phone_number)
unique	✓	✓	✓
Non-empty (null)	✓	✓	✗
Stable	✗	✓	✓
Simple	✓	✓	✓
Short	✓	✗	✓
Familiar	✓	✓	✓

The above list is not exhaustive but serves as a general guideline. The meanings of these requirements are self-telling. For example, “stable” means “not likely to change”, and “simple” means “easy to understand and use”. The *uniqueness* and *non-empty* (non-nullability) requirements are satisfied almost by default in `username` and `email`. The `phone_number` values might be null, because even if a customer doesn’t have or want to give us the phone number, we would still welcome that customer to register as a user. The rest of the metrics can be quite subjective. For example, usernames can be easier

to change than email addresses. On the other hand, email addresses tend to be longer, which affects access speed. For our case, we will settle with the `email` as the primary key for the `user` table for its stability – The Sci-Fi Collective allows users to update the usernames but not the email addresses. You can indicate which attribute is the primary key by either underlining it or putting a key symbol on its left side.



For another example, among the 7 attributes and their possible combinations in the `product` table, you can at least identify two candidate keys:

- In real world, most products have product codes, which can help identify products uniquely. Since each product has a unique code, the code can also prevent redundancy.
- The combination of product names and manufacturers can also identify products uniquely as well as prevent redundancy in the `product` table.

The comparison between these two candidate keys is as follows:

	(code)	(name, manufacturer)
unique	✓	✗
Non-empty (null)	✓	✓
Stable	✓	✓
Simple	✓	✗
Short	✓	✗
Familiar	✗	✓

The product code is a clear winner in comparison to the combo of product name and manufacturer. The product code is simpler and shorter. Although product code might be less familiar, it is not difficult to grasp. As such, we can settle with `product_code` and use it as the primary key for the `product`

table.

product	
key	code
	name
	description
	manufacturer
	photo
	price
	cost
	inventory_quantity



In both above two examples, we picked the primary key from a set of candidate keys. What if there are no candidate keys available? What if there are some candidate keys, but none of them is a good fit? We will answer this question by picking a primary key for the review table. The review table has two columns, review\_text and review\_time. Neither of the two columns can uniquely identify a row in the review table. Although the combination of review\_text and review\_time can identify rows uniquely for most cases, it is still possible for two reviews with the same texts to be recorded at the same time. That said, there are no candidate keys available for the review table.

	(review_text)	(review_time)	(review_text, review_time)
unique	✗	✗	✗
Non-empty (null)	✓	✓	✓
Stable	✗	✓	✗
Simple	✗	✗	✗
Short	✗	✗	✗
Familiar	✓	✓	✓

When you are in a situation like this, you should always think about the alternative – create a new column and use it as the primary key. You can opt

to create a numeric column, and use it as the primary key for the `review` table. Numeric primary keys are smaller, and can fit completely in computer memory, making them much faster to search through. They also scale better and are more standardized than a combination of two attributes. As such, we will create an auto-increment numeric attribute `review_id` and use it as the primary key for the `review` table. A key as such is known as a *surrogate key*.



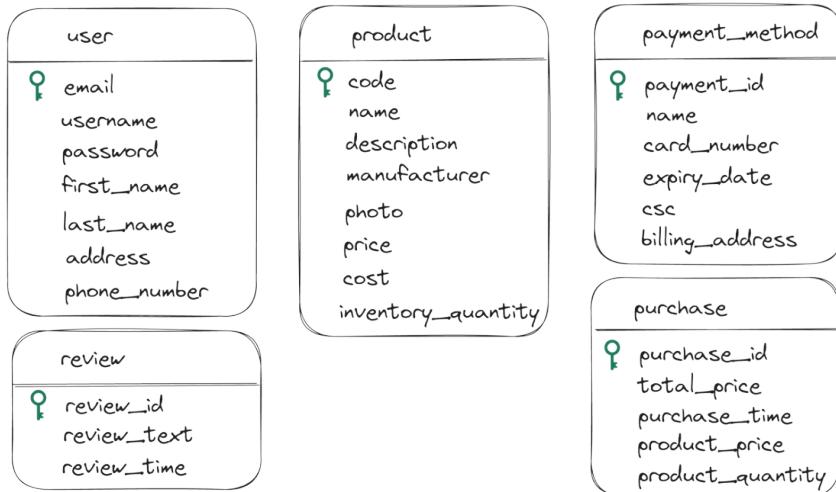
Undoubtedly, a surrogate key can identify rows uniquely in a table. However, can such a key prevent redundancy? Well, not by itself. It is still possible for two identical rows of data to be inserted into the `review` table:

review_id	review_text	review_time
...	...	...
7	"Great product"	"9/14/2023, 9:15:15 AM"
8	"Great product"	"9/14/2023, 9:15:15 AM"

Would redundancy be a problem? For tables such as `user` and `product`, it is very important to eliminate redundancy. Think about maintaining two rows of data that represent the same user – which one should be used when the user updates his or her username? In contrast, reviews are always displayed as a list of texts on the same screen. As long as a row of review data can be identified uniquely, it can be updated or deleted properly. As such, the redundancy issue table won't be much of a problem for the `review` table.

Following the same logic, you will see that the `payment_method` and `purchase` tables are in a similar situation in which identifying each row uniquely is more important than eliminating redundancy. For the `purchase` table, redundancy is tolerable. The `purchase` table represents the transaction records of users. A transaction record is immutable. For the `payment_method`

table, redundancy is not only tolerable, but also acceptable. The Sci-Fi Collective allows two different users to add the same payment method to their individual accounts. Think about when a couple use each other's credit cards. In such scenarios, auto-increment surrogate keys would be a great choice for the primary keys – we will name the surrogate key columns `payment_id` in the `payment` table and `purchase_id` in the `purchase` table. The primary keys of all entities are as follows:

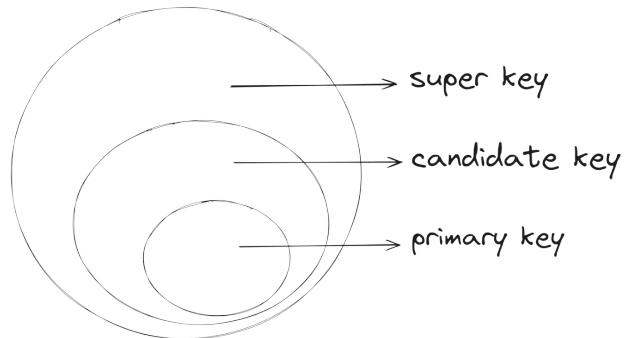


### Super keys, candidate keys and primary keys

There is another type of key that you didn't pick up from above – super keys. A super key is a set of one or more columns of a table that can uniquely identify a row in the table. However, shame on a suer key, it may contain unnecessary columns that are not required to uniquely identify a row.

Now you know what super keys are, you can see candidate keys from a different angle – a candidate key is a minimal super key, which means that it is a super key with no unnecessary columns.

Lastly, out of all the candidate keys in a table, one is chosen as the primary key. The relationships among the three types of keys are summarized below:



## 4.4 Data types

You have successfully identified or generated the primary keys of all entities in the last section. In this section, you will work to define data types for all attributes. This is a necessary step in completing your design of entities and attributes. To implement the database based on your design, the data types of attributes are required information.

Data types are not entirely a new topic. In Chapters 1 and 2, you saw how data types are used in SQL queries. In Chapter 3, you were encouraged to get sample data that helps you figure out data types of characteristics of subjects in the requirements gathering phase. In this section, you will take advantage of the sample data you accumulated, and use those to help you define data types of all attributes accurately. Meanwhile, you will also pick up some new conceptual knowledge about data types.

### 4.4.1 String data types: power of personality

String data types are one of the most used data types in databases. Before you can use string data types, you will learn a little bit about the variations of string data types first.

#### **CHAR, VARCHAR, and TEXT**

The most common types of string data are CHAR, VARCHAR, and TEXT. The

major difference among them lies in storage requirements (i.e., some require more bytes than others due to how they are physically implemented).



You may be asking, why do bytes even matter? After all, people have terabytes of available space on their disk. When databases are small, the result is negligible, but as they grow, so do computational requirements (e.g., access speed) and memory requirements. Here's an example that can help you taste the effect of data length on search speed.

1	1,231,321,546
2	3,321,547,542
3	3,432,786,876
4	5,456,789,342
5	1,546,898,321
6	7,546,768,853
7	2,345,657,789
8	9,324,546,768

Please try finding the number 8 in the left figure, and then try finding 1,546,898,321. If it took you even a millisecond longer to find the longer number, then congratulations, you are no different than a computer. The choice of data types that you make may cause data to bloat, which in turn slows down data lookups (i.e., `SELECT`) and other operations.

Now that you know bytes matter, we can derive a simple guideline in choosing which string data type to use:



- When an attribute has data of uniform length, use **CHAR**.
- When an attribute has data of a relatively short but varied lengths, use **VARCHAR**. “Relatively short” here also implies that an upper limit can be easily estimated (e.g., 500 characters).
- When an attribute is expected to have a large amount of text data that regularly exceeds a few thousand characters (e.g., 1000+ characters), use **TEXT**. This is especially true if the upper limit is hard to estimate.

Some examples will be helpful to demonstrate this guideline. If you need an attribute to store the state/province as a part of the address information of your US and Canada customers, you can use a two-character designation (e.g., WA short for Washington State). **CHAR** will be the perfect data type for this attribute. For another example, if you need an attribute to store users’ names, you can expect its data to have varied length. Additionally, the upper limit of the names can also be estimated. Most names don’t need to go beyond 30 characters. As such, **VARCHAR** will be the perfect data type for this attribute. The comparison between these two examples is summarized below:

state_province	name
WA CA OR KY GA	Bob Dingleberry Jigglypuff Wackadoodle Fizzwitz

CHAR(2)                    VARCHAR(30)

In the above figure, you may notice **CHAR** and **VARCHAR** are followed by parentheses with numbers in them. This syntax deserves some explanation. When you use either **CHAR** or **VARCHAR** to define data types, you need to declare the length. **CHAR(2)** indicates that each data record of this attribute will

occupy exactly 2 bytes. `VARCHAR(30)` indicates that each data record of its attribute can occupy up to 30 characters.

As for the data type `TEXT`, it is typically used to hold a large amount of text data, such as product description, product manual, or book texts.



### **Strings and database performance**

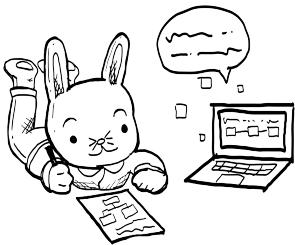
In some databases, `TEXT` may not be stored in-line along with the rest of the columns but instead, the data may exist at a secondary file storage, requiring further lookups and a performance delay. Use it sparingly and only when necessary (e.g., description fields).

Finally, text attributes are indexable through full-text indexes (specialized indexes for larger text that work like a search engine), which are not supported by all database systems (e.g., MySQL had no support for this in its InnoDB engine). In such cases, all searches on text attributes have to be linear and therefore slow. If the string data in an attribute is short, and you expect this attribute to be searched frequently, `VARCHAR` is the better choice for its data type.

### **Identify string attributes in our database**

Now that you have learned the basics of string data types, you are ready to identify the attributes in our database for which string data types are a good fit.

To complete this task, you will do the following for each entity:



1. Check the requirements and the sample data that you get from the requirements gathering phase.
2. Identify the attributes that should be strings.
3. Choose a string data type from among CHAR, VARCHAR, and TEXT, which you learned about in the last section, for the identified attributes.

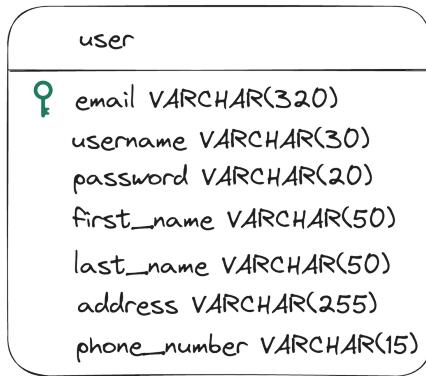
First, you will check the requirements and the sample data that you get from the requirements gathering phase. For example, our user entity has 7 attributes. The sample data for this entity we get is as follows:

username	email	password	first_name	last_name	phone_number
"dataDiva"	"diva@gmail.com"	"123456"	"Jonna"	"Diva"	
address		"Acorn Street 125, Seattle, WA"			"2602603555"
"keyKeeper"	"key128893@outlook.com"	"123qwerty"	"John"	"Keeper"	
					"Lemon tree avenue 300, Portland, OR"
					"3264564910"
"queryQueen"	"queen.query@yahoo.com"	"query**sd1"	"Berry"	"Sánchez"	
					"Amanda Ln 310, Portland, OR"
					"722347118"

Second, you will identify the attributes that should be strings. From the sample data, you can tell that all of them are strings. In other words, all 7 attributes are of String data types.

Third, you will choose a string data type for the identified attributes. From the sample data, you can tell that all attributes have variable lengths. Additionally, it is not difficult to estimate the max length for these variables. As such, VARCHAR would be the most appropriate data type. For example, a phone number cannot contain more than 15 digits (excluding formatting) no matter where you are, thanks to the international phone numbering plan. As such, the data type of phone\_number will be VARCHAR(15). You may wonder if storing the passwords as plain texts using VARCHAR would introduce security issue. Of course, it will! We will address security-related challenges in Chapter 7. For now, VARCHAR is good enough. When you identify all the data types of the attribute, you can put the data type by the right side of its

corresponding attribute to complete your design of the user entity:



In case you have difficulties in estimating a reasonable character limit for a VARCHAR attribute, you should always ask follow-up questions to the domain experts or developers.

For another example, the sample data for the product entity that we get is as follows:

name	description	manufacturer	photo
"Atomic Nose Hair Trimmer", "Trim your nose ...", "Mad Inventors Inc.", ..., "https://hosting.photobucket.com/images/n627/....jpg"			
"Selfie Toaster", "Get your face on ...", "Goofy Gadgets Corp", ..., "https://hosting.photobucket.com/images/n631/....jpg"			
"Cat-Poop Coffee", "The only coffee made", "Absurd Accessories", ..., "https://hosting.photobucket.com/images/n627/....jpg"			

From the sample data, you can tell that all the four attributes are strings. VARCHAR will be good enough for the name and manufacturer attributes. However, you can expect product descriptions and photo URLs be long strings, especially product descriptions. To be safe, we can assign TEXT to the description attribute. Considering that the photo URLs won't be unreasonably long, we can stick to VARCHAR, but declare a length that is long enough to accommodate URLs, such as 1000. The updated product entity will look as follows:

product	
♀	code
	name VARCHAR(100)
	description TEXT
	manufacturer VARCHAR(100)
	photo VARCHAR(1000)
	price
	cost
	inventory_quantity



### **Lost in translation: character sets**

In every database you design and implement, you need to make a big decision for the tables, columns, and data in it – you need to decide the character set for a database, and it's better to make that decision early than late.

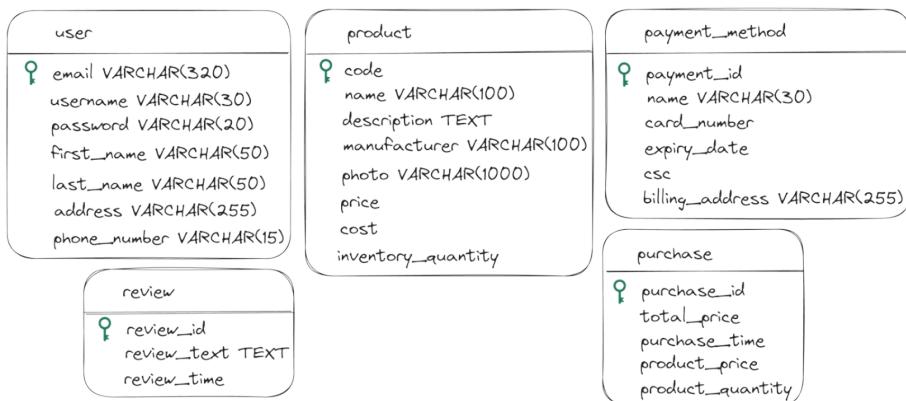
What is a character set? A character set refers to a set of rules that determines how characters are represented and stored in a database. You can think a character set as a set of guidelines that a database system follows to handle letters, numbers, symbols, and other characters from various languages and scripts.

Why do you need to decide on character set for a database? Think about languages around the world. Different languages use different letters and characters. For example, English uses the letters A, B, C, and so on, while Chinese uses characters that look very different. Different character sets support different languages. For example, there are character sets for Latin-based languages (e.g., Latin1), as well as character sets dealing with internationalization and multilingual content (e.g., UTF-8). When designing a

database, it's important to choose a character set that aligns with the languages and symbols you'll be working with to ensure data accuracy and compatibility.

The decision for character set is typically made when a database is created. Why it's better to decide on character set early than late? Some RDBMS may not allow you to change the character set. Even if it is possible, changing character sets can be complex and result in data loss or corruption.

What about the other entities? Feel free to pause reading and try out for yourself to identify and define their string attributes. If you are pressed for time, here are the declared string attributes in all entities:



#### 4.4.2 Integers: numbering your way to success

Before you can use integer data types fluently, you need to learn a little bit about the different integer data types.

#### TINYINT, SMALLINT, MEDIUMINT, INT, and BIGINT



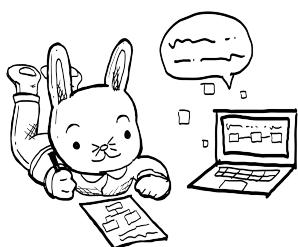
The most common types of integer data are TINYINT, SMALLINT, INT, and BIGINT. These data types allocate a fixed number of bytes based on a power of two and are 1, 2, 4, and 8 bytes, respectively. Additionally, these data types have different minimum and maximum values, which also vary depending on whether they are set up as signed or unsigned.

In some database systems, an unsigned number doesn't have a + or - sign associated with it. *In other words, an unsigned number cannot be negative. In contrast, a signed number can be negative.* The ranges of the four integer types, signed and unsigned, are as follows:

TINYINT	{	signed : -128 to 127
		unsigned : 0 to 255
SMALLINT	{	signed : -32,768 to 32,767
		unsigned : 0 to 65,535
INT	{	signed : -2,147,483,648 to 2,147,483,647 (~2 billion)
		unsigned : 0 to 4,294,967,295 (~4 billion)
BIGINT	{	signed : $-2^{63}$ to $2^{63} - 1$
		unsigned : 0 to $2^{64}$

As you can see, TINYINT can hold a maximum value of 255 as unsigned but only 127 as signed. INT can accommodate a maximum value as 4,294,967,295 as unsigned, which is big. However, if you have a numeric attribute whose data can be bigger than 4 billion in value, you will need the help from BIGINT.

### ***Identify integer attributes in our database***

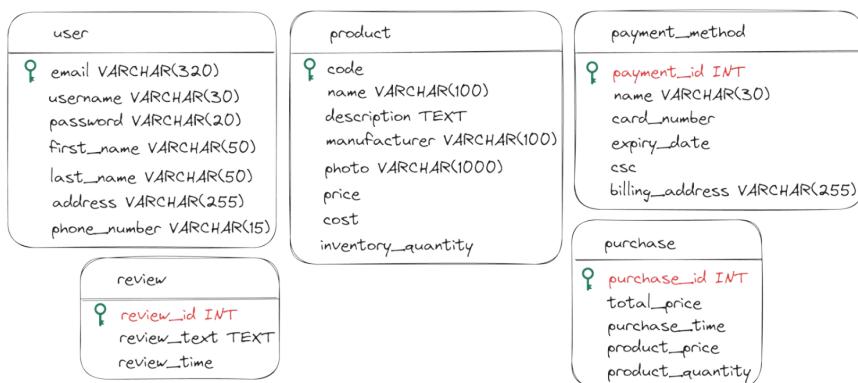


Now that you have learned the basics of integer data types, you are ready to

identify the attributes for which integer data types are a good fit in our database.

To complete this task, you will follow the same procedure as identifying string attributes. You will start by examining the requirements and sample data that you get from the requirements gathering phase, then move to identify the integer attributes, and choose an appropriate integer data type per attribute.

For example, you can focus your attention on the ID attributes. Numeric IDs are stable, unique, simple and efficient. *Following this logic, you can, for example, declare the payment\_id attribute as an integer attribute. Of course, you need to decide among the different types of integer data types. It is self-telling that TINYINT is too small and thus not a good fit. If we aim to build a super online store that sells products across the galaxy, SMALLINT is also too small, and thus not a good fit. BIGINT would work fine and is definitely big enough. However, we can comfortably settle with INT because INT is big enough for now and can yield better database performance in comparison (i.e., the fewer bytes are used, the faster the SELECT query will be). Although whether an integer attribute is signed or unsigned rarely gets reflected in an E-R diagram, it is still a choice you need to make. All primary keys have no reason to have negative values, thus unsigned.*

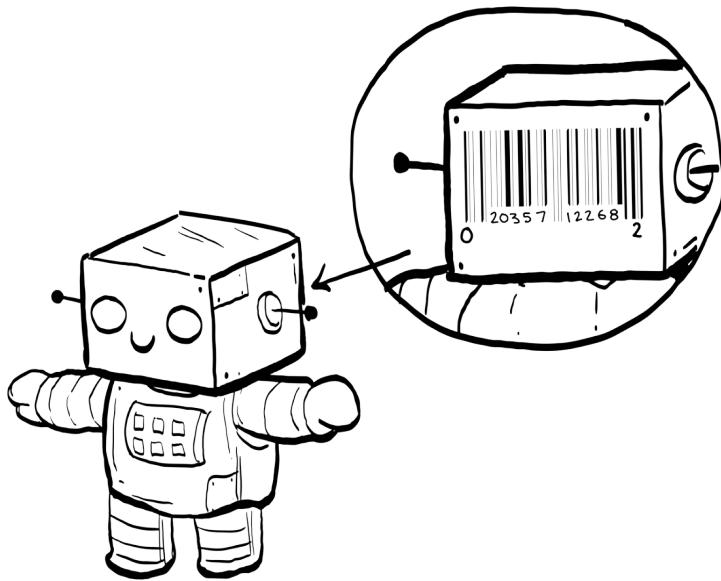


Most integer attributes in our database are easy to identify, but there are two tricky cases, the payment\_method entity and the code attribute in the product entity. Here is a quick look at the payment\_method entity in the requirements gathering phase:

<code>name</code>	<code>card_number</code>	<code>expiry_date</code>	<code>csc</code>
"master card",	5260405254103227,	0227,	777, ...
"boa visa",	3169596072241198,	1225,	557, ...
"amex",	1677288846463190,	1129,	2727, ...

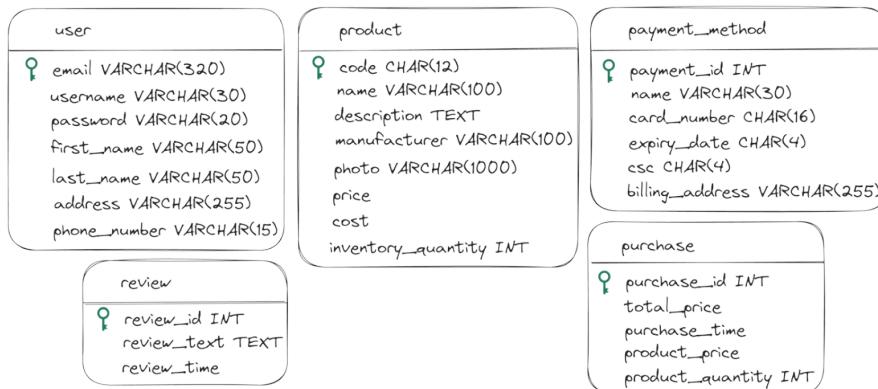
The values of `card_number`, `expiry_date` and `csc` all seem to be integers in the sample data. However, you are recommended to define them as strings instead of integers. This is one of those cases where you have to sacrifice storage efficiency for clarity, practicality, and correctness. Mathematical operations are not performed on information such as debit/credit card numbers. It is also problematic for these to be seen as a number since the industry has reserved leading zeros (0 – ISO/TC 68 and other future industry assignments). In numerical data types, the leading zeros are eliminated, and hence certain card numbers would not be stored correctly. As such, you may define the data type of card numbers as `CHAR(16)`, where 16 is the number of digits — debit/credit cards all have 16 digits. A similar problem is posed by `expiry_date` and `csc`. As such, you can opt to define them as `CHAR`. A `csc` is either a 3 or 4-digit number. You can define the data types of both `expiry_date` and `csc` as `CHAR(4)`. You may wonder whether it's safe to store the payment information, such as card numbers or CSC as plain texts? Nope. You will revisit the data types of these three attributes in Chapter 7 when the security of databases is examined. For now, plain-text payment information is good enough when you are still learning everything about data types.

Conquering the `payment_method` entity makes it easy to work on the `code` attribute in the `product` entity. The product code is based on the idea of The Universal Product Code (UPC), which is a unique 12-digit number assigned to each individual product sold in stores and online.



For the same reasons (e.g., leading zeros), it is more appropriate to define the data type of code as string rather than integer. Since all UPC values have the same length of 12 digits, we can define the code attribute as CHAR(12).

You can identify other integer attributes and define their data types in the same manner. The most important criterion to determine is the maximum practical value for a particular attribute, accounting for some growth. This will help keep your database small and efficient in the long run. Here's how the remaining integer attributes would look in all entities:



#### 4.4.3 Float like a butterfly, decimal like a data queen

You may notice that we did not assign an integer data type to the price attribute in the product entity, and wonder: what data types are more

appropriate for this and other similar attributes? Well, decimals. Before you can use decimal data types fluently, you need to learn a little bit about the different decimal data types.

## Floating-point vs. fixed-point data types



You have two choices when it comes to declaring a decimal attribute, *floating-point* types and *fixed-point* types. Common floating-point data types include `FLOAT` and `DOUBLE`. `FLOAT` is typically 32-bit; `DOUBLE` is typically 64-bit, which means that `DOUBLE` can accommodate bigger numbers. `FLOAT` and `DOUBLE` are used widely where accuracy is not strictly required, such as sensor reading, statistical analysis, and physical simulation. However, when accuracy is required to the extent that you care about every decimal value, they should be avoided. Why? Finite precision and rounding errors. Suppose that you have a database that keeps track of money exchanges through different currencies for its users (handling other people's money is fun).

$$\begin{array}{ccc} \$1,000,001 & \xrightarrow{\$1 \times 0.83€} & \begin{array}{l} \text{Actual: } 830000.83 \\ \text{32-bit Float: } 830000.8125 \\ \text{Error: } -0.0175 \end{array} \end{array}$$

If you wanted to convert 1,000,001 USD to EUR with the exchange rate as 0.83, the rounding errors under a 32-bit floating-point operation may result in 830000.8125. This result is about 1.75-euro cents less than what it should be. This may not seem like much, but users of such a system will lose money for no reason, especially the frequent users and those who transfer a huge amount of money.

Now you may know vaguely what we mean by accuracy, let's define accuracy formally before moving further. Accuracy means specific requirements for precision and scale. Precision refers to the total number of

significant digits, while scale refers to the number of digits that can be stored to the right of the decimal point. For example, consider the number 12.3456: its precision is 6 and its scale is 4.

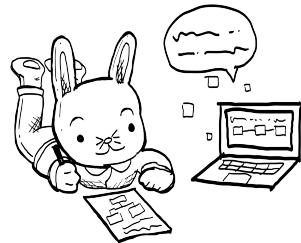
Scale Here: 4

12 . 3456

Precision Here: 6

When accuracy is required, you should think about fixed-point data types, such as `DECIMAL` or `NUMERIC`. There are no real differences between `DECIMAL` and `NUMERIC` in many RDBMS, such as MySQL and PostgreSQL. In such cases, you can use either. If you are not sure whether the RDBMS you use falls into such cases, you will need to refer to its documentation.

## Identify decimal attributes in our database



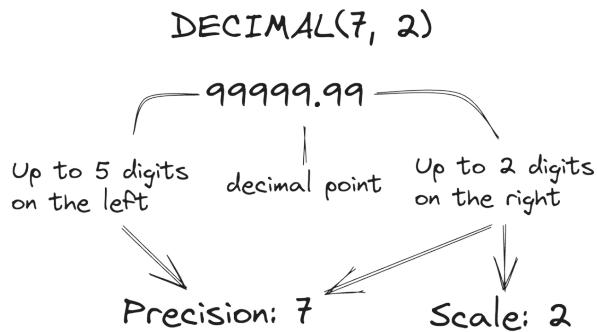
Now that you have learned the basics of decimal data types, you are ready to identify the decimal attributes in our database.

To complete this task, you will follow the same procedure as identifying string and integer attributes. For example, to identify the decimal attributes in the product table, we can take a look at the sample product data:

name	description	manufacturer	price	cost
"Atomic Nose Hair Trimmer", "Trim your nose ...", "Mad Inventors Inc."			19.99, 9.99, ...	
"Selfie Toaster", "Get your face on ...", "Goofy Gadgets Corp"			24.99, 15.02, ...	
"Cat-Poop Coffee", "The only coffee made", "Absurd Accessories"			29.99, 21.85, ...	

Did you see the values for the `price` and `cost` attributes? They represent

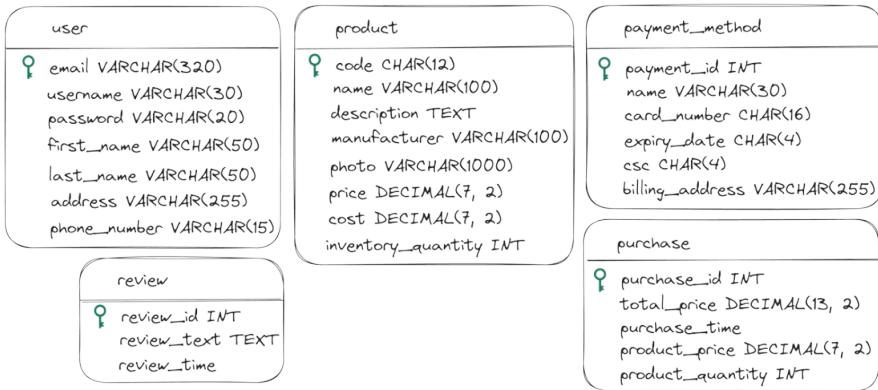
money. In other words, they require accuracy, and you should think about DECIMAL. For money, it makes sense for the scale to be 2 because that's the smallest unit of money (i.e., cents). In larger monetary operations, a higher scale may be necessary to accommodate for roundoff operations. All our products sell for less than one thousand dollars, so the precision can comfortably stay as 7:



Therefore, we will define the data type of both the price and cost attributes as DECIMAL(7, 2), where 7 specifies the precision and 2 specifies the scale. We can update our design of the product entity accordingly:

product	
code	CHAR(12)
name	VARCHAR(100)
description	TEXT
manufacturer	VARCHAR(100)
photo	VARCHAR(1000)
price	DECIMAL(7, 2)
cost	DECIMAL(7, 2)
inventory_quantity	INT

You can apply the same steps to identify decimal attributes in other tables, essentially any other attributes that are money related, such as the `product_price` and the `total_price` attributes in `purchase`. If you worry that a single transaction may lead to a large bill, you can increase the precision for the `total_price` attribute, such as `DECIMAL(13, 2)`. Here is an updated design of all the entities:



#### 4.4.4 Temporal Data Types: Time Flies When You're Querying

You may notice that we did not assign any data types to the `review_time` attribute in the `review` entity. You may wonder what data types are appropriate for this and other similar attributes? Well, date or time, which belong to the last group of data types you learn in this chapter – temporal data types. Before you can use temporal data types fluently, you need to learn a little bit about them.

#### DATE, TIME, DATETIME, and TIMESTAMP

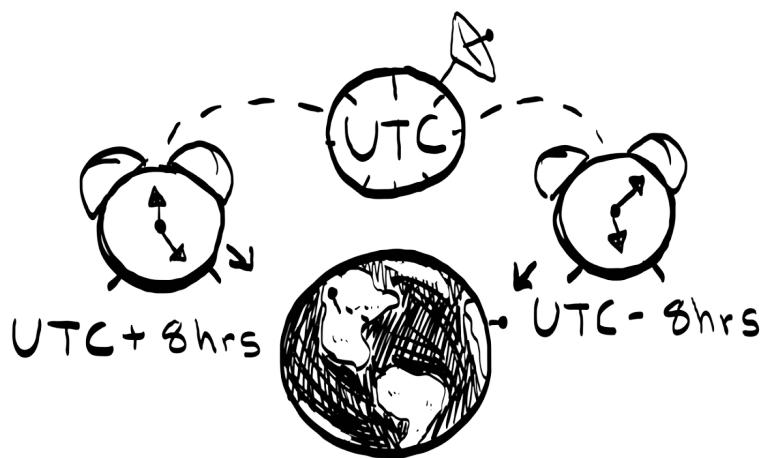
- Use `DATE` when only a date needs to be stored without any time information, such as the birthday data. For example, the date '1980-05-15' represents May 15, 1980.
- Use `TIME` when only time needs to be stored. For example, if you want to track the time at which events occur during the day, you can use the `TIME` data type. For example, the time '14:30:00' represents 2:30 PM.
- Use `DATETIME` when both date and time information need to be stored to represent historical time, future events, or other instances where the time zone is an inherent part of the time and should not be adjusted.
- Use `TIMESTAMP` to record the exact moment of an event that needs to be consistent across time zones, such as log timestamps, record creation, or modification times. The conversion of time zones of `TIMESTAMP` data is typically automatically handled by RDBMS.

If you are designing a database for an application whose users are strictly local, such as an ordering system of a restaurant, you don't need to worry

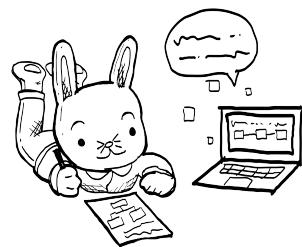
about time zone issues.

However, if you are building an application that might be used across the globe, you may wonder whether the time zone data will be stored as a part of the temporal data attributes. Regardless of the specifics of the RDBMS, you should consider storing date and time values in Coordinated Universal Time (UTC) because it ensures consistency and avoids issues such as daylight-saving time changes, different time zones, or traveling users.

What is UTC? UTC is like a big clock that everyone agrees to follow. It's the same time for everyone, no matter where they are. For example, if you live in Seattle, and your friends in Beijing want to let you know their local time without giving you the exact time explicitly, they can say, "Our local time is UTC plus 8 hours." You can not only calculate their local time, but also compare their time with yours because you know how your local time compares to UTC. That said, any web or mobile applications can calculate the local time given a UTC time.



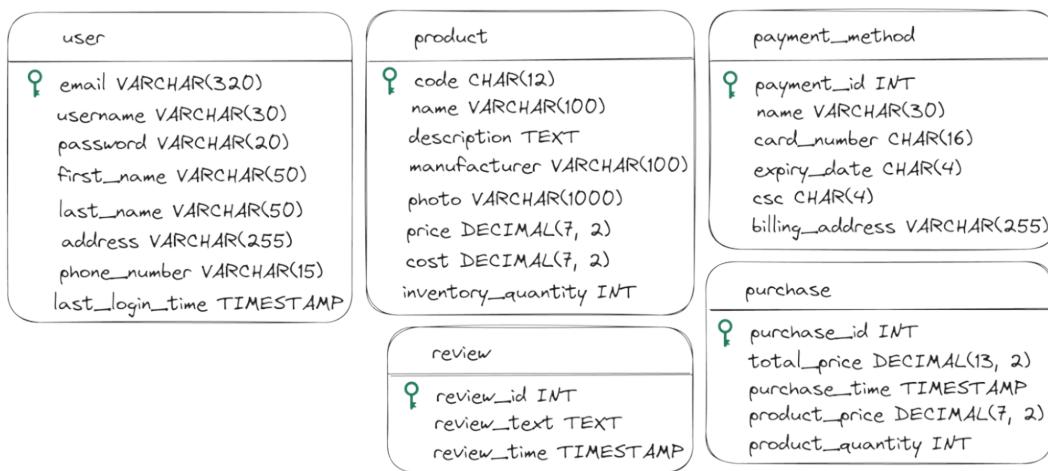
### **Identify attributes representing date and time in our database**



Now that you have learned the basics of temporal data types, you are ready to identify the attributes that represent date and time in our database.

To complete this task, you will take the same steps as declaring other data types in prior sections. You will start by referring to some sample data you collected in the requirement gathering phase. However, *in this case, you may find the task trivial because our attributes related to date and time have self-revealing names (i.e., “date” is a part of the name)*. There are two attributes that are related to date and time: the review\_time attribute in review, and purchase\_time attribute in purchase. From the requirements gathering phase, you learned that both the two attributes require date as well as time information. Both attributes represent exact moments of some events that need to be consistent across time zones. Thus, you can declare the data type of both attributes as TIMESTAMP.

Additionally, based on the requirements gathering, you learned that you need to track what the last time when a user accessed the online store by adding another TIMESTAMP attribute to the user entity, and named it last\_login\_time. *With all these updates, here is another version of the updated entities:*



So far you have successfully identified the data types of all the attributes. Take another look at your masterpiece of all entities. You did a great job!



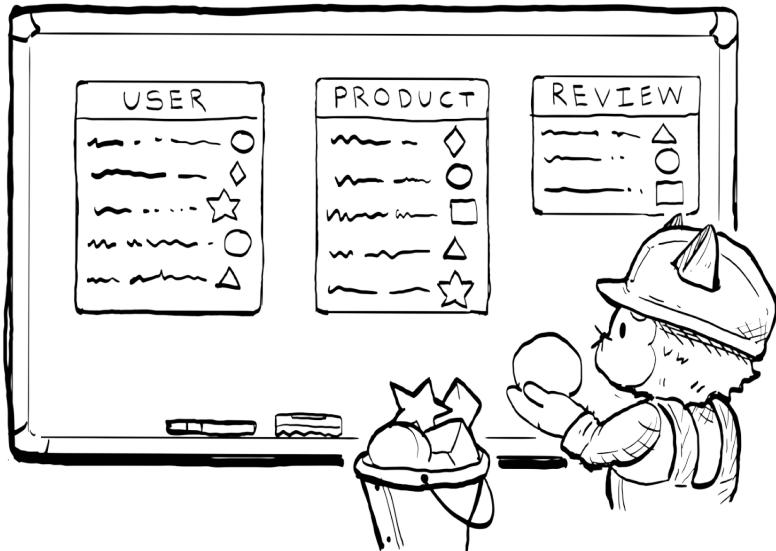
### **inconsistencies of data types across RDBMS**

Like it or not, there are inconsistencies of data types across different RDBMS. The data types that you have learned so far are common across most RDBMS. However, there are exceptions and particularities that you need to learn about each individual RDBMS. The entities and attributes you developed above are perfect for RDBMS such as MySQL, MariaDB and PostgreSQL. However, you will need to make small adjustment for other RDBMS:

- It is common to use `TEXT` in place of `TIMESTAMP` in SQLite.
- Oracle uses `NUMBER` in place of `INT` and `DECIMAL`, and `VARCHAR2` in place of `TEXT`.
- SQL Server uses `NVARCHAR(MAX)` in place of `TEXT`.

You can find such information in the GitHub repository that accompanies this book.

## **4.5 Summary**



- Design of an entity requires identification of all its attributes, the primary key, and data types of all its attributes.
- If there are no candidate keys available, or none of the candidate keys are a good fit for the primary key, you can always create a numeric attribute and use it as the primary key.
- Some common data types in databases include string types, integer types, decimal types, and temporal types. When it comes to deciding which data type to use, you need to think about what job a data type is good for as well as the demand of your particular context.
- Given an attribute, to identify its data type, you need to examine the sample data and information from the requirements gathering phase, identify the proper data type group (e.g., string), and choose the best fit within that data type group (e.g., TEXT) based on the maximum allowed values derived from your requirements analysis.
- The use of proper data types ensures that a database stores data efficiently as well as meets the demands of data querying.

# 5 Relationships

## In this chapter

- You establish relationships between entities.
- You identify the cardinality of each relationship.
- You make decisions on whether to represent some entities as weak entities.

### What you need to know

You can find the database design covered in this chapter (so far it's entities plus relationships) implemented in tools commonly used by practitioners, such as dbdiagram.io, and MySQL Workbench in the GitHub repository (<https://github.com/Neo-Hao/grokking-relational-database-design>). You can navigate to the chapter\_05 folder, and follow the instructions in README.md to load the database design into corresponding tools.

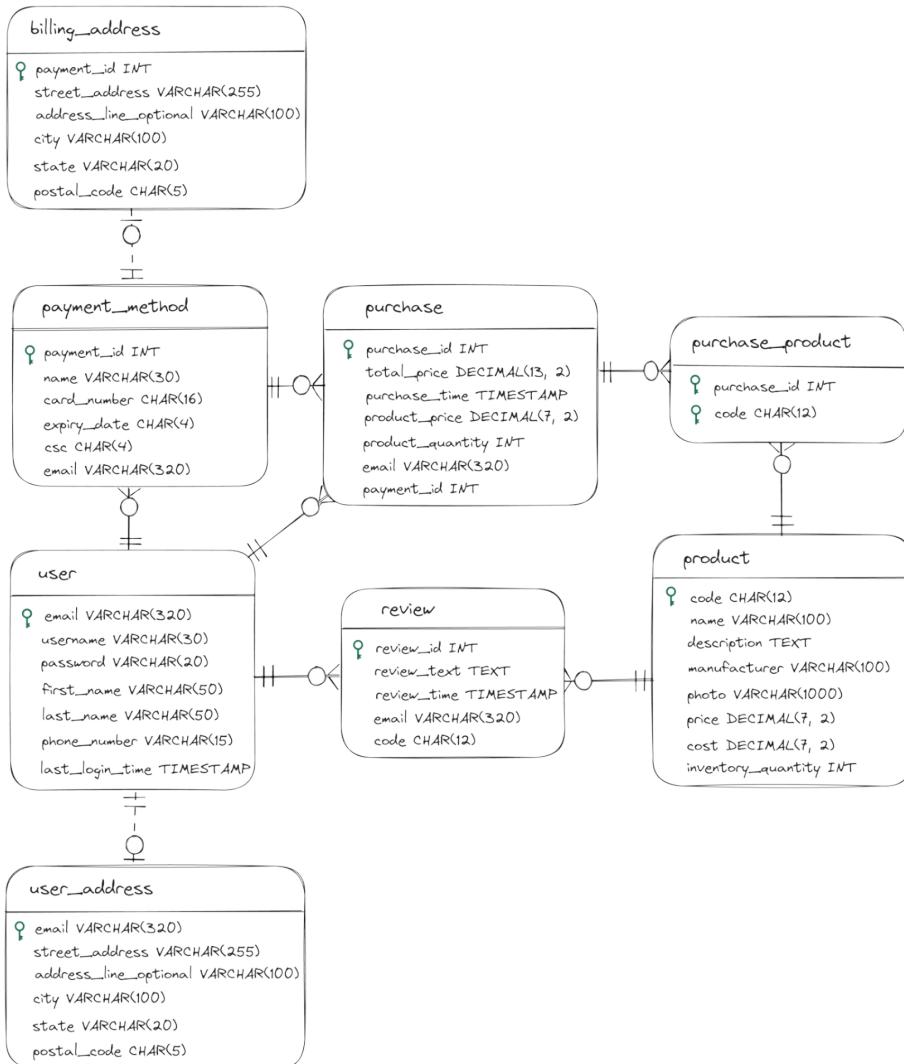
The data types that show up in this chapter are applicable to most RDBMS, such as MySQL, MariaDB, and PostgreSQL. If you are using another RDBMS such as SQL Server or SQLite, you may need to make small modifications to the design by replacing certain data types with their equivalences specific to an RDBMS. Such information can be found in README.md in the chapter\_05 folder.

## 5.1 Overview

In this chapter, you will develop your entity-relationship (E-R) diagram for The Sci-Fi Collective online store by establishing relationships among entities you identified in Chapter 4. By doing so, you will learn about important concepts in database design, such as cardinality and dependency.

## 5.2 Entity-relationship models

E-R diagrams are a graphical representation of entities and their relationships in a database. E-R diagrams are not only a visual tool, but also documentations that describe structures and rules of databases. Over the years, E-R diagrams have become a universal design language used by database designers. An E-R diagram is typically composed of boxes representing the entities, and lines representing the relationships among the entities. An E-R diagrams depicts the data structure (also known as data schema) but not the data. A table with millions of records will still be represented as a single entity. For example, the E-R diagram that you will develop by the end of this chapter looks as follows:



This diagram uses the Information Engineering notation, commonly known as Crow's Foot notation; there are other notation types which we will briefly

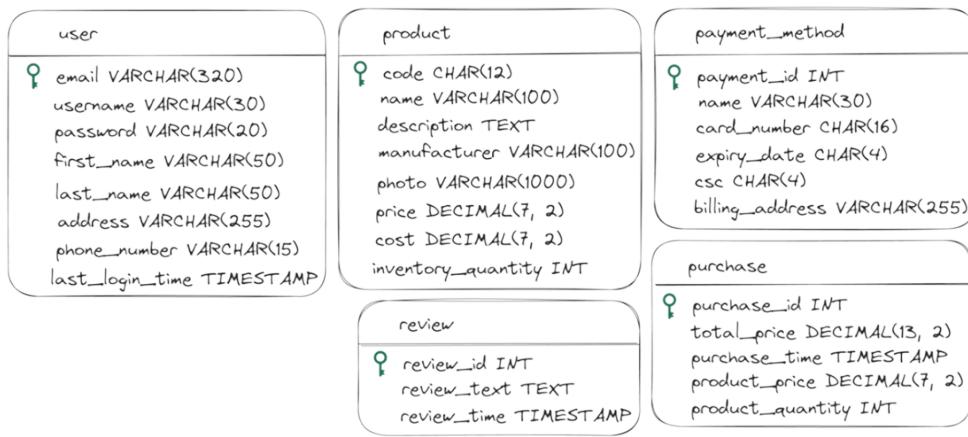
mention later on. If you are working on designing a database of a sophisticated application, you may need multiple E-R diagrams to fully describe its structure. For The Sci-Fi Collective online store, one E-R diagram is good enough.

Now that we have introduced the concept of an E-R diagram, let's start developing the E-R diagram for The Sci-Fi Collective online store. We are not starting from scratch, because we have already designed a set of entities in Chapter 4. However, we haven't represented the relationships of those entities yet. In the next few sections, you will learn more about the entity relationships and take on the task to establish relationships among entities.

## 5.3 Connecting related entities

In an E-R diagram, you use a line to connect every pair of two entities that are related to each other. Each line indicates a relationship between two entities. You will establish relationships between every two entities by (1) identifying every relationship, and (2) connecting every pair of related entities using lines.

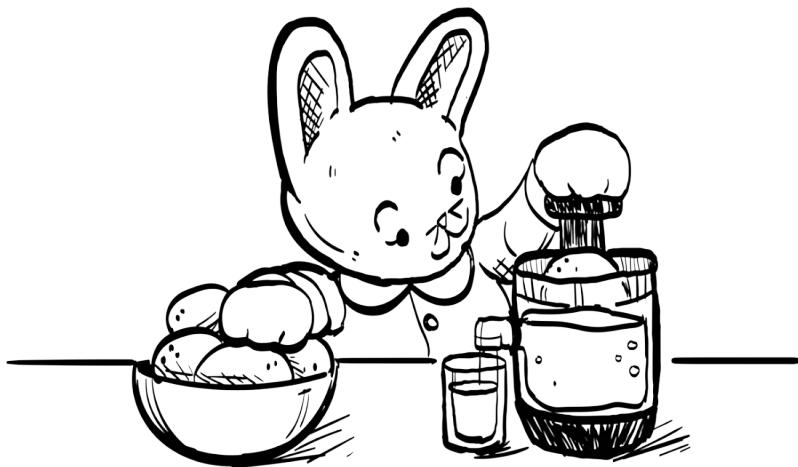
It is a great time to refresh your memory on what we have achieved so far. In Chapter 4, we established all the entities as follows:



In Chapter 3, we went over the requirement gathering phase, and collected useful information that pertains to the relationships among the above entities. We are pasting these here again adapted to reflect the current state of the ER diagram:

- A **user** can **make** multiple **purchases**. An order can be made by only one user.
- A **user** can **review** multiple **products** as long as the user bought those products. A product can be reviewed by multiple users.
- A **user** can **maintain** multiple **payment methods**. A payment method can be associated with only one user.
- A **purchase** can **have** more than one **product**. A product can show up in multiple orders.

Your first step to connect entities is to synthesize the information you gathered and the design of entities. Your goal is to generate a list of simple sentences only composed of subjects, verbs, and objects.

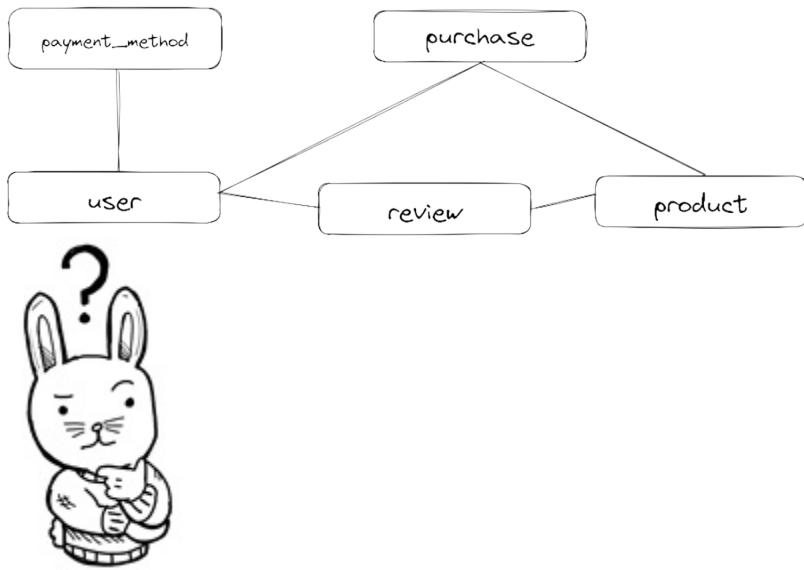


The sentences will help you understand better the relationships between every pair of entities. As you designed entities, you might have introduced changes that conflict with the gathered information. For example, you designed a **review** entity, which didn't appear as a noun in the information you gathered. Reconciling such conflicts, you will generate the following sentences:

- A user makes purchases.
- A user writes reviews.
- A product has reviews.
- A user maintains payment methods.
- A purchase contains products.

Next, you can map the above summary to a diagram. The nouns in every

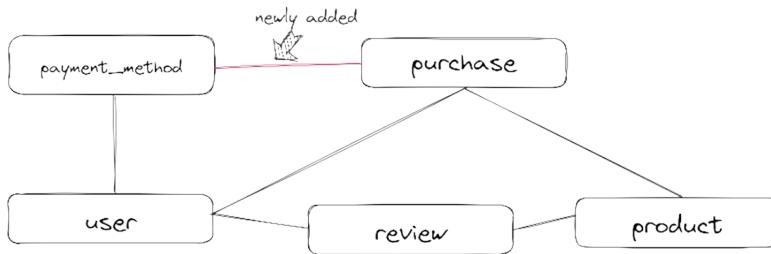
sentence represent entities. If two nouns connected by some verbs show up in one sentence, it is likely that the two entities are related. You may go through a few iterations of trial and error when mapping the summary to a diagram because of possible inaccuracy and misinterpretation. At this step, when you draw an entity in your draft E-R diagram, you may skip the attributes for now because attributes don't matter yet and it is tedious to list all of them. Based on the above summary, you will develop the following draft diagram:



When you generate a draft diagram, you should test every relationship against the information you gathered and the sample data you collected. You may also bring it to the stakeholders and explain your design rationale to them. Why? It is likely that you made some mistakes or neglected something critical in your first few iterations. For example, the software developers of The Sci-Fi Collective will point out that an online purchase can't be performed without a payment method. Based on the new information, you need to answer the following question before revising the draft diagram:

- Should `payment_method` be *related to* `purchase`?

Without the payment method information, an online order can't be finished. Otherwise, the online store can't bill its users. In other words, each purchase record needs to be mapped to a corresponding payment method. As such, a relationship between `payment_method` and `purchase` makes sense. With this question answered, you will add one more relationship:



In the next two sections, you will learn more about the characteristics of a relationship between a pair of two entities, which will empower you to further develop the draft diagram.

## 5.4 Cardinality

Cardinality is an important characteristic of a relationship between two entities. What is cardinality? Cardinality describes the number of instances of one entity that can be associated with a single instance of another entity via the relationship. Based on that, cardinality is classified into several types, and each type is represented differently in an E-R diagram.

To complete our E-R diagram, you will need to analyze the information you collected from the requirement gathering phase, identify the cardinality of each relationship, and update the draft diagram accordingly. In the following subsections, you will learn about more about cardinality and apply what you learn to update your draft E-R diagram.

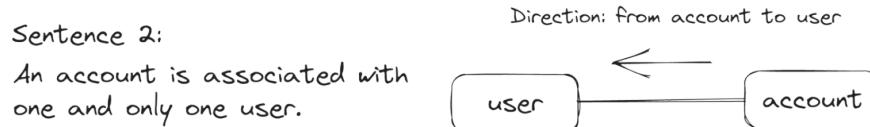
### 5.4.1 Direction and representation

To gain a deep understanding of cardinality, you will need to understand relationships better first. If directions are considered, a relationship between two entities can be broken down to two directional relationships. For example, in a banking system, user and account are two entities, and their relationship(s) can be summarized using two sentences:

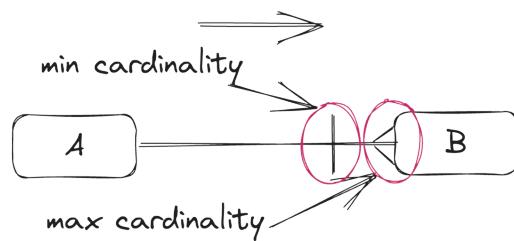
- Sentence 1: A user has zero, one or more accounts.
- Sentence 2: An account is associated with one and only one user.

Sentences 1 and 2 represent two different directional relationships between

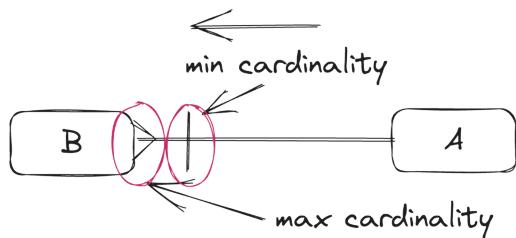
user and account. In both sentences, the direction flows from the subject to the object.



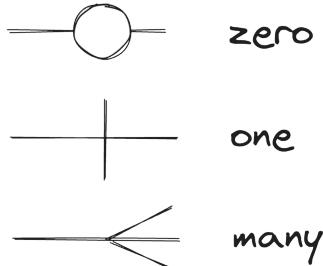
Given a directional relationship from A to B, cardinality describes the number of instances of B that a single instance of A can be associated with. Cardinality is represented using two graphic symbols on the relationship line between A and B. The symbols are used in pairs to represent the cardinalities. The symbol that is in the inner side represents the minimum instance number of B that a single instance of A needs to be associated with, or say the min cardinality. The symbol in the outer side represents the maximum instance number of B that a single instance of A needs to be associated with, or say the max cardinality.



Same as:

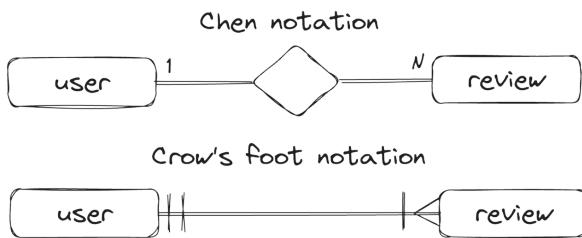


As you can see from the above figure, to represent the cardinality of the relationship from A to B, we would place the two symbols closer to B on the relationship line. What do the individual symbols mean exactly in this example? Well, they represent quantities. There are three symbols that you need to know:



### Cardinality Notation: Complicated Relationships with Math Symbols

It was not for a lack of trying but for whatever reason, there are several ways to notate the cardinality of a relationship. The two most popular ones are ‘s notation and Crow’s foot notation. Chen notation was born earlier and has historic significance. Crow’s foot notation is simpler and more popular with professionals.

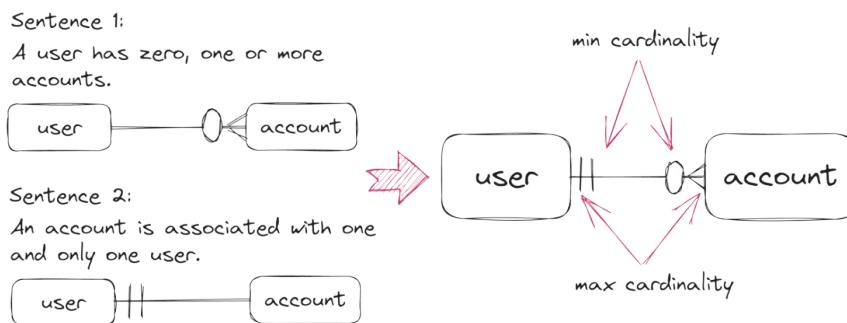


In this book, we will stick to Crow’s foot notation for its simplicity and wider popularity.

Now let’s see how to represent the cardinality of the example that you studied in the beginning of this section – the relationship between the user and account entities in a banking system:

- Sentence 1: A user has zero, one or more accounts.
- Sentence 2: An account is associated with one and only one user.

The two sentences represent two directional relationships, and they also contain the information that you need to establish their cardinalities. For instance, in the banking system, a user is associated with at least one account, which means that the min cardinality is one, and the max cardinality is unlimited, or say many, for the relationship from user to account. Similarly, an account is associated with one and only one user, which means that both the max and min cardinalities are one for the relationship from account to user.

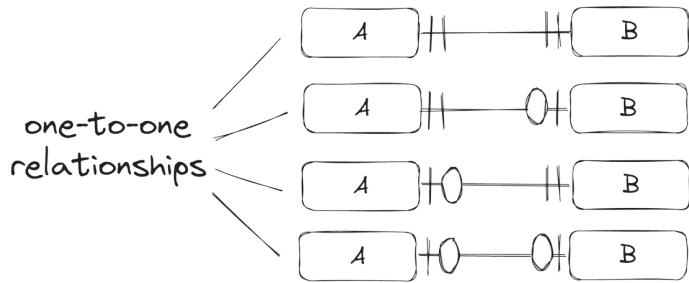


As you can see from the above figure, you can merge the two directional relationships, and use a single line to represent both. The cardinality symbols closer to account represent the cardinality of the relationship from user to account, whereas the symbols closer to user represent the cardinality of the relationship from account to user.

Now you know what cardinality is and how to represent it in E-R diagrams, you will move to learn about three common cardinality types, and apply what you learn to further develop the draft E-R diagram.

### 5.4.2 One-to-one: A perfect match

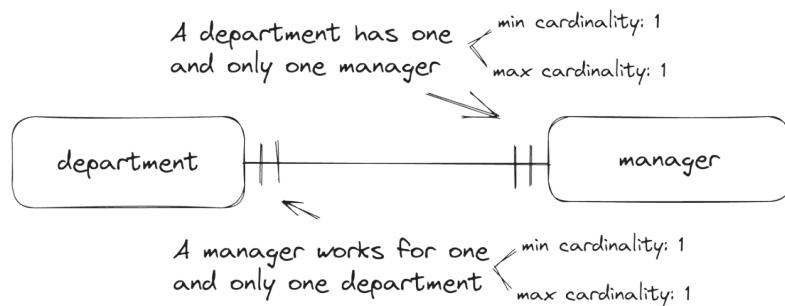
In a one-to-one relationship, each record in one entity is related to up to one record in the other entity. The “one-to-one” primarily refers to the max cardinality of both the involved directional relationships. The min cardinalities could be either zero or one.



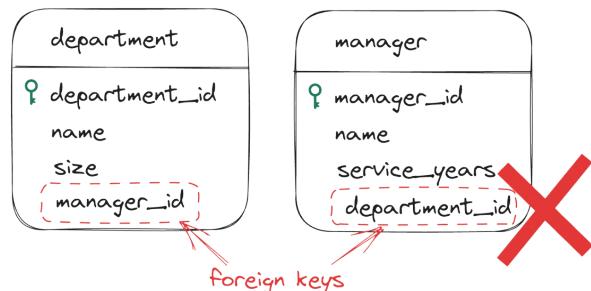
Given a one-to-one relationship, if both of the two min cardinalities are ones, one of the min cardinalities is typically converted to zero for easy implementation. For example, in the database of a large corporation, both departments and managers are represented, and their relationship is as follows:

- A department has one and only one manager.
- A manager works for one and only one department.

You can represent such a relationship as follows in an E-R diagram:



The above representation is theoretically solid but impossible to implement. To link the two entities, you need the help from foreign keys. If the above two entities have attributes, you need to place foreign keys in both tables, because each department is associated with a manager and each manager has to be associated with a department:



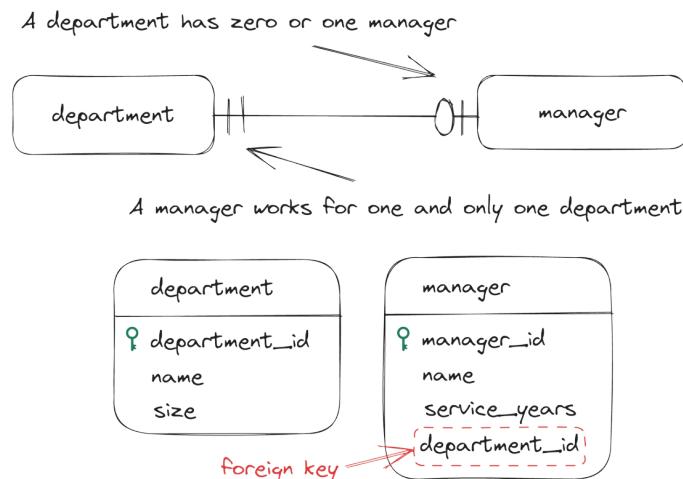


Such an implementation is problematic when it comes to data entry or insertion. When the two tables are created, they are blank. You will need to start populating the tables by entering the department information, or manager information one by one. If you enter a department record first, its manager data won't be available yet. The foreign key constraints you put in place will cause SQL to complain and reject the data entry. If you switch the order to enter manager information first, you will face the same problem.

In such a situation, you can relax one of the two min cardinalities from one to zero for easy implementation. For example, you can modify the relationship between department and manager to the follows:

- A department has zero or one manager.
- A manager works for one and only one department.

Their representation and implementation will be updated accordingly:

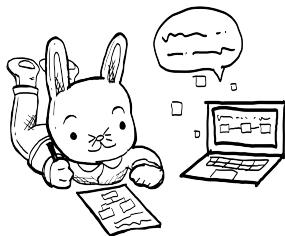


As you can see from the above figure, the min cardinality can be indicative of

where to place the foreign key in a one-to-one relationship. Given a directional relationship from table A to table B, if the minimum cardinality is 0, it means that not every instance of table A must have a related record in table B. In this case, you typically place the foreign key in the table B, which allows for the possibility of table A records having no corresponding records in table B. This change will make it possible for data entries. In our case, the table A is department, and table B is manager.

Now you know what one-to-one relationships are, it's time for you to contribute to our E-R diagram.

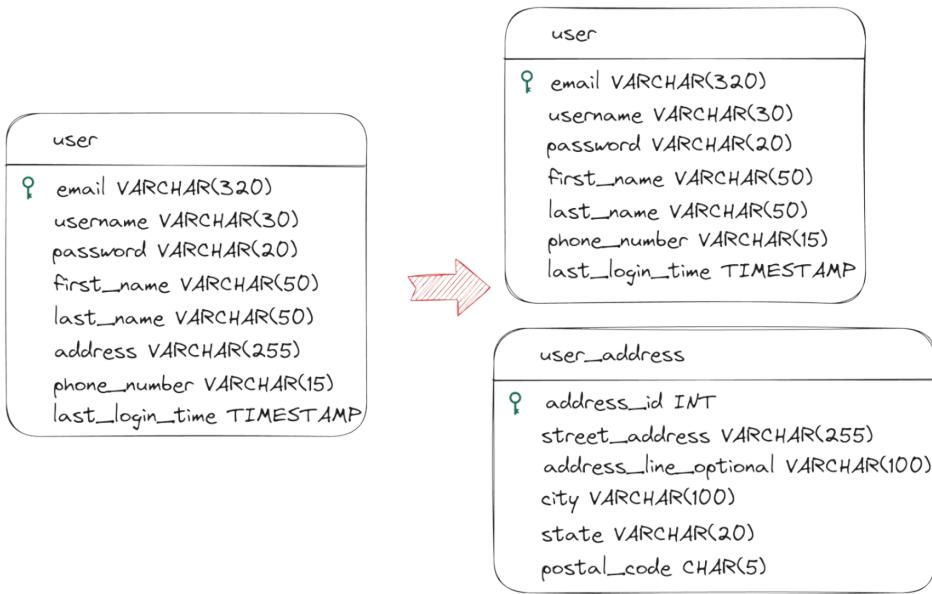
### **Identify one-to-one relationships in your database**



Based on the information you gathered from the requirement gathering phase, you don't have a one-to-one relationship between the entities that we have identified. However, some new information about users' addresses that you learned in the iterative process propels you to redesign your user entity. What is the issue? A user's address may look as follows:

20 Baldwin Rd, Shelter Island, New York, 11964.

The data analyst team needs to filter users' data based on city, state/province, or zip code to conduct targeted marketing analysis from time to time. Storing users' address information into a VARCHAR attribute will make such a task very difficult. To address this issue, you will factor the address information into a different entity with multiple attributes:



You will choose the data types on the basis of each attribute's domain (i.e., their maximum potential and types of values). To simplify the problem, you will assume that The Sci-Fi Collective only operates within the United States. With this assumption, it is not hard to find that “North Caroline” or “South Caroline” are among the longest state names (14 characters each). To be safe and accommodate any possible future changes, you can set the VARCHAR length a bit longer as 20. You may decide to use the 5-digit version for the post code. Since there is no variance in its length, you can define it as CHAR.

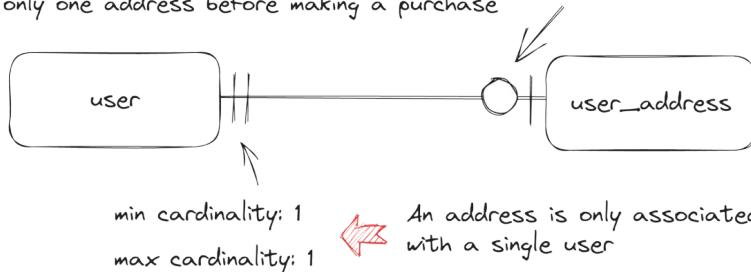
What is the relationship between `user` and `user_address`? The new information you gathered from the developer of the Sci-Fi Collective online store is as follows:

- A user may not have an address when he or she first registers an account, but a user has to have one and only one address before making a purchase
- An address is only associated with a single user

Based on such information, you can easily determine the min and max cardinalities of the two directional relationships (e.g., from `user` to `user_address`, and from `user_address` to `user`), and represent the merged relationship accordingly:

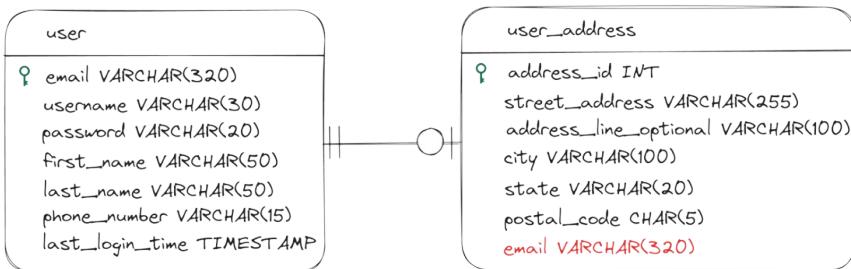
A user may not have an address when ... registers an account, but a user has to have one and only one address before making a purchase

min cardinality: 0  
max cardinality: 1



Is this a one-to-one relationship? Yes. You may notice that this representation is slightly different from the example of departments and managers we worked on. The difference lies in the min cardinality. It is possible for either or both min cardinalities to be zero(s) in a one-to-one relationship. In both examples, the max cardinalities are ones. *Indeed*, the name of “one-to-one” relationships only refers to the max cardinalities.

The representation of this relationship between user and user\_address requires the help of a foreign key. As you learned in the last subsection, the min cardinality can be indicative where the foreign key should be placed. In the directional relationship from user to user\_address, the min cardinality is zero. Thus, you will place the foreign key in the user\_address entity:

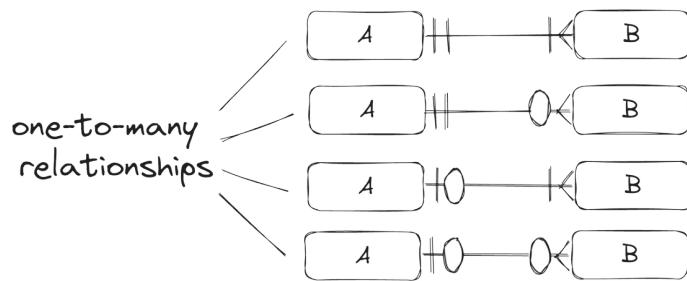


Now you know what a one-to-one relationship is and how to represent it, you

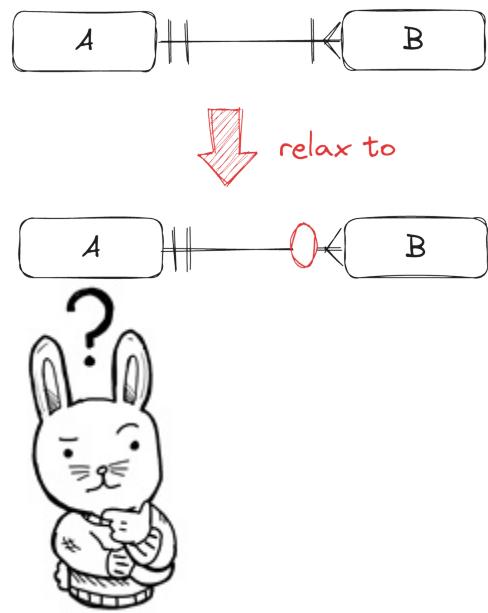
should also know that one-to-one relationships are the rarest relationship type, simply because very few one-to-one relationships exist in real life.

### 5.4.3 One-to-many: A love triangle

One-to-many relationships are the most common cardinality type. Same as one-to-one relationships, “one-to-many” primarily refers to the max cardinalities – if two entities are related, one directional relationship has the max cardinality of “one”, while the other has the max cardinality as “many”.

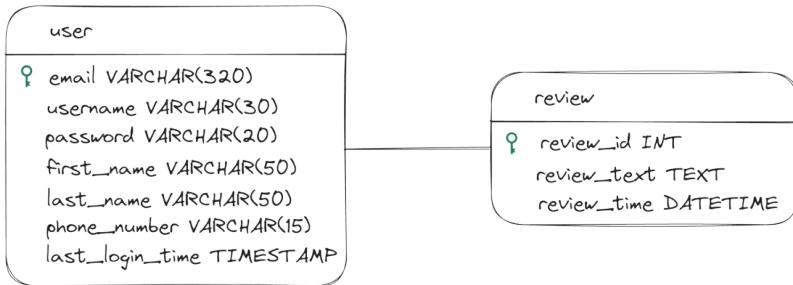


Like what you saw in the last section, in a one-to-many relationship, if both min cardinalities are ones, it will be problematic for data entry and insertion when the representation is implemented. In such a case, it is typical to relax the min cardinalities of the “many” side from one to zero.



What does a one-to-many relationship example look like? Let's look at the

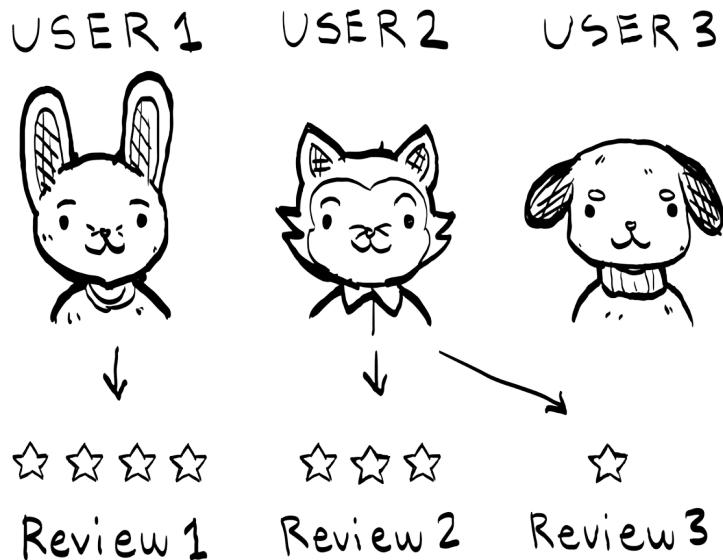
user and review entities in our draft diagram.



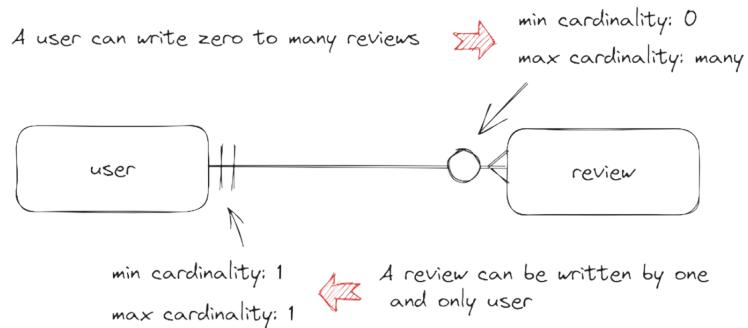
From the requirement gathering and follow-up inquiry on the relationship between the two entities, you managed to put together the following information about their relationship:

- A user can write zero to many reviews.
- A review can be written by one and only user.

You can visualize this relationship as follows:



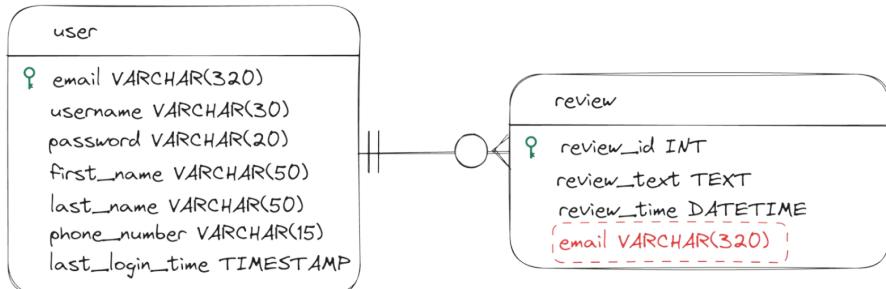
Based on such information, you can determine the min and max cardinalities of the two directional relationships (e.g., from user to review, and from review to user), and represent the merged relationship accordingly:



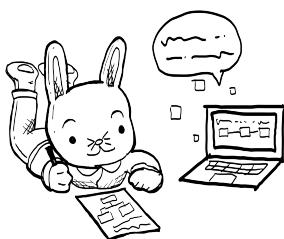
Categorizing relationships by cardinality has implications on where to place the foreign key, which is especially true for one-to-many relationships. In a one-to-many relationship, the foreign key needs to be placed in the many side (e.g., the entity pointed to by the crow's foot  $\nwarrow$ ).

Why should the foreign key be placed in the “many” side in a one-to-many relationship? The “many” side may have multiple rows that correspond to one row in the “one” side via the foreign key. If we insist putting the foreign key in the “one” side, it will make it impossible to query or update data, let alone maintaining the referential integrity.

In our case, the “many” side is the `review` entity. The foreign key in the `review` entity should be the primary key of the `user` entity. We will update the `review` entity accordingly:



## Identify one-to-many relationships in our database



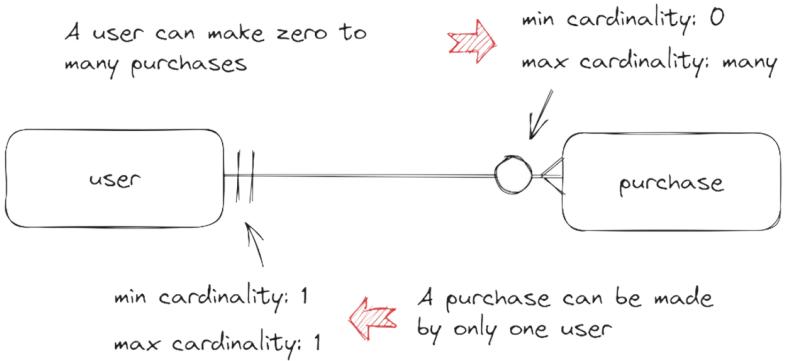
Now you know what one-to-many relationships are, and you have also updated the relationship between user and review entities in our draft diagram. You can apply the newly acquired knowledge and identify other one-to-many relationships, and update them accordingly in our draft diagram.

Let's start by reviewing the information we gathered on other relationships first:

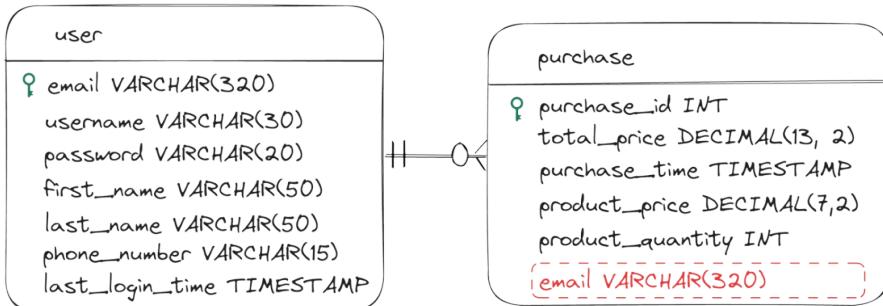
- A user can make multiple purchases. A purchase can be made by only one user.
- A user can maintain multiple payment methods. A payment method can be associated with only one user.
- [new] A payment method can be associated with multiple purchases. A purchase is associated with one payment method.
- A purchase can have more than one product. A product can show up in multiple purchases.

You can easily determine that the product and purchase relationship shouldn't fall into the one-to-many basket because if you examine its two directional relationships, neither has the max cardinality as one.

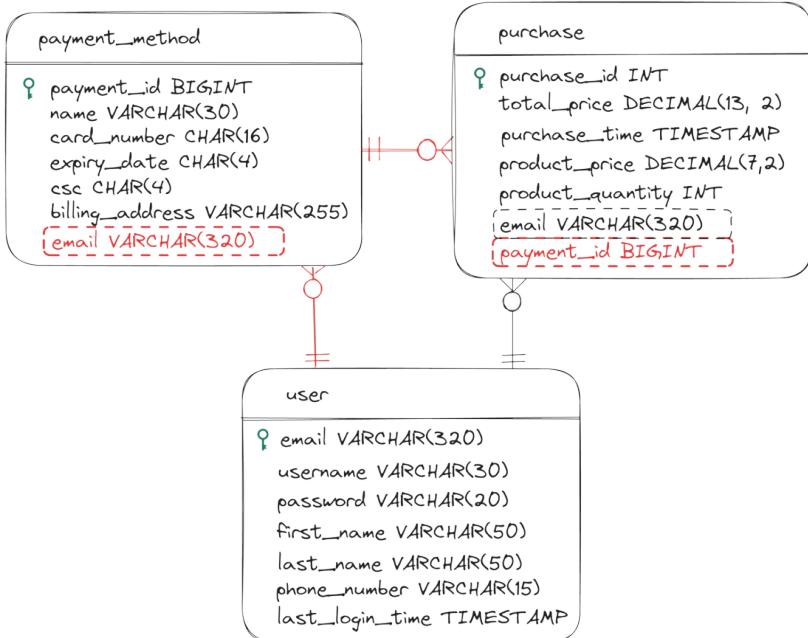
Similarly, it's not difficult to identify the user and purchase relationship as a one-to-many because one of its two directional relationships has the max cardinality as one (from purchase to user), while the other has it as many (from user to purchase). However, there is some ambiguity about the min cardinality from user to purchase: a user can make multiple transactions. After following up with the stakeholders, you will learn that the relationship is optional –think about when you browse on an online store for an hour without buying anything. In other words, this min cardinality is zero. You will update the user and purchase relationship accordingly:



Next, you will need to insert a foreign key in the “many” side to fully establish this relationship:



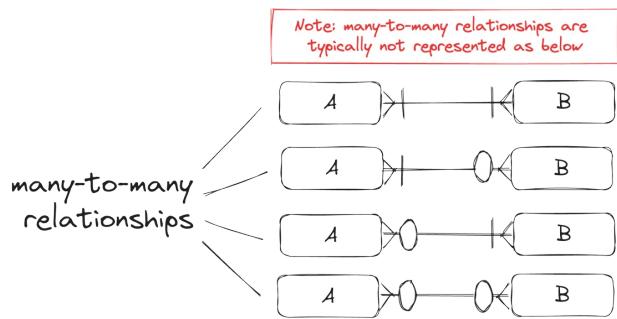
The relationships between user and payment\_method, and between purchase and payment\_method are both one-to-many. You can follow the same steps to update them accordingly:



As you can see, you have two sets of relationships going from payment\_method to user, one is through the purchase entity while the other is direct. This representation can lead to problems, which we will cover in Chapter 6.

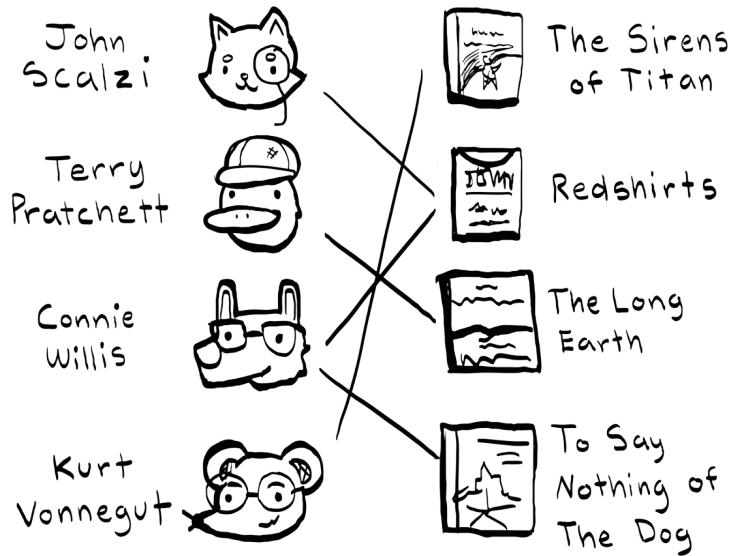
#### 5.4.4 Many to many: The more the merrier

The last cardinality type you will learn is the many-to-many relationship. Same as the other two types of relationships, “many-to-many” primarily refers to the max cardinalities – if two entities are related, the max cardinalities of both the involved directional relationships are “many”.

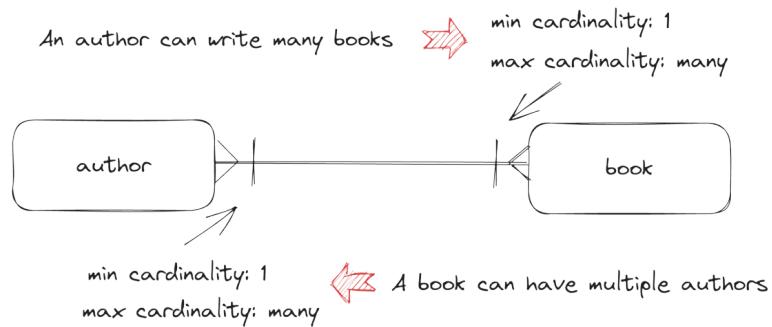


The above figure may help you grasp the gist of many-to-many relationships, but many-to-many relationships are typically represented differently. How different? In general, a many-to-many relationship is represented as two one-to-many relationships with the help from a junction table. Let's demonstrate this via an example.

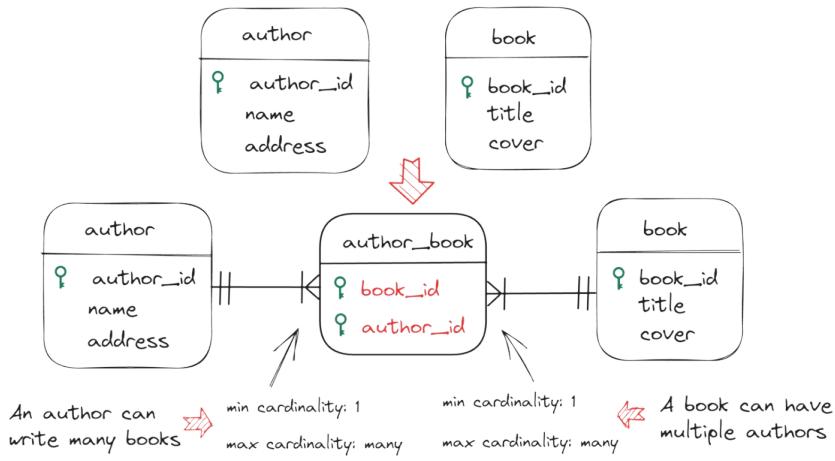
If we were to design a database for a bookstore application, we need to store all the information of authors and books in two separate tables. We all know that an author can write many books, and a book can have multiple authors:



You will know that this relationship is many-to-many as soon as you identify the max cardinalities of the two directional relationships:



To properly represent this relationship, you need to take one step further by making a junction table that contains only the IDs from the two involved entities, author and book. Additionally, you will convert the many-to-many relationship to two one-to-many relationships between the junction table and both entities. Suppose both the author and book entities have an ID and two other attributes. You can represent this many-to-many relationship as follows:



As you can see, the junction table, **author\_book**, only contains the primary keys from both **author** and **book** entities. In this junction table, the **book\_id** and **author\_id** attributes together serve as a *composite primary key*. Additionally, you need to convert the many-to-many relationship to two one-to-many relationships, in which the crow's feet always point to the junction table. The **author** and **book** entities will become indirectly related via the junction table.



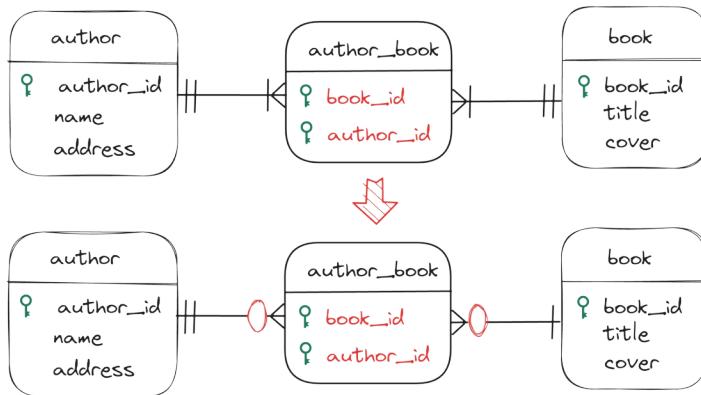
### Composite primary keys: The ultimate combo in databases

A composite primary key is a type of primary key in a relational database that consists of two or more attributes that, together, uniquely identify each row in a table. Unlike many primary keys that are based on single columns, a composite primary key requires a combination of multiple columns to form a unique identifier for each record.

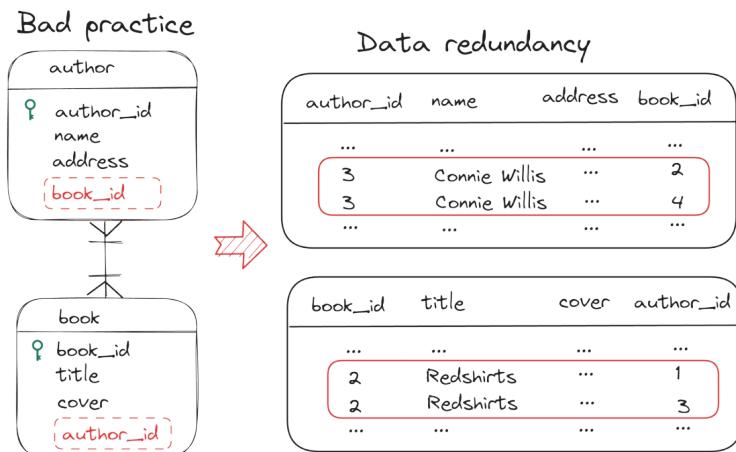
Using a composite primary key is necessary when a single column alone cannot guarantee uniqueness for each row, but the combination of multiple

columns does. It is commonly used in situations where a table has a many-to-many relationship with another table or when there is no simple key that can uniquely identify a row in the table.

Now you have two one-to-many relationships whose min cardinalities are all ones. As you have seen in the last sections, given such a one-to-many relationship, you need to relax the min cardinality on the “many side” from one to zero, so that you won’t face data entry or insertion issues when you implement such a representation.



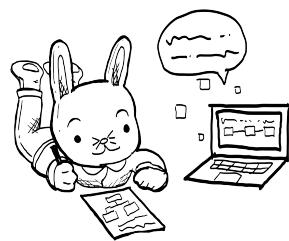
Now you know how to represent a many-to-many relationship in an E-R diagram, you may wonder why we can't simply insert a foreign key into both the entities involved in a many-to-many relationship. Well, representing a many-to-many relationship in this way will lead to data redundancy and difficulties in querying and modifying data. For example, if we represent the author and book relationship in this way, the data redundancy issue will immediately jump out:



If an author wrote multiple books, the same author information has to occupy multiple rows in the author table even if they are redundant, because they are linked to different books in the book table via the foreign key book\_id.

Similarly, if a book has multiple authors, the information of the same book has to occupy multiple rows. Needless to say, when it comes to implementing the two tables, having redundant values of primary keys, such as author\_id, is not possible. Thus, simply inserting a foreign key into both tables involved in a many-to-many relationship is not viable.

## Identify many-to-many relationships in your database



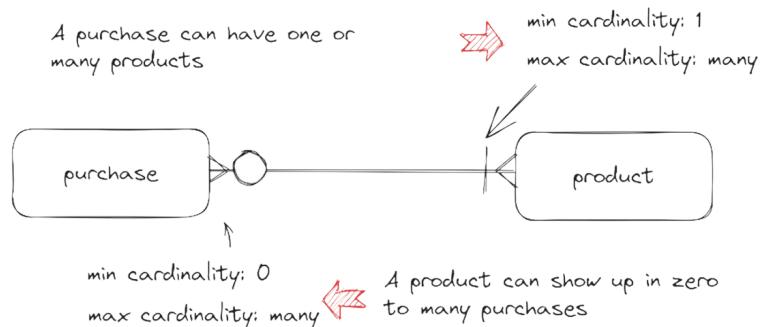
There is still one more relationship in our draft diagram that we didn't work on yet. This relationship is between the purchase and product entities:

- A purchase can have more than one product. A product can show up in multiple purchases.

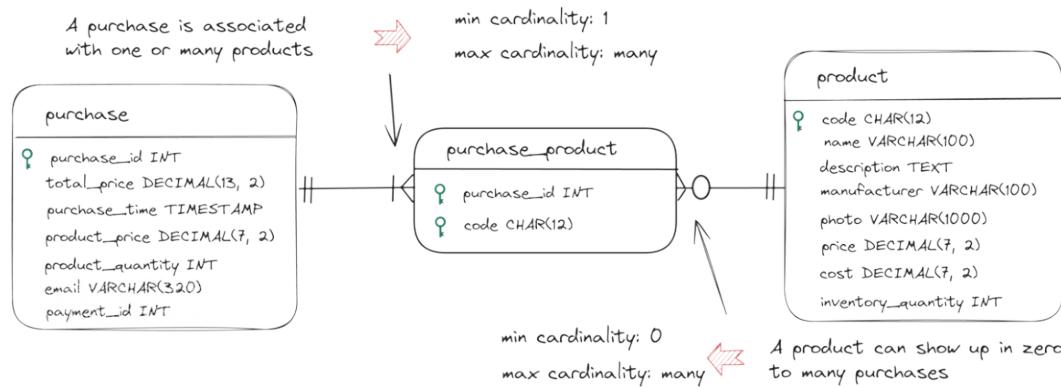
purchase	product
<pre>purchase_id INT total_price DECIMAL(13, 2) purchase_time TIMESTAMP product_price DECIMAL(7,2) product_quantity INT email VARCHAR(320) payment_id BIGINT</pre>	<pre>code CHAR(12) name VARCHAR(100) description TEXT manufacturer VARCHAR(100) photo VARCHAR(1000) price DECIMAL(7,2) cost DECIMAL(7,2) inventory_quantity INT</pre>

We should note that the previously added foreign keys in purchase are highlighted for convenience. It is also not difficult to identify this relationship as many-to-many because the two involved directional relationships have the max cardinality as many. After following up with the stakeholders for clarification, you will see that a product doesn't necessarily need to show up in a transaction, but a transaction has to be associated with at least one product. Based on such information, you can determine both the max and min

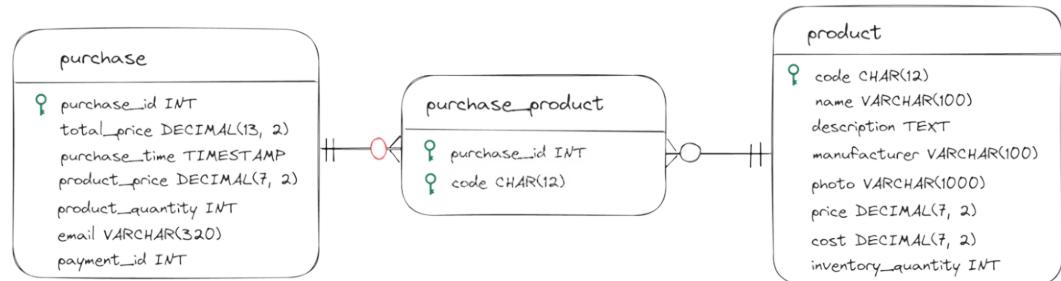
cardinalities:



To represent this many-to-many relationship, you will follow the same steps to make a junction table that only contains the IDs from the transaction and product entities, and convert the many-to-many relationship to two one-to-many relationships:

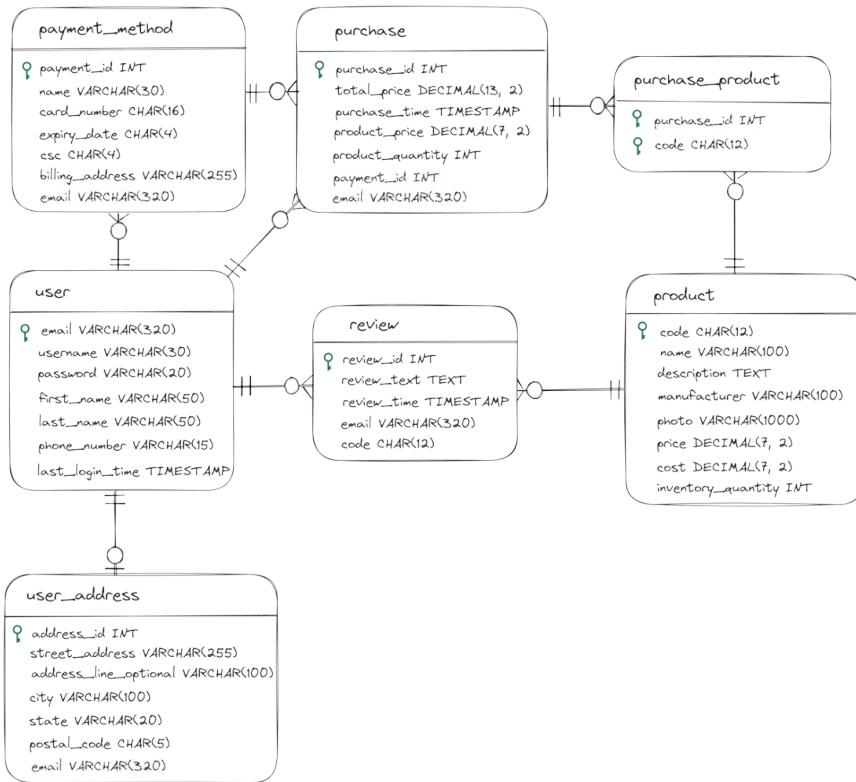


You may notice that the min cardinalities of the relationship between purchase and purchase\_product are both one. To avoid issues in data insertion when this representation is implemented, you should relax the min cardinality on the side of purchase\_product from one to zero:



To synthesize all our updates to the draft E-R diagram, it looks as follows

now:



## 5.5 Strong and weak entities

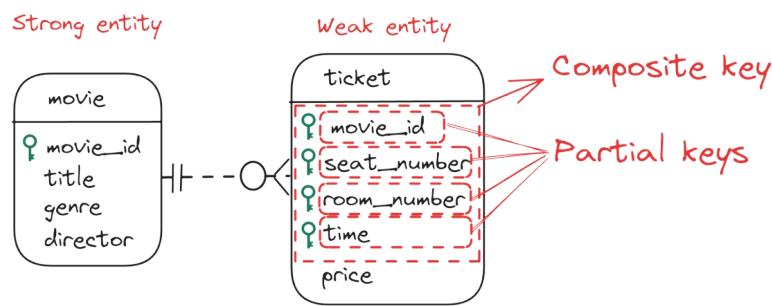
Same as cardinality, strong and weak entities are another characteristic of a relationship between two entities. This characteristic may sometimes come handy in refining E-R diagrams. You will learn about strong and weak entities in this section, and apply what you learn to refine our E-R diagram.



What are strong and weak entities? At the core, strong and weak entities are about a dependency relationship between two entities. *Given two related*

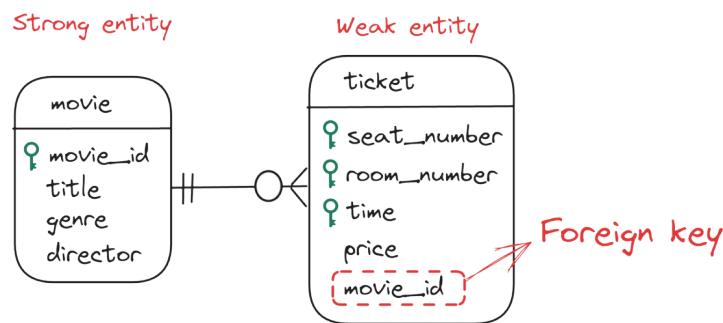
*entities, if one is strong while the other is weak, the strong entity can exist on its own, but the weak entity cannot exist without the strong entity. Let's try to understand strong and weak entities better via an example.*

Imagine that we are *designing* the database for a web application of a movie theater, and need to represent the relationship between movies (`movie`) and tickets (`ticket`). Between `movie` and `ticket`, `movie` is the strong entity as `movie` can exist independently in the database. In contrast, `ticket` is the weak entity, and it depends on `movie` to identify each of its instances. This relationship is depicted below:



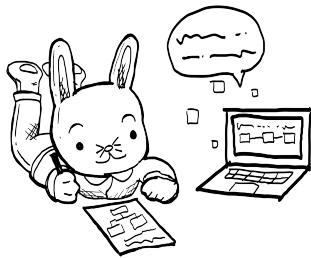
Typically, the primary key of the weak entity is a composite key that combines its own attribute(s) and the primary key of the related strong entity. As you can see, `ticket` uses a composite key composed of `movie_id` and other attributes. `movie_id` is the primary key in the `movie` entity, and the other attributes are used to distinguish different tickets of the same movie. Such attributes are also known as *partial keys*.

It is worth noting, however, that strong and weak entities can always be converted to two strong entities. For example, if tickets need to be managed independently for reselling and refund purposes, you can convert `ticket` to a strong entity:

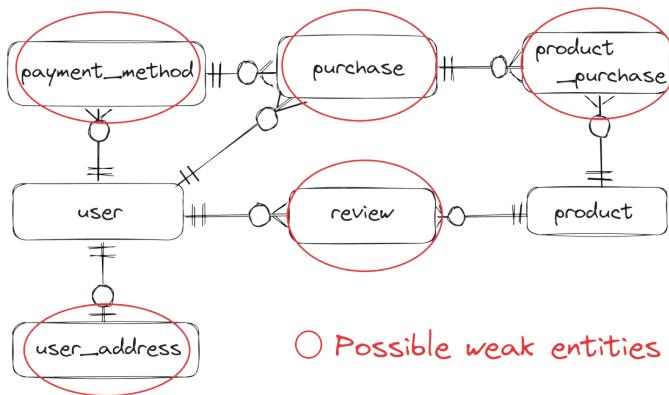


You may wonder what you should do if you have two entities that can be identified as strong and weak entities. Should you identify them as strong and weak entities, or simply treat them as two strong entities? It depends on a variety of factors such as business demands, data representation and maintenance, and performance. If both options are possible, you should always choose the one that makes E-R diagrams simpler and less susceptible to bad data. As we apply this new knowledge to further refine our E-R diagram, you will see several examples that involve decision making on whether to identify two entities as strong and weak entities.

### 5.5.1 Identify strong and weak entities in your database

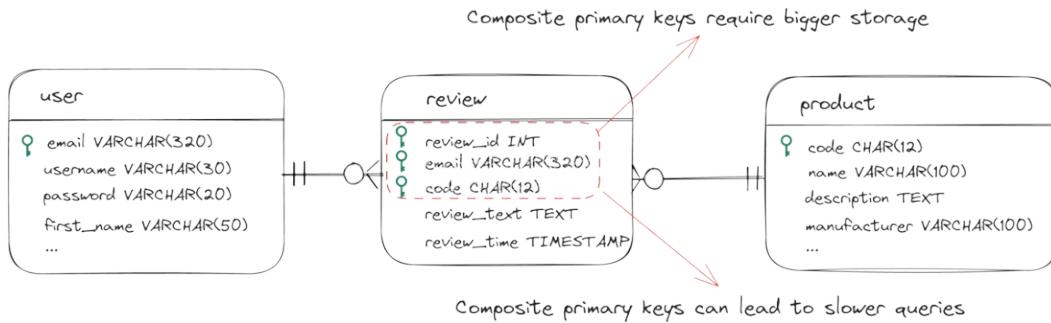


In our E-R diagram, there are several entities that fit in the definition of weak entities, such as `review` and `payment_method`. For example, the `review` entity can be seen as a weak entity that depends on the existence of `user` and `product`.



When you consider whether to convert a strong entity to a weak one, you should ask yourself if the decision will simplify the database design, or improve the data accuracy. If the answer is negative, you should stay put. For example, `review` can be converted to a weak entity that depends on `user` and

product. However, doing so means that you need to start using a composite primary key in the `review` entity, which doesn't introduce any benefits but can lead to increased storage and slower query performance – composite primary keys require wider index width and slower index lookup. As a result, `review` shouldn't be converted to a weak entity.



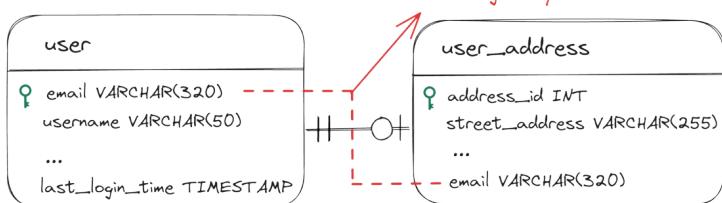
Following the same logic, you can tell that converting `payment_method` or `purchase` to weak entities is a bad idea. What about the `purchase_product` table? There's nothing about `purchase_product` you can change because it is in a perfect form of weak entities. The `purchase_product` table is a junction table that you created to link `purchase` and `product` in a many-to-many relationship. All junction tables are weak entities by their nature. In our case, `purchase` and `product` are strong entities. As the weak entity, `purchase_product` uses a composite primary key composed of the primary keys from the two strong entities.



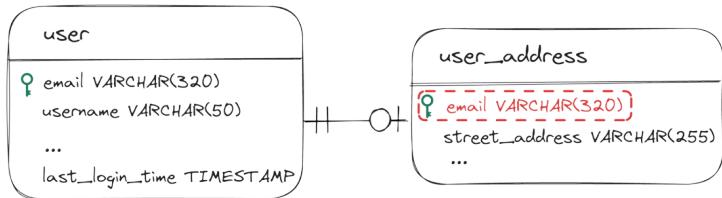
What about `user_address`? It turns out to be the only opportunity where you can apply the knowledge of strong and weak entities to refine the draft diagram. Your current design treats `user_address` as a strong entity, which uses `address_id` as the primary key to identify each of its instances, and `email` as the foreign key to maintain the link between `user_address` and `user`. In contrast, if you convert `user_address` to be a weak entity, its

primary key will become email, and the foreign key will be gone. Since user and user\_address are in a one-to-one relationship, email is good enough to identify each instance of user\_address. As a result, you no longer need a composite primary key in this case. Converting user\_address to a weak entity makes sense because it eliminates a foreign key constraint and simplifies your design.

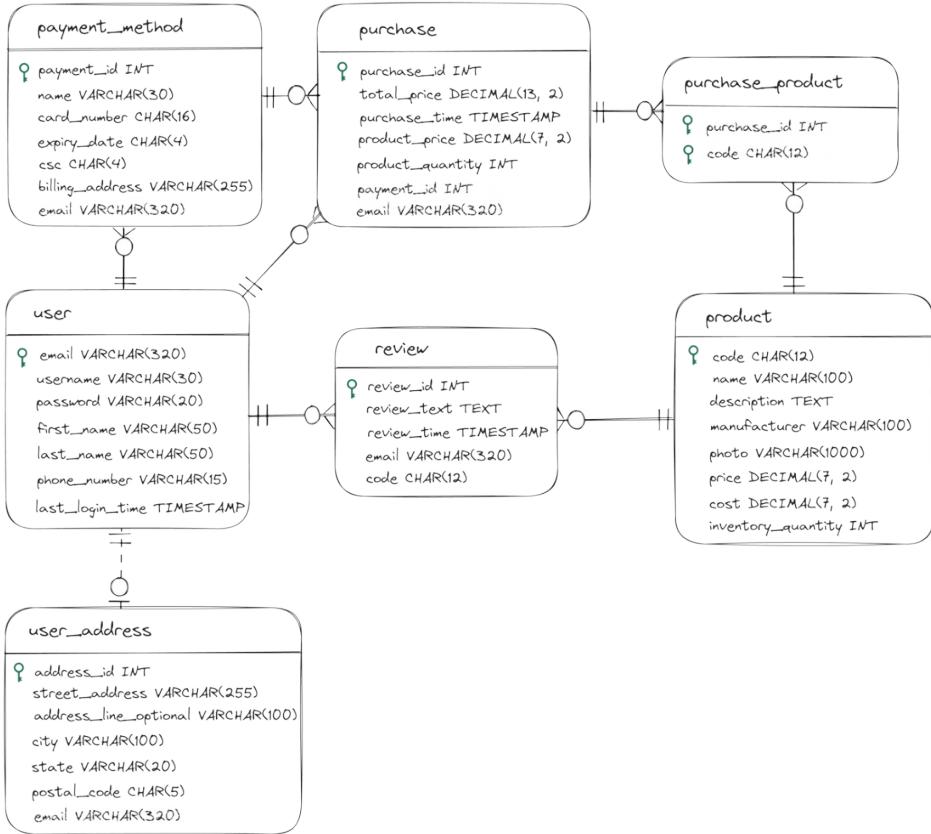
*Two strong entities:*



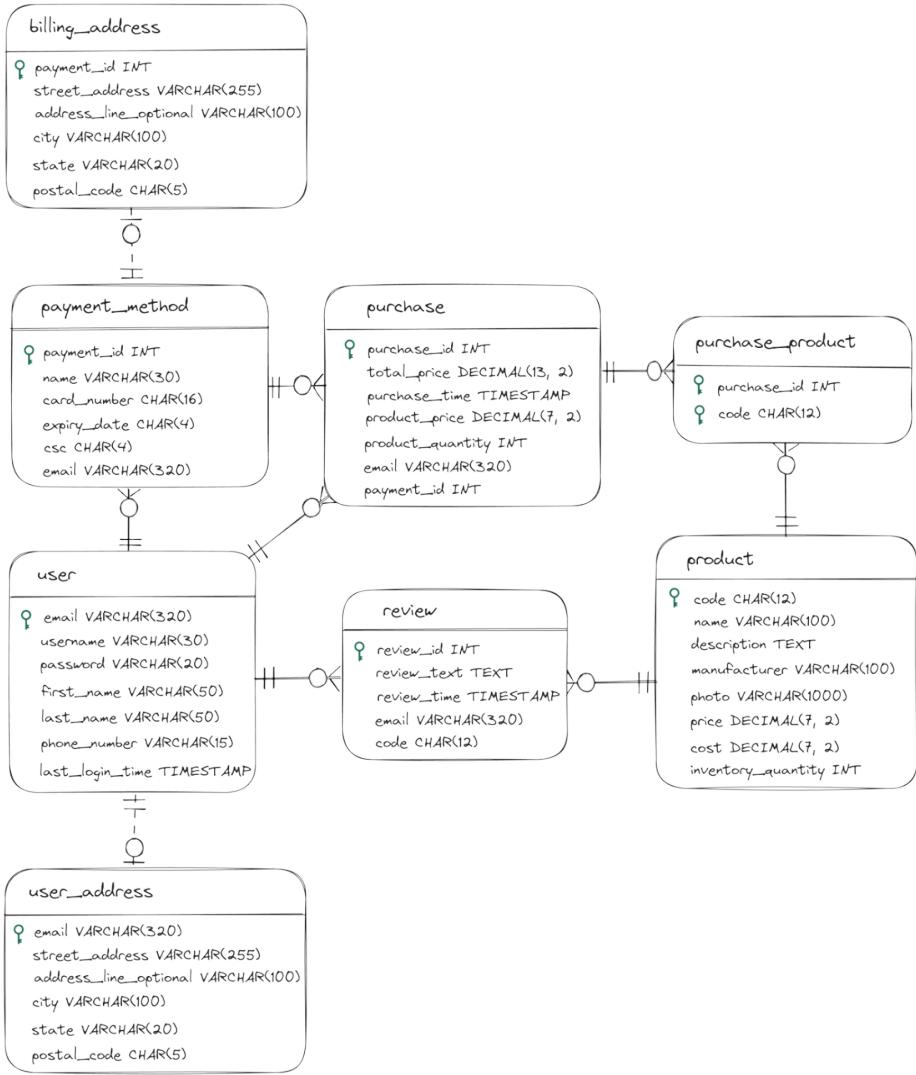
*Strong and weak entities:*



Your E-R diagram will be updated as follows when you convert user\_address to a weak entity:

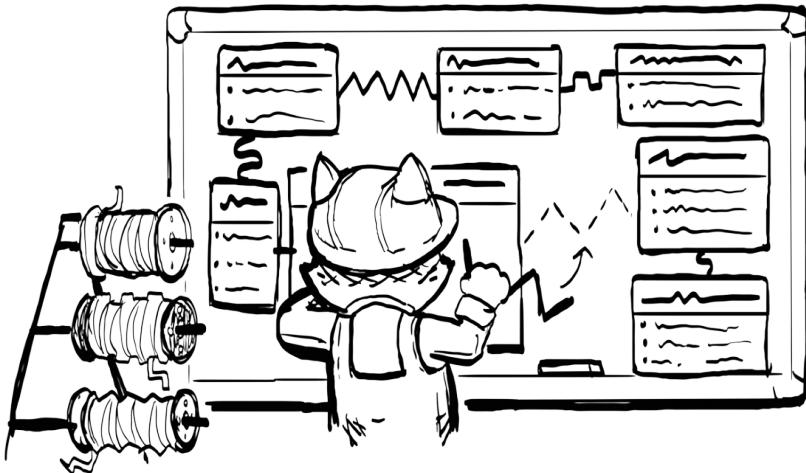


When taking a closer look at other tables and columns, you will notice that `payment_method` contains a column `billing_address` representing the billing address of a payment method. The relationship and nuances between a payment method and billing address are the same as between a user and user address. In other words, you need a table to fully represent the billing address in the same manner as how you modeled `user_address`. Following the exact same logic, we can use a table `billing_address` to represent the billing address. The `billing_address` is a weak entity; strong entity is `payment_method`. Incorporating this change, your E-R diagram will be updated as follows:



Now you should give yourself a pat on the shoulder, because you have successfully wrapped up a round of data modeling by developing and refining your E-R diagram. Even if the process is iterative, you still managed to conquer many challenges in this round. Great job!

## 5.6 Summary



- Relationships are the binding glue between entities. The representations of relationships are informed by the information from requirement gathering and further analysis.
- In an E-R diagram, you only need to represent relationships. To do so, you typically go through a process involving three steps, including (1) establishing relationships, (2) identifying the cardinality of each relationship, and (3) identifying potential weak entities and determining if they should be converted to weak entities.
- Cardinality is an important characteristic of a relationship between two entities. Cardinalities come in three flavors, one-to-one, one-to-many, and many-to-many. Different flavors of cardinalities are represented differently, and the flavor has implications on where the foreign key is placed.
- In a one-to-one relationship, the max cardinalities are both one, represented by two bars. The foreign key connecting the two entities can be placed in either side.
- In a one-to-many relationship, the max cardinalities of the two directional relationships are one and many. “Many” is represented by the crow’s foot. The foreign key should be placed in the many side pointed by the crow’s foot.
- In a many-to-many relationship, the max cardinalities of the two directional relationships are both many. The representation of a many-to-many relationship requires a junction table sitting between the two sides, and converting the relationship to two one-to-many relationship between each side and the junction table. The junction table contains the primary keys from both sides.

- Strong and weak entities are another characteristic of a relationship between two entities. If two entities are possible to be represented as strong and weak entities, you need to make decisions on if to do so, depending on whether such a representation makes the E-R diagram simpler and the database more efficient.
- A weak entity would use the primary key of its strong entity as a part of its composite primary key. Other than that, a weak entity is not represented differently in Crow's foot notation.