The necessary SQL

A COURSE FOR BEGINNERS
featuring MySQL and T-SQL

Daniel Vidali Fryer

An Open Access Textbook
Updated Nov 2021

Contents

Front matter

Preface

This book can be printed, but it's best accessed as an electronic pdf. That's because, if you see this colour, you can click it (it's a link). Many things are sneakily hyperlinked within the document. For example, clicking on an item in the table of contents will jump you to that item in the book, and clicking a reference to a figure will jump you to that figure. This book also features a glossary of terms. If you see a new term appearing in bold, you can jump to the glossary definition by clicking on the bold word.

If you already know which type of SQL you want to learn, keep in mind that we cover standard SQL, along with the MySQL and T-SQL variants. If these words mean nothing to you, then read on, friend. All will be revealed.

Introduction

Look, SQL is a polarising language. I'm not sure if anyone truly likes it. I've found it's best to come clean about this from the start. Regardless, I swear that SQL is fun. Don't believe me? Take my course and tell me you hated it. Every query is a mini puzzle to solve. Maybe SQL is less fun when you're an experienced engineer, but who cares? SQL is *powerful*. It has been around since the 1970s, and today it is the standard bridge between data engineers and data analysts.

A wise man once told me that a wise woman once told him that a drunk person's words are a sober person's thoughts. On this account, don't take it from me, but from a reddit post by a drunk senior engineer:

> "*For beginners, the most lucrative programming language to learn is SQL. F**k all other languages. If you know SQL and nothing else, you can make bank… Average joe with organizational skills at big corp? $40k. Average joe with organization skills AND SQL? Call yourself a PM and earn*

> *\$150k... I think a piece of tech is good if I hate it but I simultaneously would recommend it to a client.*"

To me, that last line is SQL in a nutshell. I may hate it, but I recommend it. And look, I suspect most of my intended audience aren't around to "make bank". You're more likely here to learn to use SQL on research projects involving some kind of dataset that doesn't seem to fit well in a CSV file. I recommend learning SQL, not so you can demand a higher salary in your next engineering job, but for two main reasons:

1. You're taking my course, so you probably already have found you need to use SQL. If not, then you're probably in a position where you could use it soon, to great effect.
2. This book will teach you a model for structuring and processing data. As we'll learn, parts of that model are used in many different languages, not just SQL. This model underlies SQL and has stood a grand test of time.

Anyway, what *is* SQL?

In the coming few pages, you and I will unpack the following quote:

> *"Structured Query Language (SQL) is a domain-specific language used in programming and designed for managing data held in a Relational Database Management System." - Wikipedia*

Just kidding, let's not unpack that quote. Right now, we don't care what a "domain specific language" is, and we can just call it a "programming language". What we will do, very briefly, is learn about *relational databases*, and the model that underlies SQL. Here is a more important quote to get you on your way:

> *"People familiar with different tools understand problems and their solutions differently." - Uldall-Espersen 2008*

The tool, in our context, is a Relational Database Management System (RDBMS), and more deeply, the relational model. Some familiarity with it will hopefully help you understand problems in the SQLian way (pronounced "Sequelian", as in, a citizen of SQL).

Perhaps you're thinking: "*hey listen here, Danny, I don't care about database management, I just want to use SQL to get my dataset so I can analyse it and be on my way.*"

Think of the Database Management System (DBMS) as a kind of oracle[1]. You declare your needs to the oracle, it does some back-end magic and, bam, you have your dataset. Like all self-respecting beings, the oracle has its own conception of reality. The **relational model** is the oracle's grand unified Theory of Everything. Unless you understand this model, talking to the oracle can be frustrating, confusing, and fruitless. Thankfully, practicing SQL and learning a bit about "relationships between tables" is pretty much all you need to do to get a good working intuition. Let's jump in.

[1] I mean oracle in the sense of a magic person who answers questions, not in the sense of Oracle Corporation which, incidentally, manufactures database systems.

Chapter 1

The relational model

1.1 Database Managemement Systems (DBMS)

A database is a purpose-built, logically coherent collection of meaningful data, representing some aspect of the real world. We can refer to this aspect of the real world as a **miniworld** [Elmasri, 2008]. A database aims to contain data that represent an accurate, up-to-date reflection of its miniworld. So, if some important aspect of the miniworld changes (for example, if a new experiment is conducted that records new data), then either the contents or structure of the database will need to change too.

Typically, a large collection of interdependent programs are employed to define, construct, manipulate, protect and otherwise manage a database. Such a collection of programs is called a Database Management System (DBMS). Microsoft SQL Server, Oracle Database, and MySQL are all examples of Database Management Systems (see Figure 1.1).

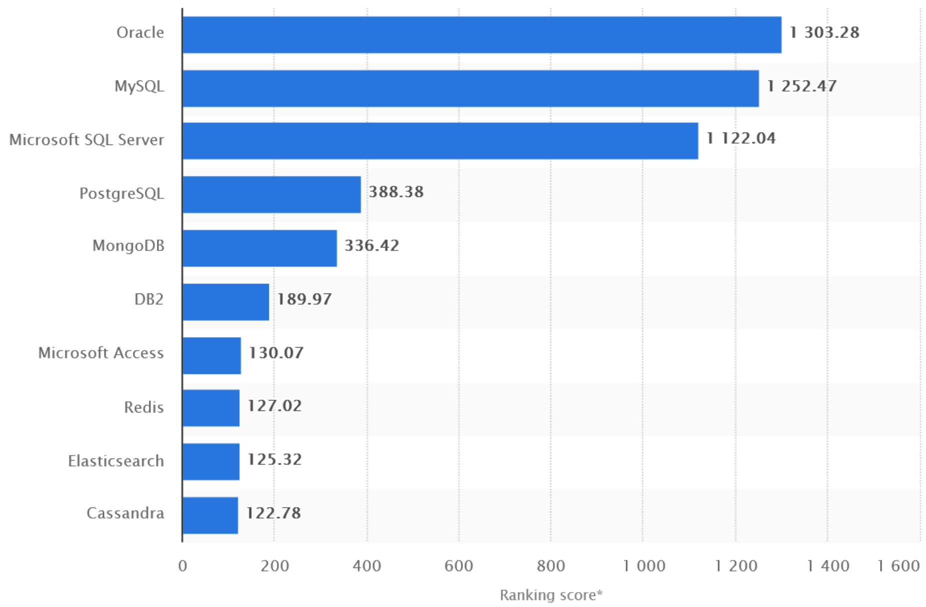

Figure 1.1: DBMS Rankings by popularity, 2019. Source: statista.com

Under the hood, a DBMS interacts with a computer via some low-level magic, mostly of interest to engineers. Above the hood, the DBMS interacts with humans. So, the DBMS aims to share a conceptual representation (i.e., a *structure*) of the data with these humans. For the DBMS, this conceptual representation is stored as a catalogue of information called **metadata** (literally, data about data). The use of metadata, kept separate from the main data, allows the DBMS to maintain a nice layer of abstraction between its low-level interactions with the machines, and its high-level interactions with its human overlords. The meta-data and the main data are the two separate silos pictured at the bottom of Figure 1.2.

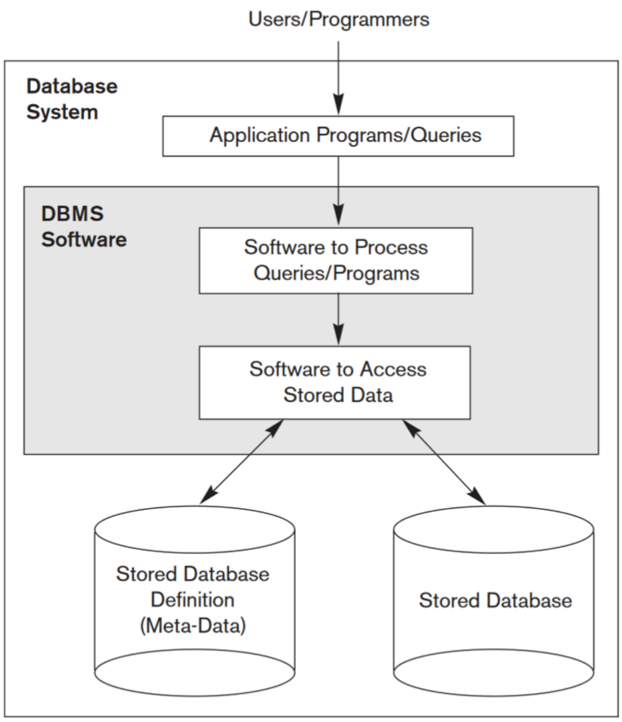

Figure 1.2: A simplified database system. Source: [Elmasri, 2008].

When commanded by the humans, the metadata allows the DBMS to easily interact with the database structure, without interfering much at all with the machine and underlying programs. In summary, *the metadata gives the database its structure*, and makes it easy for humans to define and manipulate the data. The human SQL programmers are able to define data using SQL commands like `INSERT`, `UPDATE` and `DELETE`. The SQL programmers are also able to manipulate (or **query**) data, using SQL commands like `SELECT`, `FROM` and `WHERE`. We'll learn about these (and other) data definition and manipulation commands throughout these notes.

Now, conceivably, there are endless ways in which the metadata could be used to conceptually describe the miniworld. The most popular way, by far, is to implement the *relational model*.

1.2 The relational model

The **relational model** was first introduced by an IBM researcher, Ted Codd, in 1970. If you're into it, then view his original paper [Codd, 1970]. The first sentence of that paper sums up why you're in this chapter:

> *"Future users of data banks [i.e., of databases] must be protected from having to know how the data is organised in the machine."*

The paper draws on some of the mathematical theory of *relations* (or set theory and first-order predicate logic). It applies this math to the task of modelling (i.e., structuring) a database with metadata. The resulting model, called the relational model, turns out to have some very nice mathematical properties that are super useful for a DBMS to take advantage of. Much of Ted's seminal paper on the relational model looks a bit like this:

$$
\begin{aligned}
&(1)\ \pi_1(T) = \pi_2(S),\\
&(2)\ \pi_2(T) = \pi_1(R),\\
&(3)\ T(j, s) \rightarrow \exists p\,(R(S, p) \wedge S(p, j)),\\
&(4)\ R(s, p) \rightarrow \exists j\,(S(p, j) \wedge T(j, s)),\\
&(5)\ S(p, j) \rightarrow \exists s\,(T(j, s) \wedge R(s, p)),
\end{aligned}
$$

Figure 1.3: Some undefined mathematical symbols to scare us away from Ted's paper. [Codd, 1970].

Luckily, much of the *relational algebra* underlying the relational model is easily represented using **tables**, and certain operations on tables. Tables are simple, intuitive, and easy for your standard non-mathemagical human to cope with. In fact, from now on, we will think of the relational model not as a horrifying medley of undefined mathematical symbols, nor as a cold assortment of mechanical ("boo"-lean) logic, but as a simple collection of tables, and a collection of operations on tables. To begin, here is a table:

Friends			
FriendID	FirstName	LastName	FavColour
1	X	A	red
2	Y	B	blue
3	Z	C	NULL

Figure 1.4: Our very first table.

And here is an operation on a table:

```sql
SELECT FirstName FROM Friends;
```

If the Friends table includes all the names of my friends, then the above operation will give me a list of their first names, X, Y, and Z. We will look at the `SELECT` and `FROM` keywords closely in time. For now, we're going to focus on building our vocabulary on the anatomy of tables.

1.3 Table anatomy and notation

In the relational model, the data are thought of as belonging to a collection of tables. Each piece of data (i.e., each *datum*, e.g., a person's name, a phone number, a date, etc) belongs to a particular row and column of some table, somewhere. A datum is also known as an **entry**. Each entry sits at a particular place in a table, in a certain row and column. To keep things organised, each table is thought of as a collection of rows, where each row is one realisation of the **entity** (such as a person or friend) that the table represents. The word **tuple** is also sometimes used to refer to a row, as is the word 'record'.

For example, I use my Friends table to keep track of all of my friends' favourite colours. In this case, each row of the Friends table represents one friend. That is, each row contains data on one, and only one, friend. The number of friends I have, is equal to the number of rows of the table. When I create the Friends table, I need to decide what columns the table should have. Columns are also known as **attributes**. Attributes capture the miniworld representation of each of my friends.

For example, a reasonable set of attributes might include each friend's first name (FirstName), their last name (LastName), and their favourite colour (FavColour). For good practice, I'll add a fourth column, and use it to assign each friend their own unique ID number (FriendID). Before long, we will see why these ID numbers are useful. At the very least, the ID will help me distinguish between any two friends who happen to share the same name. One can never be too prepared when it comes to keeping track of all their friends in SQL. At this stage, my table looks like this:

Friends			
FriendID	FirstName	LastName	FavColour

Figure 1.5: My empty table of friends.

Formally, a table is called a **relation** – this is where the name *relational model* comes from. We could perhaps just as easily call it the table model, at the cost of sounding less mathematically woke. We have thus defined the terms *relation* (table), *record* or *tuple* (row), and *attribute* (column). The above table has no rows, so it is an empty relation. Indeed, a table doesn't *need* to have any rows, but it does need to have columns. Could we say, then, that a table is a named collection of 1 or more columns, with 0 or more rows? Almost, but there is one ingredient to any good self-respecting table that we haven't discussed yet, the *domain*.

The **domain** tells us what sort of data (e.g., person names, phone numbers, country names, etc) that we can store in each column of the table. For my Friends table, we will choose the domain to be people's first names for the FirstName column, people's last names for the LastName column, names of colours for the FavColour column, and positive whole numbers for the FriendID column.

There is a nice and simple way to describe a table when we don't care what is inside the table: we just write the table name, then put all of the attribute names in front of it in brackets. So, for the Friends table, we write

$$\text{Friends}(\text{FriendID}, \text{FirstName}, \text{LastName}, \text{FavColour}).$$

The above describes the structure of the Friends table (provided we know what the domain of each attribute is). However, we will certainly end up needing to refer to attributes from more than one table, so we also need some convenient shorthand notation to replace clunky phrases like "the FirstName column from the Friends table". Instead, we will can just write "`Friends.FirstName`". Here, we used the full stop symbol to indicate that FirstName is a column of Friends.

If we want to talk about the rows of a table, we can enclose the comma-separated values in some brackets to show that they form one neat little record. This way, the first row of the Friends table in Figure 1.4 would be represented as:

$$(1, X, A, \text{red}).$$

The order of the elements matches the order of columns in the table. So, the above row represents a friend with FriendID number 1, whose first name is X, last name is A, and favourite colour is red. We will refer to each element of a row as one **entry** in a table. So, in this row, the colour red is one entry.

At this stage, some thrill-seeking readers may be wondering if we can set the domain of an attribute to be a collection of tables, whereby we might begin to include whole tables as entries inside rows, inside tables, producing for ourselves a horrific struggle against a hierarchical tower of tables within tables. For good reason, this is banned from the relational model.

We refer to the relational model as *flat*, and demand that each entry must be **atomic**, meaning each entry should be something that is not intended to be subdivisible (like the 'atoms' in physics[1]). For example, in my Friends table, FavColour and LastName are atomic. We would avoid merging them together as one attribute, FavColourLastName, since the result

[1]Before we subdivided them.

would be non-atomic. Similarly, if we want to store a person's address, we would aim to break the address up into atomic parts: one column for street number, one column for street name, one column for post code, etc. This flatness has various helpful consequences, not the least of which is that it makes it easy for us to search our database (e.g., "computer, give me a list of all friends who share the postcode 3000").

1.4 Relationships between tables

1.4.1 One-to-many relationships

We have just discussed the flatness principle, that every entry in a table should be atomic. You have it mostly on good faith that this is a helpful principle. However, you might already be formulating the following question. What happens if an attribute can have more than one instance for a given record? Perhaps, for example, we decide to keep track of the names (PetName) of my friends' pets, as in:

Friends(FriendID, FirstName, LastName, FavColour, PetName).

A friend could easily have more than one pet. We call this a **one-to-many relationship**, since *one* friend can have *many* pets. So, where do we put the extra pets? Do we add extra columns?

WRONG

Friends					
FriendID	FirstName	LastName	FavColour	PetName$_1$	PetName$_2$
1	X	A	red	`NULL`	`NULL`
2	Y	B	blue	Chikin	`NULL`
3	Z	C	`NULL`	Cauchy	Gauss

Figure 1.6: A dodgy table for keeping track of pets.

According to the above table, my friend X has no pets, Y has one pet (Chikin), and Z has two pets (Cauchy and Gauss). This set-up is problematic for a few reasons.

- Firstly, I have to store `NULL` in every entry where there is no pet. This takes up space and is cumbersome.
- Secondly, if I meet a new friend who has three pets, then we need to add an extra column to the table. In this case, after adding a new column (PetName$_3$), we would have to insert new `NULL` values under PetName$_3$ into every row that doesn't have 3 pets. With many friends, and many pets, the amount of `NULL` values quickly escalates.

- Finally, if we want to keep extra details on each pet, such as their birthdates (PetDOB), we'll begin to blur the lines around what this table represents. Is it a table of friends, or a table of pets? Is FavColour the favourite colour of a friend, or the favourite colour of a pet?

The solution is simple:

I create another table.

The new table will contain data on all the pets. Each pet will receive its own unique identifier, petID. A new and crucial little attribute, FriendID, will describe which friend each pet belongs to.

CORRECT

Pets			
PetID	PetName	PetDOB	FriendID
1	Chikin	24/09/2016	2
2	Cauchy	01/03/2012	3
3	Gauss	01/03/2012	3

Figure 1.7: A great table for keeping track of pets.

Now, for example, if we want to retrieve the details on the owner of the pet named Chikin, then we can start by looking up the `Pets.FriendID` for Chikin (finding that it's equal to 2) and then we match Chikin to its owner by finding out where the column `Friends.FriendID` is equal to 2.

Pets		
PetID	...	FriendID
1		2
2		3
3		3

Friends	
FriendID	...
1	
2	
3	

Figure 1.8: Finding the details of the friend who has the pet with PetID of 1.

For the above to work smoothly, each entry stored under `Pets.FriendID` *must* correspond an existing entry stored under `Friends.FriendID` (recall our notation that `Pets.FriendID` means "the FriendID column of the Pets table"). We need every `Pets.FriendID` entry to have a matching entry in `Friends.FriendID`, so that we can be guaranteed that every pet will have an owner. This requirement, in database lingo, is called **referential integrity**. In Section 1.4.3, we will learn why referential integrity is important.

Going the other way, if we want the details of all pets belonging to, say, my friend Z, then we start by finding, in the Friends table, that my friend Z has FriendID equal to 3, and then we can search for every `Pets.FriendID` entry that is also equal to 3.

Pets		
PetID	...	FriendID
1		2
2		3
3		3

Friends	
FriendID	...
1	
2	
3	

Figure 1.9: Finding the details of all pets who belong to the friend with FriendID of 3.

Take a moment to convince yourself that this works, and that the above-mentioned problems with our previous table (Figure 1.6) have been solved. By using two tables instead of one, we have removed many `NULL` values, while also making the database more flexible. Part of being "more flexible," is that each pet now has its own unique ID number (PetID). This means, if we want to, we can add a new one-to-many relationship between pets and something else (like pet toys), just the same way that we created a one-to-many relationship between Friends and Pets. That is, we would create a new table (say, PetToys) with an attribute PetID, that "points at" the PetID column of the Pets table, indicating to which pet each toy belongs (just how we indicated to which friend each pet belongs). Try it yourself now, as an exercise.

In this section, we modelled a *one-to-many* relationship. In the next section, we will expand on this idea to cover the two remaining types of relationships that can arise between two tables in a relational database: *one-to-one* relationships, and *many-to-many* relationships. All three kinds of relationships are managed by wonderful attributes called primary and foreign key pairs.

1.4.2 Primary and foreign keys

In the previous section, we avoided turning the Friends table into a hot mess (of the kind in Figure 1.6). Instead, we created a Pets table (Figure 1.7) that has a one-to-many relationship with the Friends table. To model and keep track of this relationship, we mentioned that each entry in the FriendID column of the Pets table must be equal to exactly one entry in the FriendID column of the Friends table. In this case, we call call these columns a **primary key** and **foreign key** pair.

A primary key is any column (or collection of columns) that has been chosen to uniquely identify the rows of the table it belongs to. In our example, the primary key for the Friends table is the FriendID column, and the primary key for the Pets table is the PetID column. Since the role of a primary key is to uniquely identify rows, we must ensure that ***every primary key entry is unique***. That is, no two rows in a table can share the same value for their primary key entries.

It is good practice to give every table a primary key. When creating a SQL database, it's easy to specify which attributes are primary keys. However, unfortunately, some database administrators didn't get the memo. So, in the databases that you use, you may potentially encounter tables having no clear primary key. In such cases, as we'll soon learn, when writing SQL queries, we may have to choose an attribute to play the role of primary key, even when the database doesn't recognise it officially as a primary key. The same goes for foreign keys.

A foreign key is any column (or collection of columns) where each entry is equal to one, and only one, primary key entry in some other table. Thus, given a foreign key entry, we can always find the unique primary key entry that it is equal to. For this reason, we say that the foreign key is "pointing at", or that it *references*, the corresponding primary key. When creating a foreign key, the creator writes SQL code to tell the database which primary key the foreign key is "pointing at".

1.4.3 Informal relationships and referential integrity

Before introducing one-to-one and many-to-many relationships, we're going to take a moment to discuss "informal relationships". Every table can have zero or more foreign keys, and each of the foreign keys must point at exactly one primary key somewhere. This is a strict rule about foreign keys: the record that they reference is guaranteed to exist. This rule is called **referential integrity**. Unfortunately, not all databases are designed perfectly; you may encounter tables that *should be* related, but for which the designer failed to include a foreign key to formalise the relationship. There may be no foreign keys, and there may be no primary keys, but you may know that the tables are related. This is called an informal relationship between tables.

In these cases, wanting to connect information between informally related tables, a SQL programmer will need to carefully choose their own "natural" primary and foreign key pairs from the tables – they will have to guess which columns should point at each other – often without knowing for sure if each natural foreign key entry will be guaranteed to point at an existing natural primary key entry, and often not knowing for sure if the chosen natural primary key is guaranteed to have unique entries. For example, you may be dealing with a database containing a table called Houses and a table called Suburbs. Suppose the Suburbs table has two columns, SuburbName and PostCode, and suppose the Houses table has a column

PostCode, but no column SuburbName.

Houses(Bedrooms, Bathrooms, LandSize, PostCode)
Suburbs(PostCode, SuburbName).

Looking at these tables, can you decide if either (i) or (ii) below are true?

(i) Is every PostCode entry in Suburbs unique?

(ii) For each PostCode entry in Houses, is there definitely at least one entry in Suburbs with that PostCode?

Sadly, you can't. The answers, though, have important consequences:

- If (i) is false, then two suburbs can share the same post code. For such a pair of suburbs, these tables would not allow us to decide which of the two suburbs a given house belongs to.
- If (ii) is false, then there is a house in Houses with a PostCode that doesn't exist yet as an entry in Suburbs. In this case, we cannot use the tables to find the suburb for this house.

If the PostCode columns were formally defined as a primary and foreign key pair by the database administrator, then (i) and (ii) would both be true, resolving the above dilemmas. Without this formality, it is up to the SQL programmer to decide if the relationship can be trusted. We will, of course, learn how to test (i) and (ii), given some data. However, without the formal primary and foreign key pair, our tests will have to be conducted every time the database is updated with new data.

We will get plenty of practice with primary and foreign key pairs as we go. So, don't be too concerned if your head is spinning. The important thing is to go and have another look at the one-to-many relation between Friends and Pets in Section 1.4.1 now, to remind yourself which column plays the role of primary key, (hint: it's in the Friends table), and which one plays the role of foreign key, (hint: it's in the Pets table). In the following two sections, we'll see two more of the most typical use cases for primary and foreign key pairs.

1.4.4 Many-to-many relationships

Most people need their backs scratched from time to time. So, I figured, why not keep a record of whose back is being scratched by whom? This situation is new to us, since it is a **many-to-many relationship**. That is, one friend can be the scratcher of more than one back (at different times, presumably), and one back can be scratched by more than one friend. In practice, we can model a many-to-many relationship using one new table and *two* one-to-many relationships. In other words, we make one new table, and use two primary/foreign key pairs.

Scratched(ScratcherID, ScratcheeID, Date, Time)

In this Scratched table, the foreign key ScratcherID references the primary key FriendID from the Friends table. This lets us know which friend did the back scratching. Similarly, the foreign key ScratcheeID references the primary key FriendID from the Friends table. This lets us know whose back was being scratched.

Friends		
FriendID	FirstName	...
1	X	
2	Y	
3	Z	

Friends		
FriendID	FirstName	...
1	X	
2	Y	
3	Z	

Scratched			
ScratcherID	Date	Time	ScratcheeID
1	05/09/2018	12:00pm	2
1	05/09/2018	12:30pm	3
2	06/09/2018	11:00am	1
3	07/09/2018	10:00am	1

Figure 1.10: Modelling a many-to-many relationship amongst friends.

In this example, both foreign keys reference the primary key from the Friends table. In Figure 1.10, we are visualising this as if there are two copies of Friends. In general, a many-to-many relationship can exist between any two tables, i.e., not necessarily between one table (Friends) and itself. In any case, we always model a many-to-many relationship using *two* one-to-many relationships: that is, a new table (Scratched) and two primary/foreign key pairs, as we have done in Figure 1.10.

For practice, let's model one more many-to-many relationship. This time, between pets and friends. A pet can play with more than one friend, and a friend can play with more than one pet. For whatever reason, we decide to keep count. We need a new table, PlayCount, and two primary/foreign key pairs.

Pets		
PetID	PetName	...
1	Chikin	
2	Cauchy	
3	Gauss	

Friends		
FriendID	FirstName	...
1	X	
2	Y	
3	Z	

PlayCount		
PetID	Count	FriendID
1	3	1
1	5	2
3	4	2

Figure 1.11: Modelling a many-to-many relationship between friends and pets.

We can see from the PlayCount table (Figure 1.11) that my friend X played with Chikin 3 times, Y played with Chikin 5 times, and Y played with Gauss 4 times. Nobody played with Cauchy.

So, there we have it. A many-to-many relationship between two tables is really just two one-to-many relationships, with a new intermediate table (PlayCount in Figure 1.11, or Scratched in Figure 1.10) to store the two new foreign keys, along with any attributes of the relationship itself (such as a Count attribute, or the Date and Time).

Remember, way back in Section 1.4.2, when I said that every table should have a primary key? Well, where is the primary key in PlayCount? Where is it? It's not there. We need to make it. What's more, it won't be any single column. The primary key of PlayCount will be *two columns*.

So far, we've only seen primary keys that are a *single column*. But, remember, the primary key can be *more than one column*. Of course, as we well know, the primary key also has to be unique, and it must have no `NULL` values. To achieve uniqueness in Scratched, we need to use *all four columns* as the primary key - gasp! all four columns? Yes, because one friend can scratch the same friend's back on *different dates and times*, so we need all four columns, to achieve uniqueness. So, the primary key needs to be all four columns in Scratched, but what about in PlayCount? Well, to get uniqueness in PlayCount, we only need the primary key to be composed of two columns: the two foreign keys. Indeed, when the same friend plays with the same pet on different occasions, we can just increment the corresponding Count attribute, rather than adding a whole new row to PlayCount and violating the uniqueness of the two foreign keys.

1.4.5 One-to-one relationships (and data redundancy)

Consider the following extension of the Friends table, where I have included extra attributes that describe my friends' passport details.

WRONG

Friends					
FriendID	FirstName	...	PptCountry	PptNo	PptExpiry
1	X		Australia	E1321	12/03/2021
2	Y		New Zealand	LA123	01/09/2032
3	Z		Monaco	S9876	19/06/2028

Figure 1.12: A not-so-flexible extension of the Friends table.

Assuming that each friend has only one passport, we say that there is a **one-to-one relationship** between friends and passports. The above table may often be perfectly fine for capturing this one-to-one relationship. In many cases, there is no need to introduce a new table when modelling a one-to-one relationship, at all. Indeed, individual tables capture one-to-one relationships themselves, already. For example, by including a FirstName column in the Friends table, we are implying that there is a one-to-one relationship between a friend and their own first name.

However, for keeping track of my friends passport details, I've decided I need more than one table. This implies I want to think of friends as separate entities to their passports. One reason might be that, perhaps, many of my friends do not have passports, and I don't want to create unnecessary `NULL` values (recall Figure 1.6). Another reason could be about the kind of data I want to delete when I delete a friend. Suppose I lose my friend, X. Well, I definitely want to delete their details from my Friends table. In the next table, I've deleted my ex-friend, X:

Friends					
FriendID	FirstName	...	PptCountry	PptNo	PptExpiry
2	Y		New Zealand	LA123	01/09/2032
3	Z		Monaco	S9876	19/06/2028

Figure 1.13: Goodbye, Mr X.

Look what happened though. When I deleted X, I also deleted their passport details. Now, why would I want to delete Mr X's passport details just because we are no longer friends? Those details might come in handy during any future conflicts[2]. Hence, I'm going to model the one-to-one re-

[2]Please do not use passport details for evil. Also, be careful with private info!

lationship between friends and passport details by introducing a new table:

CORRECT

Passports			
PptNo	PptCountry	PptExpiry	FriendID
E1321	Australia	12/03/2021	`NULL`
LA123	New Zealand	01/09/2032	2
S9876	Monaco	19/06/2028	3

Figure 1.14: A nice, flexible way to store passport details.

In the above table, the foreign key FriendID will reference the FriendID column of the Friends table, just as it would when modelling a one-to-many relationship. Now, if I lose a friend whose passport details I'm holding onto, then I can delete them from my Friends table and insert `NULL` into the corresponding FriendID entry in the Passports table.

This is a good time to talk about **data redundancy**. There is still a problem with my Passports table. If I have a `NULL` value in the FriendID column, then I won't be able to find the name of the person who the corresponding passport belongs to. Hmm, should I include my friend's names in the Passports table as well, so the names won't get deleted when I delete a friend? Think about this for a moment. Will this cause any issues? Yes, it may: if I keep friends' names in *both* the Friends table *and* the Passports table, then the *same piece of data will be repeated in two different locations in my database*. This is known as **data redundancy**.

The problems with data redundancy are that it takes up unnecessary space, it can lead to inconsistencies in the data (if mistakes are made during data entry), and it leads to complications when updating data (because we'll need to update the data in multiple locations). The process of restructuring a relational database to reduce redundancy and preserve the integrity of data is known as normalisation.

To solve our problem with passport names, it would be best to re-think our database a little. For example, we could have a table called Contacts, with details of all the people we know. We could have one-to-one relationships between Contacts and each of two other tables, named Friends and Enemies, that contain friend-specific and enemy-specific data (like, favourite colours for friends, and secret hideouts for enemies). Relational database design is a deep and interesting topic, lying mostly outside the scope of these notes. So, next time you meet a good database engineer, give them a high five (and then employ them).

1.5 Handwritten exercises

These exercises should all be done by hand (e.g., with pen and paper). No programming is required.

Exercise 1.5.1

We'll start with some quick questions to warm up.

1. Answer true or false to each of the following:
   a) SQL is a relational database management system.
   b) Microsoft SQL Server is a programming language.
   c) A primary key must always be unique.
   d) A primary key must never be `NULL`.
   e) A foreign key must always be unique.
   f) A foreign key must never be `NULL`.
   g) In each table, only one column can be the primary key.
2. Referential integrity demands that (choose one): (i) the primary key must always exist; (ii) the foreign key must always exist; or, (iii) for every foreign key entry, there must be a corresponding primary key entry.

Solutions to Exercise 1.5.1

1. Here are the answers:
   a) False. SQL is a programming language (and a standard).
   b) False. SQL Server is a relational database management system, which implements the T-SQL dialect of SQL.
   c) True.
   d) True.
   e) False.
   f) False.
   g) False. The primary key can be composed of multiple columns.
2. The answer is (iii): for every foreign key entry, there must be a corresponding primary key entry (in the table that it references).

Exercise 1.5.2

For the following two tables, you are told that each home can have many tenants, but each tenant lives in just one home.

Home	
HomeID	Street
1	11 Fisher Avenue
2	3 Cook Bend
3	17 Nightingale Court

Tenant	
TenantID	FirstName
1	Thomas
2	Skylar
3	Huong
4	Ananya

1. For each table, which column is most suitable as a primary key?
2. Model the relationship between Home and Tenant, using a foreign key. That means, choose a name for the foreign key, and then choose the correct table to put the foreign key in. Leave the foreign key entries blank for now.
3. You are told that:
   - tenants 1 and 2 live in home 1;
   - tenant 3 lives in home 2; and
   - tenant 4 lives in home 3.

   Using this information, fill in the entries for the foreign key you created in question 2.

Solutions to Exercise 1.5.2

1. HomeID and TenantID are suitable primary keys.
2. The foreign key must go in Tenant, since the relationship is one-to-many, with **one** home related to **many** tenants – i.e., the foreign key always goes on the 'many' side (see Section 1.4.1). A suitable name is HomeID, since the foreign key will point at the primary key of the Home table.
3. Adding the foreign key to Tenant:

Tenant		
TenantID	FirstName	HomeID
1	Thomas	1
2	Skylar	1
3	Huong	2
4	Ananya	3

Exercise 1.5.3

A European travel agent has a collection of ‘language immersion’ vacation packages. Each vacation is within one country. The VacationHistory table below, lists details of each vacation that a traveller has been on.

VacationHistory					
TravellerID	FirstName	Country	Language	StartDate	VacationID
1	Lennon	France	French	2018-01-14	1
1	Lennon	France	French	2017-05-23	1
1	Lennon	Spain	Spanish	2016-05-20	2
2	Viviana	France	French	2017-03-09	1
2	Viviana	Spain	Spanish	2018-03-22	2
2	Viviana	Germany	German	2012-11-10	3
3	Zhang	Germany	German	2018-12-31	3

1. Can TravellerID be a primary key of VacationHistory?
2. Can TravellerID and VacationID, combined, be a primary key?
3. Are there any redundant data in VacationHistory? Give examples.
4. Is there a better way to model this dataset, to avoid redundancy? How many tables would we need? Write down the tables and their data.

Solutions to Exercise 1.5.3

1. TravellerID cannot be a primary key, since TravellerID is not unique in VacationHistory.
2. The combination of TravellerID and VacationID is not unique (see the top two rows of VacationHistory), so it cannot be a primary key.
3. The table does contain redundant data. For example, the name ‘Lennon’ is repeated multiple times.
4. There are many ways to model this data. One better approach is to model this as a many-to-many relationship between travellers and their vacations. For this, we need 3 tables: Traveller, VacationRecord and Vacation, given below.

Vacation		
VacationID	Country	Language
1	France	French
2	Spain	Spanish
3	Germany	German

Traveller	
TravellerID	FirstName
1	Lennon
2	Viviana
3	Zhang

VacationRecord		
TravellerID	VacationID	StartDate
1	1	2018-01-14
1	1	2017-05-23
1	2	2016-05-20
2	1	2017-03-09
2	2	2018-03-22
2	3	2012-11-10
3	3	2018-12-31

Chapter 2

Basic SQL queries

2.1 What is SQL? Or, which SQL is it?

SQL is a language that allows us to define and manipulate databases. At the time of its inception in a laboratory at IBM in the 1970s, SQL was called SEQUEL, standing for Structured English QUEry Language. Somewhere down the line, it underwent a rebranding and is now called Structured Query Language (SQL), though it is still commonly pronounced 'sequel.'

SQL is a standard

One of the great things about SQL, is that it's more than just a language. It's also a *standard*. Since 1987, SQL has been a standard of both the American National Standards Institute (ANSI), and the International Organisation for Standardisation (ISO). This means, every SQL database management system (including Oracle, MySQL, T-SQL, SQLite, PostgreSQL, etc), implements some number of the same basic SQL standards, in some way or another. I will refer to these different implementations as '**dialects**' of SQL. Each dialect of SQL differs enough from all the others that, if you wanted to switch from one dialect to another, you would have to spend some time independently learning the new syntax. However, in theory, since all these dialects implement the same SQL standards, it shouldn't take you long. Likewise, if a company decides to invest in a new database management system, then it is much easier for them to switch from one SQL dialect to another than if they changed to an entirely different standard (i.e., stopped using SQL entirely).

The different dialects of SQL also add new features, on top of the SQL standard, that make life significantly easier to use when writing code. In the interests of variety and understanding, we'll use *two* dialects in this book. We'll use one proprietary dialect, T-SQL, owned by Microsoft corporation, and one free (open source) dialect, MySQL, managed by Oracle corporation. Both dialects are hugely popular, being among the top three database management systems globally.

Of course, *most of the code we learn will be SQL standard*, meaning it will work on both T-SQL and MySQL. Whenever we introduce something peculiar to only one dialect of SQL (e.g., to T-SQL), we'll make clear what the alternative is (if it exists) in the other dialect (e.g., in MySQL). If we don't mention T-SQL or MySQL, it means what we are introducing something that works in both dialects (i.e., a part of the standard).

What is a query?

Tipping our hats to English grammar terminology, the words '**clause**' and '**statement**' are frequently used to describe parts of the SQL language. A SQL clause is the smallest logical component of a SQL statement that lets you filter or customise how you want your data to be manipulated or returned to you. Examples are `SELECT`, `FROM`, `WHERE`, `GROUP BY`, `HAVING` and `ORDER BY`. Clauses become parts of *statements*. A SQL statement is somewhat comparable to an English language sentence. It contains one or more clauses. For example, the statement `SELECT FirstName FROM Friends` returns the first names of all my friends. In this chapter, we'll introduce and visualise the inner workings of the basic fundamental clauses in SQL, using them to build basic fundamental statements. The action of using statements to request data or information from a database is called a **query**.

Formal languages

Technically, SQL clauses are based on a pair of formal mathematical languages, called *relational algebra* and *relational calculus*. We won't be covering the mathematics in this course, since these are mostly the domain of computer scientists and theoreticians. That which we will be learning, SQL, is a practical *engineering approximation* to these formal languages. It may make us sound fancy to name-drop a couple of formal languages, but in the end you will see that all we are learning is a collection of fairly intuitive ways to chop tables up and recombine them into new tables.

2.2 Retrieving rows and columns

Our database is full of tables. A query is designed to go into the database, chop up tables, join them together, potentially aggregate or transform the results, and return to us a single table. A query never returns more than one table. Typically, we want the query to give us certain columns and/or rows of a table. We're going to start by retrieving whole columns.

2.2.1 SELECT and FROM

The FROM clause lets you specify a table that you want to access. For this reason, we're going to use FROM in almost every query we write. It is never used on it's own. It almost always appears below the SELECT clause.

What is the SELECT clause? The SELECT clause lets you retrieve the *columns* of a table – it lets you *select* columns. For example:

```
SELECT FirstName, FavColour
FROM Friends;
```

FROM Friends

Friends			
FriendID	FirstName	LastName	FavColour
1	X	A	red
2	Y	B	blue
3	Z	C	NULL

⇓

SELECT FirstName, FavColour

Friends			
FriendID	FirstName	LastName	FavColour
1	X	A	red
2	Y	B	blue
3	Z	C	NULL

⇓

RESULT

FirstName	FavColour
X	red
Y	blue
Z	NULL

Figure 2.1: The SELECT and FROM clauses.

The SQL syntax is designed to try to mimic the English language a bit. This means that the order in which you *write* clauses is not necessarily the same order that you would think about *doing* them procedurally. This can lead to a fair bit of confusion for people who have done some programming in other languages, where the order that things are written matches the order that things are actually done. For example, in the above query, the FROM clause is executed *first*, bringing up the Friends table, and the SELECT clause is executed *second*, chopping out the FirstName and FavColour columns. This order is reflected in Figure 2.1.

If we want to display all columns of a table, then we need to select all the columns with SELECT. To save time, we can use the * keyword, which is equivalent to typing all the column names separated by commas. The query below returns the entire Friends table.

```
SELECT *
FROM Friends;
```

When using SELECT, it is good practice to avoid using the * keyword at all, even when you want to select all columns of a table. This is because, in practice, you might end up writing other code that depends on your SELECT clause returning a fixed set of columns (maybe even in a fixed order). As we'll see later, it's possible for the structure of a table to be changed in the database (e.g., by adding or dropping columns), which would alter the result of calling SELECT *, and possibly break any code that depends on the originally intended result. We will, however, often use SELECT * in this course, for convenience and brevity.

When selecting attributes with SELECT, it's also easy to rename columns in the result table, using the AS keyword. For example, we can write

```
SELECT FirstName AS My_friends_name, FavColour AS Their_fav_colour
FROM Friends;
```

This would give the result:

RESULT	
My_friends_name	Their_fav_colour
X	red
Y	blue
Z	NULL

Figure 2.2: Renamed columns using the AS keyword

2.2.2 SELECT with reserved keywords and quoting

There are lists of reserved keywords in both the T-SQL documentation and the MySQL documentation. These are words like select, that should not be used for things like column names. If you're forced to use these keywords as column names, for whatever reason, then you can 'quote' them in square brackets (for T-SQL) or backticks (for MySQL). The backtick character is `, which is located to the left of the number 1 on most keyboards. Quoting also allows you to use spaces and special characters in column names:

```
-- This demonstrates a quoted column name in T-SQL
SELECT [My column] FROM MyTable;

-- This demonstrates a quoted column name in MySQL
SELECT `My column` FROM MyTable;
```

So if, for some twisted reason, you decide to name a column `select` instead of `My_friends_name`, then you could write:

```
-- A valid but terrible idea in T-SQL
SELECT FirstName AS [select]
FROM Friends;

-- A valid but terrible idea in MySQL
SELECT FirstName AS `select`
FROM Friends;
```

2.2.3 ORDER BY

We can use the `ORDER BY` clause at the end of a query, simply to order the rows of the result.

```
SELECT *
FROM Friends
ORDER BY FriendID;
```

The above will just return the Friends table, exactly as in Figure 1.4. This is because the Friends table is already ordered by FriendID. To reverse the order, we can use the keyword `DESC`, specifying a descending order:

```
SELECT *
FROM Friends
ORDER BY FriendID DESC;
```

RESULT			
FriendID	FirstName	LastName	FavColour
3	Z	C	NULL
2	Y	B	blue
1	X	A	red

Figure 2.3: Friends, in descending order of FriendID

When ordering by strings of letters, like FirstName, LastName or FavColour, the order is alphabetical:

```
SELECT *
FROM Friends
ORDER BY LastName DESC;
```

The above query returns the table in Figure 2.3, since Figure 2.3 already happens to be sorted in descending order of LastName, alphabetically.

Character strings can also contain numbers and other non-alphabetic symbols. When the 'alphabetic' order is extended to cover non-alphabetic symbols, it is called the a 'lexicographic' order. This order can cause some confusion when character strings containing numbers are ordered. For example, consider the following table, Numbers:

Numbers	
Num	NumString
111	'111'
31	'31'
32	'32'
211	'211'

Figure 2.4: A table of numbers and their character string representations.

In Figure 2.4, I've used quote marks to clarify that NumString stores the numbers as strings, rather than as numbers. Let's order by Num:

```
SELECT *
FROM Numbers
ORDER BY Num;
```

RESULT	
Num	NumString
31	'31'
32	'32'
111	'111'
211	'211'

Figure 2.5: The Numbers table, ordered by Num.

Compare the above to what happens when we order by NumString:

```
SELECT *
FROM Numbers
ORDER BY NumString;
```

RESULT	
Num	NumString
111	'111'
211	'211'
31	'31'
32	'32'

Figure 2.6: The Numbers table, ordered by NumString.

2.2.4 Data types

So far, we haven't thought much about **data types**. Every column in a relational database has a data type associated with it, and the data type dictates the kinds of data that can be stored. For example, the NumString column in the Numbers table (Figure 2.6) holds character strings, while the Num column holds numbers.

Relational database structures are, in the most basic sense, a collection of related tables. We know each table has columns, and the tables are related by primary/foreign key pairs. If you need a refresher on primary and foreign keys, we covered them in Section 1.4. What we haven't covered, is the wide range of possible data types. Data types are specified whenever a new SQL table is created. Data types dictate the kinds of data that can be stored in a table. Some of the most common data types are given below, in Table 2.1.

Table 2.1: Some of the most common data types that work in both T-SQL and MySQL.

Data type	**Description**
`INT`	Positive or negative whole numbers (integers)
`DECIMAL(p,s)`	Decimal numbers (with precision p and scale s)
`FLOAT`	Approximate decimal numbers (floating point)
`CHAR`	Fixed length character strings
`VARCHAR(n)`	Variable length character strings (maximum length n)
`DATE`	Dates (`YYYY-MM-DD`)
`TIME`	Times (`HH:MM:SS`)
`DATETIME`	Date and time (`YYYY-MM-DD HH:MM:SS`)

We will learn more about all of the data types in Table 2.1, as we progress through the course. The two that I want to comment on here are `DECIMAL(p,s)` and `VARCHAR(n)`. The `DECIMAL(p,s)` data type holds any decimal number with a maximum of p digits, and with s digits after the decimal place. So, for example, the largest number that can be stored in `DECIMAL(5,2)` is 999.99; it can store the number 0.12, but it cannot store 0.123. The `VARCHAR(n)` data type can hold any string of characters, with a maximum length of n. So, `VARCHAR(5)` can hold the word 'smart', or the string 'a12', but it can't hold the word 'smarty'.

2.2.5 Transforming entries with functions

Before entries are used in a query, they can be transformed. For example, a decimal number might be rounded up or down; the month or year might be extracted from a date; or, the initials might be extracted from a person's name. **Scalar functions** are a great set of tools for transforming entries.

We'll now see our first (and probably most important) example of a scalar function: `CAST`. This function allows us to change the data type of a column before using it. Recall, from Section 3.5.3, that ordering the Numbers table by NumString resulted in a lexicographic (or alphabetic) order. That's because NumString is a character (or string) data type. However, knowing that NumString contains strings of numbers, we can convert those strings to a number data type, such as `INT` (i.e., whole numbers):

```
SELECT NumString
FROM Numbers
ORDER BY CAST(NumString AS INT);
```

RESULT	
Num	NumString
31	'31'
32	'32'
111	'111'
211	'211'

Figure 2.7: The Numbers table, ordered by NumString as an `INT`.

Notice, in Figure 2.7, that the above query has ordered NumString using the numerical order, rather than the lexicographic order. Nice! Aside from ordering, there are many more cases where it can be beneficial to use `CAST` (and we'll see an important example in Section 3.2.2).

There are a large number of scalar functions, but three of the most useful categories of scalar functions are:

- mathematical functions;
- string functions; and
- date and time functions.

Next, I'll present some commonly used scalar functions, along with a few examples. I'll also provide some links to more of these functions, so you can expand your repertoire as you learn SQL. In most of these examples, I'll use this table of restaurant orders:

Orders			
OrderID	Item	Price	OrderDT
1	Boiled leaves	2.99	2021-12-31 15:13:00
2	Bow wow	15	2021-12-31 15:34:00
3	Cackleberry stew	32.55	2022-01-01 09:32:00
4	Mug of murk	4.40	2022-01-01 10:16:00

Figure 2.8: The Orders table.

Mathematical functions

Table 2.2: Three examples of mathematical functions.

Function	Description
SQRT	Square root
ROUND	Rounding
RAND	Generate random number

Should we desire, we can take the square roots of prices in the Orders table:

```
SELECT SQRT(Price) AS SquareRoot
FROM Orders;
```

RESULT
SquareRoot
1.7291616465790582
3.872983346207417
5.705260730238365
2.0976176963403033

Figure 2.9: The (long and ugly) square roots of prices, from Orders.

We can also 'nest' functions, one inside the other, to do a sequence of transformations on the same column. For example, since the output in Figure 2.13 is pretty hard to look at, we might want to round the numbers to only two decimal places. We can do that with the `ROUND` function:

```
SELECT ROUND(SQRT(Price),2) AS SquareRoot
FROM Orders;
```

RESULT
SquareRoot
1.73
3.87
5.71
2.1

Figure 2.10: The less ugly square roots of prices, from Orders.

Not all of the mathematical functions operate on columns. The `RAND` function, for example, just produces a random number between 0 and 1.

```
SELECT RAND();
```

The full list of T-SQL mathematical functions is available here, and the MySQL functions are available here.

String functions

Table 2.3: Two examples of string functions.

Function	Description
CONCAT	Concatenate columns
SUBSTRING	Extract characters

The `CONCAT` function (short for 'concatenation') can be used to combine multiple columns together, by merging their entries into a single string. We can also insert new characters into the strings:

```
SELECT CONCAT('The ', Item, ' cost ', Price) AS Summary
FROM Orders;
```

RESULT
Summary
The Boiled leaves cost 2.99
The Bow wow cost 15.00
The Cackleberry stew cost 32.55
The Mug of murk cost 4.40

Figure 2.11: Concatenating Item and Price, from Orders.

The `SUBSTRING` function can cut characters out of a string according to their position. For example, starting with the six-letter word 'amazed', we can use `SUBSTRING('amazed',2,4)` to cut out the letters in positions 2 to 4, returning the word 'maze'. Here's another example, on Orders:

```
SELECT SUBSTRING(Item, 1, 3) AS FirstThree
FROM Orders;
```

RESULT
FirstThree
Boi
Bow
Cac
Mug

Figure 2.12: The first three letters of all Items in Orders.

For a full list of T-SQL string functions, click here, and for MySQL string functions click here.

Date and time functions

Table 2.4: Four examples of date/time functions.

Function	Description
DAY	Extract the day (of the month)
MONTH	Extract the month
YEAR	Extract the year

The above date and time functions can be used to extract parts from date or time data. For example:

```
SELECT DAY(OrderDT) AS TheDay, MONTH(OrderDT) AS TheMonth
FROM Orders;
```

RESULT	
TheDay	TheMonth
31	12
31	12
01	01
01	01

Figure 2.13: The day and month, from Orders.

The list of T-SQL date and time functions is here, and the MySQL date and time functions are here. Note, by following these links, you can also find functions to return the current date and time.

2.2.6 WHERE

While SELECT specifies which columns to return, the WHERE clause specifies which rows to return. The returned rows are chosen based on whether they meet a **search condition**. A search condition is a logical statement that evaluates to either true or false, for any given row. For example, the search condition 1 = 1 will always be true.

```
SELECT *
FROM Friends
WHERE 1 = 1;
```

The WHERE clause in the above query is pointless because it does not exclude any rows. In essence, the role of a WHERE clause is to *exclude* rows. If no rows are to be excluded then it would be neater to avoid writing the clause, since our query would return all rows of the table just the same. The act of excluding rows of a table based on meeting a search search condition is often referred to as **filtering**.

Search conditions can get fairly complicated, to the point where they can, and often do, have whole separate queries nested inside them (but we'll open that can of worms later). Simple search conditions compare two expressions via a **logical operator**, such as the symbols = (equals), < (less than), or <= (less than or equal to). We give more details on logical operators in Section 2.4.1, and more on search conditions in Section 2.4.4.

To create a more useful search condition than 1 = 1, we can include the name of an attribute (i.e., column) as one of the expressions. For example, the search condition `FavColour = 'red'` evaluates to `true` for every row that has `'red'` in the FavColour column.

```
SELECT FirstName, LastName
FROM Friends
WHERE FavColour = 'red';
```

Recall that clauses are not executed, in practice, in the same order that they appear in the SQL syntax. The first clause to be executed is usually `FROM`. The last clause to be executed is usually `SELECT`.

`FROM Friends`

Friends			
FriendID	FirstName	LastName	FavColour
1	X	A	red
2	Y	B	blue
3	Z	C	NULL

$\Downarrow$

`WHERE FavColour = 'red'`

Friends			
FriendID	FirstName	LastName	FavColour
1	X	A	red
2	Y	B	blue
3	Z	C	NULL

$\Downarrow$

`SELECT FirstName, LastName`

ID	FirstName	LastName	FavColour
1	X	A	red

$\Downarrow$

RESULT

FirstName	LastName
X	A

Figure 2.14: The `WHERE` clause.

Remember, a table represents some **entity** that you want to describe in the miniworld. Its columns are attributes of the entity, and its rows represent different instances of the entity. So, using `SELECT` and `WHERE`, we can make queries like “give me all entries for some set of attributes for all instances of some entity that satisfy some condition". This could be, "give me all flight arrival and departure times for flights leaving Melbourne and arriving in New Zealand on the 20th of November." But, with what we've learned, we can only filter based on the rows of the table that is referred to in the FROM clause. For example, the following would produce an error:

```
SELECT FirstName
FROM Friends
WHERE PetName = 'Chikin';
```

```
Msg 207, Level 16, State 1, Line 1 Invalid column name 'PetName'.
```

Figure 2.15: Error message printed because PetName is not in the Friends table.

The above query produces an error, because PetName is not located in the Friends table. So, what do we do if the pieces of information we need are located in different tables? We will use the JOIN clause (in Section 2.3).

2.2.7 Order of execution and WHERE

In some of the previous sections, I briefly mentioned that the order of execution of the clauses in a SQL query does not necessarily match the order that they are written. This is important to understand, because it can affect the kinds of queries we are able to write. For example, recall the query that produced Figure 2.2:

```
SELECT FirstName AS My_friends_name, FavColour AS Their_fav_colour
FROM Friends;
```

It's often tempting to try to make use of an alias, like Their_fav_colour, in the WHERE clause:

```
SELECT FirstName AS My_friends_name, FavColour AS Their_fav_colour
FROM Friends
WHERE Their_fav_colour = 'blue';
```

```
Error 1054: Unknown column 'Their_fav_colour' in 'where clause'.
```

Figure 2.16: Error produced when an alias from SELECT is used in WHERE.

The above error is produced because the alias Their_fav_colour doesn't exist yet when the WHERE clause executes. This is because SELECT is executed last in the above query. Of the queries we've learned so far, the order of execution is: FROM, WHERE, SELECT, and finally ORDER BY. You may not always remember the exact order of execution, as a beginner, but if you pay attention to error messages (like that in Figure 2.16) then they will often nudge you in the right direction.

2.3 Joining tables

We've learned to chop up tables with the `SELECT` and `WHERE` clauses, and now we're going to learn to join them together. Any two tables can be joined together, but, to be joined in a reasonable way, the tables must be *related*. As we saw in Section 1.4, related tables need columns with shared entries to tell us which rows belong where. When we write a `JOIN`, we need to specify the pairs of columns that should be used to match the rows of one table to the rows of the other.

2.3.1 `JOIN` and alias

To begin, let's join the Friends and Pets tables. This join was visualised way back in Figure 1.8, where we performed the join by matching the rows of the primary key column, `Friends.FriendID`, with the rows of the foreign key column, `Pets.FriendID`. We state this requirement in SQL using the clause `ON`, with the join condition `Friends.FriendID = Pets.FriendID`.

```
SELECT FirstName, PetName
FROM Friends JOIN Pets ON Friends.FriendID = Pets.FriendID;
```

`FROM Friends JOIN Pets ON Friends.FriendID = Pets.FriendID`

Pets			
PetID	PetName	...	FriendID
1	Chikin		2
2	Cauchy		3
3	Gauss		3

Friends		
FriendID	FirstName	...
1	X	
2	Y	
3	Z	

$\Downarrow$

`SELECT FirstName, PetName`

FriendID	FirstName	LastName	FavColour	PetID	PetName	PetDOB
2	Y	B	blue	1	Chikin	24/09/2016
3	Z	C	NULL	2	Cauchy	01/03/2012
3	Z	C	NULL	3	Gauss	13/02/2017

$\Downarrow$

RESULT

FirstName	PetName
Y	Chikin
Z	Cauchy
Z	Gauss

Figure 2.17: The `JOIN` clause

The above can be achieved more succinctly, using aliases. An **alias** is a single letter, or a short word, that allows us to refer to a table without having to write its full name. In the following, we choose the letters `F` and `P` as aliases for `Friends` and `Pets`, respectively.

```
SELECT *
FROM Friends F JOIN Pets P ON F.FriendID = P.FriendID;
```

In SQL, there sometimes exists optional keywords that have no effect on the meaning of a query. For example, when writing an alias, we can optionally preceded the alias with the word `AS`, as in:

```
SELECT *
FROM Friends AS F JOIN Pets AS P ON F.FriendID = P.FriendID;
```

So, sometimes there are lots of ways to do the same thing! In fact, the following are two other equivalent ways to write the above query.

```
SELECT *
FROM Friends F INNER JOIN Pets
ON F.FriendID = Pets.FriendID;
```

```
SELECT *
FROM Friends F, Pets P
WHERE F.FriendID = P.FriendID;
```

The last approach is called an *implicit join*, because the `JOIN` command is not written explicitly but signalled by the comma between `Friends F` and `Pets P`, and because the `WHERE` clause, instead of the `ON` clause, has been used to specify the join condition `F.FriendID = P.FriendID`. We will avoid using the implicit join, because it is less clear and is no longer part of the SQL standard.

Example 2.3.1. To get a better understanding of how tables are joined, let's look at another example. Here are two tables with columns A, B, C, D and E:

Table1		
A	B	C
1	Ignorance	is
2	War	is
3	Freedom	is
4	Friendship	is

Table2		
D	E	A
slavery.	3	1
weakness.	4	2
strength.	1	3
peace.	2	4

Figure 2.18: A couple of harmless tables to join.

If we join the tables, by comparing the primary key (`Table1.A`) with the associated foreign key (`Table2.A`), then we get the intended table:

```
SELECT * FROM Table1 T1 JOIN Table2 T2 ON T1.A = T2.A
```

A	B	C	D	E
1	Ignorance	is	slavery.	3
2	War	is	weakness.	4
3	Freedom	is	strength.	1
4	Friendship	is	peace.	2

Figure 2.19: The correct way to join a couple of harmless tables.

But if we instead mistakenly join the tables using the wrong join condition, say, `Table1.A = Table2.E`, we get

```
SELECT * FROM Table1 T1 JOIN Table2 T2 ON T1.A = T2.E
```

A	B	C	D	A
1	Ignorance	is	strength.	3
2	War	is	peace.	4
3	Freedom	is	slavery.	1
4	Friendship	is	weakness.	2

Figure 2.20: A catastrophic join mistake, leading to dystopian nightmare.

Great, so we know how to join tables now, and Figure 2.20 warned us about the potentially bleak results of choosing the wrong join condition. But, what if we want our query to keep all rows of one table, even if they don't match any rows from the other table? For this, we need `LEFT JOIN`.

2.3.2 `LEFT JOIN` and `RIGHT JOIN`

You're probably hanging out to practice your `JOIN` skills now. But first, let's take a quick look at an important extension that will come in handy quite often in practice: the `LEFT JOIN`. Refer back to the `JOIN` clause execution diagram (Figure 2.17), and notice that the final table excludes any friends that have no pets. If you ever want to join two tables, but you want to keep all the records from one of them, then `LEFT JOIN` is your pal.

```
SELECT FirstName, PetName
FROM Friends LEFT JOIN Pets ON Friends.FriendID = Pets.FriendID;
```

The `LEFT JOIN` in the above query keeps everything from Friends (the table appearing to the left – hence the word 'left'). To achieve this, it will return `NULL` values in place of any corresponding missing attributes from Pets. So, since X has no pets, the query returns X's name, but inserts `NULL` in the position where X's PetName would go if they had a pet:

FirstName	PetName
X	NULL
Y	Chikin
Z	Cauchy
Z	Gauss

Figure 2.21: My friends, and the names of their pets

In the wild, you might see `LEFT JOIN` written as `LEFT OUTER JOIN`, though it does the same thing as `LEFT JOIN` (so the word 'outer' is redundant). You might also see a `RIGHT JOIN` appear in the wilderness, which does the same thing as a left join, but on the right side instead of the left side. In other words, the two queries below are equivalent.

```
-- Placing Friends on the left, and doing a 'LEFT JOIN'
SELECT FirstName, PetName
FROM Friends LEFT JOIN Pets ON Friends.FriendID = Pets.FriendID;

-- Placing Friends on the right, and doing a 'RIGHT JOIN'
SELECT FirstName, PetName
FROM Pets RIGHT JOIN Friends ON Friends.FriendID = Pets.FriendID;
```

2.4 Building and using search conditions

In this final section of the chapter, you'll learn how to create more powerful search conditions in your `WHERE` clause, by making use of logical and comparison operators. You'll also learn about wildcards, the perils of `NULL` values, and the `CASE WHEN` expression.

2.4.1 Comparison operators

In the statement `WHERE Gender = 'F'`, the symbol = is called a comparison operator. **Comparison operators** compare two or more expressions, and return either `TRUE`, `FALSE` or `NULL` (unknown). Examples of comparison operators are:

Table 2.5: The standard comparison operators, with examples that return TRUE.

Operator	Description	Returns TRUE
=	equal	1 = 1
<	less than	0 < 1
<=	less than or equal	1 <= 1
>	greater than	1 > 0
>=	greater than or equal	1 >= 1
!=	not equal	0 != 1
!<	not less than	1 !< 0
!>	not greater than	0 !> 1

2.4.2 Logical operators

Logical operators are used to combine or alter the results of comparison operators (that is, to combine or alter *logical* results, like `TRUE`, `FALSE` or `NULL`). For example, consider this:

```
(Gender = 'M') AND (Age > 35)
```

For a 25 year old male, `Gender = 'M'` evaluates to `TRUE`, and `Age > 35` evaluates to `FALSE`. So, the above will become

```
(TRUE) AND (FALSE)
```

At this point, the `AND` operator does its thing, converting the above statement into one logical value. You can use the *truth tables* in Figure 2.22, to find out the answer.

Logical operators include the familiar words `AND`, `OR` and `NOT`. They combine two or more instances of `TRUE`, `FALSE` or `NULL`, and produce new instances of `TRUE`, `FALSE` or `NULL`. This should become more clear while looking at the truth tables below.

AND				
true	AND	true	=	true
false	AND	true	=	false
true	AND	false	=	false
false	AND	false	=	false

OR				
true	OR	true	=	true
false	OR	true	=	true
true	OR	false	=	true
false	OR	false	=	false

NOT			
NOT	true	=	false
NOT	false	=	true

Figure 2.22: Truth tables for logical operators.

2.4.3 Other operators

Sometimes, the word 'logical operator' is used to refer to other operators that are perhaps more correctly referred to as 'condition operators'. The nomenclature differs a little between the documentation found in different dialects of SQL, which can be a little confusing if you get pedantic about language. So, we'll just bundle all of these tools into a table and call them *other operators*.

Table 2.6: The other standard SQL operators.

Operator
ALL
ANY
SOME
EXISTS
BETWEEN
IN
LIKE

We're not going to learn all of these right now. By the time you're done with Section 4.1, you'll be able to read SQL documentation and figure out what they do yourself. Some of them, we will look at in more detail soon. Comparison operators, logical operators, and other operators, all go hand-in-hand with search conditions, so we see them more in the next section.

2.4.4 Search conditions (and operator precedence)

In the statement `WHERE Gender = 'F'`, the bit `Gender = 'F'` is called a **search condition**. We use search conditions to exclude rows from our query results that do not satisfy the search condition. The search condition `Gender = 'F'` will make sure that our results only include rows where the column named Gender has the entry `'F'`. We can combine multiple logical and comparison operators in a single search condition. Take this example:

```
WHERE (Gender = 'M') AND (Age > 35)
```

The above search condition will exclude every row not representing a male over the age of 35. We have used the brackets to make the order of operations clearer. You don't always have to use these brackets, but it is often good for clarity. Search conditions can get about as complicated as you like. For example, the below will ensure your query results include only people who are both female and over 35, or both male and under 25.

```
WHERE ((Gender = 'F') AND (Age > 35)) OR ((Gender = 'M') AND (Age < 25))
```

Operator precedence

A quick route to great suffering is to forget about operator precedence when writing a search condition. Operator precedence refers to the order that governs which operators are executed first in a search condition.

Table 2.7: Some of the operator precedence rules in SQL.

Precedence	Operators
1	Anything in round brackets
2	=,<,>,<=,>=,!=,!<,!> (comparison operators)
3	NOT
4	AND
5	OR, ALL, ANY, SOME, EXISTS, BETWEEN, IN, LIKE

Notice from Table 2.7, that `AND` is evaluated <u>before</u> `OR`. This can cause some sneaky errors. Consider, for example, the following two search conditions:

```
-- this one evaluates to FALSE
1 = 2 AND (2 = 2 OR 3 = 3)

-- but this one evaluates to TRUE
1 = 2 AND 2 = 2 OR 3 = 3
```

More concretely, consider the following two search conditions:

```
-- matches 50 or 60 year old females only
Gender = 'F' AND (Age = 50 OR Age = 60)

-- matches 50 year old females, or anyone aged 60
Gender = 'F' AND Age = 50 OR Age = 60
```

Example 2.4.1. Table 2.7 will allow us to evaluate the following complicated (and poorly written) search condition. You may also want to refer back to the truth tables for logical operators, in Figure 2.22.

```
1 < 2 AND 2 = 2 OR 1 = 1 AND NOT (TRUE OR FALSE)
```

We can evaluate the above in the following steps:

1. Starting with 'anything in round brackets', and noting that the expression `(TRUE OR FALSE)` evaluates to `(TRUE)`, we have:
   ```
   1 < 2 AND 2 = 2 OR 1 = 1 AND NOT (TRUE)
   ```

2. Next, evaluating all the comparison operators gives:
   ```
   TRUE AND TRUE OR TRUE AND NOT TRUE
   ```

3. Now, evaluating `NOT`, noting `NOT (TRUE)` evaluates to `FALSE`, gives:
   ```
   TRUE AND TRUE OR TRUE AND FALSE
   ```

4. Next in line is `AND`, giving:
   ```
   TRUE OR FALSE
   ```

5. Finally, evaluating the `OR` operator gives:
   ```
   TRUE
   ```

Queries within search conditions

It is also possible to include whole queries within search conditions, as nested queries. Soon, in Section 3.4.1, we will look at this:

```
SELECT Name
FROM RandomPeople
WHERE Gender IN (SELECT Gender
                 FROM RandomPeople
                 GROUP BY Gender
                 HAVING AVG(Age) > 40);
```

There's a lot going on above. We haven't yet learned the clauses `GROUP BY`, or `HAVING`, or anything about `AVG`. But, if you squint at the query, you can see there is a *whole second query* that appears *nested* in brackets after the keyword `IN`. That whole query is actually part of the search condition - so, that whole *nested query* is actually part of the `WHERE` clause. We'll find out nested queries work later, in Section 3.4.

2.4.5 Wild cards in search conditions

A wild card (the `%` symbol in SQL), allows us to match a variety of character strings in our search conditions. For example, we can ask for "all words beginning with the letter A." The wild card is most often used with the operator `LIKE`. If we want to exclude all people whose name does not start with 'A', then we can use

```
WHERE Name LIKE 'A%'
```

We aren't restricted to single letters, or the start of words, either. If we want to exclude all people without the letters 'mit' appearing *anywhere* in their name, then we can use

```
WHERE Name LIKE '%mit%'
```

This process is called *pattern matching*. Along with `%`, standard SQL provides another pattern matching character, `_` (the underscore). The underscore matches *any single character*. So, the statement

```
WHERE Name LIKE 'Bi___'
```

having three underscores, will match any name starting with 'Bi' and being exactly five characters in length.

Later, we will see that the T-SQL dialect adds additional wildcard characters, and that MySQL (but not T-SQL) also has its own additional version of the much more powerful *regular expression* pattern matching, which introduces many more wildcard tools, allowing MySQL programmers to perform very imaginative pattern matching. If you're keen to look at that now, more details on these can be found under the T-SQL documentation for `LIKE`, and under the MySQL documentation for pattern matching (but these may go a little over your head at this stage).

2.4.6 The perilous `NULL`

The value `NULL` needs particularly special attention when working with comparison operators. The `NULL` value represents an entry that is either unknown or does not exist. Take a moment to decide what you think will be returned if I use the symbol = to compare two `NULL` values, like this:

```
NULL = NULL
```

It doesn't return `TRUE`! In fact, it returns `NULL`. This makes sense, when you realise that `NULL` literally means 'unknown' or 'does not exist,' and that every `NULL` value is treated as distinct from every other `NULL` value (that is, no two unknowns are necessarily the same). In fact, the same kind of thing happens when we compare anything at all to `NULL`. Take, for example

$$10 = \texttt{NULL}.$$

The above operation returns `NULL`. This also makes sense, because we cannot be sure whether the unknown thing, represented by the `NULL`, is actually equal to 10 or not, so we have to return `NULL` to indicate that the result is unknown.

This behaviour of `NULL` with comparison operators can lead to some particularly sneaky mistakes, in SQL code, that can produce incorrect results without ever causing any errors (so you'll possibly never notice the mistake). The solution is often to use the standard SQL clause `IS NULL`, as we will see in the following example.

Example 2.4.2. Perhaps the most common mistake that I've seen creep up with `NULL` values, begins with some variation of the following. We want to retrieve all Friends whose favourite colour is not blue, and we implicitly expect the result to include all Friends whose favourite colour is `NULL`. The mistake is that we write:

```sql
SELECT FirstName, FavColour
FROM Friends
WHERE FavColour != 'blue';
```

Think about it like this: the `WHERE` clause will only keep rows where the search condition `FavColour != 'blue'` evaluates to `TRUE`. Now, since the expression `NULL != 'blue'` evaluates to `NULL`, the rows with FavColour `NULL` are discarded, alongside the rows with FavColour blue. The result is:

RESULT	
FirstName	FavColour
X	red

Figure 2.23: Friends whose FavColour is definitely not blue (so, excluding `NULL` values).

If we want to include NULL values in the result, we can make the search condition return TRUE for NULL values, by making use of the IS NULL clause:

```
SELECT FirstName, FavColour
FROM Friends
WHERE FavColour != 'blue' OR FavColour IS NULL;
```

2.4.7 CASE WHEN

The CASE WHEN expression is a very flexible way to use search conditions to transform the values of entries within a query. Later, we'll see that CASE WHEN can be used in many places. For now, the SELECT clause is the easiest place to see it in action. Here is the syntax (which will be explained below):

```
CASE WHEN search_condition THEN true_output ELSE false_output END
```

In the above, search_condition represents any search condition we want to use. We have to replace the words true_output and false_output with whatever data we want to return. If the search condition evaluates to TRUE, then true_output will be returned; otherwise, false_output will be returned. For example, using the search condition FirstName = 'X', we could write:

```
CASE WHEN FirstName = 'X' THEN 'Dr. X' ELSE FirstName END
```

The above expression will return 'Dr. X' for anyone with FirstName X. Otherwise, it will just return their FirstName. Here it is within a query:

```
SELECT
    CASE WHEN FirstName = 'X' THEN 'Dr. X' ELSE FirstName END
    AS NewNames
FROM Friends;
```

RESULT
NewNames
Dr. X
Y
Z

Figure 2.24: Using CASE WHEN to rename X

The CASE WHEN syntax is very verbose, so I had to move it (as well as the alias, AS NewNames) to a new line. New lines don't affect the behaviour of the query at all – they are just there for aesthetics. Hopefully, it's still easy to see that `CASE WHEN` is positioned like any other column in the `SELECT` clause.

Example 2.4.3. As another example, let's return to the simple query that produced Figure 2.1. That query was just the following:

```
SELECT FirstName, FavColour
FROM Friends;
```

Now, suppose we want to alter the FavColour column, to return the word 'yes' when a friend has a favourite colour, and 'no' when the favourite colour is null. This means we'll be using the search condition `FavColour IS NULL`. Here is the query:

```
SELECT FirstName,
   CASE WHEN FavColour IS NULL THEN 'no' ELSE 'yes' END AS HasFavCol
FROM Friends;
```

RESULT	
FirstName	HasFavCol
X	yes
Y	yes
Z	no

Figure 2.25: Using `CASE WHEN` to replace FavColour

An awesome feature of `CASE WHEN` is that it can be extended to account for multiple cases. The syntax for three cases looks like this:

```
CASE WHEN search_condition_1 THEN output_1
      WHEN search_condition_2 THEN output_2
      WHEN search_condition_3 THEN output_3
      ELSE final_output END
```

For the above, the output will correspond to whichever is the first search condition to evaluate to `TRUE`. If none of them are `TRUE`, then `final_output` is returned. Here, we'll use multiple cases for renaming friends:

```
SELECT CASE WHEN FirstName = 'X' THEN 'Dr. X'
            WHEN FirstName = 'Y' THEN 'Prof. Y'
            ELSE FirstName END AS NewNames
FROM Friends;
```

RESULT
NewNames
Dr. X
Prof. Y
Z

Figure 2.26: Using CASE WHEN to rename X

2.5 Handwritten exercises

These exercises should all be done by hand (e.g., with pen and paper). No programming is required.

Exercise 2.5.1

This exercise should be done by hand (e.g., pen and paper). We'll practice using SELECT, FROM, CASE WHEN, ORDER BY and aliases. You're given the one table:

Alphanumeric		
Number	Letter	NumString
1	a	'34'
2	b	'121'

Note, the NumString column uses quote marks to indicate that the numbers are stored as strings rather than actual numbers.

1. Write down the result of the query below.

```
-- This is written in T-SQL syntax
SELECT T.Letter AS [Letter of Alphabet], T.Number AS Num
FROM Alphanumeric T;
```

2. Can you write the above query in MySQL syntax?
3. Fill in the blanks below, for the CASE WHEN syntax (described in Section 2.4.7), to do the following:
   - When Letter is 'a', change the entry to 'The letter A'
   - When Letter is 'b', change the entry to 'The letter B'
   - The letter column should be renamed to 'LETTERS'

```
SELECT T.Number,
       CASE WHEN ... THEN ...
            WHEN ... THEN ...
            END AS ...
FROM Alphanumeric T;
```

When finished filling in the blanks, write down the result of the query.

4. Write down the result of the following query.

```
SELECT *
FROM Alphanumeric
ORDER BY NumString DESC;
```

Note, `DESC` indicates the rows will be ordered from greatest to least (descending order).

5. The following query produces an error. Can you explain why?

```
SELECT FirstName AS MyFriendName
FROM Friends
WHERE MyFriendName = 'X';
```

Note that the order of execution of SQL clauses is not necessarily the same as the order they are written. The clause `FROM` executes before `WHERE`, which in turn executes before `SELECT`.

Solutions to Exercise 2.5.1

1. The result of the query is:

RESULT	
Letter of Alphabet	Num
a	1
b	2

2. The MySQL and T-SQL syntax taught in this course are rarely different, thanks to the SQL standard. However, this is one case where they differ. In MySQL, column names can be quoted using the backtick character instead of square brackets:

```
-- This is written in MySQL syntax
SELECT T.Letter AS `Letter of Alphabet`, T.Number AS Num
FROM Alphanumeric T;
```

3. Filling in the blanks gives:

```
SELECT T.Number,
      CASE WHEN T.Letter = 'a' THEN 'The letter is A'
           WHEN T.Letter = 'b' THEN 'The letter is B'
           END AS LETTERS
FROM Alphanumeric T;
```

The resulting table is:

RESULT	
Number	LETTERS
1	The letter is A
2	The letter is B

4. NumString is stored using strings rather than numbers, so the order is lexicographic. It follows that '34' is greater than '121'. So, Alphanumeric is already ordered by NumString:

RESULT		
Number	Letter	NumString
1	a	'34'
2	b	'121'

5. The query produces an error because the alias MyFriendName is produced in the `SELECT` clause, so it cannot be used in the `WHERE` clause (since `WHERE` executes before `SELECT`).

Exercise 2.5.2

This exercise should be done by hand (e.g., with pen and paper). We will practice joining two tables together, and using a simple `WHERE` clause. You are given the following two tables.

Home	
HomeID	Street
1	11 Fisher Avenue
2	3 Cook Bend
3	17 Nightingale Court

Tenant		
TenantID	FirstName	HomeID
1	Thomas	1
2	Skylar	1
3	Huong	2
4	Ananya	3

1. Is the relationship between Home and Tenant one-to-one, one-to-many, or many-to-many?
2. Write down the table produced by joining Home and Tenant, using the appropriate primary and foreign key pair.

3. Write down, by hand, the table produced by the following query.

```
SELECT FirstName, Street
FROM Home JOIN Tenant ON Home.HomeID = Tenant.HomeID
WHERE HomeID = 1;
```

Solutions to Exercise 2.5.2

1. The relationship is one-to-many.
2. The joined table is:

RESULT			
TenantID	FirstName	HomeID	Street
1	Thomas	1	11 Fisher Avenue
2	Skylar	1	11 Fisher Avenue
3	Huong	2	3 Cook Bend
4	Ananya	3	17 Nightingale Court

3. The query takes the result in answer 1, filters out any rows where HomeID is not equal to 1, and then selects only the FirstName and Street columns.

RESULT	
FirstName	Street
Thomas	11 Fisher Avenue
Skylar	11 Fisher Avenue

Exercise 2.5.3

With this exercise, we'll learn how to join 3 tables together. You should do this exercise by hand.

1. You are given the three tables below:

Table1	
A	X
1	$x1$
2	$x2$

Table2	
X	Y
$x1$	$y2$
$x2$	$y1$

Table3	
B	Y
1	$y1$
2	$y2$

a) Join Table1 to Table2, using column X from both tables as the primary/foreign key pair.

b) To join multiple tables together in SQL, we can simply chain multiple `JOIN` clauses, one after the other.

```
SELECT T1.A, T2.X, T2.Y, T3.B
FROM Table1 T1 JOIN Table2 T2 ON T1.X = T2.X
               JOIN Table3 T3 ON T2.Y = T3.Y;
```

Write down the result of the query above. **Hint:** you should take your answer from part 1, and join that to Table3, using column Y as the primary/foreign key pair.

c) Write down the result of the following query.

```
SELECT T1.A, T2.X, T2.Y, T3.B
FROM Table3 T3 JOIN Table2 T2 ON T3.Y = T2.Y
               JOIN Table1 T1 ON T2.X = T1.X;
```

2. You are given the three tables below.

PlayCount		
PetID	Count	FriendID
1	3	1
1	5	2
3	4	2

Friends		
FriendID	FirstName	...
1	X	
2	Y	
3	Z	

Pets		
PetID	PetName	...
1	Chikin	
2	Cauchy	
3	Gauss	

a) Is the PlayCount relationship (between Friends and Pets) one-to-one, one-to-many, or many-to-many?

b) Write down the result of joining the three tables.

Solutions to Exercise 2.5.3

1. The solutions are as follows.

a) The result of joining Table1 to Table2 is:

RESULT		
A	X	Y
1	$x1$	$y2$
2	$x2$	$y1$

b) The result of the 3-way join is:

RESULT			
A	X	Y	B
1	$x1$	$y2$	2
2	$x2$	$y1$	1

c) The result of this query is the same as part b, since it does not matter what order the tables are joined in, provided Table1 joins to Table2 and Table2 joins to Table3.

2. The solutions are as follows.
   a) The relationship is many-to-many.
   b) The result of joining the three tables is:

RESULT						
...	PetName	PetID	Count	FriendID	FirstName	...
	Chikin	1	3	1	X	
	Chikin	1	5	2	Y	
	Gauss	3	4	2	Y	

Exercise 2.5.4

We'll now join 3 tables together in a more realistic scenario. You will need to link information from one table, to information from another table, but you will not be able to do it without joining via a third table. You are given the following three tables.

Person							
P_ID	FName	LName	S_ID	BirthYr	Y_ID	Z_ID	E_CF
32	Bob	Smith	24	2004	2	E2	H2
1	Sam	Smith	12	2002	2	J8	I7
16	Ivy	Smith	32	1997	8	M5	66
5	Joy	Jones	NULL	1999	7	B4	32
9	Sky	Jones	NULL	2011	8	E3	9

Suburb			
S_ID	Name	PostCode	D_ID
24	Balwyn	3103	1
12	Glen	3146	1
32	Hawthorn	3122	3

Demographics					
D_ID	G_ID	M_ID	T_ID	StartBracket	EndBracket
3	32	3	4	50000	100000
1	1	7	39	150000	200000
2	4	2	38	100000	150000

We are told that P_ID is the primary key of the Person table, S_ID is the primary key of the Suburb table, and D_ID is the primary key of the Demographics table. Suppose we want to link a person's age to the average annual income bracket (StartBracket and EndBracket) of their suburb.

1. If we want to join 3 tables, how many primary/foreign key pairs do we need to use?
2. Which primary/foreign key pairs do we need to use to link a person's BirthYear to the income bracket of their suburb?
3. Write a query that produces a table with three columns: the BirthYr of each person, beside the StartBracket and EndBracket of the suburb they live in.
4. Write down the table produced by your solution to question 3.

Solutions to Exercise 2.5.4

1. We need to use two primary/foreign key pairs (four keys in total).
2. To link Person to Suburb, we need to use S_ID. To link Suburb to Demographics, we need to use D_ID.

3. This query produces the desired result:

```
SELECT P.BirthYr, D.StartBracket, D.EndBracket
FROM Person P JOIN Suburb S ON P.S_ID = S.S_ID
              JOIN Demographics D ON S.D_ID = D.D_ID;
```

4. The result of the query in solution 3 is:

RESULT		
BirthYr	StartBracket	EndBracket
2004	150000	200000
2002	150000	200000
1997	50000	100000

Exercise 2.5.5

In this exercise, we will practice working with search conditions. You may want to refer carefully to the operator precedence rules in Table 2.7. I will try to trick you into making mistakes regarding operator precedence and `NULL` values.

1. Write down what each of the following expressions evaluates to.
   a) `1 = 1 AND 'a' = 'a'`
   b) `NOT 0 < 1`
   c) `1 != 1 OR 2 = 2`
   d) `NULL != NULL`
   e) `NULL = NULL`
   f) `1 = 1 OR 2 < 3 AND 3 != 3`
   g) `NOT 1 > 2 AND 'blue' = 'green'`
2. Consider the following table of house sales.

HouseSales		
Suburb	Bedrooms	PriceThousands
Bundoora	4	550
Bundoora	2	700
Alphington	5	1200

   a) Write down the result of the following query (think carefully).

```
SELECT *
FROM HouseSales
WHERE Suburb = 'Bundoora'
  AND PriceThousands < 600 OR Bedrooms > 3;
```

   b) Write down the result of the following query (which *is* different).

```
SELECT *
FROM HouseSales
WHERE Suburb = 'Bundoora'
  AND (PriceThousands < 600 OR Bedrooms > 3);
```

3. Consider the following incomplete table of atomic masses.

Atoms		
Element	Num	Mass
Argon	18	39.948
Potassium	19	NULL
Calcium	NULL	NULL
Scandium	21	44.956

   a) Write down the result of the following query.

```
SELECT *
FROM Atoms
WHERE Element LIKE '%iu%' AND Mass IS NULL;
```

   b) Write down the result of the following query.

```
SELECT *
FROM Atoms
WHERE Num = NULL;
```

Solutions to Exercise 2.5.5

1. Here are the answers:
   a) `TRUE AND TRUE` becomes `TRUE`
   b) `NOT TRUE` becomes `FALSE`
   c) `FALSE OR TRUE` becomes `TRUE`
   d) `NULL != NULL` evaluates to `NULL`
   e) `NULL = NULL` evaluates to `NULL`
   f) Remember, `AND` must be evaluated before `OR`. So, the expression `TRUE OR TRUE AND FALSE` becomes `TRUE OR FALSE`, which is `TRUE`.
   g) Remember, `NOT` must be evaluated before `AND`. So, the expression `NOT FALSE AND FALSE` becomes `TRUE AND FALSE`, which is `FALSE`.
2. Here are the answers:
   a) Since `AND` is evaluated before `OR`, the search condition matches any house in Bundoora that is cheaper than 600 (thousand), or any house *at all* with more than 3 bedrooms. The result is:

RESULT		
Suburb	Bedrooms	PriceThousands
Bundoora	4	550
Alphington	5	1200

b) Due to the parentheses, `OR` is evaluated first. The result is:

RESULT		
Suburb	Bedrooms	PriceThousands
Bundoora	4	550

3. Here are the answers:
   a) The wild cards (`'%'`) are at both the beginning and end of the search string `'%iu%'`. So, the `LIKE` operator will match any Element containing ‘iu’. The result is:

RESULT		
Element	Num	Mass
Potassium	19	NULL
Calcium	NULL	NULL

   b) The search condition `Number = NULL` can never evaluate to `TRUE`, since `NULL = NULL` will evaluate to `NULL`. So, the result is an empty table:

RESULT		
Element	Num	Mass

Exercise 2.5.6

In this exercise you will practice using `LEFT JOIN`. You are given the following two tables.

Atoms		
Element	Num	Mass
Argon	18	39.948
Potassium	19	NULL
Calcium	NULL	NULL
Scandium	21	44.956

Components	
Molecule	Num
SO_3Ar	18
Ar·HCCH	18
ArCa	18
ArCa	20

1. We'll warm up with a regular join (also known as an inner join). Write down the result of the following query.

```
SELECT A.Element, A.Num, C.Molecule
FROM Atoms A JOIN Components C ON A.Num = C.Num;
```

2. Write down the result of the following query.

```
SELECT A.Element, A.Num, C.Molecule
FROM Atoms A LEFT JOIN Components C ON A.Num = C.Num;
```

Solutions to Exercise 2.5.6

1. The result of the inner join is:

RESULT		
Element	Num	Molecule
Argon	18	SO_3Ar
Argon	18	Ar·HCCH
Argon	18	ArCa

2. For a left join, every row from the left table (Atoms) is included in the result at least once. However, the inner join (from solution 1) is also present. The result is:

RESULT		
Element	Num	Molecule
Argon	18	SO_3Ar
Argon	18	Ar·HCCH
Argon	18	ArCa
Potassium	19	NULL
Calcium	NULL	NULL
Scandium	21	NULL

2.6 SQL editor exercises

This section gives some basic exercises to help you become familiar with your SQL editor. In the workshop, I provide a tutorial on how to use your editor. If you are not in the workshop, you should search online for a guide. There are no solutions to these questions.

1. Go to the set-up page on the course repository and follow the instructions to set up a free local MySQL or T-SQL database management system, and fill it with the data provided. This guide will also instruct you to choose and download an appropriate SQL editor for writing and executing code.

2. There are many SQL resources, but I personally like the excellent MySQL tutorial at www.selectstarsql.com. Read the front matter if you like, bookmark the tutorial, and come back to it later. I believe it's always important to learn the basics from more than one author. This particular tutorial also lets you execute MySQL code in the browser.

3. Search online for a simple syntax guide. For MySQL, I like the w3schools guide. For T-SQL, I like the one from dofactory. Don't spend too long on this. It's just to let you know they exist. Later, we'll also learn to read the proper T-SQL and MySQL documentation directly.

4. After completing the setup step (item 1 above), you have access to multiple databases. Use the directory tree in your SQL editor to begin investigating them. You can switch databases with the `USE` keyword. For example, to use the Sandpit database, execute:

```
USE Sandpit;
GO -- 'GO' is for T-SQL only, remove this line for MySQL.
```

5. In the Sandpit database you can find all the tables from these notes. T-SQL organises tables into 'schemas'. For example, Friends is in the Notes schema, so it is named Notes.Friends. MySQL, on the other hand, doesn't support schemas using the '.' symbol. So, in MySQL, I have named them with underscores instead. This means, the Friends table in the Notes schema is called Notes_Friends, in MySQL. This naming convention allows us to pretend MySQL has schemas. Use the directory tree to determine some of the names of other schemas.

6. Use the directory tree to figure out some of the table names and column names in the Notes schema. You may notice that some column names are slightly different to the ones in these notes. Why might that be? Any ideas?

7. Use your editor's interface to view the columns that are present in the Notes.Friends table. What do you see? Can you determine the data types of each column? Can you determine whether `NULL` values are allowed in each column? If you like, you can learn more about data types (and find the data types **varchar** and **int**) in the T-SQL or MySQL documentation (note, in MySQL, **int** is sometimes called **integer**). Don't spend too long on the docs now!

8. Open a new query tab. In the Sandpit database, execute the following query that selects all of the rows and columns of the Notes.Pets table.

```
SELECT *
FROM Notes.Pets;
```

9. SQL is not sensitive to empty spaces, upper-case/lower-case letters, or new lines. So, if you really want, you can write any query on one massive line in lower-case, or you can use no indentation, or YoU cAn EvEn WrItE iN sPoNgEbOb CaSe! This freedom can be a curse. Execute the following query.

```
                 select
firstNAME,lastNAME from NOTES.friends;
```

10. **Challenge question**. A number of exercises will be labelled 'challenge question'. You should not do these if you are a beginner programmer or short on time. For this challenge question, set up a GitHub repository for your solutions. Create a short README file, and add your SQL solutions to this repository as you go. Consider using GitHub Desktop. If you're brave, make the repository public, and share the link with me, so future students can compare their work to yours!

2.7 Coding exercises

Note: if you are working in MySQL (not T-SQL), then, in all table names below, you should should replace the period (`.`) with an underscore (`_`). However, aliases are not part of the table name, so you should still use the period for aliases.

Exercise 2.7.1

Complete each of the tasks below, using SQL queries. They all relate to the tables in the Notes schema of the Sandpit database. We will practice using `SELECT`, `FROM` and `WHERE`.

1. Retrieve only the names of all pets.
2. Retrieve the names of all pets that belong to the friend with FriendID equal to 3. Do not join any tables (we'll do that later).
3. Display the first and last names of all friends whose favourite colour is red.

4. We have not yet worked with dates, but we will now. The Scratched table contains a column ScratchDate. Execute the following query and explain what it does:

```
SELECT ScratcherID, ScratchDate, ScratchTime, ScratcheeID
FROM Notes.Scratched -- in MySQL, use Notes_Scratched instead
WHERE ScratchDate = '20180905';
```

5. Replace the search condition in the above query with one that returns all records where ScratchDate is on or before 6th Sep, 2018. Note, the date format is `'YYYYMMDD'`, and you can use the <= comparison operator.
6. Retrieve the ScratcheeID and ScratcherID for all people who have participated in back scratching between the hours of 11AM and 12PM (inclusive). You can use the comparison operators <= and >=, as well as the logical operator `AND`. The time format is `HH:MM:SS` (24 hour format).
7. Retrieve the ScratcherID for all people who scratched a back either at 11AM on Sep 6th, 2018, or at 10AM on Sep 7th, 2018. You can use the logical operator `OR`.

Solutions to Exercise 2.7.1

1.

```
SELECT PetName
FROM Notes.Pets; -- In MySQL, write Notes_Pets instead
```

2.

```
-- In T-SQL
SELECT PetName
FROM Notes.Pets P
WHERE P.FriendID = 3;

-- In MySQL
SELECT PetName
FROM Notes_Pets P -- 'Notes' is part of the table name
WHERE P.FriendID = 3; -- P is an alias
```

3.

```
SELECT FirstName, LastName
FROM Notes.Friends
WHERE FavColour = 'red';
```

4. The query returns all records with ScratchDate 5th Sep, 2018.

5.
```
SELECT ScratcherID, ScratchDate, ScratchTime, ScratcheeID
FROM Notes.Scratched
WHERE ScratchDate <= '20180906';
```

6.
```
SELECT ScratcherID, ScratcheeID
FROM Notes.Scratched
WHERE ScratchTime >= '11:00:00' AND ScratchTime <= '12:00:00';
```

7.
```
SELECT ScratcherID
FROM Notes.Scratched
WHERE
(ScratchDate = '2018-09-06' AND ScratchTime = '11:00:00')
OR
(ScratchDate = '2018-09-07' AND ScratchTime = '10:00:00');
```

Exercise 2.7.2

We'll now practice using `CASE WHEN`, `ORDER BY` and column aliases (with quoting). For this question, we will use the Colours table in the Ape schema, as well as the Scratched table in the Notes schema. Both are in the Sandpit database.

1. Retrieve the Colours table, but rename the Comments column to 'Ape Opinions'.
2. Order the Colours table, in descending order of colour names (using the alphabetic order).
3. You can order by two columns at once:
   ```
   ORDER BY column1, column2
   ```
   Order the Scratched table according to date of scratching. For each date, make sure the rows are also ordered according to the time of scratching.
4. Edit your query from question 3, so that ScratchDate is in descending order and ScratchTime is in ascending order.
5. The apes do not appreciate fancy colour names. Reproduce the Colours table, but rename the colour 'magenta' to purple, and the colour 'turquoise' to 'blue'.

Solutions to Exercise 2.7.2

1. In T-SQL, identifiers are quoted with square brackets. In MySQL, they are quoted with backticks. The backtick is typically located to the left of the number 1, at the top left of your keyboard.

   ```
   -- In T-SQL use square brackets
   SELECT ColourID, ColourName, Comments AS [Ape Opinions]
   FROM Ape.Colours;

   -- In MySQL use backticks
   SELECT ColourID, ColourName, Comments AS `Ape Opinions`
   FROM Ape_Colours;
   ```

2. ```
   SELECT *
   FROM Ape.Colours
   ORDER BY ColourName DESC;
   ```

3. ```
   SELECT *
   FROM Notes.Scratched
   ORDER BY ScratchDate, ScratchTime;
   ```

4. ```
   SELECT *
   FROM Notes.Scratched
   ORDER BY ScratchDate DESC, ScratchTime;
   ```

5. ```
   SELECT ColourID,
          Comments,
          CASE WHEN ColourName = 'magenta' THEN 'purple'
                WHEN ColourName = 'turqoise' THEN 'blue'
                ELSE ColourName END AS ColourName
   FROM Ape.Colours;
   ```

Exercise 2.7.3

We will now practice writing search conditions with the operators `BETWEEN`, `IN`, `LIKE` and `NOT`. These questions all relate to the Houses and Suburbs tables, in the Notes schema of the Sandpit database.

1. Use the `IN` operator to return the names of all home owners in the post codes 3128, 3142 and 3083.
   Hint: the condition `MyColumn IN (1,2,3)` returns `TRUE` when an entry of MyColumn equals either 1,2 or 3.
2. Use the `LIKE` operator to get the street address (ignoring suburb name)

of all houses that are on an avenue. This means, any house where the street address ends in 'Ave'. You can read about `LIKE` on page 50 of these notes.

3. Use `NOT`, with `LIKE`, to get the house_ID of every house that is not on an avenue.
4. Use `LIKE` (and other operators) to get the street address (ignoring suburb name) of all houses that have a post code starting with '31' and that also cost strictly less than $300,000.
5. We haven't seen the `BETWEEN` operator yet.
   The following two search conditions are equivalent:

```
-- using >=, and <=
MyNumber >= 1 AND MyNumber <= 2

-- using BETWEEN (equivalent to the above)
MyNumber BETWEEN 1 AND 2
```

   Use `BETWEEN` to find all suburbs with a 40%–70% vaccination rate.

Solutions to Exercise 2.7.3

1.

```
SELECT house_owner
FROM Notes.Houses
WHERE post_code IN (3128, 3142, 3083);
```

2. I have used an empty space in `'% Ave'` to make sure 'Ave' is preceded by an empty space.

```
SELECT house_address
FROM Notes.Houses
WHERE house_address LIKE '% Ave';
```

3.

```
SELECT house_address
FROM Notes.Houses
WHERE house_address NOT LIKE '% Ave';
```

4. To get prices 'strictly less', we use the < operator.

```
SELECT house_address
FROM Notes.Houses
WHERE post_code LIKE '31%' AND house_price < 300000;
```

5.

```
SELECT *
FROM Notes.Suburbs
WHERE vaccination_rate BETWEEN 0.4 AND 0.7;
```

Exercise 2.7.4

We will now practice dealing with NULL values.

1. Run the following query and explain what is wrong with it.
   ```
   SELECT *
   FROM Notes.Friends -- In MySQL, replace with Notes_Friends
   WHERE FavColour = NULL;
   ```
2. Use the IS NULL operator to find all houses where the post code is unknown.
3. Find all houses where the post_code is not unknown.
4. Retrieve all house IDs and post codes from the Houses table, but for any NULL post codes, change the entry to 'UNKNOWN'. The post code column in the result table should be called 'post_code_modified'.

Solutions to Exercise 2.7.4

1. The expression FavColour = NULL always evaluates to NULL, so it never returns TRUE.
2.
   ```
   SELECT *
   FROM Notes.Houses
   WHERE post_code IS NULL;
   ```
3.
   ```
   SELECT *
   FROM Notes.Houses
   WHERE post_code IS NOT NULL;

   -- The following will also work
   SELECT *
   FROM Notes.Houses
   WHERE NOT post_code IS NULL;
   ```
4.

```
SELECT house_ID,
      CASE WHEN post_code IS NULL THEN 'UNKNOWN'
      ELSE post_code END AS post_code_modified
FROM Notes.Houses;
```

Exercise 2.7.5

Now it's time to practice using `JOIN`, `LEFT JOIN` and `RIGHT JOIN`. This exercise will use tables from the Ape and Notes schemas, in the Sandpit database.

1. Join the Friends and Pets tables, using the correct primary/foreign key pair. From the result, how many pets does the friend named 'Z' have, and what are their names?
2. Join Table1 and Table2 using the correct primary and foreign key pair. First, use your SQL editor to find out which columns are the primary and foreign keys, in each table. In your result, retrieve only columns B and C from Table1, and only column D from Table2.
3. To get the initials of an ape, we can use the `SUBSTRING` function (see Section 2.2.5). The expression `SUBSTRING(FirstName, 1, 1)` extracts the first letter of FirstName. Try it out first with the following query:

```
SELECT SUBSTRING(FirstName, 1, 1) AS FirstInitial,
       SUBSTRING(LastName, 1, 1) AS LastInitial
FROM Ape.Friends; -- replace with Ape_Friends in MySQL
```

   Now, for all apes that have a favourite colour, list their initials, next to the name of their favourite colour.
4. Modify your solution to question 3, so that the result also includes any apes that do not have a favourite colour.
5. Modify your solution to question 3, so that the result also includes any colours that are not the favourite of any ape.

Solutions to Exercise 2.7.5

1. The friend named 'Z' has two pets, Cauchy and Gauss.

```
SELECT *
FROM Notes.Friends F JOIN Notes.Pets P
    ON F.FriendID = P.FriendID;

-- Remember to use underscores in MySQL table names
SELECT *
FROM Notes_Friends F JOIN Notes_Pets P
    ON F.FriendID = P.FriendID;
```

2. The primary key of Table1 is column A. The primary key of Table2 is column E. The only foreign key is in Table2, and is column A. The correct join is therefore:

```
SELECT T1.B, T1.C, T2.D
FROM Notes.Table1 T1 JOIN Notes.Table2 T2
    ON T1.A = T2.A;
```

3.

```
SELECT SUBSTRING(FirstName, 1, 1) AS FirstInitial,
       SUBSTRING(LastName, 1, 1) AS LastInitial,
       ColourName
FROM Ape.Friends F JOIN Ape.Colours C
    ON F.FavColourID = C.ColourID;
```

4. We just replace `JOIN` with `LEFT JOIN`:

```
SELECT SUBSTRING(FirstName, 1, 1) AS FirstInitial,
       SUBSTRING(LastName, 1, 1) AS LastInitial
       ColourName
FROM Ape.Friends F LEFT JOIN Ape.Colours C
    ON F.FavColourID = C.ColourID;
```

5. We just replace `JOIN` with `RIGHT JOIN`:

```
SELECT SUBSTRING(FirstName, 1, 1) AS FirstInitial,
       SUBSTRING(LastName, 1, 1) AS LastInitial
       ColourName
FROM Ape.Friends F RIGHT JOIN Ape.Colours C
    ON F.FavColourID = C.ColourID;
```

Exercise 2.7.6

In this exercise, we will practice joins involving three tables. This exercise will use tables from the Ape and Notes schemas, in the Sandpit database. The syntax for a three way join should look like this:

```
SELECT *
FROM Table1 T1
  JOIN Table2 T2 ON T1.attribute1 = T2.attribute2
  JOIN Table3 T3 ON T2.attribute3 = T3.attribute4;
```

1. In the Ape schema, join the EatingFrom table to both the BananaTree table and the Friends table, using the appropriate primary/foreign key pairs.
2. Modify your solution to question 1, so that it only produces results for trees planted in July 2016.
3. Produce a single table that holds the first name of each friend who

scratched a back, the first name of the friend whose back was scratched, and the date and time of the scratching. Finally, order the result by the date of scratching (in ascending order). Make sure that you use column aliases to give the resulting FirstName columns appropriate names (e.g., ScratcherName and ScratcheeName), so that the two can't be confused.

Solutions to Exercise 2.7.6

1.
```
SELECT *
FROM Ape.BananaTree B
  JOIN Ape.EatingFrom E ON B.TreeID = E.TreeID
  JOIN Ape.Friends F ON F.FriendID = E.FriendID;
```

2.
```
SELECT *
FROM Ape.BananaTree B
  JOIN Ape.EatingFrom E ON B.TreeID = E.TreeID
  JOIN Ape.Friends F ON F.FriendID = E.FriendID
WHERE B.YearPlanted = 2016 AND B.MonthPlanted = 7;
```

3.
```
SELECT Sr.FirstName AS ScratcherName,
      Se.FirstName AS ScratcheeName,
      S.ScratchDate,
      S.ScratchTime
FROM Notes.Friends Sr
  JOIN Notes.Scratched S ON Sr.FriendID = S.ScratcherID
  JOIN Notes.Friends Se ON Se.FriendID = S.ScratcheeID
ORDER BY ScratchDate;
```

Chapter 3

Aggregating, grouping and windowing

In the previous chapter, we learned how to write queries that chop up tables and join them together, taking advantage of `SELECT`, `FROM` and `JOIN`, while using search conditions and various operators in the `WHERE` clause. But, so far, all of our skills just retrieve 'raw' data as it appears already in the database. Our next step, is to start learning to derive new data from this raw data.

Aggregation and grouping are fundamental SQL concepts for deriving data. Aggregating queries work by partitioning the rows of a table into groups, and then applying *aggregation functions* to return a single value for each group. There are many applications for this, but one worth mentioning is, when we want to avoid extracting a very large dataset, aggregating queries allow us to summarise the dataset – extracting only the smaller aggregated dataset.

As we will see next, the `GROUP BY` clause determines how the groups are partitioned. Then, the `HAVING` clause decides which (if any) groups to discard. Finally, an *aggregating function* can be used to get basic summary information (such as the average, or the standard deviation) within each of the groups.

3.1 GROUP BY

The `GROUP BY` clause does pretty much what it says on the tin: it groups the rows of a table (using the entries within one or more columns). The easiest way to understand it is with a few examples. The following query groups the Pets table by FriendID, and then selects the FriendID column.

```
SELECT FriendID
FROM Pets
GROUP BY FriendID;
```

Pay attention to the fact that `GROUP BY` is executed *before* `SELECT`, even though `SELECT` was written first:

`FROM Pets`

Pets			
PetID	PetName	PetDOB	FriendID
1	Chikin	24/09/2016	2
2	Cauchy	01/03/2012	3
3	Gauss	01/03/2012	3

$\Downarrow$

`GROUP BY FriendID`

PetID	PetName	PetDOB	FriendID
(2)	(Chikin)	(24/09/2016)	2
(2,3)	(Cauchy, Gauss)	(01/03/2012, 01/03/2012)	3

$\Downarrow$

`SELECT FriendID`

PetID	PetName	PetDOB	FriendID
(2)	(Chikin)	(24/09/2016)	2
(2,3)	(Cauchy, Gauss)	(01/03/2012, 01/03/2012)	3

$\Downarrow$

`RESULT`

FriendID
2
3

Figure 3.1: The `GROUP BY` clause

After grouping (with `GROUP BY`), in the second step within Figure 3.1, each row of the table represents *one group*. The last two rows of the Pets table were placed into a single group, together, because they both shared the same FriendID. So, after grouping, the table had two rows instead of three. We told SQL to group by FriendID, so SQL made sure it returned only one value for each row in the FriendID column. However, we didn't tell SQL what to do with the values in the other columns (PetID, PetName and PetDOB), so it placed those values into **tuple entries**, which we are representing here with red parentheses.

After grouping, we were able to execute `SELECT FriendID` with no issues, in the third step within Figure 3.1. However, if in that step we had chosen to `SELECT` any of the other columns, then SQL would have produced an error. An example of this error is displayed Figure 3.2.

```
SELECT PetDOB
FROM Pets
GROUP BY FriendID;
```

PetID	PetName	PetDOB	FriendID
(2)	(Chikin)	(24/09/2016)	2
(2,3)	(Cauchy, Gauss)	(01/03/2012, 01/03/2012)	3

```
Msg 8120, Level 16, State 1, Line 1 Column 'Pets.PetDOB' is
invalid in the select list because it is not contained in either
an aggregate function or the GROUP BY clause.
```

Figure 3.2: Error message printed if PetDOB column is selected with tuples in it.

The above error was returned because SQL cannot return a `RESULT` table that has any tuple entries. The reason SQL cannot return tuple entries like this, is that it wants to return only *one value for each entry*. This property of entries is referred to as **atomicity**. After applying `GROUP BY`, in Figure 3.2, the entry in the second row of the PetDOB column contains two values, Cauchy *and* Gauss. So, the error was caused by trying to `SELECT PetDOB`. In general, when there are tuple entries, SQL doesn't automatically check if the tuples contain only one unique value in them.

In summary, when we use `GROUP BY`, we can't select any columns that will end up with tuple entries. That is, we can't select any columns that we haven't also included in our `GROUP BY` clause. How restrictive! This causes a lot of confusion for new SQL programmers, but it's a very natural restriction when you get used to it. It (hopefully) begins to become natural when you understand that all entries in SQL query results must be atomic.

Thankfully, the tuple entries are designed to be dealt with in various ways. One way to deal with those pesky tuples is to add their columns to the `GROUP BY` clause. SQL will only group rows together if all of the columns in the `GROUP BY` clause share the same values. Since Cauchy and Gauss were born on the same day, see what happens if we `GROUP BY PetDOB, FriendID`:

`GROUP BY PetDOB, FriendID`

PetID	PetName	PetDOB	FriendID
(2)	(Chikin)	24/09/2016	2
$(2, 3)$	(Cauchy, Gauss)	01/03/2012	3

Figure 3.3: An intermediate step in query execution, grouping by *both* PetDOB and FriendID

Keep in mind that we can't execute `GROUP BY` on its own without a `SELECT` statement; the above illustration displays only the intermediate step achieved by `GROUP BY`. Rows are formed into groups based on whether or not *all the columns* in the `GROUP BY` clause have matching entries. We can see this in action in Figure 3.3. By chance, the two pets that have FriendID equal to 3 also share the same birthday. So, the groups were unchanged compared to Figure 3.2. What did change, is that now we don't have tuples in the PetDOB column. Since the tuples are gone from PetDOB, we can now `SELECT` PetDOB in our query as well, without causing an error.

Example 3.1.1. For this `GROUP BY` example, we'll use a table called Letters:

Letters		
A	B	Num
a	b	1
a	c	2
a	b	3
a	c	4

Figure 3.4: The Letters table.

If we group by column B, using `GROUP BY B`, then the grouping is:

Letters		
A	B	Num
a	b	1
a	c	2
a	b	3
a	c	4

$\Longrightarrow$

A	B	Num
(a, a)	b	(1, 3)
(a, a)	c	(2, 4)

Figure 3.5: Grouping Letters by column B.

The 'a' entries in column A weren't grouped together, because we didn't ask SQL to check them, so they were just placed into tuple entries, according to the groups they belong to, as determined by column B. If we instead choose to `GROUP BY A`, we get:

Letters		
A	B	Num
a	b	1
a	c	2
a	b	3
a	c	4

$\Longrightarrow$

A	B	Num
a	(b, c, b, c)	(1, 2, 3, 4)

Figure 3.6: Grouping Letters by column A.

If we group by both A and B with `GROUP BY A,B` then we get

Letters		
A	B	Num
a	b	1
a	c	2
a	b	3
a	c	4

$\Longrightarrow$

A	B	Num
a	b	(1, 3)
a	c	(2, 4)

Figure 3.7: Grouping Letters by both columns A and B.

Notice that, unlike last time we grouped by A, the four rows containing 'a' in column A were not all merged into one row. This is because we also grouped by B at the same time, and rows are only merged if *all columns in the `GROUP BY` clause match*. Now we can `SELECT` either A, or B, or both, if we like, because both are in the `GROUP BY` clause, so neither column is left with any tuples in it.

3.2 Aggregation functions

In the previous section, when applying `GROUP BY`, we faced some pesky red tuple entries. We learned that, before using `SELECT` on them, we could make the tuples go away by adding their column(s) to the `GROUP BY` clause. But what if we don't *want* to group by those extra columns? Consider this table of people whose ages will become outdated as this book matures:

RandomPeople		
Name	Gender	Age
Beyoncé	F	37
Laura Marling	F	28
Darren Hayes	M	46
Bret McKenzie	M	42
Jack Monroe	NB	30

Figure 3.8: The RandomPeople table

Executing a `GROUP BY Gender` gives

Name	Gender	Age
(Beyoncé, Laura Marling)	F	(37, 28)
(Darren Hayes, Bret McKenzie)	M	(46, 42)
(Jack Monroe)	NB	(30)

Figure 3.9: The RandomPeople table, grouped by Gender.

We could now `SELECT Gender`, which would be useful if we only wanted to get a table of the different genders. If we want to extract more information about the genders, then we need a function that returns just *one value for each tuple entry* in the grouped rows. Observe the built-in SQL function `AVG`:

```
SELECT Gender, AVG(Age) AS AverageAge
FROM RandomPeople
WHERE Gender = 'F'
GROUP BY Gender;
```

`FROM RandomPeople WHERE Gender = 'F'`

RandomPeople		
Name	Gender	Age
Beyoncé	F	37
Laura Marling	F	28
Darren Hayes	M	46
Bret McKenzie	M	42
Jack Monroe	NB	30

⇓

`GROUP BY Gender`

Name	Gender	Age
(Beyoncé, Laura Marling)	F	(37, 28)

⇓

`AVG(Age) AS AverageAge`

Name	Gender	AverageAge
(Beyoncé, Laura Marling)	F	AVG((37,28))

⇓

`SELECT Gender, AverageAge`

Name	Gender	AverageAge
(Beyoncé, Laura Marling)	F	32.5

⇓

RESULT

Gender	AverageAge
F	32.5

Figure 3.10: The `AVG` aggregation function

The above query returns the average age of all females in the RandomPeople table. Pay close attention to the order in which the clauses in the above query are executed. Remember, the actual order of execution does not match the order in which things are written. In particular, notice that

`WHERE` is executed before `GROUP BY`. This order will be important to recall when we start using the `HAVING` clause.

We call `AVG` an **aggregation function**. There are a host of other aggregation functions available, which vary slightly between dialects of SQL. In Section 4.1, we will learn to read the best source of information on these functions: the (T-SQL or MySQL) documentation. Here is a table of simple and useful aggregation functions:

Table 3.1: Some of the aggregation functions in T-SQL and MySQL.

Function (MySQL)	**Function (T-SQL)**	**Purpose**
AVG	AVG	Average
STDDEV_SAMP	STDEV	Sample standard deviation
STDDEV_POP	STDEVP	Population standard deviation
VAR_SAMP	VAR	Sample variance
VAR_POP	VARP	Population variance
COUNT	COUNT	Count number of rows
MIN	MIN	Minimum
MAX	MAX	Maximum
SUM	SUM	Sum

Table 3.1 tells us we can do many kinds of aggregations on grouped data. If we use any of these functions on ungrouped data, they treat the entire table as one group. For example:

```
SELECT COUNT(Gender)
FROM RandomPeople;
```

The above query first groups all of the entries from Gender, forming a single tuple entry (F,F,M,M,NB). Then, it counts the number of values in that tuple. The query returns a single number, 5. Clearly, we would also get the number 5 if we instead chose to `COUNT(Name)`, or `COUNT(Age)`. What's the point of counting any specific column then? There is a point.

The `COUNT` function skips over any `NULL` values in whatever column you feed it. Remember the FavColour column of the Friends table (Figure 1.4)? If we `SELECT COUNT(FavColour) FROM Friends`, then the FavColour tuple will look like (red, blue, `NULL`), and the query will return 2, rather than 3, ignoring the `NULL` value. If we don't want to ignore `NULL` values at all when counting, then the `*` expression comes to the rescue. For example, writing the query `SELECT COUNT(*) FROM RandomPeople`, is like asking explicitly to "count the number of rows of RandomPeople".

3.2.1 COUNT(DISTINCT)

We have just seen that COUNT(ColumnName) returns the number of non-NULL entries in the column called ColumnName. We can also modify COUNT with the DISTINCT keyword. For example:

```
SELECT COUNT(DISTINCT Gender) AS NumGenders
FROM RandomPeople;
```

The above query will return a table with a single row and column, containing the number 3: that is, the number of distinct genders in RandomPeople (Figure 3.8). We can, of course, group the records before applying the COUNT(DISTINCT) function, as we do here (with Letters, Figure 3.4):

```
SELECT B, COUNT(DISTINCT A) AS NumA, COUNT(A) AS NumRows
FROM Letters
GROUP BY B;
```

RESULT		
B	NumA	NumRows
b	1	2
c	1	2

Figure 3.11: Counting the distinct entries in A, grouping by B (from Letters table).

If we wish to count distinct records across multiple columns, we can concatenate the columns together with CONCAT (see Section 2.2.5). For example:

```
SELECT COUNT(DISTINCT CONCAT(A,B)) AS NumAB
FROM Letters;
```

3.2.2 CAST, and the perilous INT

You might have predicted that aggregation functions like AVG and SUM won't work on strings of letters. The bad news is that these functions can also cause sneaky problems to do with different types of numbers. The INT data type, in SQL, represents whole numbers only (with no decimal values allowed). In contrast, the DECIMAL(5,2) data type (see Section 2.2.4) allows 5 digit numbers with 2 decimal places. Aggregation functions generally return the same data type as the column they are applied to. So, if we use the AVG function on an INT column, then it will always return an INT!

Here's what would happen if the Age column in RandomPeople was stored as an INT:

```
SELECT Gender, AVG(Age) AS AverageAge
FROM RandomPeople
WHERE Gender = 'F'
GROUP BY Gender;
```

RESULT	
Gender	AverageAge
F	32

Figure 3.12: An example of calling `AVG` on an `INT`.

Compare the above to the output of the same query in Figure 3.10, where it was assumed that Age was a decimal number. You'll notice the result should be 32.5, not 32. There is a way around this problem, via 'casting' the Age column to a different data type, like `DECIMAL(5,2)`, using `CAST(Age AS DECIMAL(5,2))`, before aggregating:

```
SELECT Gender, AVG(CAST(Age AS DECIMAL(5,2))) AS AverageAge
FROM RandomPeople
WHERE Gender = 'F'
GROUP BY Gender;
```

Note, the above query will produce average ages with more than 2 decimal places. This is because aggregation functions will, when applied to `DECIMAL` data types, will 'play it safe' by returning numbers with more decimal places than they started with.

3.2.3 Grouping with `CASE WHEN`

We first learned about `CASE WHEN` in Section 2.4.7, as a flexible way to transform data using search conditions. Here, we'll see we can also use that transformed data to control the formation of groups.

The following query uses `CASE WHEN` on the RandomPeople table (from Figure 3.8), to group by the Name column. However, rather than grouping by the individual names, it will form just two groups: one for the case when Name starts with the letter 'B', and the other for the case when Name does not start with 'B'. Note, `LIKE` was introduced in Section 2.4.5.

```
SELECT COUNT(*) AS NumPeople
FROM RandomPeople
GROUP BY CASE WHEN Name LIKE 'B%' THEN 'B people'
              ELSE 'non-B people' END;
```

The above query achieves what I set out to do, but it doesn't display the NumPeople counts next to the labels for each group. I would like to see 'B people' and 'non-B people' displayed next to the counts. Now, this is one place where SQL can be annoyingly verbose: we have to repeat the whole `CASE WHEN` expression in the select list.

```
SELECT CASE WHEN Name LIKE 'B%' THEN 'B people'
            ELSE 'non-B people' END AS NameGroup,
      COUNT(*) AS NumPeople
FROM RandomPeople
GROUP BY CASE WHEN Name LIKE 'B%' THEN 'B people'
              ELSE 'non-B people' END;
```

RESULT	
NameGroup	NumPeople
B people	2
non-B people	3

Figure 3.13: RandomPeople, grouped by whether Name starts with 'B'

In MySQL only, the above query can be simplified using the alias from the `SELECT` list (which we called NameGroup). This does not work in T-SQL:

```
-- only works in MySQL, not T-SQL
SELECT CASE WHEN Name LIKE 'B%' THEN 'B people'
            ELSE 'non-B people' END AS NameGroup,
      COUNT(*) AS NumPeople
FROM RandomPeople
GROUP BY NameGroup;
```

Another trick to avoid repeating `CASE WHEN`, that works in both MySQL and T-SQL, is to use the `WITH` clause. The `WITH` clause is a little above our skill level at the moment, but we will learn about it in Section 4.4.3.

3.3 HAVING

In the previous section (Figure 3.10), we saw an example in which the `WHERE` clause was used to discard all the rows that didn't satisfy `Gender = 'F'`. When using `GROUP BY`, it's also possible to discard entire groups of rows, based on the output of aggregation functions. In other words, it's possible to use search conditions like `AVG(Age) > 40`. The natural thing to try is `WHERE AVG(Age) > 40`, putting the search condition in the `WHERE` clause. However, this produces an error!

```
WHERE AVG(Age) > 40
```

```
Msg 147, Level 15, State 1, Line 3 An aggregate may not appear in
the WHERE clause unless it is in a subquery contained in a HAVING
clause or a select list, and the column being aggregated is an
outer reference.
```

Figure 3.14: Error message printed if an aggregation function is used in the `WHERE` clause.

The `HAVING` clause was introduced to SQL because **aggregation functions can't be used in the `WHERE` clause**. So, for any search condition with an aggregation function, the correct place for it is in the `HAVING` clause. This makes some sense, if we study the (logical) order of execution of the clauses that we've learned so far:

Table 3.2: The logical order of execution of the clauses we have learned so far

Clause	Logical order of execution
FROM	1
WHERE	2
GROUP BY	3
HAVING	4
SELECT	5

Notice from Table 3.2, that `WHERE` executes *before* `GROUP BY`, and having executes after `GROUP BY`. So, the `WHERE` clause acts on individual rows, while the `HAVING` clause acts on groups of rows. Here's an example of how we discard entire groups of rows based on a search condition with an aggregation function:

```
SELECT Gender, AVG(Age) AS AverageAge
FROM RandomPeople
GROUP BY Gender
HAVING AVG(Age) > 40;
```

Again, pay careful attention to the order of execution below.

`FROM RandomPeople GROUP BY Gender`

Name	Gender	Age
(Beyoncé, Laura Marling)	F	(37, 28)
(Darren Hayes, Bret McKenzie)	M	(46, 42)
(Jack Monroe)	NB	(30)

⇓

`HAVING AVG(Age) > 40`

Name	Gender	Age	
(Beyoncé, Laura Marling)	F	(37, 28)	} 32.5
(Darren Hayes, Bret McKenzie)	M	(46, 42)	} 44
(Jack Monroe)	NB	(30)	} 30

⇓

`AVG(Age) AS AverageAge`

Name	Gender	AverageAge
(Darren Hayes, Bret McKenzie)	M	AVG((46,42))

⇓

`SELECT Gender, AverageAge`

Name	Gender	AverageAge
(Darren Hayes, Bret McKenzie)	M	44

⇓

`RESULT`

Gender	AverageAge
M	44

Figure 3.15: The `HAVING` clause

In the above execution diagram (Figure 3.15), we see that the search condition `AVG(Age)> 40`, in the `HAVING` clause, was executed after `GROUP BY`, so that `AVG` was able to act on the grouped data.

In the query for Figure 3.15, notice that we used `AVG(Age)` twice: once in the `HAVING` clause and once in the `SELECT` clause. Why is that? In the second step of Figure 3.15, we see that the clause `HAVING AVG(Age)> 40` did not actually insert the average ages into the `RESULT` table. The ages weren't inserted into the `RESULT` table until `SELECT Gender, AVG(Age) AS AverageAge` was executed. This is useful, for example, when we want to *discard groups* using one aggregation function, and *select columns* using a different one, as in this next query. This next query uses `STDEV(Age)` to return the sample standard deviation of ages, for each gender whose average age is greater than 40:

```
SELECT Gender, STDEV(Age) AS AverageAge
FROM RandomPeople
GROUP BY Gender
HAVING AVG(Age) > 40;
```

The `HAVING` clause works with any aggregation function (see examples in Table 3.1). For the above query, we used the search condition `AVG(Age) > 40`, but a variety of search conditions are possible, ranging from very simple to highly complicated. We covered search conditions back in Section 2.4.4. In the next section, we'll see just how complicated search conditions can get.

3.4 Nested queries

I thought long and hard about where to place the section introducing nested queries (also known as subqueries). They aren't as tightly bound to the idea of aggregation as are `GROUP BY`, `HAVING`, and aggregation functions. However, in my experience, they are best introduced as a way to work with aggregating queries. So, here they are, the chapter on aggregation and grouping. The question at the start of the next section should make this clear.

3.4.1 Basic nested queries

Using the tools we have so far, you can write queries for both the following:

(i) Find all the RandomPeople with `Gender = 'F' OR Gender = 'NB'`.

(ii) Find all the genders in RandomPeople having `AVG(Age) < 40`.

Naturally, there should be a way to combine (i) and (ii) into a single query that says, “find all the RandomPeople whose Gender has an average age less than 40.” The easiest way to do this, is with a **nested query**.

Many people will, at some point, try to do the above in an apparently *even easier* way, without a nested query, by writing something like this:

```
SELECT Name
FROM RandomPeople
GROUP BY Gender
HAVING AVG(Age) < 40;
```

```
Msg 8120, Level 16, State 1, Line 1 Column 'RandomPeople.Name' is
invalid in the select list because it is not contained in either an
aggregate function or the GROUP BY clause.
```

Figure 3.16: Another error produced due to the atomicity constraint

Unfortunately, it doesn’t work. Remember the principle of **atomicity**? It helped us understand the error we saw way back in Figure 3.2. In fact, Figure 3.16 gives us essentially the same error message as Figure 3.2. So, why is it happening? Let’s break it down:

`FROM RandomPeople GROUP BY Gender HAVING AVG(Age) < 40`

Name	Gender	Age	
(Beyoncé, Laura Marling)	F	(37, 28)	32.5
(Darren Hayes, Bret McKenzie)	M	(46, 42)	44
(Jack Monroe)	NB	(30)	30

⇓

`SELECT Name`

Name	Gender	Age
(Beyoncé, Laura Marling)	F	(37,28)
(Jack Monroe)	NB	(30)

Figure 3.17: An execution diagram highlighting atomicity troubles.

In the final step of Figure 3.17, we try to `SELECT Name`, but `Name` has some pesky red tuple entries in it, that were generated by `GROUP BY`. In other words, the `Name` column has become non-atomic.

In summary, I can see, in the last step of Figure 3.17, that the names belonging to any gender with average age less than 40 are: Beyoncé, Laura Marling and Jack Monroe. But, I can't `SELECT` the Name column after I `GROUP BY` Gender, because the Name column will contain the pesky red tuples in it. At this point, many people might try adding Name to the `GROUP BY` clause:

```
SELECT Name
FROM RandomPeople
GROUP BY Gender, Name
HAVING AVG(Age) < 40;
```

In the above, I grouped by both Gender and Name, so the name column will have no red tuples, so I won't get an error. However, the query doesn't achieve my aim, because the groups will now be separated according to Gender *and* Name. So, when I then run `HAVING AVG(Age) < 40`, the averages won't be calculated within whole genders, but instead within groups of people who share both the same gender *and* the same name. In the RandomPeople table, this amounts to each person belonging to their own one person group, so the average ages would just be each person's own age. Well, how do I get the names I want? Why is it so damn hard, when I can see the names *right there*??!

The solution is to use a nested query. To wrap our heads around how this will work, let's for a moment use the word `RESULT` to denote the nested query. Consider the following query as a step towards "find all the RandomPeople whose Gender has an average age less than 40":

```
SELECT Name
FROM RandomPeople
WHERE Gender IN (RESULT);
```

The above query uses a search condition with a command we haven't seen in action yet: the `IN` operator. We saw the `IN` operator in Table 2.6, but we didn't see what it does. That's because I wanted to wait until we could use `IN` with a nested query. For the above query, `IN` checks if Gender matches anything contained in `RESULT`. So, to find the people we are after, we need `RESULT` to contain (F,NB).

Here, `RESULT` can be a whole query of its own, in which case we call the query 'nested'. We can replace the word `RESULT` with the query that produces the `RESULT` we want. So, plugging in a query that returns (F,NB):

```
SELECT Name
FROM RandomPeople
WHERE Gender IN (SELECT Gender
                 FROM RandomPeople
                 GROUP BY Gender
                 HAVING AVG(Age) < 40);
```

And there we have it, our first use of a nested query. It's really just two queries, where the 'inner' (i.e., nested) query executes first, and then the result of that execution gets passed to the `IN` operator as part of the search condition. This means the whole nested query is part of the search condition in the `WHERE` clause. Nested queries can be used in many places (including in `FROM`, `HAVING`, `SELECT` and even `GROUP BY`). We will see some more examples as we go, as well as in the exercises.

3.4.2 Correlated nested queries

I believe correlated nested queries are, by far, the most confusing thing to learn in introductory SQL. I'm not sure if it's good pedagogy to start with the pretense of confusion, but in this instance I think we need a disclaimer. That said, maybe I didn't learn them right the first time, or you'll just find them simpler than I did.

There are three good reasons to learn about correlated nested queries: (1) they will help grow your awareness of aliases (since, with an alias, you can *accidentally* create a correlated nested query); (2) they can help us think a little bit about efficiency; and (3) once you understand them, they can achieve some pretty complicated things, often in intuitive ways. Look carefully at the aliases in the following nested query, and decide if anything is out of the ordinary.

```
SELECT Name
FROM RandomPeople RP
WHERE age > (SELECT AVG(age)
             FROM RandomPeople
             WHERE gender = RP.gender);
```

The alias `RP` is defined in the *outer* query, but it is used in the `WHERE` clause of the *inner* query. With a basic nested query, we think of the inner query being executed first, followed by the outer query. That can't happen here, because the inner query depends on part of the outer query. This dependence on the outer query is the reason for the name 'correlated'. This is a correlated nested query. The execution of this query is very different to a regular nested query. Here, we have to think of the nested query being executed multiple times: *once for each row of the RandomPeople table*. The following execution diagram will help us visualise this.

`(SELECT AVG(Age) FROM RandomPeople WHERE Gender = RP.Gender)`

Row 1

RP.Name	RP.Gender	RP.Age
Beyoncé	F	37

`RP.Gender = 'F'`

Name	Gender	Age
Beyoncé	F	37
Laura Marling	F	28

} 32.5

Row 2

RP.Name	RP.Gender	RP.Age
Laura Marling	F	28

`RP.Gender = 'F'`

Name	Gender	Age
Beyoncé	F	37
Laura Marling	F	28

} 32.5

Row 3

RP.Name	RP.Gender	RP.Age
Darren Hayes	M	46

`RP.Gender = 'M'`

Name	Gender	Age
Darren Hayes	M	46
Bret McKenzie	M	42

} 44

Row 4

RP.Name	RP.Gender	RP.Age
Bret McKenzie	M	42

`RP.Gender = 'M'`

Name	Gender	Age
Darren Hayes	M	46
Bret McKenzie	M	42

} 44

Row 5

RP.Name	RP.Gender	RP.Age
Jack Monroe	NB	30

`RP.Gender = 'NB'`

Name	Gender	Age
Jack Monroe	NB	30

} 30

$\Downarrow$

`FROM RandomPeople RP WHERE age > (RESULTS)`

Name	Gender	Age	RESULTS
Beyoncé	F	37	32.5
Laura Marling	F	28	32.5
Darren Hayes	M	46	44
Bret McKenzie	M	42	44
Jack Monroe	NB	30	30

$\Downarrow$

`SELECT name`

Name
Beyoncé
Darren Hayes

Figure 3.18: Execution diagram for a correlated nested query. The table RandomPeople `RP` was given its alias, `RP`, in the outer query. So, the nested query executes once for each row of RandomPeople `RP`

We can describe the procedure in Figure 3.18 with the following steps. In the steps, we use a different colour for RandomPeople RP than we do for RandomPeople, to emphasize that these are treated as distinct copies of the same table.

(1) First, split the RandomPeople RP table into rows, giving 5 rows. For each row, do the following:

   (i) Extract the RP.Gender entry from the current row, and use this value of RP.Gender in the nested query. For example, for row 1, the gender is `'F'`, so use `WHERE Gender = 'F'` in the nested query, to find the two rows of RandomPeople that have Gender `'F'`.

   (ii) For the rows found in step (i), compute `AVG(Age)`. For example, using `WHERE Gender = 'F'`, the average age will be 32.5.

(2) After repeating steps (i) and (ii) for all 5 rows of RandomPeople RP, we will have 5 average ages, giving the tuple $(32.5, 32.5, 44, 44, 30)$. Call this tuple RESULTS, and add it as a temporary column to RandomPeople.

(3) Execute `SELECT name FROM RandomPeople WHERE age > RESULTS`, replacing RandomPeople with the temporary table found in step (2).

Hopefully, by studying the above steps along with Figure 3.18, you will develop a good sense of how a correlated nested query works. In particular, since the nested query was executed 5 times (called 'looping'), there are more steps involved than a regular (uncorrelated) nested query, which only executes once. So, clearly, a correlated nested query may be less efficient than a regular nested query, in terms of execution time. However, remember, it is the job of the Database Management System (DBMS) to try to optimise your query for you, 'under the hood'. So, in many cases, a correlated nested query may end up being very efficient, due to optimisations carried out by the DBMS, that we don't really need to learn about. In practice, you should worry about writing queries that you understand first. Then, if they are slow, you can try to make them faster.

If you have grasped the execution pattern, then you'll soon see, during the exercises, that we can achieve some complicated things fairly easily, by taking advantage of the 'looping' behaviour of a correlated nested query.

3.5 Windowing

Window functions are the last big piece of the puzzle in our chapter on grouping and aggregation. They aren't usually introduced in a beginner SQL course, but I think they should be, because they give a complete picture on how grouping can be used. The kind of 'grouping', that we do in this section, is actually called *windowing* or *partitioning*, rather than 'grouping', since we don't use the `GROUP BY` clause to achieve it. Regardless, windowing a variation on the same basic concepts behind grouping. The clause that we use for windowing is called the `OVER` clause.

3.5.1 `OVER` and `PARTITION BY`

Both clauses, `GROUP BY` and `OVER`, start by 'partitioning' a table. However, the key difference between `GROUP BY` and `OVER` is that `GROUP BY` forces the query to return one result row *per group* – it merges the entries within a group, as we visualised with red tuples in Figure 3.1. Whereas, `OVER` forces a query to return one result row *per row*. That is, `OVER` does not merge the entries together, but still allows us to make use of the 'groups' (which we now call **partitions** to highlight that their entries aren't merged).

To illustrate, let's use a very similar example to the one in Figure 3.10, but we'll compare the effect of `OVER` to the effect of `GROUP BY`. The example we start with is:

```
SELECT Gender, MIN(Age) AS MinimumAge
FROM RandomPeople
GROUP BY Gender;
```

Since we used `GROUP BY Gender`, the above will return one row for each gender, giving the minimum age within each gender:

RESULT	
Gender	MinimumAge
F	28
M	42
NB	30

Figure 3.19: The minimum age for each gender.

Now, if we just drop the `GROUP BY` clause from the query, we get:

```
SELECT Gender, MIN(Age) AS MinimumAge
FROM RandomPeople;
```

But now the query results in the following error:

```
Msg 8120, Level 16, State 1, Line 1 Column 'RandomPeople.Gender' is
invalid in the select list because it is not contained in either an
aggregate function or the GROUP BY clause.
```

Figure 3.20: An eerily familiar error message.

The error in Figure 3.20 is caused because the aggregation function `MIN`, in the absence of `GROUP BY`, still returns one row per group. The difference is that, in the absence of `GROUP BY`, the whole table is treated as *one group*. So, `MIN(Age)` just returns a *single* value (the minimum age from the whole table, 28). The error happens because we included Gender in `SELECT Gender, MIN(Age)`: we violated the atomicity constraint, by trying to select multiple genders, (F, M, NB), for just a *single* value of age, 28.

Now, let's modify the query, so that it forms **partitions**, rather than groups. We do this with the `OVER` clause:

```
SELECT Gender, MIN(Age) OVER() AS MinimumAge
FROM RandomPeople;
```

RESULT	
Gender	MinimumAge
F	28
F	28
M	28
M	28
NB	28

Figure 3.21: Every row gets the minimum age from the whole table.

The query has given us the minimum age from the whole table, 28, returned once *for each row* of the whole table. So, the returned table in Figure 3.21 has as many rows as RandomPeople, and it has 28 in every row. In summary, the 'window' for the aggregation function was the whole table, but the result was returned once for every row.

Notice that, unlike `GROUP BY`, the `OVER` clause went in the `SELECT` clause and attached directly to the `MIN` function. This is because, unlike `GROUP BY`, the `OVER` clause always needs to work with a function. In the next section, we discuss all the types of functions that work with `OVER`. But, first, notice that `OVER` has some round brackets next to it. Those round brackets are begging for us to input some information about what columns we want to use for partitioning, so we don't just partition by the whole table.

Here's an example where we partition by Gender, within the `OVER` clause. To do this, we must use the `PARTITION BY` command:

```
SELECT Gender, MIN(Age) OVER(PARTITION BY Gender) AS MinimumAge
FROM RandomPeople;
```

RESULT	
Gender	MinimumAge
F	28
F	28
M	42
M	42
NB	30

Figure 3.22: Every row gets the minimum age from that gender.

By calling `OVER(PARTITION BY Gender)`, we ended up with the table in Figure 3.22, still having as many rows as RandomPeople, but now every gender appears next to the minimum age from that gender.

3.5.2 Window functions

Any function that works with `OVER` is called a **window function**. The window functions unlock a wide range of creative possibilities for ways to use `OVER`. Window functions include all of the aggregation functions in Table 3.1, but they also include **analytic functions**. A function is called analytic if it works on partitions, just like an aggregation function; but, an analytic function may return a different value for each row in the partition. Contrast this with an aggregation function, which always returns the same value for every row in a partition. Here is a table of analytic functions that all exist in both T-SQL and MySQL:

Table 3.3: Some of the analytic functions in T-SQL and MySQL. These work with the `OVER` clause, along with any function in Table 3.1.

Function	**Returns**
FIRST_VALUE	Entry in first row of partition
LAST_VALUE	Entry in last row of partition
LAG	Entry one row behind current row
LEAD	Entry one row ahead of current row
ROW_NUMBER	Number of current row in partition
RANK	Rank of current row in partition
DENSE_RANK	Rank, without gaps
PERCENT_RANK	Percentage of rank value
CUME_DIST	Cumulative distribution value
NTILE	Bucket numbers (like histogram)

Table Table 3.3 includes a number of **ranking functions**. Ranking functions are just types of analytic functions that return a *rank* for each row in a partition.

You have your whole life to experiment with window functions (*how exciting*), and we're going to get practice during the exercises. For now, let's take a look at ordering rows within the `OVER` clause.

3.5.3 `ORDER BY` within `OVER`

To learn how to use `ORDER BY` in the `OVER` clause, I'm going to introduce a new table, and we'll see a useful application of window functions. The new table we'll work with contains sales from a six day sausage sizzle starting on New Year's Eve, 1999. Here it is:

SausageSizzleSummary		
SaleDate	Product	Sales
1999-12-31	pork	3
1999-12-31	veggie	3
2000-01-01	pork	2
2000-01-01	veggie	7
2000-01-02	pork	6
2000-01-02	veggie	6
2000-01-03	pork	6
2000-01-03	veggie	2
2000-01-04	pork	1
2000-01-05	veggie	5

Figure 3.23: The SausageSizzleSummary table.

The SausageSizzleSummary table gives the quantity of product sold (veggie or pork sausages), for each day of the sizzle. If there were no sales of a particular product, on a particular day, then the corresponding row is omitted from the table. Using these data, our goal will be to rank the daily sales, both overall and by product. Ranks are one way to provide answers to questions like:

- Which day(s) had the highest sales overall, for a single product type, and which product was it?
- Which day(s) had the highest pork sales?
- Which day(s) had the lowest veggie sales?

You can easily answer these questions yourself, just by examining the SausageSizzleSummary table, but we would like an automated way to compute *all* of the ranks. To start with, we'll examine the differences between the three window functions `ROW_NUMBER`, `RANK` and `DENSE_RANK`.

```
SELECT Sales,
      ROW_NUMBER() OVER(ORDER BY Sales) AS row_num_sales,
      RANK()       OVER(ORDER BY Sales) AS rank_sales,
      DENSE_RANK() OVER(ORDER BY Sales) AS dense_rank_sales
FROM SausageSizzleSummary;
```

RESULT			
sales	row_number	rank	dense_rank
1	1	1	1
2	2	2	2
2	3	2	2
3	4	4	3
3	5	4	3
5	6	6	4
6	7	7	5
6	8	7	5
6	9	7	5
7	10	10	6

Figure 3.24: Various ways to rank sausage sizzle sales.

In the above query, `ORDER BY Sales` tells the ranking functions to use Sales from lowest to highest (ascending order). Examining the result (Figure 3.24), we can see that the `ROW_NUMBER` function gives a distinct number for each row, while `RANK` and `DENSE_RANK` only change when Sales changes. We can also see that `RANK` skips the next available ranking value after a tie, while `DENSE_RANK` does not skip any values.

Our goal is to rank the daily sales, both overall and by product. So, for our purpose, we will use `DENSE_RANK` twice: once with no partition, and once with a partition by Product. We also want to override the default (ascending) ordering: the keyword `DESC` will tell `DENSE_RANK` to order by descending number of sales.

```
SELECT Product, SaleDate,
     DENSE_RANK() OVER(ORDER BY Sales DESC) AS overall_sales_rank,
     DENSE_RANK() OVER(PARTITION BY Product ORDER BY Sales DESC) AS
         product_sales_rank,
     sales
FROM SausageSizzleSummary;
```

RESULT				
product	saleDate	overall_sales_rank	product_sales_rank	sales
pork	2000-01-02	2	1	6
pork	2000-01-03	2	1	6
pork	1999-12-31	4	2	3
pork	2000-01-01	5	3	2
pork	2000-01-04	6	4	1
veggie	2000-01-01	1	1	7
veggie	2000-01-02	2	2	6
veggie	2000-01-05	3	3	5
veggie	1999-12-31	4	4	3
veggie	2000-01-03	5	5	2

Figure 3.25: Sales ranked overall and by product.

From the result in Figure 3.25 we can see:

- The day with the overall highest selling product was New Year's Day, and it was veggie sausages (with 7 sales).
- The highest pork sales were on Jan 2$^{\text{nd}}$ and 3$^{\text{rd}}$ (tied at 6 sales).
- The lowest veggie sales were on Jan 3$^{\text{rd}}$ (with 2 sales).

Aside from answering questions like those above, ranks have many applications. For example, ranks are a common tool in non-parametric statistics, and are particularly useful for computing quantiles, including the median.

3.6 Handwritten exercises

These exercises should all be done by hand (e.g., with pen and paper). No programming is required.

Exercise 3.6.1

In this exercise, we'll look at grouping and aggregating (that is, using aggregation functions like `AVG` or `COUNT`). We will also look at using `CAST` to change data types.

1. Order the following by which one executes first (not which is written first in a query): `GROUP BY`, `FROM`, `HAVING`, `WHERE`, `ORDER BY`, `SELECT`.
2. This query makes use of the Letters table, below.

Letters		
A	*B*	Num
a	b	1
a	c	2
a	b	3
a	c	4

Write down the result of each of the following queries.

a)

```
SELECT MAX(Num)
FROM Letters
GROUP BY A;
```

b)

```
SELECT B, MAX(Num)
FROM Letters
GROUP BY B;
```

c)

```
SELECT A, B, MAX(Num) AS MaxNum, MIN(Num) AS MinNum
FROM Letters
GROUP BY A,B;
```

3. This question uses Friends and RandomPeople, shown below.

Friends			
FriendID	FirstName	LastName	FavColour
1	*X*	*A*	red
2	*Y*	*B*	blue
3	*Z*	*C*	NULL

RandomPeople		
Name	Gender	Age
Beyoncé	F	37
Laura Marling	F	28
Darren Hayes	M	46
Bret McKenzie	M	42
Jack Monroe	NB	30

For each of the following queries, state whether or not it produces an error. If a query produces an error, explain why. If a query does not produce an error, explain in plain English what it does.

a)

```
SELECT FirstName
FROM Friends
GROUP BY FavColour;
```

b)

```
SELECT FavColour
FROM Friends
GROUP BY FavColour;
```

c)

```
SELECT AVG(Age)
FROM RandomPeople
WHERE AVG(Age) < 55
GROUP BY Gender;
```

d)

```
SELECT AVG(Age)
FROM RandomPeople
GROUP BY Gender
HAVING Age > 20;
```

e)

```
SELECT Gender, MAX(Age) AS AgeMax, MIN(Age) AS AgeMin
FROM RandomPeople
GROUP BY Gender
HAVING COUNT(*) < 3;
```

f)

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

4. This question makes use of the SausageSizzleSummary table, below.

SausageSizzleSummary		
SaleDate	Product	Sales
1999-12-31	pork	3
1999-12-31	veggie	3
2000-01-01	pork	2
2000-01-01	veggie	7
2000-01-02	pork	6
2000-01-02	veggie	6
2000-01-03	pork	6
2000-01-03	veggie	2
2000-01-04	pork	1
2000-01-05	veggie	5

You are told that the Sales column uses an `INT` data type. The following query aims to calculate the exact average number of sales for each SaleDate. Explain what is wrong with the query, and then write a new query that fixes the problem.

```
SELECT SaleDate, AVG(Sales) AS AvgSales
FROM SausageSizzleSummary
GROUP BY SaleDate;
```

Solutions to Exercise 3.6.1

1. The order of execution is: `FROM`, `WHERE`, `GROUP BY`, `HAVING`, `SELECT`, `ORDER BY`. **Comments**: in particular, notice that `WHERE` executes before `GROUP BY` and `HAVING` executes after `GROUP BY`. This explains why search conditions that make use of aggregation functions can be used in `HAVING` but not in `WHERE` (since those search conditions are designed to filter out groups).
2. The results are:

a)

RESULT
MAX(Num)
4

b)

RESULT	
B	MAX(Num)
b	3
c	4

c)

RESULT			
A	B	MaxNum	MinNum
a	b	3	1
a	c	4	2

3. The answers are as follows.

   a) Error: `GROUP BY FavColour` causes FirstName to contain tuple entries that are not dealt with, so they cannot be selected (even though we know those tuples contain only one value).

   b) No error: it returns a list of every distinct favourite colour.

   c) Error: the search condition `AVG(Age)< 55` contains an aggregation function. Search conditions with aggregation functions must be used in `HAVING`, not in `WHERE`.

   d) Error: the search condition `Age > 20` does not contain an aggregation function. Search conditions without aggregation functions must be used in `WHERE`, not in `HAVING`.

   e) No error: it returns the maximum and minimum age for each gender, but only for the genders that have less than 3 people.

   f) No error: It returns the total number of people in the table.

4. The query does not produce an error, but it does not produce the exact average sales. We are told that Sales is stored as an `INT`, implying the result will be rounded down to the nearest whole number. To fix this, we can cast Sales to a `DECIMAL`:

```
SELECT SaleDate, AVG(CAST(Sales AS DECIMAL)) AS AvgSales
FROM SausageSizzleSummary
GROUP BY SaleDate;
```

Exercise 3.6.2

In this exercise, we will practice using `GROUP BY` with the `CASE WHEN` expression. You are given the following EduStudy table.

EduStudy		
Id	Income	Education
EI13	low	5
EI122	low	1
EI281	low-mid	4
EI3332	middle	3
EI4751	high-mid	3
EI12	high	2

In order to study the relationship between income and education, 5000 survey participants were categorised according to their education level (with 1 low to 5 high), and their income bracket. The above EduStudy table holds a subset of the results.

1. The research team would like to categorise all participants according to the following rules:
   - if Income is either low or low-mid, and Education is strictly greater than 3, the category should be 'Group A';
   - if Income is either high or high-mid, and Education is strictly less than 3, the category should be 'Group B'; and
   - otherwise, the category should be `NULL`.

   By hand, add a column to EduStudy, called Category, and fill it in according to the above rules.
2. Fill in the blanks to create the Category column from question 1.

```
SELECT *,
CASE WHEN ...
     THEN ...
     WHEN ...
     THEN ...
     END AS Category
FROM EduStudy;
```

   Note, the expression `Income IN ('low', 'low-mid')` will return `TRUE` if `Income` is low or low-mid, and `FALSE` otherwise.
3. Now, the research team would like to count the number of EduStudy participants in each Category. Assuming the blanks have been filled in correctly, explain why the following will produce an error (although, technically it <u>does</u> work in MySQL, it just does not work on most dialects of SQL):

```
SELECT COUNT(*) AS NumParticipants,
CASE WHEN ...
     THEN ...
     WHEN ...
     THEN ...
     END AS Category
FROM EduStudy
GROUP BY Category;
```

4. Modify the query from question 3 to produce a query that works in both MySQL and T-SQL.

Solutions to Exercise 3.6.2

1. With the new Category column, EduStudy becomes:

RESULT			
Id	Income	Education	Category
EI13	low	5	Group A
EI122	low	1	NULL
EI281	low-mid	4	Group A
EI3332	middle	3	NULL
EI4751	high-mid	3	NULL
EI12	high	2	Group B

2. Filling in the blanks:

```
SELECT *,
CASE WHEN Income IN ('low', 'low-mid') AND Education > 3
     THEN 'Group A'
     WHEN Income IN ('high', 'high-mid') AND Education < 3
     THEN 'Group B'
     END AS Category
FROM EduStudy;
```

3. The query returns an error (in T-SQL, at least), because the alias called Category is created in the `SELECT` clause, so it cannot be used in the `GROUP BY` clause (which is executed before `SELECT`).
4. The query will work in both MySQL and T-SQL if we copy the whole `CASE WHEN` expression into the `GROUP BY` clause, in place of Category.

```
SELECT COUNT(*) AS NumParticipants,
CASE WHEN Income IN ('low', 'low-mid') AND Education > 3
     THEN 'Group A'
     WHEN Income IN ('high', 'high-mid') AND Education < 3
     THEN 'Group B'
     END AS Category
FROM EduStudy
GROUP BY
CASE WHEN Income IN ('low', 'low-mid') AND Education > 3
     THEN 'Group A'
     WHEN Income IN ('high', 'high-mid') AND Education < 3
     THEN 'Group B'
     END;
```

There are other ways to achieve the above, without repetition (e.g., using the `WITH` clause, which we will cover in Section 4.4.3). For now, the above is reliable.

Exercise 3.6.3

In this exercise, we will experiment with nested queries. We will use the EduStudy table again.

EduStudy		
Id	Income	Education
EI13	low	5
EI122	low	1
EI281	low-mid	4
EI3332	middle	3
EI4751	high-mid	3
EI12	high	2

1. Our goal is to obtain the Id of every participant who belongs to an income bracket with average education level at least 3.

   a) Explain why the following query produces an error.

   ```
   SELECT Id
   FROM EduStudy
   GROUP BY Income
   HAVING AVG(Education) >= 3;
   ```

   b) Explain in words what the following query does.

```
SELECT Income
FROM EduStudy
GROUP BY Income
HAVING AVG(Education) >= 3;
```

c) Given your answer above, fill in the appropriate subquery (replacing the word `subquery`) below, to achieve our goal (as stated in the question 1 intro).

```
SELECT Id
FROM EduStudy
WHERE Income IN (subqeury);
```

2. We will now see an example of a subquery used in the `FROM` clause. Our goal is to list, next to every participant in EduStudy, the total number of participants in their income group.

   a) Write down the table produced by following query.

   ```
   SELECT Income, COUNT(*) AS Num
   FROM EduStudy
   GROUP BY Income;
   ```

   b) The table resulting from part a contains the total number of participants in each income group. We can join this table to EduStudy to get our desired result, but which columns should we use as the primary/foreign key pair?
   c) Given your answers above, fill in the appropriate subquery and the appropriate join condition, in the following query, in order to achieve our goal.

   ```
   SELECT T2.*, Num
   FROM (subquery) T1 JOIN EduStudy T2 ON ...;
   ```

Solutions to Exercise 3.6.3

1. The solutions are as follows.

   a) The query produces an error because Id contains tuple entries after grouping by Income (atomicity is violated).
   b) The query obtains a list of the distinct income brackets that have average education level at least 3.
   c) We simply replace `subquery` with the query provided in part b of the question.

```
SELECT Id
FROM EduStudy
WHERE Income IN (SELECT Income
                 FROM EduStudy
                 GROUP BY Income
                 HAVING AVG(Education) >= 3);
```

2. The solutions are as follows.

   a) The query produces the following table.

RESULT	
Income	Num
low	2
low-mid	1
middle	1
high-mid	1
high	1

   b) We should join the two tables using their Income columns.
   c) For the subquery, we can use the query provided in part a of the question. For the join condition, we can use the Income columns (mentioned in part b).

```
SELECT T2.*, Num
FROM (SELECT Income, COUNT(*) AS Num
       FROM EduStudy
       GROUP BY Income) T1
JOIN EduStudy T2 ON T1.Income = T2.Income;
```

3.7 Coding exercises

Note: if you are working in MySQL (not T-SQL), then, in all table names below, you should should replace the period (`.`) with an underscore (`_`). However, aliases are not part of the table name, so you should still use the period for aliases.

Exercise 3.7.1

In this exercise we will practice basic usage of `GROUP BY`.

1. Execute the following query and explain what it does:

```
SELECT B
FROM Notes.Letters
GROUP BY B;
```

2. Now we will execute a query that fails and returns an error that is very common when using `GROUP BY`. Execute the below, and explain why the error occurred. You may need to revise Section 3.1, and in particular page 79.

```
SELECT A, B
FROM Notes.Letters
GROUP BY B;
```

3. In the query below, we've edited question 2, so that it uses the aggregation function `MAX`. This prevents the error from occurring. Execute it and explain what the query does (and why it prevents the error).

```
SELECT MAX(A), B
FROM Notes.Letters
GROUP BY B;
```

   Note: When used on collections of characters or strings, `MAX` returns the highest value using the lexicographic order.

4. Write a query that groups the rows of the Notes.Letters table by columns A and B, then selects columns A and B from the result.
5. Write a query that uses `GROUP BY` to return a table with 3 rows, one for each Gender (male, female and non-binary) in Notes.RandomPeople.
6. Execute the query below, and explain what it does.

```
SELECT DISTINCT Gender
FROM Notes.RandomPeople;
```

Solutions to Exercise 3.7.1

1. The query uses column B to group the rows of the table. Since column B is then selected, the result displays a list of the distinct entries in B.
2. The error occurs because column A is selected after grouping by column B. Column A contains 'tuple entries' (non-atomic).
3. The `MAX` aggregation functions returns a single value for each group, so `MAX(A)` obeys the atomicity constraint.
4.

```sql
SELECT A, B
FROM Notes.Letters
GROUP BY A, B;
```

5.
```sql
SELECT Gender
FROM Notes.RandomPeople
GROUP BY Gender;
```

6. It achieves the same thing as question 5, but without using `GROUP BY`.

Exercise 3.7.2

In this exercise, we will practice basic aggregation functions.

1. Execute the following and explain what it does.
```sql
SELECT AVG(Height) AS AvgTreeHeight
FROM Ape.BananaTree;
```

2. Retrieve the sample standard deviation of the widths of banana trees in Ape.BananaTree.
3. Retrieve the population standard deviation of the widths of banana trees in Ape.BananaTree.
4. Run the following query and explain why it produces an error.
```sql
SELECT AVG(Height) AS AvgTreeHeight, TreeID
FROM Ape.BananaTree;
```

5. Fix the query from question 4 using `GROUP BY TreeID`, but then explain why the result is not very useful.
6. How long do banana trees take to grow? Retrieve the average height of banana trees for each year planted.
7. Use the aggregation function `COUNT` to display the number of times that each FavColourID appears in Ape.Friends (including `NULL`). The `COUNT` function counts the number of *rows* (keeping in mind that anything in the `SELECT` clause will be executed after `GROUP BY`). The `COUNT` function will not count `NULL` values in the column that it is asked to count. For example, `COUNT(FavColourID)` will not count rows where FavColourID is `NULL`.
8. The apes want to know if planting in certain months will lead to taller banana trees. For each month planted in Ape.BananaTree, get the maximum height of the trees. But, restrict your results to trees planted before 2017, so they've had time to grow.
9. Get the minimum width for each month and year in Ape.BananaTree.

10. The column named Ripe in Ape.Banana indicates whether a banana is ripe (Ripe = 1) or unripe (Ripe = 0). Retrieve the average TasteRank for ripe and unripe bananas in Ape.Banana. **Hint:** the result should not be an integer.

Solutions to Exercise 3.7.2

1. The query returns the average of all tree heights in the table.
2. Retrieve the sample standard deviation of the widths of all banana trees in Ape.BananaTree.

```
-- In MySQL, replace STDEV with STDEV_SAMP
SELECT STDEV(width) AS stdev_width
FROM Ape.BananaTree;
```

3. Retrieve the population standard deviation of the widths of all banana trees in Ape.BananaTree.

```
-- In MySQL, replace STDEVP with STDEV_POP
SELECT STDEVP(width) AS stdevp_width
FROM Ape.BananaTree;
```

4. The aggregation function `AVG(Height)` produces one value for the whole table, so selecting TreeID results in non-atomic entries.
5. The 'fixed' query, below, is not very useful, because every row of BananaTree is in its own group.

```
SELECT AVG(Height) AS AvgTreeHeight, TreeID
FROM Ape.BananaTree
GROUP BY TreeID;
```

6.

```
SELECT YearPlanted, AVG(Height) AS AvgTreeHeight
FROM Ape.BananaTree
GROUP BY YearPlanted;
```

7.

```
-- alternatively, you could use COUNT(*)
SELECT FavColourID, COUNT(FriendID) AS ColourCount
FROM Ape.Friends
GROUP BY FavColourID;
```

8.

```
SELECT MonthPlanted, MAX(Height) AS MaxTreeHeight
FROM Ape.BananaTree
WHERE YearPlanted < 2017
GROUP BY MonthPlanted;
```

9. We have to group by both columns:

```
SELECT YearPlanted, MonthPlanted, MIN(Width) AS MinWidth
FROM Ape.BananaTree
GROUP BY MonthPlanted, YearPlanted;
```

10. TasteRank is stored as an `INT`, so we need to `CAST` it to a `DECIMAL` first.

```
SELECT Ripe, AVG(CAST(TasteRank AS DECIMAL)) AS Avg_TasteRank
FROM Ape.Banana
GROUP BY Ripe;
```

Exercise 3.7.3

We'll now practice basic usage of the `HAVING` clause.

1. Execute the query below, and explain why it produces an error.

```
SELECT AVG(Height) AS AvgTreeHeight, MonthPlanted, YearPlanted
FROM Ape.BananaTree
WHERE AVG(Height) > 5
GROUP BY MonthPlanted, YearPlanted;
```

2. Modify the query from question 1, to fix the error.
3. Retrieve the maximum and minimum widths of all trees in `Ape.BananaTree`, but discard any cases where the average height is less than 5.
4. Edit your solution from question 3, so that it only returns results from trees with TreeIDs $1, 4, 6, 16$ and 18.

Solutions to Exercise 3.7.3

1. Aggregation functions cannot be used in the `WHERE` clause, since `WHERE` is executed before `GROUP BY`.
2.

```
SELECT AVG(Height) AS AvgTreeHeight, MonthPlanted, YearPlanted
FROM Ape.BananaTree
GROUP BY MonthPlanted, YearPlanted
HAVING AVG(Height) > 5;
```

3.
```
SELECT MIN(width) AS MinWidth,
       MAX(width) AS MaxWidth
FROM Ape.BananaTree
HAVING AVG(height) >= 5;
```

4.
```
SELECT MIN(width) AS MinWidth,
       MAX(width) AS MaxWidth
FROM Ape.BananaTree
WHERE TreeID IN (1,4,6,16,18)
HAVING AVG(height) >= 5;
```

Exercise 3.7.4

In this exercise, we will practice using nested queries (i.e., subqueries).

1. Sometimes, nested queries are unnecessary. Consider the nested query below. Execute it, and explain what it does.
```
SELECT T.colourID, T.colourName, T.comments
FROM (SELECT *
      FROM Ape.Colours C LEFT JOIN Ape.Friends F
        ON F.favColourID = C.colourID) T
WHERE T.favColourID IS NULL;
```
   Using what you know about order of execution regarding `WHERE` and `LEFT JOIN`, can you modify the above query to be simpler (and avoid using a nested query)? You might want to start by copying out the subquery and executing it on its own.
2. Below is *another* way to do the same thing as the query in question 1… but wait! No it isn't! The query below returns an empty table. It looks like it should work. What a headache. Why is this happening?
```
SELECT C.colourID, C.colourName, C.comments
FROM Ape.Colours C
WHERE C.colourID NOT IN (SELECT F.FavColourID
                         FROM Ape.Friends F);
```
3. The EduStudy table is included in the Notes schema. The query below involves repetition of the same `CASE WHEN` expression twice (once in `SELECT`, and once in `GROUP BY`). Rewrite the query, using a subquery (appearing in a `FROM` clause) to remove this repetition.

```
SELECT COUNT(*) AS NumParticipants,
CASE WHEN Income IN ('low', 'low-mid') AND Education > 3
      THEN 'Group A'
      WHEN Income IN ('high', 'high-mid') AND Education < 3
      THEN 'Group B'
      END AS Category
FROM Notes.EduStudy
GROUP BY
CASE WHEN Income IN ('low', 'low-mid') AND Education > 3
      THEN 'Group A'
      WHEN Income IN ('high', 'high-mid') AND Education < 3
      THEN 'Group B'
      END;
```

4. Consider the correlated nested query (explained on page 91):

```
SELECT PersonName
FROM Notes.RandomPeople RP
WHERE age > (SELECT AVG(age)
             FROM Notes.RandomPeople
             WHERE gender = RP.gender);
```

Correlated nested queries can sometimes be inefficient and confusing. Can you rewrite it as a join? Try without reading the hint.

Hint: you can join RandomPeople to the result of a nested query.

```
FROM RandomPeople RP JOIN (RESULT) R ON ...
```

In the above, `R` is an alias for `RESULT` (just like a table alias). Note also that you may want to use `GROUP BY` within the nested query. Try again without reading the next hint.

Second hint: you can use the following as the nested query:

```
SELECT gender, AVG(age) AS AverageAge
FROM Notes.RandomPeople
GROUP BY gender;
```

Solutions to Exercise 3.7.4

1. The query displays all colours that are not the favourite of any ape. All joins are part of the `FROM` clause, so they execute before `WHERE`. This means we can simply use `WHERE` to identify `NULL` values in the join result.

```
SELECT C.colourID, C.colourName, C.comments
FROM Ape.Colours C LEFT JOIN Ape.Friends F
  ON F.favColourID = C.colourID
WHERE FavColourID IS NULL;
```

2. The resulting table is empty because the subquery (copied below) contains `NULL` values. If a list contains one unknown number, then it's impossible to confirm that any number is not in the list.

```
SELECT F.FavColourID
FROM Ape.Friends F;
```

3. We do the categorising in the subquery, and everything else happens in the 'outer' query.

```
SELECT COUNT(*) AS NumParticipants, T.Category
FROM (SELECT
      CASE WHEN
            Income IN ('low', 'low-mid') AND Education > 3
          THEN 'Group A'
          WHEN
            Income IN ('high', 'high-mid') AND Education < 3
          THEN 'Group B'
          END AS Category
      FROM Notes.EduStudy) T
GROUP BY T.Category;
```

4.

```
SELECT RP.PersonName
FROM Notes.RandomPeople RP JOIN (
     SELECT gender, AVG(age) AS AverageAge
     FROM Notes.RandomPeople
     GROUP BY gender) R
ON RP.gender = R.gender
WHERE RP.age > R.AverageAge;
```

Exercise 3.7.5

In this exercise we will explore windowed aggregation functions, using the `OVER` clause. We will also learn an unnecessarily difficult way to calculate the variance of a column.

1. The EduStudy table is in the Notes schema. Start by writing a query that retrieves the whole EduStudy table, but also appends a new column with every entry equal to the average education level of all participants.

2. If you haven't already, then make sure your solution to question 1 is returning a decimal, and not an integer.
3. Edit your solution to question 2. This time, return a single column that holds the difference of each participants education level from the average. Name this column avg_edu_diff.
4. The mathematical function `POWER(expression, 2)` will raise any numeric expression to the power of 2. We want to apply this function to the avg_edu_diff column in your result from question 3. The query will get messy if you try to apply it directly. So, use your result from question 3 as a subquery in the `FROM` clause before calculating `POWER(avg_edu_diff, 2)`. Name the resulting column square_deviation.
5. Take the average of all entries in the square_deviation column in your result from question 4. The result is equal to the population variance of the Education column. You can check this, by comparing it to the result of this query:

```
SELECT VARP(CAST(Education AS DECIMAL))
FROM Notes.EduStudy;
```

Solutions to Exercise 3.7.5

1.

```
SELECT *, AVG(Education) OVER() AS avg_edu
FROM Notes.EduStudy;
```

2.

```
SELECT *, AVG(CAST(Education AS DECIMAL)) OVER() AS avg_edu
FROM Notes.EduStudy;
```

3.

```
SELECT Education - AVG(CAST(Education AS DECIMAL))
      OVER() AS avg_edu_diff
FROM Notes.EduStudy;
```

4.

```
SELECT POWER(T.avg_edu_diff, 2) AS square_deviation
FROM (
SELECT Education - AVG(CAST(Education AS DECIMAL))
      OVER() AS avg_edu_diff
FROM Notes.EduStudy
) T;
```

5.
```
SELECT AVG(POWER(T.avg_edu_diff, 2))
FROM (
  SELECT Education - AVG(CAST(Education AS DECIMAL))
         OVER() AS avg_edu_diff
  FROM Notes.EduStudy
) T;
```

Exercise 3.7.6

In this exercise we'll practice using windowed aggregation functions with the `OVER` and `PARTITION BY` clauses.

1. Start by joining the Ape.Banana and Ape.BananaTree tables, using appropriate primary/foreign key pairs. What does each row in the result represent?
2. Use the `OVER` clause with `PARTITION BY`, to edit your solution to question 1, so that the query returns the BananaID, TasteRank, TreeID, YearPlanted, and the minimum TasteRank over all bananas for given TreeID.
3. Edit your solution to question **??**, to discard all bananas that come from any tree with a minimum TasteRank of 1.

Solutions to Exercise 3.7.6

1. Each row represents one banana and the tree it came from.
```
SELECT *
FROM Ape.Banana B
   JOIN Ape.BananaTree T ON B.TreeID = T.TreeID;
```

2.
```
SELECT B.BananaID,
       B.TasteRank,
       B.TreeID,
       T.YearPlanted,
       MIN(B.TasteRank) OVER(PARTITION BY B.TreeID) AS MinTaste
FROM Ape.Banana B
   JOIN Ape.BananaTree T ON B.TreeID = T.TreeID;
```

3. Since the window function is computed in the `SELECT` clause, we will use a subquery to discard the rows:

```
SELECT *
FROM (
  SELECT B.BananaID,
         B.TasteRank,
         B.TreeID,
         T.YearPlanted,
         MIN(B.TasteRank)
            OVER(PARTITION BY B.TreeID) AS MinTaste
  FROM Ape.Banana B
     JOIN Ape.BananaTree T ON B.TreeID = T.TreeID
) S
WHERE S.MinTaste != 1;
```

Chapter 4

Independent development

In this chapter, our goal is to develop the three most important skills for working independently on SQL queries: reading documentation, creating data, and developing test cases. Reading SQL documentation includes understanding the esoteric ‘Syntax Conventions’. Creating test data involves creating tables, altering tables, inserting data, and updating data. Developing test cases involves taking a structured approach to thinking about how a SQL query might produce incorrect results.

4.1 Reading the docs

This short section is intended to give you the bare minimum to get you started reading the official T-SQL documentation. In my experience, few people actually read official documentation, but it is so valuable!

There are two secrets to being an effective programmer at any skill level. The first is that you will borrow a lot of code from other people. Often, somebody else has already done what you want to do, or at least something very similar. Online forums, like StackExchange and specialty discussion boards, are great places to ask questions. There are also tens, if not hundreds, of beginner and intermediate tutorials available online. It is worth trying a few tutorials, until you find one that suits you. My personal favourite is SelectStarSQL (click here). Before laying out his principles of pedagogy, the SelectStarSQL author, Kao, writes

> “ *I struggled to find something I could stand behind because I felt that a good resource had to be free, not require registration, and care about pedagogy – it had to genuinely care about its users and there was nothing like that around. By overcoming some minor technical hurdles, I believe that Select Star SQL has met this standard.*”

The second secret to being an effective programmer at any skill level is to be able to *read the docs*. Whatever language you're coding in, the documentation provided by the creator/maintainer is nearly always the most comprehensive and reliable source of information. As far as readability goes, well, let's just say that docs in general aren't famous for being beginner friendly. That's why I'm devoting this section to teaching you how to read some of the T-SQL and MySQL documentation. I should stress that the documentation is made for somewhat experienced programmers. Reading documentation is definitely a skill in itself, so give it some time to develop and don't be disheartened if it doesn't immediately make sense. The rewards are worth the journey, and a little understanding can go a long way.

4.1.1 How to read the documentation

The first thing you'll need is a reference sheet of the **Syntax Conventions**. The Syntax Conventions are essentially a set symbols that are used to make the documentation clearer and more succinct. You can find the Syntax Conventions via the two following links:

- MySQL Syntax Conventions
- T-SQL Syntax Conventions

Those reference sheets allow you to start making sense of the rest of the documentation. The essential parts are only a page long, and many conventions are shared by both dialects (T-SQL and MySQL). We will explain the conventions in detail in this section.

Once you have a reference sheet of conventions, start by heading over for a quick peek at the **Data Manipulation Language (DML)** pages, via the pair of links below. DML is the part of SQL that is devoted to inserting, updating, deleting and querying data. DML exists in contrast to the **Data Definition Language (DDL)** – the part of SQL that is dedicated to creating, altering and dropping whole tables. Here are links to the DML pages:

- MySQL DML docs
- T-SQL DML docs

You can poke around there, for a while, being overwhelmed by it all. And, once you've done that, you can head over to the most important part, for most people. The `SELECT` documentation:

- MySQL SELECT docs
- T-SQL SELECT docs

If you scroll to the large syntax box (that has something resembling code in it), the first thing you'll notice is, the description of `SELECT` looks like a completely horrible mess. The second thing you may notice is, if you look very carefully, *almost every clause that we've covered so far is actually in that mess*. You can find `SELECT`, `FROM`, `WHERE`, `JOIN`, `GROUP BY`, `HAVING`, and `ORDER BY`, all dispersed within a terrifying medley of symbols, that only reflect the cold indifference of a world of computer scientists and engineers, who exist only to communicate with machines, and who, no doubt, completely despise us for wanting to fly too close to the sun.

In reality, I have much respect for the people who thought of this succinct way to describe large parts of the language. With a little work, hopefully, you'll see why the Syntax Conventions are useful. Indeed, these people do not hate us entirely. In the case of T-SQL, you'll see our friends at Microsoft have put some thought into simplifying it, with a kind of simplified summary box, at the top of the `SELECT` docs page. Here's my own version of that summary box, with a few slight modifications, focusing *mostly* on things that we've seen so far:

```
SELECT [ALL | DISTINCT] select_expr [, select_expr] ...

[ INTO <new_table> ]

[ FROM <table_source> [,...n] ]

[ WHERE <search_condition> ]

[ GROUP BY {col_name | expr}, ... ]

[ HAVING <search_condition> ]

[ ORDER BY {order_by_expression [ ASC | DESC ]} [,...n] ]
```

Figure 4.1: A simplified version of the `SELECT` documentation, showing a mixture of T-SQL and MySQL syntax conventions.

Syntax Conventions indicate the kinds of SQL statements that are valid. So, we can reference Figure 4.1 to derive a large number of acceptable SQL statements. In particular, the Syntax Conventions are a tool to indicate: which clauses are optional versus necessary, which expressions can be repeated multiple times, and the order to write the clauses in. What Syntax Conventions don't do, is they don't explain what each valid SQL statement *actually does*. They give you a clue for <u>what</u> you can write, not why you should write it.

I've intentionally made Figure 4.1 a little more confusing than it needs to be, because I've mixed MySQL and T-SQL Syntax Conventions, for a challenge! This approach will give us material to work with below, and will ultimately help us get comfortable with the conventions faster.

In Figure 4.1, there are a few keywords we haven't seen yet (`ALL`, `DISTINCT`, and `INTO`). There are also a few lower-case words, vertical lines, square brackets, and curly braces. The Sytax Conventions table, below, explains those symbols:

Table 4.1: The most important Syntax Conventions in MySQL and T-SQL.

MySQL	T-SQL	Description
`[a]`	`[a]`	`a` is optional
`{a}`	`{a}`	`a` is not optional
`[a|b]`	`[a|b]`	optionally, choose `a` or `b`
`{a|b}`	`{a|b}`	not optional, choose `a` or `b`
`a [, a] ...`	`a [,...n]`	optionally repeat `a`, separated by commas
`label`	`<label>`	a label/placeholder that will be defined elsewhere

Take a moment to compare the Syntax Conventions in Table 4.1, to the simplified `SELECT` documentation in Figure 4.1. See if you can spot at least one instance of each convention. Notice, these T-SQL and MySQL conventions differ only by whether a placeholder has angle brackets, and by the convention for 'optionally repeat, separated by commas' (these are the last two rows of Table 4.1).

To get a grip on the Syntax Conventions, it helps to work through a few examples.

Example 4.1.1. For the first example, consider the first line from Figure 4.1:

```
SELECT [ALL | DISTINCT] select_expr [, select_expr] ...
```

The parts in square brackets are optional. So, the above tells us we must start with the word `SELECT`, and this must be followed with something called a `select_expr`. The `select_expr` is a placeholder. The MySQL SELECT docs define it with a sentence:

```
Each select_expr indicates a column that you want to retrieve.
```

Figure 4.2: The definition of `select_expr` from the MySQL SELECT docs.

The part `[, select_expr] ...` means we can optionally add as many column names as we want, separated by commas. So, we can validly write:

```
SELECT FirstName
```

And we can also validly write:

```
SELECT FirstName, LastName, FavColour
```

We are also given the option [ALL | DISTINCT]. This means we can optionally write either ALL or DISTINCT, but not both. Despite not having seen these two keywords before, we now know that we can validly write:

```
SELECT ALL FirstName, LastName, FavColour
```

Or, we can write:

```
SELECT DISTINCT FirstName, LastName, FavColour
```

Searching far down the MySQL SELECT docs, we get the explanation of these two options:

```
The ALL and DISTINCT modifiers specify whether duplicate rows
should be returned. ALL (the default), specifies that all matching
rows should be returned, including duplicates. DISTINCT specifies
removal of duplicate rows from the result set. It is an error to
specify both modifiers. DISTINCTROW is a synonym for DISTINCT.
```

Figure 4.3: The explanation for [ALL DISTINCT] from the MySQL SELECT docs.

We will practice applying DISTINCT during the exercises. Keep in mind, the above explanation in MySQL also applies to T-SQL, for this and many other clauses, being part of the SQL standard.

Example 4.1.2. As a second example, consider the excerpt from Figure 4.1:

```
SELECT [ALL | DISTINCT] select_expr [, select_expr] ...

[ FROM <table_source> [,...n] ]

[ GROUP BY {col_name | expr}, ... ]
```

We already dissected the first line, in Example 4.1.1. The second line tells us we can optionally include FROM <table_source>. Then, the T-SQL convention [,...n] tells us we can repeat <table_source>, separated by commas. The MySQL would have been [, <table_source>] ...

Finding the exact meaning of the <table_source> placeholder requires a bit of digging. In the T-SQL SELECT docs there are links to the T-SQL

FROM clause page, where the following can be found:

```
<table_source> Specifies a table, view, table variable, or derived
table source, with or without an alias, to use in the Transact-SQL
statement.
```

Figure 4.4: The definition of <table_source>, from the T-SQL FROM clause page.

The `GROUP BY` is also optional. However, if we choose to `GROUP BY`, then we are told, by `{col_name | expr}`, that we must either include `col_name` or `expr`. These two placeholders are defined in the MySQL docs (I told you I would mix T-SQL and MySQL for a challenge!). A little prior knowledge and guesswork tells us that `col_name` must be the name of the column we want to group by. An `expr` ('expression') is a bit more detailed. There is a whole MySQL docs page dedicated to explaining what constitutes a valid expression, and a similar docs page in T-SQL. Finally, the whole segment `{col_name | expr}, ...` is yet another way to tell us that we can optionally add more column names (or other expressions), separated by commas.

4.1.2 Going deeper with placeholders

In addition to the details in Figure 4.4, the same T-SQL FROM docs page provides a much more complex specification of the <table_source>, which looks something like:

```
<table_source> ::=
{
  complicated stuff
}
```

The symbol `::=` is used by T-SQL to indicate the beginning of a definition for the valid set of statements that can replace a placeholder. So, simplifying the complicated stuff, we might see:

```
<table_source> ::=
{
  table_or_view_name [ [ AS ] table_alias ]
 | user_defined_function [ [ AS ] table_alias ]
}
```

The above tells us that the <table_source> can be a `table_or_view_name` or a `user_defined_function` (we haven't yet seen what a view or a user defined function is, but we at least know what a table name is). It also tells us that the table name can be given an alias, and, if we like, the alias can be preceded by the keyword `AS`. Indeed, these are the alias and optional `AS` keyword that we learned about in Section 2.3.

Example 4.1.3. For practice, here's a toy example, that applies a few basic Syntax Conventions to an English language sentence, rather than SQL:

```
<greeting> ::= Hello. [Do you {love | hate} reading the docs?]
```

The above implies a valid `<greeting>` can be any one of these:

- Hello.
- Hello. Do you love reading the docs?
- Hello. Do you hate reading the docs?

The curly braces, together with the vertical bar, have insisted that we choose between 'love' and 'hate.'

4.1.3 Was this a waste of time?

I think, looking at the Syntax Conventions, and the depth of the documentation, it's tempting to say that you're better off just searching on forums for solutions when you're stuck. This is often true. I encourage it.

However, particularly when I forget syntax, yet have some idea of what I want, I've found the the official documentation to be an invaluable quick reference. It also comes with a bonus: when I'm sifting through the docs, looking for valid syntax, I almost always come across new keywords that spur some curiosity. I'll often dig a little deeper and find useful tools. Once I find new tools, I'll get some good ideas on how to use them, by searching through online forums.

Example 4.1.4. Trying to find the definition of `<select_list>`, from the T-SQL docs, I stumbled across the 'SELECT Clause' page (not to be confused with the SELECT docs page). Below is an excerpt, simplified a little by removing some optional parts.

```
SELECT [ ALL | DISTINCT ]
[ TOP ( expression ) [ PERCENT ] [ WITH TIES ] ]
<select_list>
<select_list> ::=
    {
      *
      | { table_name | table_alias }.*
      | {
          [ { table_name | table_alias }. ]
              { column_name }
        }
    } [ ,...n ]
```

Figure 4.5: A simplified excerpt from the T-SQL SELECT Clause page.

Some things jumped out at me. For example, it says I can write:

```
SELECT table_alias.*, table_alias.column_name
```

So, knowing in advance that `SELECT` can be paired with the `FROM` and `JOIN` clauses, I can write:

```
SELECT F.*, P.PetName
FROM Friends F JOIN Pets P ON F.friendID = P.friendID;
```

Knowing also what `SELECT *` does, I guessed that the query above would give me all columns of the Friends table, along with just the PetName column from the Pets table (after joining the two tables). At that point, I just executed the above query on my database, and found that, indeed, that's what it does!

That's not all I found. Looking at Figure 4.5 again, I saw I could write:

```
SELECT TOP ( expression ) <select_list>
```

Pairing this with `FROM`, I can write:

```
SELECT TOP ( 2 ) FriendID, FirstName
FROM Friends;
```

Executing the above query on my T-SQL database confirms that this extracts the first two rows of the Friends table. Executing it in MySQL returns an error, confirming that this syntax is specific to T-SQL, and so it must not be in the SQL standard.

That's quite enough of reading documentation for now! A large part of development is about experimentation. In the following section, we'll learn to create and manipulate data – skills that come in handy when it's time to experiment.

4.2 Creating databases and tables

Up until now, everything that we've learned has been part of the SQL Data Manipulation Language (DML). The DML is all about using an existing database structure. In this section, we'll learn the most important parts of the SQL Data Definition Language (DDL). The DDL is all about creating a database structure. The DDL includes creating, altering and dropping tables.

4.2.1 Creating, using and deleting databases

There are three handy commands to be aware of before we start creating and editing tables. These three commands allow us to create databases, activate them, and then delete them entirely when we're done. A SQL server consists of multiple databases, and new databases can easily be created, used and deleted. The three useful commands for this are `CREATE DATABASE`, `USE`, and `DROP DATABASE`. With these, we can create, use and delete whole databases without affecting other databases.

You can specify a database that you want to use with `USE`. For example, the following will specify that you want to use the database named Sandpit (assuming Sandpit exists, i.e., that someone has already created it).

```
USE Sandpit;
```

However, it's often useful to create your own fresh database from scratch, so that you can experiment without changing the structure of an existing database. The following will create a database named MyExperiments:

```
CREATE DATABASE MyExperiments;
```

Then, you can start using MyExperiments with:

```
USE MyExperiments;
```

Once you're done experimenting, or just want to start fresh, you can delete the entire MyExperiments database, with the following. **Warning: dropping a database deletes all tables and data in that database.**

```
DROP DATABASE MyExperiments;
```

4.2.2 The T-SQL `GO` keyword

Something to keep in mind with T-SQL (but not with MySQL), is that the above commands should all be followed by the keyword `GO`. In T-SQL, `GO` indicates the end of a 'batch'. Essentially, using `GO` just makes sure that all your statements are executed in the right order. So, in T-SQL, we would write, for example:

```
CREATE DATABASE MyExperiments;
GO

USE MyExperiments;
GO

-- add your experiments in here,
-- e.g., create tables, create data
-- and write some queries.

DROP DATABASE MyExperiments;
GO
```

While in MySQL we can just write:

```
CREATE DATABASE MyExperiments;

USE MyExperiments;

-- add your experiments in here,
-- e.g., create tables, create data
-- and write some queries.

DROP DATABASE MyExperiments;
```

4.2.3 Data types and `CREATE TABLE`

Usually, when a SQL table is created, it is empty, and only the structure is specified. The table structure consists in a table name, column names, primary/foreign key specifications, and a data type for each column. Additionally, each individual column can optionally be flagged `not null`, to disallow `NULL` values. We can see all of this happening below, with code that will create the Friends and Pets tables:

```
-- create the Friends table
CREATE TABLE Friends (
  FriendID INT not null,
  FirstName VARCHAR(20),
  LastName VARCHAR(20),
  FavColour VARCHAR(20),
  PRIMARY KEY (FriendID)
);

-- create the Pets table
CREATE TABLE Pets (
  PetID INT not null,
  PetName VARCHAR(20),
  PetDOB DATE,
  FriendID INT not null,
  FOREIGN KEY (FriendID) REFERENCES Friends (FriendID),
  PRIMARY KEY (PetID)
);
```

Figure 4.6: Creating the Friends and Pets tables

Here are some useful comments on the syntax shown in Figure 4.6:

- The data type `VARCHAR(20)` specifies a variable length character string, with maximum length of 20. Providing a maximum length helps the database management system 'plan ahead' to reduce overall use of memory space. The maximum length 20 means that, for example, if I meet a friend with a FirstName that is longer then 20 characters, then I will only be able to store the first 20 letters of their name.

- The condition `not null`, appearing after the data types for FriendID and PetID, ensures that these columns can never contain `NULL` values. In the case of the FriendID in the Pets table, this ensures that we never store the details of a pet whose owner is unknown. Excluding `NULL` values is also necessary for any column that we plan to make into a primary key, since primary keys must never be `NULL`.

- The line `PRIMARY KEY (FriendID)` specifies that the column FriendID is a primary key. Lastly, the line `FOREIGN KEY (FreindID) REFERENCES Friends (FriendID)` specifies that the FriendID in the Pets table is a foreign key 'pointing at' (i.e., referencing) the FriendID in the Friends table.

Now that you've seen the example syntax in Figure 4.6 and the corresponding comments, here is a simplified general `CREATE TABLE` syntax, making use of some of the Syntax Conventions we learned about in Section 4.1.1 (as summarised in Table 4.1):

```
CREATE TABLE table_name (
  [column_name data_type [not null]], ...
  [PRIMARY KEY (column_name)],
  [FOREIGN KEY (column_name) REFERENCES table_name (column_name)]
);
```

Figure 4.7: The basic general syntax for creating tables in SQL.

When you need to store numeric data, it is almost always best to use a `DECIMAL(p,s)` data type. For example, `DECIMAL(5,2)` stores 5 digit numbers with 2 decimal places, meaning the largest number that can be stored in `DECIMAL(5,2)` is 999.99. If you are only storing whole numbers, and you don't plan to aggregate the numbers, then it may be fine to use `INT` instead (see Section 3.2.2 for a discussion on the perils of aggregating `INT` data).

Aside from the need to learn more about data types, such as `DATE` and `TIME` (which we will do in Section 4.3.2), the syntax in Figure 4.7 gives you everything you need, to create most kinds of tables. However, there are many more fancy things that can be done with the `CREATE TABLE` statement, that we don't cover in these notes. If you take a moment to briefly check out the MySQL or T-SQL docs for creating tables, you'll see that the `CREATE TABLE` syntax can get much more complicated. Regardless, I have rarely needed to go beyond the basic syntax described in Figure 4.7.

4.2.4 ALTER TABLE

It's not uncommon to want to add or remove columns from tables, or to change the data types of columns. This is not part of the typical daily use of a relational database. The columns in tables usually remain static in a relational database, by design. Typical daily use involves inserting or deleting rows, and updating data (which we learn about in Section 4.3.2), or performing queries to retrieve data. Regardless, altering the table column structure is not uncommon, as a database matures and its purpose changes over time.

We'll return to our Friends table as an example case for `ALTER TABLE`. The following statement will alter the Friends table to add two new columns, called StartDate and StartTime, to record the exact date and time we became friends. These two new columns will have the `DATE` and (you guessed it) `TIME` data types, respectively.

```
-- (T-SQL only) add StartDate and StartTime
ALTER TABLE Friends
ADD StartDate DATE, StartTime TIME;

-- (MySQL only) add StartDate and StartTime
ALTER TABLE Friends
ADD ( StartDate DATE, StartTime TIME );
```

After executing the statement above, the new columns will automatically be filled with NULL values. So, Friends will look like this:

Friends					
FriendID	FirstName	LastName	FavColour	StartDate	StartTime
1	X	A	red	NULL	NULL
2	Y	B	blue	NULL	NULL
3	Z	C	NULL	NULL	NULL

Figure 4.8: The Friends table with new StartDate and StartTime columns.

We will see how to update those NULL values, in the Section 4.3.2, but for now we'll keep exploring what ALTER TABLE can do. We've seen that adding a column is straightforward. Dropping a column is usually straightforward. The following statement deletes the StartDate column that we just created.

```
ALTER TABLE Friends
DROP COLUMN StartDate;
```

The above works fine, but what if we try to drop the foreign key FriendID from the Pets table?

```
ALTER TABLE Pets
DROP COLUMN FriendID;
```

```
Error: cannot drop column 'FriendID': needed in a foreign key
constraint 'pets_ibfk_1'
```

Figure 4.9: Error dropping the foreign key FriendID from Pets.

The error in Figure 4.9 has prevented us from accidentally losing the relationship between Friends and Pets, since this relationship is essentially stored in the foreign key FriendID. The name pets_ibfk_1 was automatically assigned by the database management system when we created the foreign key FriendID, so it's not something we've seen before. Regardless, now that we've seen the above error, we can get around it by specifically dropping the foreign key constraint pets_ibfk_1. This implies we are intentionally removing the relationship between Friends and Pets. The following statement uses ALTER TABLE to drop a constraint.

```
ALTER TABLE table_name
DROP CONSTRAINT pets_ibfk_1;
```

Aside from adding and deleting columns, the ALTER TABLE clause can also be used to change the data type of an existing column. Recall from Figure 4.6 that the data type for FirstName is VARCHAR(20), meaning we can store names up to 20 characters in length. If we meet a friend with a longer first name, we might want to increase that capacity. We'll alter the FirstName column, changing the data type, to double its maximum length (to 40 characters). Annoyingly, this statement has a slightly different syntax in T-SQL to MySQL:

```
-- change a data type in MySQL (uses 'modify column')
ALTER TABLE Friends
MODIFY COLUMN FirstName VARCHAR(40);

-- change a data type in T-SQL (uses 'alter column')
ALTER TABLE Friends
ALTER COLUMN FirstName VARCHAR(40);
```

Altering data types can cause problems when the new data type is not compatible with the old data type, though this issue only arises if the column already contains data that aren't NULL . The database management system will automatically try to convert the existing data to the new data type, and if it fails to do the conversion without data loss, it will produce an error. For example, suppose we try to convert FavColour to a VARCHAR of length 3 (keeping in mind that the word 'blue' is already in the FavColour column, and has length 4):

```
-- using the MySQL syntax
ALTER TABLE Friends
MODIFY COLUMN FavColour VARCHAR(3);
```

```
Error Code: 1265. Data truncated for column 'FavColour'
```

Figure 4.10: An error produced when we try to reduce the FavColour VARCHAR length to something shorter than the word 'blue'

The error in Figure 4.10 is returned, refusing to truncate the word 'blue' to only 3 letters. This error prevents unexpected data loss. In the next section, we'll see how we can first manually truncate the data using UPDATE, to avoid this error.

4.2.5 DROP TABLE

Dropping (i.e., deleting) tables is usually quick and easy. The two things to keep in mind are:

- **Warning:** dropping a table deletes the whole table and all the data in the table, often without warning.
- A table can't be deleted if there is a foreign key pointing at it. This is to maintain referential integrity – a concept we introduced way back in Section 1.4.1.

We can try to drop the Friends table, with the following:

```
DROP TABLE Friends;
```

```
Error: cannot drop table 'friends' referenced by a foreign key
constraint 'pets_ibfk_1' on table 'Pets'.
```

Figure 4.11: Error dropping the Friends table, because Pets has a foreign key pointing at it.

The error in Figure 4.11 is produced by the above code, because the Pets table has the foreign key friendID, pointing at the Friends table. We can circumvent this issue by first dropping the foreign key constraint from the Pets table, using what we learned in Section 4.2.4:

```
-- remove the foreign key from Pets
ALTER TABLE Pets
DROP CONSTRAINT pets_ibfk_1;

-- delete the Friends table
DROP TABLE Friends;
```

We've learned how to create, alter and drop tables. We've also seen that altering and deleting tables can sometimes get a little hairy (e.g., when foreign key constraints are involved or when new data types are not compatible with existing data). When you're experimenting with SQL (creating tables to test your own queries) it's often easiest to use the commands in Section 4.2.1 to create a whole new database, use it for your tests or experiments, then drop the whole database when you're done. We'll see examples of the whole process in Section 4.4.

4.3 Creating data and views

In Section 4.2, we learned about the Data Definition Language (DDL), allowing us to create and modify databases and tables. In this section, we return to the Data Manipulation Language (DML). We'll learn to insert data, update existing data, delete specific rows of data, and save the results of queries into tables. After that, we'll learn about 'views' – objects that look and feel like tables, but are actually just conveniently stored queries.

4.3.1 `INSERT` rows

Let's just go ahead and insert two new rows of data into the Friends table. Note, we're working with the original Friends table here, from way back in Figure 1.4 (that is, ignoring any alterations we made in Section 4.2).

```
INSERT INTO Friends
VALUES
(4, 'Kimmy', 'Jenkins', 'yellow'),
(5, 'Jimmy', 'Jenkins', NULL);
```

We now have two new friends, Kimmy and Jimmy Jenkins! The Friends table now looks like this:

Friends			
FriendID	FirstName	LastName	FavColour
1	X	A	red
2	Y	B	blue
3	Z	C	`NULL`
4	Kimmy	Jenkins	yellow
5	Jimmy	Jenkins	`NULL`

Figure 4.12: The Friends table with two new rows of data.

Here are some comments on the above code:

- The data must be written in the same order as the column names: if we wrote `('Kimmy', 4, 'Jenkins', 'yellow')`, then we would have set the FriendID to `'Kimmy'` and the FirstName to `4`.
- We can add as many friends as we like: each row should be enclosed in round brackets, and separated by commas.
- Character strings like `'Kimmy'` must be written with quote marks, while numbers like `4` should be written without quote marks. Entries for `DATE` and `TIME` also require quote marks, but they must be written in

the right format, which we discuss in Section 4.3.2 (and the format was also given in the description column of Table 2.1).

- Inserted data must match the data type of the column. If we tried to insert `'Kimmy'` for the FriendID, then we'd get an error: `Incorrect integer value: 'Kimmy' for column 'FriendID'`. This is because the FriendID is an integer (defined using `INT` in Section 4.2.3), while `'Kimmy'` is a character string (defined using `VARCHAR`).

For Jimmy Jenkins, we inserted a `NULL` value for FavColour. Another way to insert `NULL` values is to include a list of column names for which the data are not `NULL`. Here's an example, where we insert two more friends, Niko and Sage, whose last names and favourite colours are `NULL`:

```
INSERT INTO Friends
(FriendID, FirstName)
VALUES
(6, 'Niko'),
(7, 'Sage');
```

When people first learn how to insert data in SQL, it's usually not long before they ask how to insert a whole Excel file or CSV. I'll pre-empt that question with an important clarification: standard SQL doesn't have that feature! Don't get me wrong, there are tools for it. Most decent MySQL or T-SQL code editors will almost definitely have a built-in interface (or so-called 'wizard') for importing CSV or Excel files and inserting the data into SQL tables. Those tools are not SQL though.

If you have a CSV or Excel data file, and you want (or need) to use standard SQL code to insert the data, there is a really handy online conversion tool available at konbert.com/convert/csv/to/sql. It converts CSV files to SQL `INSERT` statements. If you're using Excel, you'll need to export the Excel file as a CSV first (which can be done easily in Excel).

4.3.2 `UPDATE` and `DELETE` rows

In Section 4.2.4, we altered the structure of the Friends table, adding columns StartDate and StartTime. We saw (in Figure 4.8) that these two columns were then automatically filled with `NULL` values. Here's a summary of steps taken to reach that point:

```
-- first create the structure
CREATE TABLE Friends (
  FriendID INT not null,
  FirstName VARCHAR(20),
  LastName VARCHAR(20),
  FavColour VARCHAR(20),
  PRIMARY KEY (FriendID)
);

-- then insert our 3 friends
INSERT INTO Friends
VALUES
(1,'X','A','red'),
(2,'Y','B','blue'),
(3,'Z','C', NULL);

-- (T-SQL only) add StartDate and StartTime
ALTER TABLE Friends
ADD StartDate DATE, StartTime TIME;

-- (MySQL only) add StartDate and StartTime
ALTER TABLE Friends
ADD COLUMN ( StartDate DATE, StartTime TIME );
```

Now we're going to update StartDate and StartTime. When inserting or updating `DATE` data, the date should be specified as a character string with the format `'YYYY-MM-DD'`. For `TIME` data, the format is `'HH:MM:SS'`. The following will set StartDate to December 30th, 1999, and StartTime to 4:30pm. **Warning: update statements like the below will happily overwrite every entry in whatever columns we specify! This can easily lead to unexpected data loss.**

```
UPDATE Friends
SET StartDate = '1999-12-30', StartTime = '16:30:00';
```

Friends now looks like this:

Friends					
FriendID	FirstName	LastName	FavColour	StartDate	StartTime
1	*X*	*A*	red	1999-12-30	16:30:00
2	*Y*	*B*	blue	1999-12-30	16:30:00
3	*Z*	*C*	NULL	1999-12-30	16:30:00

Figure 4.13: The same StartDate and StartTime for every row in Friends.

Our `UPDATE` statement overwrote the StartDate and StartTime for every row in the Friends table. That's not usually desirable. The `UPDATE` clause can use a search condition (in a `WHERE` clause) to determine where to insert data. The correct way to use `UPDATE` is usually to specify just one row to update. The best way to specify one row, is via the primary key. In the following, we update the StartDate and StartTime, just for our friend X, who has primary key entry 1:

```
UPDATE Friends
SET StartDate = '2000-01-03', StartTime = '08:00:00'
WHERE FriendID = 1;
```

Friends now looks like this:

Friends					
FriendID	FirstName	LastName	FavColour	StartDate	StartTime
1	X	A	red	2000-01-03	08:00:00
2	Y	B	blue	1999-12-30	16:30:00
3	Z	C	NULL	1999-12-30	16:30:00

Figure 4.14: Friends after updating StartDate and StartTime for our friend X.

The `DELETE` statement works much like `UPDATE`: if no `WHERE` clause is given in a `DELETE` statement, **then every row of the table will be deleted**. As with `UPDATE`, it is usually best to use match a single primary key value when deleting data:

```
DELETE FROM Friends
WHERE FriendID = 999;
```

Since there is currently no FriendID equal to 999, the above statement will have no effect on the Friends table.

4.3.3 Insert data with `SELECT`

All of Chapters 2 and 3 were dedicated to using `SELECT` statements to chop up tables, join them, and produce new data (such as aggregates or ranks). It's common to want to save the results of those `SELECT` statements into tables. But, why save the results of a query into a table, when you can easily just run the query again to get the same table?

One common scenario is that a query might take a long time to execute, especially if it involves complex processing of large amounts of data. In such cases, saving the results into a table means they will be computed in advance, and can be rapidly accessed when needed. This works well when the database is not updated frequently, or when the query results do not need to be perfectly up to date.

Another common scenario is that the database may be changing over time, and you may want to save a snapshot of the data, either in its original form or as some kind of aggregate. For example, an organisation may frequently insert and delete members from their Membership table, and may want to count how many members there are each month. The results could be saved into a table, along with the date, to provide a historical record of membership tallies.

Example 4.3.1. Here's an example of a table that records membership:

```
CREATE TABLE Membership (
 memberID INT not null,
 memberName VARCHAR(100),
 phone VARCHAR(20),
 joinDate DATE,
 PRIMARY KEY (memberID)
);
```

In this organisation, when a member joins, a row is inserted:

```
INSERT INTO Membership
VALUES (12231, 'Denali Dune', '+61 03 97229917', '2021-06-21');
```

When a membership is cancelled, the corresponding row is deleted:

```
-- deleting member Denali Dune
DELETE FROM Membership
WHERE memberId = 12231;
```

Here's some example Membership data for us to work with:

```
INSERT INTO Membership
VALUES
(12688, 'Reilly Bierman', '+61 03 9269 1200', '2021-05-01'),
(12233, 'Shiloh Henry', '+61 03 9479 6000', '2021-05-13'),
(12565, 'Tristan Gaumond', '+61 03 9905 4000', '2021-05-04'),
(12223, 'Rene Brassard', '+61 03 9903 2000', '2021-06-30'),
(12668, 'Tanner Hubert', '+61 03 9035 5511', '2021-07-29');
```

Counting the number of members at any given time corresponds to counting the number of rows of the Membership table. We can include the date in our results, using the useful function `SYSDATE` (in MySQL), or `SYSDATETIME` (in T-SQL).

```
-- in T-SQL replace SYSDATE with SYSDATETIME
SELECT COUNT(*) AS MemberCount, SYSDATE() AS ExecutionDateTime
FROM Membership;
```

If we executed the above query on the 30th of July 2021, at 2:15pm, the result would have looked like this:

RESULT	
MemberCount	ExecutionDateTime
5	2021-07-30 14:15:00

Figure 4.15: Member count result, executed on 30th of July 2021.

If we want to run the above query and save the result as a table, then in MySQL we can use the `CREATE TABLE ... SELECT` statement, below. Subsequently, each month, we can perform the query again, and save the results with the `INSERT INTO ... SELECT` statement. In T-SQL, on the other hand, both tasks are achieved with a single statement, called `SELECT INTO`.

```
-- MySQL only: table creation and initial insert
CREATE TABLE MemberCountHistory
SELECT COUNT(*) AS MemberCount, SYSDATE() AS ExecutionDateTime
FROM Membership;

-- MySQL only: subsequent inserts (execute once per month)
INSERT INTO MemberCountHistory
SELECT COUNT(*) AS MemberCount, SYSDATE() AS ExecutionDateTime
FROM Membership;

-- T-SQL only: initial table creation (and execute once per month)
SELECT COUNT(*) AS MemberCount, SYSDATE() AS ExecutionDateTime
INTO MemberCountHistory
FROM Membership;
```

4.3.4 `CREATE VIEW`

A view is a valuable and frequently used tool in SQL. A view is a derived table that gets derived again every time you use it. On the surface, a view behaves (almost) the same way as a table. However, under the hood, a view is essentially a stored query.

Any `SELECT` statement can be stored as a view, by simply prepending it with the line `CREATE VIEW MyView AS`. For example, if my typical daily use of the Friends and Pets tables mainly involves the need to pair friends names with their pets names, then I might store this data in a view:

```
CREATE VIEW FriendsPets AS
SELECT F.FirstName, P.PetName
FROM Friends F JOIN Pets P ON F.FriendID = P.FriendID;
```

Now, any time I want to access the above results, I can use the FriendsPets view, rather than typing out the whole above query.

```
SELECT *
FROM FriendsPets;
```

RESULT	
FirstName	PetName
Y	Chikin
Z	Cauchy
Z	Gauss

Figure 4.16: The output of `SELECT * FROM FriendsPets;`.

Example 4.3.2. Recall the SausageSizzleSummary table from Figure 3.23. That table provided quantities of daily sales of pork and veggie sausages from a six day sausage sizzle starting on New Year's Eve, 1999. For our purposes in Section 3.5.3, it was useful to have access to the daily total sales. However, at the sausage sizzle, data were recorded for each individual sale, using the SausageSizzle table. Here are the first few rows of SausageSizzle:

SausageSizzle					
SaleId	SaleDate	Product	Quantity	FriendId	Paid ($)
1	1999-12-31	pork	1	NULL	3
2	1999-12-31	veggie	3	NULL	9
3	1999-12-31	pork	2	1	2
4	2000-01-01	veggie	4	NULL	4
5	2000-01-01	veggie	2	2	6

Figure 4.17: The first few rows of the SausageSizzle table.

Each time someone approaches the store and buys some sausages, the sale date, type of sausage (product), and quantity of product, are recorded. So, if someone buys 3 veggie sausages and 1 pork sausage, then two rows will need to be added to the table. The table also includes a FriendID foreign key column, to keep track of any sales to friends (since those sales were discounted). Finally, there is a column 'Paid ($)' to indicate how much money changed hands during the sale.

We can easily make SausageSizzleSummary from SausageSizzle. It should be fine to save SausageSizzleSummary as a table, using what we learned in Section 4.3.3, since the sausage sizzle is over and we don't expect SausageSizzle to change. However, back in 1999, during (or even before) the sausage sizzle, it may have been ideal to create SausageSizzleSummary as a view. As a view, SausageSizzleSummary will always be an up to date reflection of the contents of SausageSizzle.

```
CREATE VIEW SausageSizzleSummary AS
SELECT Product, SaleDate, SUM(Quantity) AS Sales
FROM SausageSizzle
GROUP BY Product, SaleDate;
```

Now, any time we want to see daily quantities of sausage sales, we can just `SELECT * FROM SausageSizzleSummary`, returning the table in Figure 3.23.

4.4 Test cases and development workflow

At the start of the current chapter, I stated our goal to develop the three most important skills for working independently in SQL: reading documentation, creating data, and developing test cases. Arguably, developing test cases is the most important of the three.

Test cases are small sets of artificial data, created by hand as we write new queries; they are example data, for which the expected output of a query is known in advance. A query can be syntactically correct (i.e., run without errors), yet produce logically incorrect results (i.e., results that are senseless or unintended). By running a query on test cases, we can make sure the query produces logically correct outputs.

In addition to confirming that a query produces correct results, test cases can speed up the development process. Paired with a good workflow, they become a means to writing far more complicated queries, faster. We will learn how to use test cases to build large queries piece-by-piece, testing each piece as we go.

Finally, test cases allow us to quickly explore new clauses and functions, in the most hands-on way possible. Stumble upon a new tool in the documentation and think you know how it works? Write some test cases and experiment. The surest way to understand how things work is to experiment, using carefully handcrafted data. Save your test cases somewhere, and you'll soon find you have a small database of valuable examples to sharpen your skills on.

In summary, test cases allow you to independently build robust queries, with logically correct results, more rapidly, while deepening your knowledge of SQL clauses, functions and other tools – all while writing your personal collection of example data, that you'll know back to front. At this stage, you could probably dive in and intuit your own approach to writing test cases. However, in this section, I'll lay down some fundamentals for writing better test cases, and I'll try to exemplify a good workflow.

4.4.1 Our first real dataset (peek with `TOP` and `LIMIT`)

To illustrate testing and development, we'll use our first real dataset: the Stack Exchange Data Explorer platform. This database includes (at the time of writing) over 37 million answers and 100 million comments, for more than 26 million questions, from Stack Exchange – an excellent collection of community Q&A sites. You can browse the Stack Exchange Data Explorer platform and find thousands of user generated SQL queries, making it a great learning resource. We'll focus on a smaller subset of this database, containing questions and answers from the Statistical Analysis Site.

Our goal

In statistics, two dominant schools of reasoning exist that have apparently conflicting philosophical interpretations of probability. These two schools are referred to as 'Bayesians' and 'frequentists'. Our goal is to query the Stack Exchange Statistical Analysis posts to compare community appreciation for posts mentioning the word 'frequentist', to those mentioning the word 'Bayesian', for each month, across all years prior to (and including) 2020. So, grouping by month, we want to see the average score, average number of views, and total number of posts, in each of the two categories, 'frequentist' versus 'Bayesian'. Note that by no means does this constitute a thorough investigation, or even a thorough research question – we're just doing some good old exploration.

Accessing the data

This dataset can be queried using T-SQL, direct from a web browser, at data.stackexchange.com/stats/query/new. Since the platform uses T-SQL, and you may be using MySQL, we'll make sure to write our queries using syntax that works in both dialects, wherever possible.

Since this dataset is quite large, queries take a long time to execute. This won't bother us, because we'll be developing our own fake test data to refine our code, and we'll only execute our queries on the real dataset when ready. Of course, we'll start by taking a peek at the real data, so we know what we're dealing with. MySQL and T-SQL both have the capability to extract only the first few rows of a dataset, but the syntax differs:

```
-- T-SQL only: extract first 10 rows of Posts table
SELECT TOP(10) *
FROM Posts;

-- MySQL only: extract first 10 rows of Posts table
SELECT *
FROM Posts
LIMIT 10;
```

Head over now to the Statistical Analysis Site query page, and execute the above (T-SQL code only) to view the first 10 rows of the Posts table. Note, you may want to create an account and log in first, so you don't have to keep completing the CAPTCHA each time you execute a query. The columns that we're interested in from Posts are Id, CreationDate, Score, ViewCount and Body. On the right of that page, you can see database schema details, giving the data type for each column in each table. Note there is also a data dictionary available on the site.

Creating a test table

Before we can start writing our own fake data as test cases, we need a database and table to store them in. The following will get us started. In our local server (not the web browser), we can execute the following:

```
CREATE DATABASE StackOverflowTesting;
GO -- only use GO in T-SQL, remove GO for MySQL

USE StackOverflowTesting;
GO -- only use GO in T-SQL, remove GO for MySQL

CREATE TABLE Posts (
  Id INT NOT NULL,
  CreationDate DATETIME,
  Score INT,
  ViewCount INT,
  Body VARCHAR(100),
  PRIMARY KEY (Id)
);
```

Note the Stack Exchange database schema uses `NVARCHAR` for Body, which can can fit more types of special characters (like smiley faces). We have used `VARCHAR` instead, because our test data will be simpler.

4.4.2 Fastball testing

The first rule of testing is to test early. The second rule of testing is to test thoroughly. We're going to practice 'fastball' testing for testing early, and 'validity' testing to test thoroughly. **Fastball testing** is the process of writing quick and dirty test data to run through a query as early as possible. **Validity testing** is the process of carefully choosing test cases that push the limits of a query. Fastball testing makes sure that queries execute without error and produce the most basic desired output. Validity testing makes sure queries are robust to unexpected or strange inputs.

For fastball testing, we want to quickly make a small amount of data that we can use to check that our query is working. We can then add more data as we build the query. To get started, we can run the following in our local database (not the web browser):

```
INSERT INTO Posts
VALUES
(1, '2020-01-01',1,200,'dummy text'),
(2, '2020-02-01',1,200,'dummy frequentist'),
(3, '2020-03-01',1,200,'dummy text'),
(4, '2020-03-01',1,200,'dummy bayesian');
```

Now we want to start building our query piece-by-piece, executing it as we go, and comparing the results to our test data. This means we need to stay aware of the output that we expect to produce. For this reason, it's often a good idea to use a pen and paper to jot down what you expect your output to look like. It is possible to store your expected output in a SQL table, for a more direct comparison, but that is laborious and best left for the validity testing phase.

An online search (or documentation search) tells us we can use `MONTH` to extract just the month part of a `DATETIME` column. We will also use `CASE WHEN` (introduced in Section 2.4.7) to categorise posts according to the presence of words 'Bayesian' or 'frequentist'. Within `CASE WHEN`, our search condition uses `LIKE`, with the wild card '%' (as introduced in Section 2.4.5), to check if a word appears anywhere in the body. We'll keep the query simple, for now.

```
SELECT MONTH(CreationDate) AS CreationMonth,
       CASE WHEN Body LIKE '%frequentist%' THEN 'F'
            WHEN Body LIKE '%bayesian%' THEN 'B'
            END AS Category
FROM Posts;
```

Checking the output of the above query, we see that Category is `NULL` whenever Body does not contain 'frequentist' or 'Bayesian'. This is fine, since we plan to discard those rows soon anyway. After becoming confident that `MONTH` and `CASE WHEN` are working as expected, we can add a `WHERE` clause to filter out posts we're not interested in. We'll exclude all years after 2020, and remove rows where Body does not contain 'frequentist' or 'Bayesian'. Note that, annoyingly, we can't access Category in the `WHERE` clause, because it is created in the `SELECT` clause (see Section 2.2.7 for a refresher on this). To check the `WHERE` clause is working well, we'll add a row from 2021.

```
-- insert a row with year 2021, to be filtered by WHERE
INSERT INTO Posts VALUES
(5, '2021-01-01',1,200,'dummy frequentist');

-- extend the query with WHERE
SELECT MONTH(CreationDate) AS CreationMonth,
       CASE WHEN Body LIKE '%frequentist%' THEN 'F'
            WHEN Body LIKE '%bayesian%' THEN 'B'
            END AS Category
FROM Posts
WHERE YEAR(CreationDate) <= 2020
      AND Body LIKE '%bayesian%' OR Body LIKE '%frequentist%';
```

The result includes a row for the group January/frequentist. That's a problem, because our test data includes only one January 2020 row, which has no ‘frequentist’ in the Body. Since rows are not being excluded properly, we can narrow the problem down to the `WHERE` clause. Can you find it? **Hint:** see Section 2.4.4, and note the search condition in the `WHERE` clause is:

```
YEAR(CreationDate) <= 2020 AND
   Body LIKE '%bayesian%' OR Body LIKE '%frequentist%'
```

The problem is to do with operator precedence. The above search condition matches any row with ‘Bayesian’ in the Body, regardless of the year. Round brackets will fix this operator precedence issue:

```
YEAR(CreationDate) <= 2020 AND
   (Body LIKE '%bayesian%' OR Body LIKE '%frequentist%')
```

Next, we can add a `GROUP BY` clause. We will group by CreationMonth and Category – repeating the same `CASE WHEN` expression, within `GROUP BY` (as discussed in Section 3.2.3). Within each group, we’ll aggregate Score with `AVG(Score)`. However, currently, our test data has at most one row in each group. Since we want to see the effect of `AVG`, we’ll add one more row of data to the March/Bayesian group:

```
-- insert another row for March/Bayesian group
INSERT INTO Posts VALUES
(6, '2020-03-01',2,200,'dummy bayesian');

-- look at the current state of Posts
SELECT * FROM Posts;

-- extend the query with GROUP BY and AVG
SELECT AVG(Score) AvgScore,
       MONTH(CreationDate) AS CreationMonth,
       CASE WHEN Body LIKE '%frequentist%' THEN 'F'
            WHEN Body LIKE '%bayesian%' THEN 'B'
            END AS Category
FROM Posts
WHERE YEAR(CreationDate) <= 2020
      AND (Body LIKE '%bayesian%' OR Body LIKE '%frequentist%')
GROUP BY MONTH(CreationDate),
       CASE WHEN Body LIKE '%frequentist%' THEN 'F'
            WHEN Body LIKE '%bayesian%' THEN 'B'
            END;
```

The above gives us a side-by-side comparison of our query results with our test data. I encourage you to run the code yourself, to look at the outputs. You’ll notice that our query returns an AvgScore of 1 for every row. This is a problem, because we expect to see an AvgScore of 1.5 in the March/Bayesian group. Take a moment to try to solve the problem. **Hint:** see Section 3.2.2.

After some head scratching, or by finding an online post for a similar problem, you may realise: since Score is an `INT`, the output of `AVG(Score)` is being truncated to an `INT`. We can fix this by 'casting' Score to a `DECIMAL` data type, with `CAST(Score AS DECIMAL)`. Along with this fix, we'll add the `AVG` ViewCount, and we'll also `COUNT` the number of answers:

```
SELECT AVG(CAST(Score AS DECIMAL)) AvgScore,
       AVG(CAST(ViewCount AS DECIMAL)) AS AvgViews,
       COUNT(*) AS NumPosts,
       MONTH(CreationDate) AS CreationMonth,
       CASE WHEN Body LIKE '%frequentist%' THEN 'F'
            WHEN Body LIKE '%bayesian%' THEN 'B'
            END AS Category
FROM Posts
WHERE YEAR(CreationDate) <= 2020
      AND (Body LIKE '%bayesian%' OR Body LIKE '%frequentist%')
GROUP BY MONTH(CreationDate),
       CASE WHEN Body LIKE '%frequentist%' THEN 'F'
            WHEN Body LIKE '%bayesian%' THEN 'B'
            END;
```

Running the above query, it looks like our `AVG` problem is fixed. Our query is starting to look ready for a more thorough approach to testing. However, the query has a lot of repetition, and is getting difficult to manage. We should first remedy some of that via the `WITH` clause.

4.4.3 Reducing repetition via `WITH`

Our goal is now to make the query a bit more manageable, without changing the output of the query at all. We might not necessarily make the query *shorter*, but we will reduce repetition.

The `WITH` clause allows us to give an alias to an entire query, so that the query behaves like a table (similar to a `VIEW`, as covered in Section 4.3.4). The main difference is, unlike a `VIEW`, the alias isn't stored anywhere for use later; it can only be used immediately after the `WITH` clause. When using the `WITH` clause, the alias is called a 'Common Table Expression (CTE)'. You can read the MySQL or T-SQL documentation for the full `WITH` clause syntax. A simplified version meeting most use cases is:

```
WITH cte_name AS (subquery)
```

Here, `cte_name` is the alias given to `subquery`, and `subquery` is the query that you want to give an alias to. The subquery should be followed by a `SELECT` clause. Here's a simple example, using the name PostCats in place of `cte_name`.

```
WITH PostCats AS (
    SELECT MONTH(CreationDate) AS CreationMonth,
           CASE WHEN Body LIKE '%frequentist%' THEN 'F'
                WHEN Body LIKE '%bayesian%' THEN 'B'
                END AS Category
    FROM Posts
    WHERE YEAR(CreationDate) <= 2020
    )
SELECT *
FROM PostCats;
```

The above causes the PostCats alias to behave like a table with two columns: CreationMonth and Category. It then simply displays the whole PostCats table with `SELECT *`. One advantage of creating PostCats is that we can now create a `GROUP BY` clause that uses Category, instead of repeating the whole `CASE WHEN` expression (as we were forced to do previously). Similarly, we can use Category in the `WHERE` clause to filter out posts that don't contain 'Bayesian' or 'frequentist':

```
WITH PostCats AS (
    SELECT MONTH(CreationDate) AS CreationMonth,
           CASE WHEN Body LIKE '%frequentist%' THEN 'F'
                WHEN Body LIKE '%bayesian%' THEN 'B'
                END AS Category
    FROM Posts
    WHERE YEAR(CreationDate) <= 2020
    )
SELECT CreationMonth, Category
FROM PostCats
WHERE Category IS NOT NULL
GROUP BY CreationMonth, Category;
```

To be able to get averages after grouping, we need Score and ViewCount to be available within PostCats (as `DECIMAL` values):

```
WITH PostCats AS (
    SELECT MONTH(CreationDate) AS CreationMonth,
            CASE WHEN Body LIKE '%frequentist%' THEN 'F'
                  WHEN Body LIKE '%bayesian%' THEN 'B'
                  END AS Category,
    CAST(Score AS DECIMAL) AS Score,
    CAST(ViewCount AS DECIMAL) AS ViewCount
    FROM Posts
    WHERE YEAR(CreationDate) <= 2020
    )
SELECT CreationMonth, Category,
        AVG(Score) AS AvgScore,
        AVG(ViewCount) AS AvgViews,
        COUNT(*) AS NumPosts
FROM PostCats
WHERE Category IS NOT NULL
GROUP BY CreationMonth, Category;
```

Our query now involves much less repetition. Another benefit is that the subquery PostCats is a whole separate query that we can independently check during validity testing, if we need to. This can sometimes separate testing into more manageable components.

For the most part, our query is done! For some satisfaction and a sanity check, we can run this query on real data, in a web browser, on the Stack Exchange data platform. However, if this query is to be taken seriously, we'll also need to test it more thoroughly.

4.4.4 Temporary test data via `WITH`

When we start validity testing, we won't put all our test data in one table. If we do, then having too much data in one table means the output will become increasingly complex and unmanageable. There is a neat trick to creating temporary test tables, using the `WITH` clause:

```
WITH Posts (Id, CreationDate, Score, ViewCount, Body) AS (
  SELECT 1,'2020-01-01',1,200,'dummy text'
  UNION ALL
  SELECT 2,'2020-02-01',1,200,'dummy frequentist'
  UNION ALL
  SELECT 3,'2020-03-01',1,200,'dummy text'
),
PostCats AS (
    SELECT MONTH(CreationDate) AS CreationMonth,
           CASE WHEN Body LIKE '%frequentist%' THEN 'F'
                WHEN Body LIKE '%bayesian%' THEN 'B'
                END AS Category,
    CAST(Score AS DECIMAL) AS Score,
    CAST(ViewCount AS DECIMAL) AS ViewCount
    FROM Posts
    WHERE YEAR(CreationDate) <= 2020
    )
SELECT CreationMonth, Category,
       AVG(Score) AS AvgScore,
       AVG(ViewCount) AS AvgViews,
       COUNT(*) AS NumPosts
FROM PostCats
WHERE Category IS NOT NULL
GROUP BY CreationMonth, Category;
```

I know, it doesn't look very neat. The syntax could be a lot nicer. Regardless, we carry on. There are a few things to point out about the above code. The columns Id, CreationDate, Score, ViewCount and Body must all be named next to the Posts alias at the start of the `WITH` clause. Then, each row of test data is preceded by `SELECT`, and followed by `UNION ALL`. After the test data, a second alias (PostCats) is defined – we are using `WITH` to define two aliases: one called Posts (holding the test data) and one for PostCats. The rest of the query is the same as the one we developed in Section 4.4.3. It is large and messy, so from now on, to avoid taking up too much space in the notes, I will write the test data only, so the above will look like:

```
WITH Posts (Id, CreationDate, Score, ViewCount, Body) AS (
  SELECT 1,'2020-01-01',1,200,'dummy text'
  UNION ALL
  SELECT 2,'2020-02-01',1,200,'dummy frequentist'
  UNION ALL
  SELECT 3,'2020-03-01',1,200,'dummy text'
)
test_query
```

4.4.5 Validity testing

Fastball testing enabled a workflow that quickly brought our query up to scratch. Now, validity testing will help us make the query more robust. Compared to fastball testing, we're going to spend a lot more time on writing test data and making sure we know what results we expect. There really is no substitute for experience when it comes to validity testing, since you need time to develop a good awareness of what constitutes 'tricky' or 'strange' inputs to a query.

Look at each clause in the query and try to think of data that might cause it to go 'wrong'. Pay particular attention to values that are being used in search conditions or functions (such as aggregation functions). For example, we expect our query to behave differently for years above and below 2020. So, viewing 2020 as a 'boundary', we should write test data on either side of the boundary (2019, 2020, and 2021). For Score, we should consider values that don't average to whole numbers; we should also consider negative scores. For the Body, consider different placements of the target word in the string (at the start, the end, and the middle), and consider uppercase and lowercase letters in the target word. For all columns where `NULL` values are allowed, test data should include at least one `NULL`.

For each way you think the query might go wrong, write one separate set of test data, using the `WITH` syntax in Section 4.4.4. I will give two examples below, but they won't be exhaustive! Trying to thoroughly test the query, it's easy to become overwhelmed with possibilities. Don't let this cause you to throw your hands up and avoid validity testing entirely. Your testing might not be perfect, but it will be much better than nothing.

Example 4.4.1. For this example, we'll test the behaviour about the 'boundary' year, 2020, for the CreationDate column. Notice that, for all columns that we aren't testing, we use the same (simple) values in every row. This way we avoid complicating the output. We also throw in a `NULL` CreationDate, for good measure.

```
WITH Posts (Id, CreationDate, Score, ViewCount, Body) AS (
  SELECT 1,'2019-01-01',1,200,'dummy frequentist'
  UNION ALL
  SELECT 2,'2020-01-01',1,200,'dummy frequentist'
  UNION ALL
  SELECT 3,'2021-01-01',1,200,'dummy frequentist'
  UNION ALL
  SELECT 4,NULL,1,200,'dummy frequentist'
)
test_query
```

Given the above test data, we can easily determine the expected output. We should see the third row get filtered out (having a year above 2020), and the fourth row should be filtered out (having NULL year). The other two rows should appear in the result (both in the January/frequentist group). Comparing these expectations to the output tells us the query is behaving well:

RESULT				
CreationMonth	Category	AvgScore	AvgViews	NumPosts
1	F	1.0	200.0	2

Figure 4.18: The output of our validity test for the boundary year 2020

Example 4.4.2. For this example, we'll test the Body column to make sure the LIKE operator is picking up on lowercase and uppercase letters, as well as not being affected by the position of the target word. We will also include a row with NULL Body.

```
WITH Posts (Id, CreationDate, Score, ViewCount, Body) AS (
  SELECT 1,'2020-01-01',1,200,'dummy FREQUENTIST'
  UNION ALL
  SELECT 2,'2020-01-01',1,200,'dummy FREQUENTIST dummy'
  UNION ALL
  SELECT 3,'2020-01-01',1,200,'FREQUENTIST dummy'
  UNION ALL
  SELECT 4,'2020-01-01',1,200,NULL
)
test_query
```

Given the above test data, our expected output has 3 posts in the group January/frequentist. The output we receive tells us the query is behaving well:

RESULT				
CreationMonth	Category	AvgScore	AvgViews	NumPosts
1	F	1.0	200.0	3

Figure 4.19: The output of our validity test for the boundary year 2020

4.5 Simplifying complicated queries with aliases

This brief section aims to quickly make you aware of a technique that I most often use to simplify complicated-looking SQL queries. Below is an example of a query that looks complicated due to the presence of long column names. This query was made available by Stats NZ as an example to be executed on the New Zealand Integrated Data Infrastructure research database.

```
SELECT
year(IDI_Clean_20181020.moh_clean.PRIMHD.moh_mhd_activity_start_date)
    AS StartYear,
IDI_Clean_20181020.moh_clean.PRIMHD.snz_moh_uid
FROM
IDI_Clean_20181020.moh_clean.PRIMHD
WHERE
IDI_Clean_20181020.moh_clean.PRIMHD.moh_mhd_activity_type_code !=
    'T35'
GROUP BY
year(IDI_Clean_20181020.moh_clean.PRIMHD.moh_mhd_activity_start_date),
IDI_Clean_20181020.moh_clean.PRIMHD.snz_moh_uid
ORDER BY
year(IDI_Clean_20181020.moh_clean.PRIMHD.moh_mhd_activity_start_date);
```

Often, a query can be simplified just by introducing aliases. The `FROM` clause in the above query contains one table name:

```
FROM IDI_Clean_20181020.moh_clean.PRIMHD
```

The above table name was not given an alias, so the full table name is being reused as a prefix to every column name in the query. Introducing a short alias can make the query easier on the eyes. In many cases, it's sufficient to stop there. However, in more complicated queries, the lengths of the column names can still be a burden. So, it can help to also give short aliases to the column names. To be able to use aliases for the column names, we need to assign them via a `WITH` clause (see Section 4.4.3). Here is the query after being simplified via short aliases for the table and column names:

```
WITH Shortened AS (
  SELECT M.moh_mhd_activity_start_date AS astart,
         M.snz_moh_uid AS muid,
         M.moh_mhd_activity_type_code AS activity
  FROM IDI_Clean_20181020.moh_clean.PRIMHD AS M
)
SELECT year(astart) AS StartYear, M.muid
FROM Shortened
WHERE M.activity != 'T35'
GROUP BY year(astart), M.muid
ORDER BY year(astart);
```

The above query is now much easier to read than the original. Furthermore, all of the alias assignments happen in the `WITH` clause, which now serves as a reminder of the original table and column names.

4.6 Grokking SQL

This section is an unnecessarily verbose recommendation for you to head over and check out a blog post called 'Against SQL'. As a beginner, it's very likely that most of the problems (and the way they are explained in that blog post) will go over your head. If you don't want to hear me wax lyrical about the word 'grok', then save yourself some time and skip this section.

The word 'grok' is a bit of popular computer programmer slang, originating in Robert A. Heinlein's 1961 sci-fi novel *Stranger in a Strange Land*. In a very basic sense, to grok something means to have a powerful familiarity with it, but the meaning is much deeper than this. The programmers' Jargon File mentions:

> "*When you claim to 'grok' some knowledge or technique, you are asserting that you have not merely learned it in a detached instrumental way but that it has become part of you, part of your identity.*"

The Wikipedia page for grok quotes critic Istvan Csicsery-Ronay Jr. saying that the major theme of the book by Heinlein "can be seen as an extended definition of the term." I haven't read Heinlein's novel, so I won't pretend to truly grok the word grok. Though I do like this next passage from the book, also quoted on Wikipedia:

> "*…It means 'fear', it means 'love', it means 'hate' – proper hate, for by the Martian 'map' you cannot hate anything unless you grok it, understand it so thoroughly that you merge with it and it merges with you – then you can hate it. By hating yourself. But this implies that you love it, too, and cherish it and would not have it otherwise.*"

Now, people hate SQL for many reasons, but I believe that to truly hate SQL requires hard work, and deep familiarity. Conversely, to grow familiar with SQL I think we should spend time understanding how to hate it. With that in mind, I think you should check out Jamie Brandon's blog post, *Against SQL*. As much of it is pretty technical, I suggest coming back to it from time as you learn SQL, so that your mild discomfort might mature into true Martian hate.

4.7 Coding exercises

Exercise 4.7.1

In this exercise, we will investigate the perils of joining tables that are not related by primary/foreign key pairs.

1. Join the Notes.Houses and Notes.Suburbs tables, via post_code. Why can't you tell what the vaccination rate is for the house with HouseID H0001? Write a query (by modifying the join you just wrote) that counts, for each houseID, the number of possible suburbs it is in.
2. Modify your solution to question 1, using `HAVING COUNT(*)> 1` (and some other changes), so that it returns (only) the house ID, street address and post code, for the two houses that have more than one potential suburb.
3. Search online (e.g., the documentation) for how to use the aggregation function `STRING_AGG` (in T-SQL) or `GROUP_CONCAT` (in MySQL). Modify your solution to question 2, so that it includes a column that lists all potential suburb names for each of the two houses.
4. Duplicate post codes are not the only problem here. Write a query that joins Houses to Suburbs, but does not discard any records from Houses. Retrieve the house_ID, house_address, post_code and suburb_name. What do you notice about houses H0003, H0004 and H0008? What about houses H0011 and H0012? Now, modify the keep only the houses where suburb_name is `NULL`.
5. Search online for how to use the `UNION` clause. Then, use `UNION` to combine the results from your solutions to questions 3 and 4, into a single table. The result is a table of all houses whose suburb names are not (uniquely) identifiable.
6. Use either the `WITH` clause (see Section 4.4.3), or a subquery in the `FROM` clause, to count the number of houses in your result from question 5.

Solutions to Exercise 4.7.1

1.
```
SELECT H.house_id, COUNT(*) AS num_suburbs
FROM Notes.Suburbs S
     JOIN Notes.Houses H ON S.post_code = H.post_code
GROUP BY H.house_ID;
```

2.
```
SELECT H.house_id, H.house_address, H.post_code
FROM Notes.Suburbs S
    JOIN Notes.Houses H ON S.post_code = H.post_code
GROUP BY H.house_ID, H.house_address, H.post_code
HAVING COUNT(*) > 1;
```

3. In MySQL, you should replace `STRING_AGG(suburb_name, ', ')` with `GROUP_CONCAT(suburb_name SEPARATOR ', ')`.

```
SELECT H.house_id,
       H.house_address,
       H.post_code,
       STRING_AGG(suburb_name, ', ') AS potential_suburbs
FROM Notes.Suburbs S
JOIN Notes.Houses H ON S.post_code = H.post_code
GROUP BY H.house_ID, H.house_address, H.post_code
HAVING COUNT(*) > 1;
```

4.
```
SELECT H.house_id, H.house_address, H.post_code, S.suburb_name
FROM Notes.Houses H
    LEFT JOIN Notes.Suburbs S ON S.post_code = H.post_code
WHERE S.suburb_name IS NULL;
```

5. Simply place `UNION` between the two queries (but don't forget to remove the semi-colon from the first query).

```
SELECT H.house_id,
       H.house_address,
       H.post_code,
       string_agg(suburb_name, ', ') AS potential_suburbs
FROM Notes.Suburbs S
JOIN Notes.Houses H ON S.post_code = H.post_code
GROUP BY H.house_ID, H.house_address, H.post_code
HAVING COUNT(*) > 1
UNION
SELECT H.house_id, H.house_address, H.post_code, S.suburb_name
FROM Notes.Houses H
    LEFT JOIN Notes.Suburbs S ON S.post_code = H.post_code
WHERE S.suburb_name IS NULL;
```

6. Simply place the entire query from solution 5 into a `WITH` clause (see Section 4.4.3), and then count the number of rows with `COUNT(*)`:

```
WITH T AS (
  SELECT H.house_id,
         H.house_address,
         H.post_code,
         string_agg(suburb_name, ', ') AS potential_suburbs
  FROM Notes.Suburbs S
  JOIN Notes.Houses H ON S.post_code = H.post_code
  GROUP BY H.house_ID, H.house_address, H.post_code
  HAVING COUNT(*) > 1
  UNION
  SELECT H.house_id, H.house_address, H.post_code,
      S.suburb_name
  FROM Notes.Houses H
      LEFT JOIN Notes.Suburbs S ON S.post_code = H.post_code
  WHERE S.suburb_name IS NULL;
)
SELECT COUNT(*) AS NumHouses
FROM T;
```

Exercise 4.7.2

In this exercise, we will create a database. I will call it MyDatabase.

1. Create an empty database named MyDatabase (using `CREATE DATABASE`), and activate it (via `USE`). If you are using T-SQL, don't forget to put the `GO` keyword between statements.
2. Create the following two tables in MyDatabase (using `CREATE TABLE`), and then add the data to them (using `INSERT INTO`). Do not create primary or foreign keys, yet. Do not specify `not null`.

Animals		
AnimalID	AnimalName	StatusID
1	Black Rhino	1
2	Saola	1
3	Asian Elephant	2
4	Green Turtle	2
5	Dugong	3
6	Giant Panda	3

Endangered	
StatusID	Specification
1	Critically endangered
2	Endangered
3	Vulnerable

3. Explain why the following produces an error. Note: annoyingly (for my teaching), this does not produce an error in MySQL, but don't execute it yet (just explain why it would produce an error in T-SQL).

```
ALTER TABLE Animals ADD PRIMARY KEY (AnimalID);
```

4. Execute the following two statements and explain what they achieve.

```
-- In MySQL, replace 'ALTER COLUMN' with 'MODIFY COLUMN'
ALTER TABLE Animals
ALTER COLUMN AnimalID INT NOT NULL;

GO -- In MySQL, delete this 'GO' keyword

ALTER TABLE Animals
ADD PRIMARY KEY (AnimalID);
```

5. Explain why the following produces an error.

```
ALTER TABLE Animals
ADD FOREIGN KEY (StatusID) REFERENCES Endangered (StatusID);
```

6. Add a primary key constraint to the Endangered table, and then add the appropriate foreign key constraint to Animals.

Solutions to Exercise 4.7.2

1.

```
-- start by dropping, for good measure
DROP DATABASE IF EXISTS MyDatabase;
GO -- In MySQL, delete the 'GO' keywords
CREATE DATABASE MyDatabase;
GO
USE MyDatabase;
GO
```

2.

```
CREATE TABLE Animals (
  AnimalID int,
  AnimalName varchar(20),
  StatusID int
);
GO -- In MySQL, delete the 'GO' keywords
CREATE TABLE Endangered (
  StatusID int,
  Specification varchar(30)
);
GO
INSERT INTO Endangered VALUES
(1, 'Critically endangered'),
(2, 'Endangered'),
(3, 'Vulnerable');
GO
INSERT INTO Animals VALUES
(1, 'Black Rhino' , 1),
(2, 'Saola'       , 1),
(3, 'Asian Elephant', 2),
(4, 'Green Turtle' , 2),
(5, 'Dugong'      , 3),
(6, 'Giant Panda' , 3);
```

3. AnimalID allows `NULL` s, so it cannot be a primary key.
4. Note: delete the `GO` keyword if you are using MySQL.

```
-- In MySQL, replace 'ALTER COLUMN' with 'MODIFY COLUMN'
ALTER TABLE Endangered
ALTER COLUMN StatusID int not null;
GO -- In MySQL, delete the 'GO' keywords
ALTER TABLE Endangered
ADD PRIMARY KEY (StatusID);
GO
ALTER TABLE Animals
ADD FOREIGN KEY (StatusID) REFERENCES Endangered (StatusID);
```

Exercise 4.7.3

In this exercise, we will continue working with the database created in Exercise 4.7.2.

1. Write a query that joins the Animals and Endangered tables, and retrieves only the columns AnimalName and Specification. Save the result using `CREATE VIEW`.
2. Explain why the following produces an error.

```
INSERT INTO Animals VALUES
(1, 'Hawksbill Turtle', 1);
```

3. This one also produces an error. Explain why.

```
INSERT INTO Animals VALUES
(7, 'Hawksbill Turtle', 4);
```

4. Insert the ‘Hawksbill Turtle’ into Animals, with StatusID 2.
5. Use `UPDATE` to set the StatusID of the ‘Hawksbill Turtle’ to 1.
6. Write a query (using `SELECT`) to display the contents of the view you created in question 1. The Hawksbill Turtle was created after you created the view, so why is it showing up in the view now?

Solutions to Exercise 4.7.3

1.

```
CREATE VIEW MyView AS
SELECT A.AnimalName, E.Specification
FROM Animals A JOIN Endangered E
ON A.StatusID = E.StatusID;
```

2. The error is produced because the primary key value (AnimalID = 1) already exists in the Animals table.
3. The error is produced because the foreign key value (StatusID = 4) is not present in the Endangered table (foreign key constraint).
4.

```
INSERT INTO Animals VALUES
(7, 'Hawksbill Turtle', 2);
```

5. A view is a stored query that behaves like a table. So, the results of the view are always up to date.

```
-- display the contents of the view
SELECT *
FROM MyView;
```

Exercise 4.7.4

We’re now going to extend and improve on the Orders table, which we first saw way back in Figure 2.13. You can look at the Orders table in the Notes schema of the Sandpit database. It’s a strange table, really. For one, there are no table numbers. Also, the items and their prices are stored alongside each order, which will lead to data redundancy when the same item is ordered twice. We're going to create a separate table for items, and we'll add a table number column into Orders.

1. To start with, we'll preserve the original Orders table by copying it into MyDatabase, which we created in Exercise 4.7.2. Execute the following code and explain what it does.

```
-- using T-SQL syntax
USE MyDatabase;
GO
CREATE SCHEMA Restaurant;
GO
SELECT *
INTO Restaurant.Orders
FROM Sandpit.Notes.Orders;

-- using MySQL syntax
USE MyDatabase;
CREATE TABLE Restaurant_Orders
SELECT *
FROM Sandpit.Notes_Orders;
```

2. In the new Restaurant schema, use a single statement to create a table named Menu, and copy into it the Item and Price columns from Orders.
3. Now drop the Price column from the Restaurant.Orders table.
4. Add a column called TableNo to Restaurant.Orders. Then, set the table number to 1 for OrderIDs 1 and 2 (only).
5. Now add the appropriate foreign key constraint to represent the relationship between Orders and Menu.

Solutions to Exercise 4.7.4

1. The T-SQL code creates a Restaurant schema in MyDatabase, and then copies the Orders table (from Sandpit) into the Restaurant schema. The MySQL code does almost the same thing, but without creating a schema (since MySQL doesn't use schemas).
2.

```
-- T-SQL syntax
SELECT Item, Price
INTO Restaurant.Menu
FROM Restaurant.Orders;

-- using MySQL syntax
CREATE TABLE Restaurant_Menu
SELECT Item, Price
FROM Restaurant_Orders;
```

3.

```
-- T-SQL syntax
ALTER TABLE Restaurant.Orders
DROP COLUMN Price;

-- MySQL syntax
ALTER TABLE Restaurant_Orders
DROP COLUMN Price;
```

4. This syntax works in both MySQL and T-SQL. In MySQL, just replace Restaurant.Orders with Restaurant_Orders.

```
ALTER TABLE Restaurant.Orders
ADD TableNo INT;

UPDATE Restaurant.Orders
SET TableNo = 1
WHERE OrderID IN (1,2);
```

5. We need to add the `not null` constraint, add the primary key constraint, and then finally add the foreign key constraint.

```
-- In MySQL replace 'ALTER COLUMN' with 'MODIFY COLUMN'
ALTER TABLE Restaurant.Menu
ALTER COLUMN Item varchar(30) not null;

ALTER TABLE Restaurant.Menu
ADD PRIMARY KEY (Item);

ALTER TABLE Restaurant.Orders
ADD FOREIGN KEY (Item) REFERENCES Restaurant.Menu (Item);
```

Exercise 4.7.5

In this exercise, we will work with a pair of practice databases modelled after a small section of the New Zealand Integrated Data Infrastructure (IDI). Note, many of the tables in this fake IDI database are empty. They are just there to simulate table and schema naming conventions. You will need to use the ACC_Clean data dictionary. The IDI_Clean database contains individual level data, and the IDI_Metadata database contains non-individual level data.

1. The real IDI contains very large tables. To have a peek at these tables, it would be very inefficient to simply execute `SELECT * FROM MyTable`. Write a query that looks at the first 3 rows of the Serious_Injury table in ACC_Clean.
2. Explain what the query below does (and, if you are using T-SQL, then try running it). You may have to search online for details about the T-SQL 'Information Schema'.

```
SELECT *
FROM IDI_Clean.Information_Schema.Columns
WHERE Table_Catalog = 'IDI_Clean';
```

3. Explain what the following query does.

```
WITH E AS (
SELECT snz_uid,
      CASE WHEN acc_cla_ethnic_grp1_snz_uid = 1
           THEN 1 ELSE NULL END AS ethnic_grp1,
      CASE WHEN acc_cla_ethnic_grp2_snz_uid = 1
           THEN 2 ELSE NULL END AS ethnic_grp2,
      CASE WHEN acc_cla_ethnic_grp3_snz_uid = 1
           THEN 3 ELSE NULL END AS ethnic_grp3
FROM IDI_Clean.ACC_Clean.Serious_Injury S
), M AS (
SELECT *
FROM
IDI_Metadata.clean_read_CLASSIFICATIONS.acc_ethnicity_code
)
SELECT E.snz_uid,
      M1.description AS ethnic_grp1,
      M2.description AS ethnic_grp2,
      M3.description AS ethnic_grp3
FROM E
LEFT JOIN M AS M1 ON E.ethnic_grp1 = M1.ethnic_grp
LEFT JOIN M AS M2 ON E.ethnic_grp2 = M2.ethnic_grp
LEFT JOIN M AS M3 ON E.ethnic_grp3 = M3.ethnic_grp;
```

4. How many injuries is each employer involved in? Write a query that lists the number of injuries caused by each employer. Use the key named snz_employer_ird_uid to identify employers, since you do not have access to the employer names. Also display the total amount of claim costs related to each employer.
5. Of those people who have engaged with the ACC, how many have en-

gaged with the Ministry of Health (MOH)? Correspondence between department identifiers is described in the concordance table of the security schema.

6. This question is challenging and the resulting query will be long. The query in question 3 performs many joins, making a 'wide' table. We would like to do it using a 'long' table instead. First, use a query on Serious_Injury, with multiple `UNION` clauses, to return a table where every row represents a unique combination of snz_uid and ethnicity code. Make sure the result includes all 6 ethnicity codes, not just the 3 from question 3. Then (joining the result to the acc_ethnicity_code table, and grouping by snz_uid) use `STRING_AGG` (in T-SQL) or `GROUP_CONCAT` (in MySQL) to combine the multiple text values of ethnicity into one entry for each individual.

Solutions to Exercise 4.7.5

1.

```
-- T-SQL syntax
SELECT TOP(3) *
FROM IDI_Clean.ACC_Clean.Serious_Injury;

-- MySQL Syntax
SELECT *
FROM IDI_Clean.ACC_Clean_Serious_Injury
LIMIT 3;
```

2. It displays all the schemas, tables and columns in IDI_Clean.
3. In the Serious_Injury table, ethnicity is encoded using 6 separate columns. Each column contains a 0 or 1 (binary), indicating if the person has claimed a particular ethnicity category. A person can claim multiple ethnicity categories. The English text representation of ethnicity can be found in the acc_ethnicity_code table in the database IDI_Metadata. The query only considers the first 3 of the 6 ethnicity columns. First it recodes the binary ethnicity variables to a number that matches the keys in acc_ethnicity_code. It then performs 3 separate left joins in order to replace the keys with the text representation of ethnicity.
4.

```
SELECT snz_employer_ird_uid,
      COUNT(*) AS NumInjuries,
      SUM(acc_cla_claim_costs_to_date_ex_gst_amt) AS TotalCost
FROM ACC_Clean.Serious_Injury
GROUP BY snz_employer_ird_uid;
```

5.

```
SELECT COUNT(*) AS NumPeople
FROM Security.concordance
WHERE snz_acc_uid IS NOT NULL
AND snz_moh_uid IS NOT NULL;
```

6. For the query below, in MySQL, use the `GROUP_CONCAT` function instead of `STRING_AGG`.

```
WITH E AS (
  SELECT snz_uid, 1 AS ethnicity_code
  FROM IDI_Clean.ACC_Clean.Serious_Injury S
  WHERE acc_cla_ethnic_grp1_snz_uid = 1
  UNION
  SELECT snz_uid, 2 AS ethnicity_code
  FROM IDI_Clean.ACC_Clean.Serious_Injury S
  WHERE acc_cla_ethnic_grp2_snz_uid = 1
  UNION
  SELECT snz_uid, 3 AS ethnicity_code
  FROM IDI_Clean.ACC_Clean.Serious_Injury S
  WHERE acc_cla_ethnic_grp3_snz_uid = 1
  UNION
  SELECT snz_uid, 4 AS ethnicity_code
  FROM IDI_Clean.ACC_Clean.Serious_Injury S
  WHERE acc_cla_ethnic_grp4_snz_uid = 1
  UNION
  SELECT snz_uid, 5 AS ethnicity_code
  FROM IDI_Clean.ACC_Clean.Serious_Injury S
  WHERE acc_cla_ethnic_grp5_snz_uid = 1
  UNION
  SELECT snz_uid, 6 AS ethnicity_code
  FROM IDI_Clean.ACC_Clean.Serious_Injury S
  WHERE acc_cla_ethnic_grp6_snz_uid = 1
), M AS (
SELECT *
FROM
IDI_Metadata.clean_read_CLASSIFICATIONS.acc_ethnicity_code
)
SELECT snz_uid, STRING_AGG(description, ', ') AS ethnicities
FROM E JOIN M
ON E.ethnicity_code = M.ethnic_grp
GROUP BY snz_uid;
```

Chapter 5

Unleashing SQL, with R

This chapter assumes very basic knowledge of R. Below, I've linked a guide (by ModernDive) for installing R and RStudio, as well as an R script that launches a tutorial, in case you're learning R for the first time, or want to brush up. This chapter does not aim to teach you R, but aims to show you some ways to apply your newfound SQL skills in R.

- Click here for the guide to installing R and RStudio
- Click here for the basic R usage tutorial

In particular, as prerequisites for this chapter, you should already know:

- What an R script is, and how to execute R code in RStudio.
- What the assignment operator does (`<-`).
- What an R function is, and what a function argument is.
- What a `data.frame` is, and how to create one from scratch.
- What an R package is, and how to install one.

There are two main motivations for covering R in a SQL programming course. The first is that R is a language for data analysis, visualisation and general programming, while SQL is not. Heck, SQL has essentially one job: databases. There *are* advantages to limitations, though. Thanks to limited scope and the strict relational model, SQL language designers, educators and programmers reap the benefits of creative constraints. R, on the other hand, is a much more flexible programming language.

The second motivation is that, in learning SQL, you're learning a way of thinking about data that has been incorporated into R. It's perhaps no surprise that some principles of the relational model have found their way into R. The R community includes many scientists, statisticians and analysts, who share a passion for data. Naturally, some R developers have been inspired by the advantages of SQL.

We'll see some of the benefits to the SQL way of thinking, as understood by R programmers, in Section 5.1. Then, in Section 5.3, we'll look at some great tools that have been developed to combine the flexibility of R with the constraints of SQL, forming what is often referred to as an 'opinionated' approach to working with data in R.

5.1 Tidy data

Parts of the SQL way of thinking (that is, of the relational model) have been adopted as the principles of 'tidy data', through the very popular work of Hadley Wickham, who is an adjunct professor at The University of Auckland, and Chief Scientist at RStudio.

> "*The principles of tidy data are closely tied to those of relational databases and Codd's relational algebra, but are framed in a language familiar to statisticians*" – [Wickham, 2014].

A tidy dataset satisfies three conditions (which you may recognise!):

(i) Every variable is a column.

(ii) Every observation is a row.

(iii) Every cell holds a single (atomic) value.

Datasets that satisfy conditions (i) and (ii) are often referred to as being in **long format**. Condition (iii) is sensible, for reasons we discussed in the last two paragraphs of Section 1.3. This simple tidy data standard allows datasets to be read and understood universally, and makes it easier for tools (like R packages) to work together on tidy datasets.

> "*Happy families are all alike; every unhappy family is unhappy in its own way*" – Leo Tolstoy.

> "*Tidy datasets are all alike, but every messy dataset is messy in its own way.*" – Hadley Wickham.

To understand tidy data better, we best look at messy data. For each messy dataset, we'll determine which of the above three tidy data conditions is violated. Then, we'll see how to tidy the messy data. To make tidying more convenient, Hadley Wickham developed the tidyr package.

```
install.packages("tidyr")
library(tidyr)
```

Example 5.1.1. The tidyr package comes with its own built-in datasets to practice on. Our first dataset is called `relig_income`, and it can be viewed by executing:

```
relig_income
```

Below are the first 5 columns and 3 rows of `relig_income`.

Table 5.1: The first 5 columns of `relig_income` from tidyr.

religion	<$10k	$10-20k	$20-30k	$30-40k
Agnostic	27	34	60	81
Atheist	12	27	37	52
Buddhist	27	21	30	34

In Table 5.1 we see that a count is being observed for each religion, at each income level (where the income levels are <$10k, $10-20k, etc). Are any of the three tidy data conditions not being met? The income variable is stored in the column names. This means condition (i) is violated: not every variable is a column. Are any of the other conditions violated?

Yes, condition (ii) is violated: not every row is an observation. Can you see why? For each religion, multiple observed counts are stored in each row. In other words, there is a different observation at each income level. A dataset like Table 5.1 is said to be in a **wide format**, since the counts are 'grouped' together in each row. We can undo this grouping, using the tidyr function `pivot_longer`, with the arguments below. This function is designed to take a dataset from a wide format to a long format.

```
pivot_longer(
  data = relig_income,
  cols = !religion,
  names_to = "income",
  values_to = "count"
)
```

Here are some notes to help you understand the arguments above:

- The argument `data = relig_income` asks `pivot_longer` to use the dataset called `relig_income`.
- The expression `!religion` translates to 'not religion'.
- The argument `cols = !religion` asks `pivot_longer` to turn all column names, except 'religion', into the entries of one new column.

- The argument `names_to = "income"` asks `pivot_longer` to name the new column ‘income’.
- To perform this pivot, the entries in all columns (except `religion`) will be placed into another new column. Argument `values_to = "count"` asks `pivot_longer` to name this new column ‘count’.

Below, we see the output is now tidy.

Table 5.2: The data from Table 5.1 are now tidy, after applying `pivot_longer`.

religion	income	count
Agnostic	<$10k	27
Agnostic	$10-20k	34
Agnostic	$20-30k	60
Agnostic	$30-40k	81
Atheist	<$10k	12
Atheist	$10-20k	27
Atheist	$20-30k	37
Atheist	$30-40k	52
Buddhist	<$10k	27
Buddhist	$10-20k	21
Buddhist	$20-30k	30
Buddhist	$30-40k	34

Example 5.1.2. For this example, we will look at a dataset where condition (iii) is violated: more than one value stored in a cell. We’ll make this dataset ourselves.

```
address_data <- data.frame(
  name = c("Raleigh Smith",
           "Hoa Pham",
           "Ram Singh"),
  address = c("109 Tenterfield Rd, North Sydney, NSW, 2060",
              "36 Feather St, Kings Beach, QLD, 4551",
              "90 Sunnyside Rd, Renmark, SA, 5341")
)
```

The data now look like this:

Table 5.3: Some data we created with names and addresses.

name	address
Raleigh Smith	109 Tenterfield Rd, North Sydney, NSW, 2060
Hoa Pham	36 Feather St, Kings Beach, QLD, 4551
Ram Singh	90 Sunnyside Rd, Renmark, SA, 5341

The name and address columns both contain multiple values. We can separate the address column using the tidyr function `separate`.

```
separate(
  data = address_data,
  col = address,
  into = c("street","city","state","postcode"),
  sep = ", ")
```

The following notes explain the code above.

- The argument `col = address` specifies that we want to separate the address column.
- The `into` argument specifies the names of new columns that we want to create.
- The argument `sep = ", "` tells `separate` to split the contents of the address column wherever it encounters a comma followed by an empty space.

The result is given below.

name	street	city	state	postcode
Raleigh Smith	109 Tenterfield Rd	North Sydney	NSW	2060
Hoa Pham	36 Feather St	Kings Beach	QLD	4551
Ram Singh	90 Sunnyside Rd	Renmark	SA	5341

The above is almost tidy, but it is probably preferable to separate the name column as well, into first and last names. This would make it easier to, for example, filter the data by last name.

Example 5.1.3. We'll now see a trickier example of a dataset in wide format. This dataset, called Anscombe's Quartet, is often used an example of why it's important to plot data before performing linear regression. The dataset is available by default in R. Note, you do not need to understand linear regression to follow this example.

```
anscombe
```

Table 5.4: The Anscombe's Quartet dataset.

$x1$	$x2$	$x3$	$x4$	$y1$	$y2$	$y3$	$y4$
10.00	10.00	10.00	8.00	8.04	9.14	7.46	6.58
8.00	8.00	8.00	8.00	6.95	8.14	6.77	5.76
13.00	13.00	13.00	8.00	7.58	8.74	12.74	7.71
9.00	9.00	9.00	8.00	8.81	8.77	7.11	8.84
11.00	11.00	11.00	8.00	8.33	9.26	7.81	8.47
14.00	14.00	14.00	8.00	9.96	8.10	8.84	7.04
6.00	6.00	6.00	8.00	7.24	6.13	6.08	5.25
4.00	4.00	4.00	19.00	4.26	3.10	5.39	12.50
12.00	12.00	12.00	8.00	10.84	9.13	8.15	5.56
7.00	7.00	7.00	8.00	4.82	7.26	6.42	7.91
5.00	5.00	5.00	8.00	5.68	4.74	5.73	6.89

Table 5.4 contains the x and y values for four different 'sets' of data. However, they are arranged with four columns of x values and four columns of y values. This means there are four different observations in each row. To tidy this data, we need a new column, which we will call 'set'. Again, we can achieve this with `pivot_longer`.

```
pivot_longer(
  data = anscombe,
  names_pattern = "(.)(.)",
  names_to = c(".value", "set"),
  cols = everything()
)
```

Below are some notes for understanding the arguments above. Feel free to skip these details if you are a beginner. It will not be required knowledge for these notes. The details are not simple, and many new R programmers will have difficulty with this. Focus instead on understanding why Table 5.4 is not tidy, while Table 5.4 below is tidy.

- The argument `names_pattern = "(.)(.)"` asks `pivot_longer` to split each column name from Table 5.4 into two parts. For example, $x1$ becomes x and 1.

- The argument `names_to = c(".value", "set")` does two things. First, `".value"` indicates that the first part of each column name (for example, the x in $x1$) will define a new column name. Then, `"set"` indicates that the second part of each column name (for example, the 1 in $x1$) will become an entry in a new column called 'set'.

- The argument `cols = everything()` then asks `pivot_longer` to apply the previous two arguments to every column in Table 5.4.

After applying `pivot_longer`, the data from Table 5.4 are tidy. Below are the first 8 rows of the result. These 8 rows correspond to the first row of Table 5.4.

set	x	y
1	10.00	8.04
2	10.00	9.14
3	10.00	7.46
4	8.00	6.58
1	8.00	6.95
2	8.00	8.14
3	8.00	6.77
4	8.00	5.76

For fun, I have included below a plot of the four sets of data in Anscombe's Quartet, with linear regression lines. Note that all four lines of best fit are the same! This is not directly relevant to our example, however.

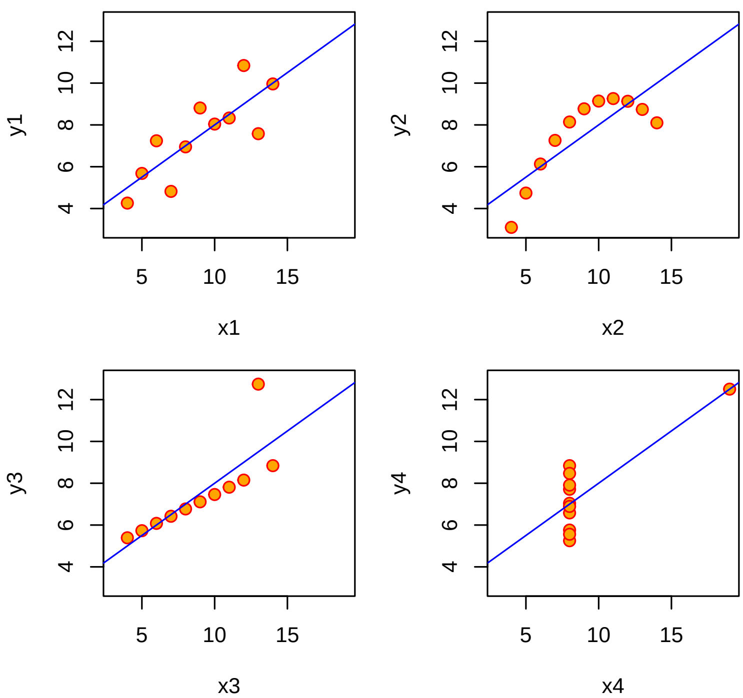

Figure 5.1: Plots of the four sets of data in Anscombe's Quartet dataset, with linear regression lines included.

5.2 Talking to SQL, with R

5.3 SQL and the tidyverse

```
SELECT Score, ViewCount, MONTH(CreationDate) AS CreationMonth,
      YEAR(CreationDate) AS CreationYear,
      CASE WHEN body LIKE '%happy%' THEN 'happy'
           WHEN body LIKE '%covid%' THEN 'covid' END AS Category
FROM Posts
WHERE YEAR(CreationDate) = 2020
  AND (Body LIKE '%happy%' OR Body LIKE '%covid%');
```

Bibliography

[Codd, 1970] Codd, E. F. (1970). A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387.

[Elmasri, 2008] Elmasri, R. (2008). *Fundamentals of database systems*. Pearson Education India.

[Wickham, 2014] Wickham, H. (2014). Tidy data. *Journal of statistical software*, 59(1):1–23.

Glossary

relational model The framework for how SQL "thinks of" data. It describes tables, and relationships between tables (e.g., with primary and foreign key pairs). 7, 11

miniworld The part of the real world (or any imaginary world) that a given database is designed to capture or represent. 9

metadata Literally means "data about data". Metadata is used internally by databases to describe their structure. 9

query A SQL query is a request for data or information from a database. 10, 28

table A SQL table represents an abstract entity (such as a friend or a pet), or may represent the relationship between two entities (such as the act of one friend scratching another friend's back). A table is composed of rows (representing instances of the entity) and columns (representing attributes of the entity). Each cell in a table contains one atomic unit of information. 11

entry One data value, in one particular row and column of a table. 12, 13

entity In the relational model, an entity is an object or thing in the mini world, that we choose to logically from other objects or things in the mini world. Entities are represented by tables, and the columns of the tables are the attributes of the entity. Friends and Pets are two examples of entities that we use in this text. 12, 39

tuple A tuple is a collection of entries. Usually we use the word tuple to refer to one row of a table. Each row is one instance of the entity represented by the table. 12

attribute One column of a table. The columns are essentially a set of labels that define, conceptually, the data to be contained in the table. 12

relation A table. The fundamental unit of organisation in a relational database. 12

domain The collection of possible values for each attribute in a table. The domain tells us what type of data (e.g., person names, phone numbers, country names etc) that we can store in each column of the table. 13

atomic A property of all entries in all tables in the relational model. It demands that every entry contains one, and only one, value, of the chosen data type. For example, the FirstName entry of a particular row in the Friends table must contain only one FirstName. 13, 177

one-to-many relationship When one record (i.e., row) in a table can be associated with multiple records in another table via a primary and foreign key pair. 14

referential integrity This a requirement of every primary and foreign key pair. It states that every foreign key entry must have at least one corresponding primary key entry in the table that it points to (i.e., in the table that it "references"). 15, 17

primary key A primary key is any column (or collection of columns) that has (or have, together) been chosen to uniquely identify the rows of the table it belongs to. The entries in a primary key must be unique. 16

foreign key A column whose entries correspond to entries of the primary key in some (usually different) table in the database. 16

many-to-many relationship When one record (i.e., row) in a table can be associated with multiple records in another table, and vice versa. 18

one-to-one relationship When one record (i.e., row) in a table can be associated with at most one record in another table, via a primary and foreign key pair. 21

data redundancy When the same piece of data is unnecessarily repeated in more than one place in a database. This can lead to inconsistencies in the data. 22

dialect This is a word I use to refer to the different languages belonging to different database management systems, that each implement large parts of the SQL standard. I refer to these as different dialects of SQL. 27

clause A SQL clause is the smallest logical component of a SQL statement that lets you filter or customise how you want your data to be queried. Examples are `SELECT`, `FROM`, `WHERE`, `GROUP BY`, `HAVING` and `ORDER BY`. 28

statement A SQL statement is a block SQL of syntax that is written to perform a single query or data manipulation task. An example is the statement `SELECT FirstName FROM Friends`, which retrieves all of the first names of my friends. 28

data type Data types are specified when SQL tables are created. They determine the kinds of data that can be stored in a given column. Examples include `INT` (positive or negative whole numbers), `FLOAT` (approximate decimal numbers), `VARCHAR` (variable lengths character strings), `DATE` (dates) and `TIME` (times). 33

scalar function A scalar function is a tool used to transform entries in a query. For example, a number might be rounded up or down; the month or year might be extracted from a date; or, the initials might be extracted from a person's name.. 34

search condition A logical statement that evaluates to either True or False, for any given row. 38, 48

filtering The act of choosing rows from a table, based on meeting a search condition, is often referred to as filtering. In SQL, this is achieved by the `WHERE` clause (or, when filtering based on aggregation functions, it is achieved by the `HAVING` clause). 38

logical operator A symbol that denotes a logical operation. A logical operation returns either `TRUE`, `FALSE` or `NULL` . 38, 46

alias An alias is a single letter, or a short word, that allows us to refer to a table without having to write its full name. 42

comparison operator A symbol used to compare two things and return either `TRUE`, `FALSE` or `NULL` (unknown). 45

tuple entries Tuple entries may be better referred to as 'arrays', but I use this terminology instead to keep things (hopefully) less confusing for beginners. Tuple entries are collections of multiple values that occupy a single entry in a table. These cannot be returned by `SELECT` in standard SQL. 77

atomicity See **atomic**. 77, 89

aggregation function An aggregation function returns one single result for each group formed from a `GROUP BY` clause. These can only be used wthin the `SELECT` or the `HAVING` clause. 82

nested query A query that would be a whole valid query if it appeared on its own, but it is currently nested within another query, so that the other query can easily use its results. 89

partition A partition is formed by the `PARTITION BY` command within the `OVER` clause. A partition is similar to a 'group' (of the kind formed by `GROUP BY`). However, unlike a 'group', a partition does not merge entries within the 'groups'. So, using partitions, we can write queries that still return one row *per row*, rather than being forced to return one row *per group* (as we are forced to do when using `GROUP BY`). 94, 95

window function A window function is any function that can work with the `OVER` clause. These include aggregation functions, ranking functions, and analytic functions. 96

analytic function An analytic function is a type of window function that returns a different value for each row in a partition. Contrast this with an aggregation function, which returns the same value for every row in a partition. 96

ranking function A ranking function is a type of analytic function that returns a *rank* for each row in a partition. 96

Syntax Conventions The Syntax Conventions are a set of symbols that are used in MySQL and T-SQL documentation to help communicate the kinds of syntax that form valid SQL statements. 120

Data Manipulation Language (DML) DML is the part of the SQL language that is dedicated to inserting, updating, deleting and querying data. Contrast this with Data Definition Language (DDL). 120

Data Definition Language (DDL) DDL is the part of the SQL language that is dedicated to defining objects, such as tables. DDL is used to create, alter and drop whole tables. 120

fastball testing Fastball testing is the process of writing quick and dirty test data to run through a query as early as possible. Fastball testing makes sure that queries execute without error and produce the most basic desired output. 144

validity testing Validity testing is the process of carefully choosing test cases that push the limits of a query. Validity testing makes sure queries are robust to unexpected or strange inputs. 144

long format A dataset is said to be in a long format if it satisfies: (i) every variable is a column, and (ii) every observation is a row. 167

wide format If multiple observations are present in each row, then a dataset is said to be in a wide format. 168