Course Notes on SQL

WITH A FOCUS ON T-SQL

Daniel Vidali Fryer

An Open Access Textbook
Updated Nov 2020

Contents

Using SQL is fun. It is true that when you're starting out, SQL is *more* fun. Every query is a mini puzzle to solve. Maybe SQL is less fun when you're experienced, but then, SQL is *powerful*. 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 (RDBMS)." - Wikipedia*

On second thought, 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*. 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), or more deeply, the relational model for database management. Some familiarity with it will hopefully help you understand problems in the SQLian way (pronounced "Sequelian", as in, a citizen in the nation of SQL).

Possibly you: "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 in ⟨statistical-programming-language⟩!"

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, Shazamo, 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 queries 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

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**.

Typically, a large collection of interdependent programs is 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.

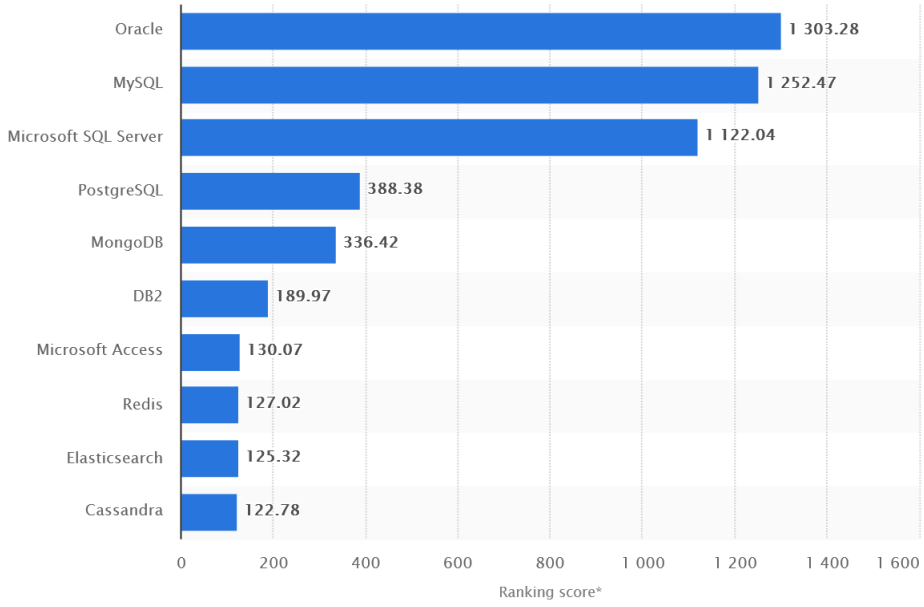


Figure 1.1: DBMS Rankings by popularity. Source: statista.com

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 of the data (sometimes referred to as its *miniworld*) with the 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 (that is, separate from the *stored database*), allows the DBMS to implement a layer of abstraction between its low-level interactions with the machines, and its high-level interactions with its human overlords.

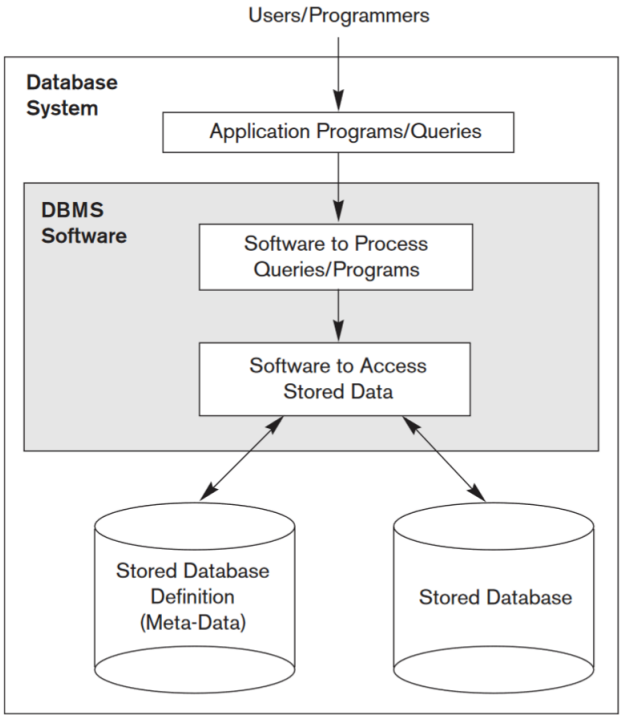


Figure 1.2: A simplified database system. Source: [1].

A database aims to contain data that are an accurate, up-to-date reflection of its miniworld. So, if some important aspect of the miniworld changes, then the contents or structure of the database may change to mirror it.

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. As SQL programmers, we think of defining data using the commands *insert*, *update* and *delete*; and we think of manipulating data using the *select* command, and others. We'll learn about these 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 [2]. 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. [2].

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:

```
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 get a look at the bigger picture.

1.3 The bigger picture in tables

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 piece of data at a particular place in a table (at a certain row and column) is often referred to as an **entry**. To keep things organised, each table is thought of as a collection of rows, where each row is one realisation of the abstract entity (such as a person or friend) that the table represents. A row is also known as **tuple** or 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. When I create the Friends table, I need to decide what columns the table should have. Columns are also known as **attributes**.

For example, a reasonable set of columns to include might be 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. We have thus defined the terms *relation* (table), *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

Friends(FriendID, FirstName, LastName, FavColour).

The above fully describes the structure of the table, provided that we know what the domain of each attribute is. This representation will come in handy later, when we want to describe the important operations that a SQL programmer can do on tables. 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 tuple 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 tuple as one **entry** in a table. So, in this tuple, 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 start including whole tables as entries inside tuples, inside tables, in some kind of horrible Kafkaesque hierarchical tower of tables within tables. For good reasons, this kind of thing is banned from the relational model. We refer to this model as *flat*, and say that each entry in a table must be **atomic**, meaning that each entry should be something that is not intended to be subdivisible. For example, in my Friends table, FavColour and LastName are atomic. We would avoid merging them together as one attribute, FavColourLastName, since the result 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 adds clutter.
- 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.
- Also, if we want to keep extra details on each pet, such as their birthdates (PetDOB), we'll just end up generating more NULL values in new columns.

The solution is simple:

I create another table.

The new table will contain data on all the pets, and an attribute (FriendID) will describe which friend each pet belongs to.

Pets CORRECT			
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.

For this to work, the entries stored under FriendID *must* correspond to existing entries stored under FriendID in the Friends table. This way, each pet will have one existing friend that it belongs to. So, if we want the details on the owner of Chikin, then (noting that Chikin has FriendID equal to 2) we can search the Friends table for the friend with FriendID 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.

Going the other way, if we want the details of all pets belonging to, say, my friend Z, then (noting that the FriendID of Z in the Friends table is 3) we can search for all rows of the Pets table with FriendID 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. By "more flexible," I mean that each pet now has it's own unique ID number (PetID). So, if we wanted, we could add a new one-to-many relationship between pets and something else (like pet toys), much in the same way that we just 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 which pet each toy belongs to). Try it yourself now, as an exercise.

In this section, we just looked at *one-to-many relationships*. In the next section, we will look in more detail at the kinds of attributes we just created, called primary and foreign key pairs. Then, we'll look at the remaining two types of relationships that arise between tables in SQL: those that are many-to-many, and those that are one-to-one.

1.4.2 Primary and foreign keys

In the previous section, we avoided turning the Friends table into an unwieldy mess (as 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 (or have, together) 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 – but, 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.

A foreign key is any column (or collection of columns) whose each entry is equal to one, and only one, primary key entry in some other table that the foreign key is designated to "point at". Thus, given a foreign key entry, we can always identify the unique primary key entry that it is equal to (i.e., pointing at). For this reason, we say that the foreign key is "pointing at", or that it *references*, the primary key.

1.4.3 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. Unfortunately, again, some database administrators didn't get the memo here – 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, an 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 Houses table has a column PostCode, but no column SuburbName, and suppose the Suburbs table has two columns, SuburbName and PostCode.

- Does each PostCode entry in Houses point at a unique PostCode entry in Suburbs? or, is the uniqueness violated, with some suburbs sharing the same post code?
- Can you definitely determine which SuburbName from Suburbs corresponds to each PostCode entry in Houses? or, is there a risk that there is a PostCode entry in Houses that doesn't exist yet as an entry in Suburbs?

These questions would be easily answered if the PostCode columns were formally defined as a primary and foreign key pair by the database administrator. Without that formality, it is up to the SQL programmer to decide if any relationships can be trusted.

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.

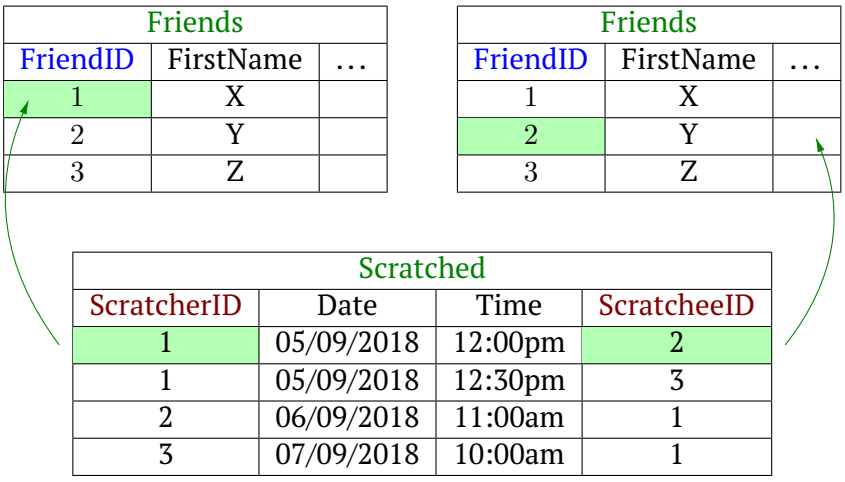


Figure 1.10: Modelling a many-to-many relationship amongst friends.

In this example, both foreign keys reference the primary key from the Friends table. So, in the above diagram 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 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 above.

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 some strange 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.

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.

Friends WRONG					
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 (and, of course, each passport belongs to only one friend), 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 have decided that *I do need more than one*

table. One reason for making this decision might be that, perhaps, many of my friends do not have passports, and I don't want to generate many extra `NULL` values (recall our discussion of Figure 1.6). In this case though, my reason is that whenever I lose a friend (maybe due to some disagreement over Android versus iPhone), I definitely want to delete their details from my Friends table. In the next table, I've deleted my ex-friend, Mr 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 Mr X, I also deleted his passport details. Now, why would I want to delete Mr X's passport details just because we are no longer friends? Since Mr X and I had our falling out, I just can't be sure that we stand on neutral ground. So, those details might come in handy during any future conflicts[1]. Hence, I'm going to model the one-to-one relationship 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 just insert `NULL` into the FriendID column of the Passports table. There is still a problem with this set-up, though. 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 that they won't get deleted when I delete a friend? If I keep their names in *both* the Friends table *and*

[1] I swear I am not a psychopath, and this is just a joke. Although, there is a nice lesson here: be careful who we trust with our data!

the Passports table, then the *same piece of data will be repeated in two different locations in my database*. This is known as a **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 can cause us to have to do more work if data needs to be updated (because we'll have to update the data in multiple locations).

In fact, to solve our problem with passport names, it may be best to rethink our database a little. Maybe we should 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 (e.g., Friends and Enemies) that contain friend-specific and enemy-specific data (like, favourite colours for friends, and secret hideouts for enemies). Database design is a deep and interesting topic, lying mostly outside the scope of these notes. So, next time you meet a database designer, give them a high five.

Chapter 2

Basic SQL retrieval queries

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 along the line, it underwent a rebranding and is now called Structured Query Language (SQL), though it is still commonly pronounced "sequel."

In this chapter, we introduce and visualise some of the basic fundamental clauses in SQL. In SQL, the clauses are based on a pair of formal mathematical languages, called *relational algebra* and *relational calculus*. We won't be learning these formal languages in this course. That which we will be learning, SQL, is an *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.

These notes follow the particular flavour of SQL that is used with Microsoft SQL Server, called Transact-SQL, or T-SQL. However, the basic syntax is roughly the same, across all implementations of SQL. This is due to the wonderful fact that SQL is a standardised language, and these standards are maintained by the International Organisation for Standardisation (ISO), and the American National Standards Institute (ANSI).

Our database is full of tables. A query is designed to go into the database, chop up tables, join them together, potentially aggregate the results, and return to us a single result 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. Often, when two tables have a relationship, we want to join them together, so that each record in the joined table corresponds to one fact about the miniworld. This section will give us such powers.

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 chop a table up by *columns* – it lets you *select* certain columns of the table. For example:

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

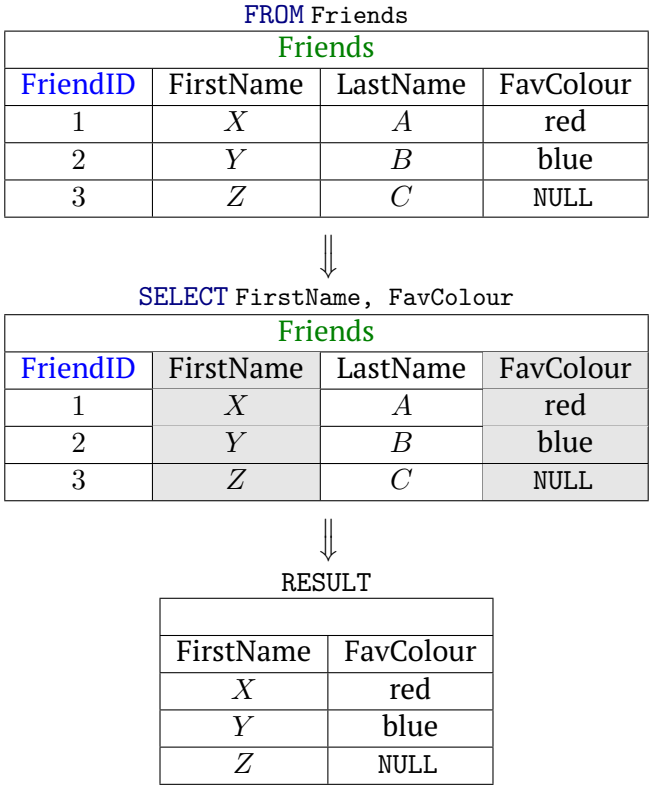


Figure 2.1: The SELECT and FROM clauses.

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.

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;
```

2.2 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. Really, 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.

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 4.3, and more on search conditions in Section 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.

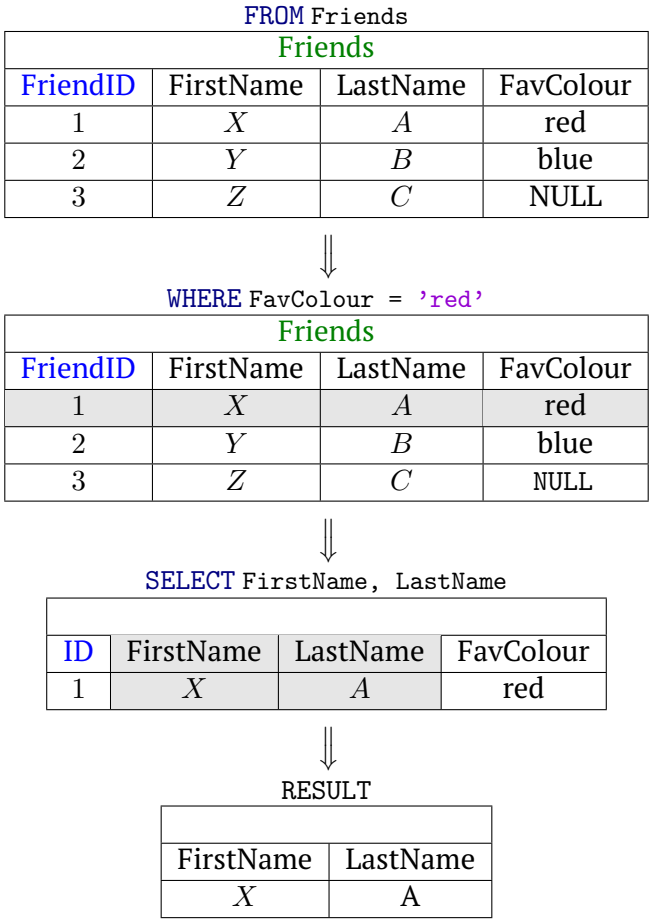


Figure 2.2: The `WHERE` clause.

2.3 JOIN

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. Thus, when we write a `JOIN` we need to tell the `JOIN` which columns have the important shared entries. For example, we can join the Friends and Pets tables (see Figure 1.8), by comparing the primary key, FriendID, from Friends with the foreign key, FriendID, from Pets (using the command `ON Friends.FriendID = Pets.FriendID`).

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

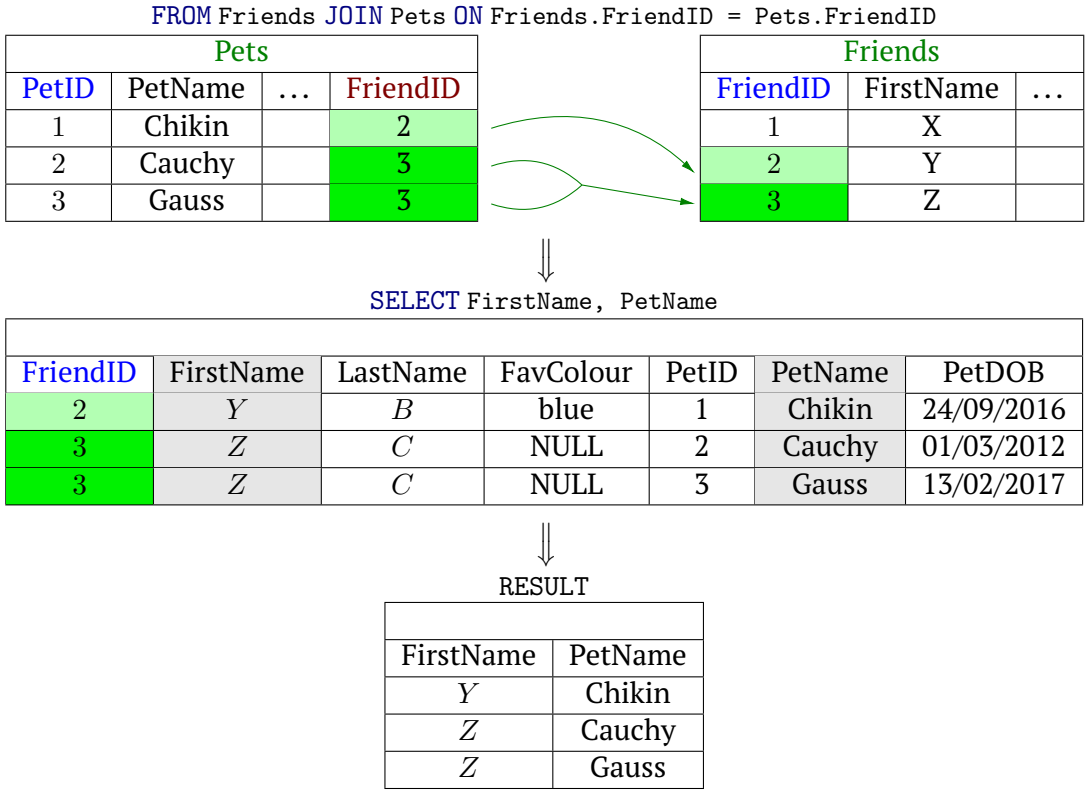


Figure 2.3: The JOIN clause

The above can be achieved more succinctly using aliases. Aliases are single letters or short words that allow 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, by writing `Friends F` and `Pets P`.

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

Lots of ways to do the same thing, huh? You've seen nothing. The following are all equivalent ways to write the above query.

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

```
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 the `WHERE` clause, instead of the `ON` clause, has been used to specify the join condition `F.FriendID = P.FriendID`. From now on, we will use the implicit join, since it is more succinct and readable.

To get a better understanding of how the 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

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, Table2 T2 WHERE 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

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

```
SELECT * FROM Table1 T1, Table2 T2 WHERE 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

2.4 LEFT 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`. Take another look at the `JOIN` clause execution diagram (Figure 2.3), 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 the tables, 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 Mr X has no pets, the query returns Mr X's name, but inserts `NULL` in the position where Mr X's PetName would go if he had a pet:

FirstName	PetName
X	NULL
Y	Chikin
Z	Cauchy
Z	Gauss

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 it does it on the right side instead of the left side. Finally, deep in the jungle, you might find a `FULL OUTER JOIN`, but this is a very rare occurrence, so I'll leave it up to you to figure out exactly what it

does (hint: it is both a left join and a right join in one, so it keeps all records from both tables – okay, that was more of an answer than a hint).

Chapter 3

Aggregating queries

We have learned how to retrieve and join tables, and now it's time to learn about aggregating queries!

At risk of oversimplifying, I'll say an aggregating query is the SQL way to somehow partition the rows of a table into groups and then somehow return *a single value* for each group. This is especially useful when we want to avoid extracting a very large dataset that we can instead summarise – 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, and finally an aggregating function can be used to get basic summary statistics (such as the average, or the standard deviation) within 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 by the values of 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:

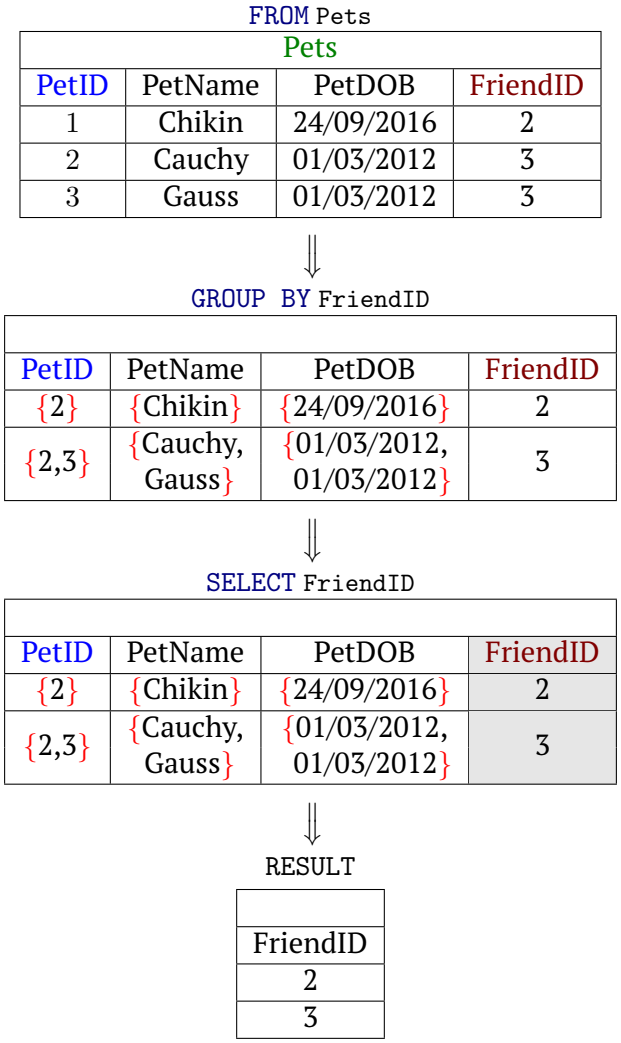


Figure 3.1: The `GROUP BY` clause

After grouping (with `GROUP BY`), each row of the table represents *one group*. The last two rows of the Pets table were placed into one 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 it 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 lists, which we are representing here with red curly brackets.

After grouping, we were able to execute `SELECT FriendID` with no issues. However, if we had chosen to `SELECT` any of the other columns, then SQL would have produced an error (see Figure 3.2)! This is because SQL cannot

```
Msg 8120, Level 16, State 1, Line 1 Column 'Notes.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 lists in it.

return a RESULT table that has any lists in it. The reason SQL cannot return any lists is that it wants to return only *one value for each entry*. For example, the entry in the second row of the PetName column contains two values (Cauchy and Gauss). In general, when there are lists, SQL doesn't check if the lists contain only one value in them. So, when we use GROUP BY, we can't select any columns that will end up with lists in them. How restrictive!

Thankfully, the lists are designed to be dealt with in various ways. One way to deal with those pesky lists 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, PetID

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

By chance (or by instructive design), the two pets that have FriendID equal to 3 also shared the same birthday, so the groups were unchanged compared to our previous result. What did chance is that now we don't have lists in the PetDOB column. Since the lists are gone from PetDOB, we can now SELECT that column in our query as well, without causing an error. 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. Here's an example of how that works, using a table called Letters.

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

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}

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 lists according as to whether they belonged to rows with matching values in column B. If we `GROUP BY A` instead, 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}

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}

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 lists in it.

3.2 Aggregation functions

In the previous section, when applying `GROUP BY`, we faced some pesky red lists. We learned that, before using `SELECT` on them, we could make the lists 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 3.3: 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}

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 list* 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;
```

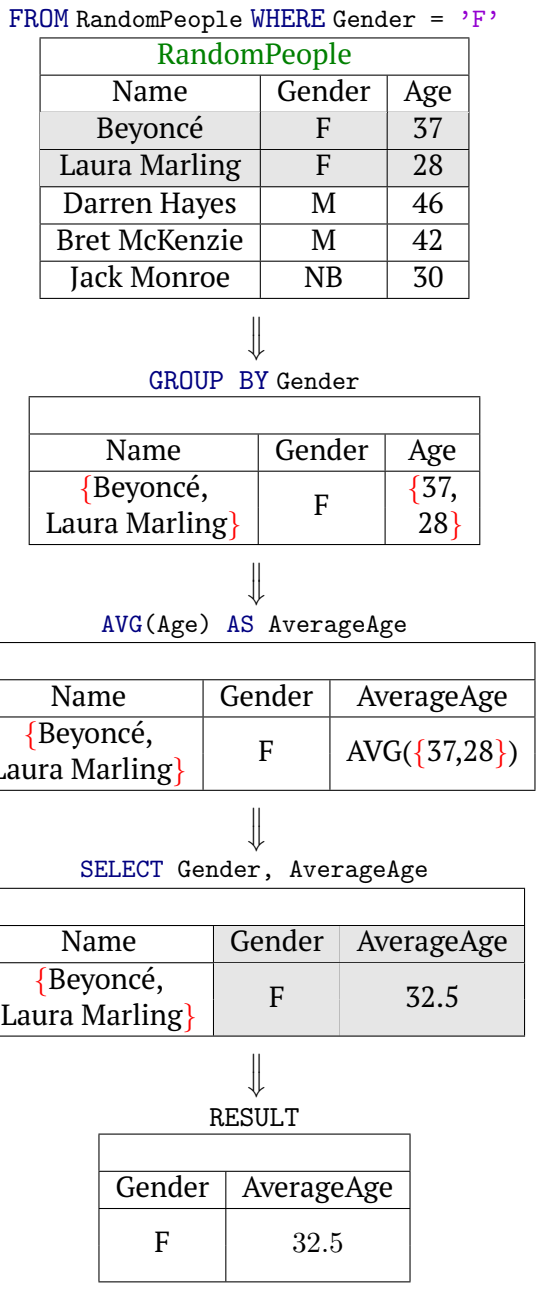


Figure 3.4: The AVG aggregation function

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 in most implementations of SQL. In Section 4, we will learn to read the Microsoft SQL online documentation, which is the best source of information on these functions, for Microsoft SQL (i.e., Transact-SQL). You can check them out at the website here (or if you're reading a printed copy of these notes then go ahead and type "Transact-SQL aggregation functions" into a good search engine). Here is a table of simple and useful aggregation functions:

Function	Purpose
AVG	Average
STDEV	Sample standard deviation
STDEVP	Population standard deviation
VAR	Sample variance
VARP	Population variance
COUNT	Count number of rows
MIN	Minimum
MAX	Maximum
SUM	Sum

Table 3.1: Some of the aggregation functions in Microsoft's Transact-SQL

3.3 HAVING

In the previous section we saw an example in which the `WHERE` clause was used to discard all the rows that didn't satisfy `Gender = 'F'`. The `HAVING` clause was introduced to SQL because **aggregation functions can't be used in the `WHERE` clause**. In other words, if we want to retain only the groups that have, say, average age greater than, say, 40, then we can specify the condition `HAVING AVG(Age) > 40`, like this:

```
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.

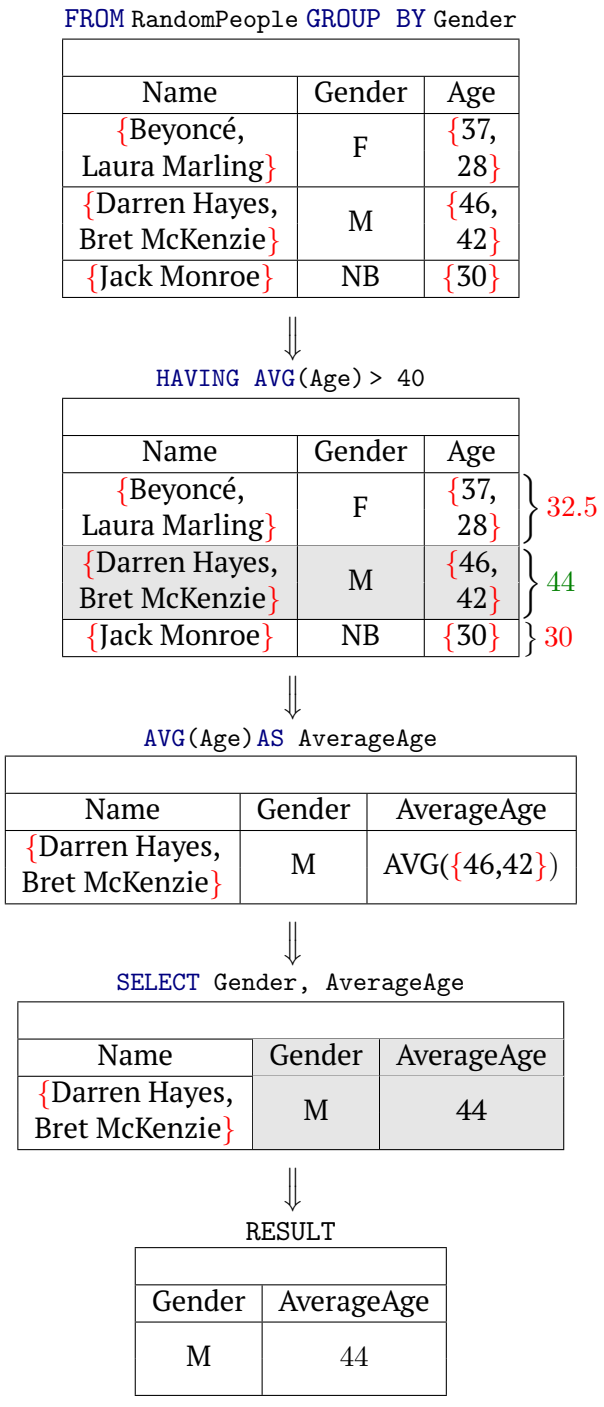


Figure 3.5: The `HAVING` clause

Notice that executing `HAVING AVG(Age) > 40` did not actually insert the

average ages into the table. The ages weren't inserted into the 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 query uses `STDEV(Age)` to return the sample standard deviation of ages in 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 of the aggregation functions (for some of these, see 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 cover search conditions in Section 4.4.

3.4 Nested queries

I thought long and hard about where to place the section on nested queries. 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. The question at the start of the next section should make this clear.

3.4.1 Basic nested queries

Here's a question: what if we need to aggregate ages in RandomPeople (Figure 3.3) – say, within each gender – but then we want to get a list of all the names that belong to one of the aggregated genders? For example, suppose we want a list of all the names of people who belong to every gender that has an average age greater than 40. We will learn to do this sort of thing with a nested query.

I can see in the execution diagram (Figure 3.5) that the names belonging to a gender with average age greater than 40 are Darren Hayes and Bret McKenzie, but I can't `SELECT` the Name column after I `GROUP BY Gender`, because it will contain the pesky red lists in it. What if I do the following?

```
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 lists, 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 subsequently 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. So, how do I get the names? Why is it so damn hard, when I can see the names *right there*??!

The solution is to use a **nested query**, saying "hey, SQL, see those names? can you please grab them from the RandomPeople table?" Then, SQL responds with, "which names exactly are you talking about there, buddy?" And you think carefully and say, "the names whose genders are present in the `RESULT` table of my previous query (Figure 3.5)!"

So, to wrap our heads around this, let's for a moment use the word `RESULT` to denote the previous query. This should do the trick:

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

Since `RESULT` represents a whole query of its own, we call it a nested query. To actually run this in SQL, we have to include the whole nested query itself. So, plugging that in place of `RESULT` we get:

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

I should point out that we've just used a command we haven't seen yet: the `IN` logical operator. We cover logical operators in Section 4.3.

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 get us thinking about efficiency; and (3) once you understand them, they can achieve some pretty complicated things, often in intuitive ways (with the caveat of potentially poor efficiency). 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 and referenced in 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. We can think of this query as being executed as follows:

1. First, extract a list of all the genders from RandomPeople, in order of appearance, giving Gender_list = {F, F, M, M, NB}.
2. For each element of Gender_list, the inner query is executed once, with `RP.gender` referring to one element from Gender_list. The result is a list of the average ages, `RESULT`_list = {32.5, 32.5, 44, 44, 30}.
3. Now we have one `RESULT` in `RESULT`_list for each row of RandomPeople. The outer query is now executed: the age of each person is compared to the corresponding `RESULT`, using the logical operator >.
4. The outer query now returns a table of the names of those people whose age is greater than the average age in their own gender.

The idea of using lists, as I have introduced here, is for intuition only. So, this is not exactly how the query executes, but it helps us reason about it logically. For example, since the inner query is executed once for each row of the outer query, we can intuitively see that this could be very inefficient. Also, if you have grasped the execution pattern, then you'll soon see (hopefully during the exercises) that you can achieve some complicated things fairly easily with a correlated nested query.

Chapter 4

Reading the docs

This short chapter 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 hundreds of beginner and intermediate tutorials available online. It is worth trying a few until you find one that suits you. My personal favourite, and one I highly recommend, is SelectStarSQL (click here). It's rare to find a truly passionate teacher offering their work for free, so make the most of it! Before laying out his principles of pedagogy, the SelectStarSQL author, Kao, writes

> " *When I was a data scientist at Quora, I used to have people ask me for resources for learning SQL. 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. My hope is that like Learn You a Haskell for Great Good! and Beautiful Racket have done for Haskell and Racket, Select Star SQL will become the best place on the internet for learning SQL.*"

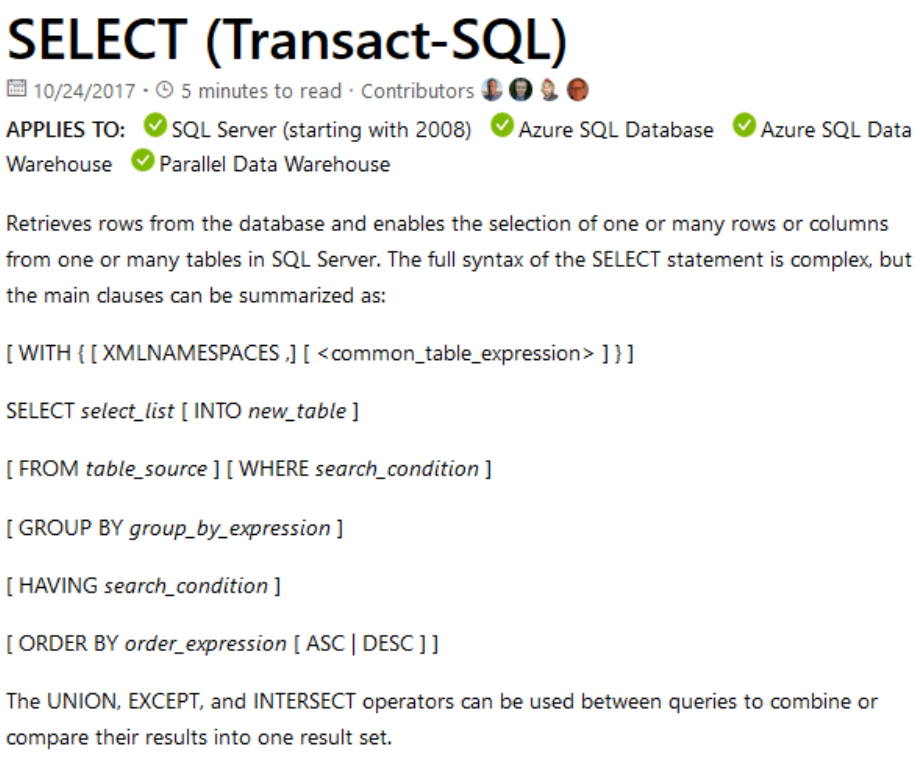
The second secret to being an effective programmer at any skill level is to be able to *read the docs*. Whatever language you're coding in, the documentation provided by the creator/maintainer is nearly always the most comprehensive and reliable source of information. As far as readability goes, well, let's just say that docs in general aren't famous for being beginner friendly. That's why I'm devoting this whole section of these notes

to teaching you how to read a tiny little bit of Microsoft's Transact-SQL 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 the docs don't immediately make sense. The rewards are worth the journey and a little understanding can go a long way.

4.1 How to read the docs

The first thing you'll need is a reference sheet of the T-SQL Syntax Conventions. You can find it in the T-SQL docs (click here). The reference sheet allows you to start making some sense of what the rest of the docs are saying. The essential part is only a page long.

Once you have your reference sheet handy, head over to the Queries page, found at this link (click here). In the navigation menu to the left of that page, click on `SELECT`. This takes you to the `SELECT` documentation page (click here). Here's a screenshot from the top of that page:



SELECT (Transact-SQL)

10/24/2017 · 5 minutes to read · Contributors

APPLIES TO: SQL Server (starting with 2008) Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

Retrieves rows from the database and enables the selection of one or many rows or columns from one or many tables in SQL Server. The full syntax of the SELECT statement is complex, but the main clauses can be summarized as:

[WITH { [XMLNAMESPACES ,] [<common_table_expression>] }]

SELECT *select_list* [INTO *new_table*]

[FROM *table_source*] [WHERE *search_condition*]

[GROUP BY *group_by_expression*]

[HAVING *search_condition*]

[ORDER BY *order_expression* [ASC | DESC]]

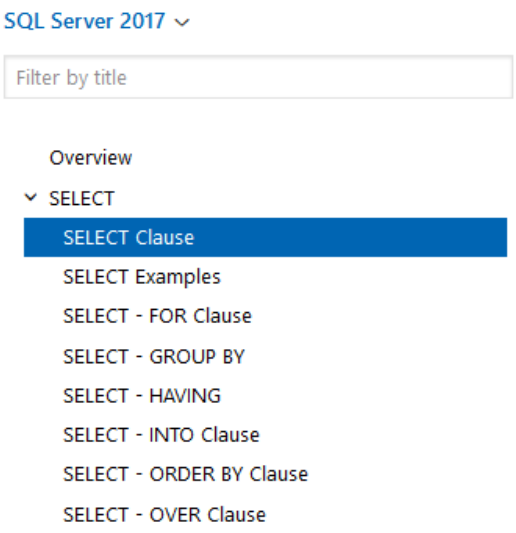
The UNION, EXCEPT, and INTERSECT operators can be used between queries to combine or compare their results into one result set.

There's a bunch of upper-case keywords there that you should recognise if you've been following these notes until now: `SELECT`, `FROM`, `WHERE`, `GROUP`

BY and HAVING. The other things that jump out are a few keywords we don't know yet, a few lower-case italicised words, and some different kinds of brackets. Since the majority of the brackets are square, let's refer to our T-SQL Syntax Conventions to find out what square brackets mean.

[ ] (brackets)	Optional syntax items. Do not type the brackets.

Ok, we see that the square brackets signify that whatever is inside the brackets is optional (and you shouldn't type the brackets themselves). So, the only part of the above query that is never optional is SELECT *select_list*. Take a moment to think back to all the queries that we have discussed in this chapter. They all have SELECT in them! So, really, the only queries that we have looked at in these notes so far are called SELECT queries, and they are all described in the SELECT section of the Queries documentation (click here). Now, to figure out what *select_list* means, we have to dig deeper into the SELECT docs. In the navigation menu to the left of the Queries documentation webpage, click on SELECT Clause (or click here). It's like:



On that page, under the heading **Arguments**, you can find ⟨*select_list*⟩ and an explanation for what it does:

< select_list > The columns to be selected for the result set. The select list is a series of expressions separated by commas. The maximum number of expressions that can be specified in the select list is 4096.

Great! So when we see the word *select_list* it just means that we can put a bunch of column names there (up to 4096 of them in fact), separated by commas. This is what we have done in every query in these notes so far, though keep in mind we can use the * symbol to represent *all* the columns. Wait, where in the docs does it say we can use the * symbol? Well, a little higher up on the SELECT Clause page there is a little box that looks like this:

Syntax

Copy

```
SELECT [ ALL | DISTINCT ]
[ TOP ( expression ) [ PERCENT ] [ WITH TIES ] ]
<select_list>
<select_list> ::=
    {
      *
      | { table_name | view_name | table_alias }.*
      | {
          [ { table_name | view_name | table_alias }. ]
              { column_name | $IDENTITY | $ROWGUID }
          | udt_column_name [ { . | :: } { { property_name | field_name }
            | method_name ( argument [ ,...n] ) } ]
          | expression
          [ [ AS ] column_alias ]
         }
      | column_alias = expression
    } [ ,...n ]
```

Now, this is a mess. So, in your mind, or on a piece of paper, remove everything that is optional (i.e., everything in square brackets). You'll get:

```
SELECT
<select_list>
<select_list> ::=
    {
      *
      | { table_name | view_name | table_alias }.*
      | {
```

```
            { column_name | $IDENTITY | $ROWGUID }
          | udt_column_name
          | expression
        }
    | column_alias = expression
}
```

Still kind of a mess. There are still a bunch of symbols we don't understand. In particular, we don't understand the `::=` thing, or the curly braces `{ }`, the vertical bar symbol `|`, nor a bunch of the words and dollar signs `$`. Working our way down from the top, let's check the T-SQL Syntax Conventions to figure out what some of them mean.

<label> ::=	The name for a block of syntax. This convention is used to group and label sections of lengthy syntax or a unit of syntax that can be used in more than one location within a statement. Each location in which the block of syntax can be used is indicated with the label enclosed in chevrons: <label>.

Ok, so in this case the `::=` thing is a way of saying that everything that comes after it defines ⟨*select_list*⟩. Now, how about the curly braces?

{ } (braces)	Required syntax items. Do not type the braces.

Ok, the curly braces tell us that whatever is inside the curly braces is not optional. I guess they're a way of grouping things together while making it clear that they aren't optional (as opposed to square brackets). Read on to the english language example below to see why curly braces are needed. Well, what about the vertical bars?

| (vertical bar)	Separates syntax items enclosed in brackets or braces. You can use only one of the items.

Great. Vertical lines tell us that we can use one, and only one, of whatever is on either side. Here's a fun example that combines all the above and uses the english language rather than SQL:

⟨*greeting*⟩ ::= Hello. [Do you {love | hate} reading the docs?]

The above tells us that ⟨*greeting*⟩ can be any of these:

- Hello.
- Hello. Do you love reading the docs?
- Hello. Do you hate reading the docs?

We can see from the above why the curly braces are useful. The curly braces have allowed us to make it clear that the vertical line | wants us to choose between "love" and "hate." Vertical bar, you are always asking us the hard questions. Now, go back up to the definition of ⟨*select_list*⟩ and make sure that you can see why simply writing * is allowed.

The journey ahead is long and arduous, but noble. Sadly, I cannot go with you, for I have limited resources and it is outside the scope of these notes. Besides, like I said at the start, this short chapter is intended to give you only the bare minimum to get started. I encourage you to read the docs frequently and with patience. If something you read doesn't make sense, then a quick online search is your friend.

4.2 A note on reserved keywords

When writing SQL queries, you may sometimes have to come up with your own names for things. For example, in Section 3.2 we wrote a query that calculates the average age of all females in the the RandomPeople table:

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

The returned table has a column called `AverageAge`, since we wrote `AVG(Age) AS AverageAge`. This name, `AverageAge`, was made up by me, but I could have picked *any name*, right? Well, you *can* choose any name, but you have to be careful. The T-SQL documentation includes a *list of reserved keywords (click here)*. These are words like `SELECT`, that should not be used for things like column names. If you want to use these keywords as column names (though you shouldn't), then you can enclose them in square brackets or quotation marks. So, if for some twisted reason you decide to name the column `SELECT` instead of `AverageAge`, then you could write:

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

4.3 Logical and comparison operators

In the statement `WHERE Gender = 'F'`, the symbol "=" is called a comparison operator. **Comparison operators** compare two things and return either `TRUE`, `FALSE` or `NULL` (unknown). **Logical operators** are similar but they can be used to compare more than two things. We first encountered "=" way back in Section 2.2, where we also encountered our first search condition. Search conditions go hand-in-hand with logical and comparison operators, and we talk about them more in the next section (Section 4.4).

A table of comparison operators (click here) and a table of logical operators (click here) can both be found under the language elements section of the T-SQL docs. We will have more practice with logical and comparison operators during the exercises and in Section 4.4.

4.3.1 A note on `NULL`

The word `NULL` deserves a special mention when discussing logical operators. Take a moment to decide what you think will be returned if I use the "=" symbol to compare two `NULL` values, like this:

$$\texttt{NULL} = \texttt{NULL}$$

It doesn't return `TRUE`! In fact, it returns `NULL`. This makes sense when you realise that `NULL` represents "unknown" or something that "does not exist," and every `NULL` is treated as being distinct from every other `NULL` (i.e., no two unkowns 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 makes sense because we cannot be sure whether the "unknown" represented by the `NULL` on the right hand side is actually equal to 10 or not, so we have to return "unkown".

This behaviour of `NULL` with logical 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 to use the SQL keyword `IS NULL` (click here) and/or the T-SQL built-in function `ISNULL` (click here), which we will practice using during the exercises.

4.4 Search conditions

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 brackets but it is often good for clarity. Search conditions can get about as complicated as you like. For example, the below will ensure results include only people who are both male and over 35, or both female and under 25.

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

It is also possible to include whole queries within search conditions, as nested queries. Recall what we looked at in Section 3.4.1:

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

The query that appears in brackets after the keyword `IN` is actually part of the search condition.

4.4.1 Wild cards in search conditions

A wild card allows us to conduct searches like "all words beginning with the letter 'A'." The wild card is a `%` symbol, and is most often used in search conditions with the logical 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%'
```

There are other wildcard characters that, combined with `%`, allow you to make some very powerful and imaginative searches. More details on these can be found under the T-SQL documentation for `LIKE` (click here).

Chapter 5

Exercises

5.1 A note on style

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 is a curse for anyone who needs to read SQL code (including the original programmers themselves).

Please be thoughtful about how you structure and present your code. The worst is when you go off and spend quality time with your family after writing an excellent SQL query, then come back and face beating yourself to death with your own keyboard for not using some friendly formatting standards: upper-case letters for keywords, simple clear comments, line breaks, and indentations.

Also, strictly speaking, you don't have to end your queries with a semicolon (;) in T-SQL, but it is a good idea, since there are other flavours of SQL that do require semicolons. Semicolons also help make it clear where one query ends a new one begins.

5.2 Exercises

In these exercises, it is assumed that you have your preferred SQL development environment installed, that you have connected to a database server, and that the server contains a database (that we will call PlayPen) that contains the Notes and Ape schemas. Guidelines for creating and accessing a database with these schemas are available at the course repository (by clicking here). Solutions are provided with these notes, but it's usually best not to read them until you're done.

5.2.1 Introductory exercises

1. Search online for your own beginner SQL tutorial or use this excellent tutorial: SelectStarSQL (click here). 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.

2. Search online for a simple syntax guide. I like the one from dofactory (click here). Don't spend too long on this. After you get more practice with SQL you'll have a better idea of which guide suits you. Keep in mind we are using T-SQL. If you encounter some commands in a guide that don't work as expected, it may be because they are using a different flavour of SQL (such as MySQL or SQLPlus).

3. Use the directory tree in SSMS or Azure Data Studio to begin investigating PlayPen and any other databases you have access to. Figure out some of the **table names** and **column names** in the Notes schema. The Notes schema contains all the tables defined in these notes. However, notice that some column names are different to the ones in these notes. Why might that be? Any ideas?

4. Expand the Notes.Friends table directory in the directory tree, then expand the 'columns' directory. What do you see? Can you determine the data types of each column? Can you determine whether `NULL` values are allowed? When a new table is created, the creator can decide whether to allow `NULL` values in each column. You can learn about data types and find the data types **varchar** and **int** in the T-SQL documentation (click here). Don't spend too long on the docs now! Come back to it later.

5. Right click on the Notes.Friends table in the directory tree and click 'Select Top 1000.' An SQL query is generated that selects the first 1000 rows, and the results are displayed. Why are there square brackets around the table, schema, and column names in the query? What will happen if you remove the square brackets? Try it.

5.2.2 Using the `SELECT` and `WHERE` clauses

6. Right-click on the PlayPen database icon in the directory tree and click "New Query" to open a new query editor linked to the PlayPen database. In the new editor, write and execute an SQL query that selects all of the rows and columns of the Notes.Pets table.

7. Write a query retrieving the PetName column of the Notes.Pets table.

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

```
SELECT PetName, FriendID
FROM Notes.Pets
WHERE FriendID = 3;
```

9. Write a query that displays the FirstName and LastName of every friend in the Notes.Friends table whose favourite colour is `'red'`.

10. Write a query that displays the ScratcherID and the ScratcheeID for all records in the Notes.Scratched table that occurred **on or before** the 5th of September 2018. Choose the comparison operator carefully, and specify the date using a string in the format `'YYYYMMDD'`. For example, to specify 12th of February 2016, you'd write `'20160212'`.

11. Write a query that displays all columns of the Notes.Scratched table and only returns records that have a ScratchTime between 11 AM and 12 PM (inclusive). You will need to use the comparison operator(s) <= and/or >=, as well as the logical operator `AND`.

12. Write a query displaying all columns of the records in Notes.Scratched whose ScratchTime is between 11 AM and 12 PM (inclusive) *and* whose ScratchDate is the 5th of September 2018.

13. Write a query displaying the full names of all of my friends in Notes.Friends whose favourite colour is either `'red'` *or* `'blue'`.

14. Write a query displaying all columns of the Notes.RandomPeople table for the rows where the PersonName starts with the letter 'B'. Use the `LIKE` logical operator and the % wild card (read about them on page 48 of these notes).

15. Edit your query from Question 14 (still using the `LIKE` logical operator and the % wild card) so that it displays only rows where the letters 'ar' appear together *anywhere* in PersonName.

16. We will now practice dealing with `NULL` values. First, write a query that displays the entire Notes.Friends table. Have a look at the result and pay attention to any `NULL` values. A `NULL` in the FavColour column indicates that a friend either doesn't have a favourite colour or that we don't know what it is. We want to write a query that lists all of the friends who have `NULL` favourite colour. What happens when you run the following? Why does it happen?

```
SELECT *
FROM Notes.Friends
WHERE FavColour = NULL;
```

If you're confused, then read "**A note on** `NULL`" on page 47. Now, fix the query using the `IS NULL` keyword. I have not shown you how to use this keyword, but I'm hoping you can use intuition to know where to write it within the search condition.

5.2.3 Joining tables

17. In the directory tree, find the list of columns of the Notes.Friends and Notes.Pets tables in the PlayPen database. From here, determine the name of the primary key in Notes.Friends, and the name of the primary and foreign keys in Notes.Pets (there is only one foreign key in this table).

18. Open a new query editor linked to the PlayPen database. Execute the following query and explain what it does:

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

From the results, how many pets does my friend named 'Z' have and what are their names?

19. Execute the following query and explain what it does. Is the result any different to the result from Question 18? Why or why not?

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

20. Use a `JOIN` to combine Notes.Table1 with Notes.Table2. Use an implicit join or an explicit join as you prefer (they do the same thing, only the syntax is different). The syntax in Question 18 is for an implicit join and the syntax in Question 19 is for an explicit join.

    Join the two tables by matching `Table1.A` with `Table2.A` (this is the intended way, matching the primary key with the foreign key). Then, try instead joining them by matching `Table1.A` with `Table2.E` (an unintended way). What happens if you join in another unintended way: matching `Table1.C` with `Table1.C`? Why did this happen?

21. This query will involve the Ape schema, which is also located in the PlayPen database. Write a query that produces a table containing the full name of each ape in the Ape.Friends table, along with the name of their favourite colour (from the Ape.Colours table).

22. Edit the query from Question 21 so that it only returns entries where the favourite colour is blue. Use the logical operator `AND`.

23. Now we will join 3 tables together. We want a table that tells us how many times each of my friends from Notes.Friends has played with each pet from Notes.Pets, and also includes full details on each friend and pet that played together. The play count is stored in the table Notes.PlayCount which is described on page 18 of these notes. Use the directory tree to confirm that Notes.PlayCount has two foreign keys, and to confirm their names.

    In the T-SQL docs, the `JOIN` clause is actually part of the `FROM` clause. It is quite complicated to interpret, but part of it looks like this:

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

    Referring to the T-SQL Syntax Conventions page (click here), we see `[ ,...n ]` tells us we can repeat `<table_source>` as many times as we want, separated by commas. See if you can use this knowledge to guess the syntax for joining the 3 tables. Ask for help if you're stuck.

24. There were duplicate columns present in the result for Question 23. To produce a cleaner result after a join, select only the columns that you need (instead of using `*`). Edit your solution to Question 23 so that no duplicate columns are produced. There is a tricky side to this: If you select a column that appears in more than one of the joined tables then you will get an error saying that the column is ambiguously defined. You can avoid this error by either writing the full table name before the column name, as in `SELECT Notes.Pets.PetID`, or by giving the table name an alias, as in `FROM Notes.Pets P`, and later referring to it in the `SELECT` clause using the alias, as in `SELECT P.PetID`.

25. We will again join 3 tables together, but this time two of the tables will be *the same table*. The Notes.Scratched table keeps record of which friend scratched whose back (and the time and date). We would like to join this with Notes.Friends in such a way that each row of the resulting table contains the first names of both the scratchee and the scratcher, as well as the date and time (4 columns). Make sure that you use the keyword `AS` to give the resulting FirstName columns appropriate names (e.g., ScratcherName and ScratcheeName) so that the two can't be confused.

26. In the Ape schema, produce a table of the details of every ape, along with the details of their favourite colour. If an ape does not have a favourite colour, or their favourite colour is unknown, then the table should have `NULL` values in place of the corresponding colour details.

27. The following query can be achieved with a `LEFT JOIN`, but can you do it with a `RIGHT JOIN`? In the Ape schema, produce a table of all the details of every colour, along with all the details of every ape who chose that colour as their favourite. If a colour isn't liked by any apes, then it should still appear in the table.

28. In these notes you have seen nested queries used within search conditions (e.g., in the `WHERE` clause). You will now write a nested query in the `FROM` clause. This means that the result of the nested query will be treated like any other table in the `FROM` clause. Using your answer to exercise 27 as `RESULT`, write a nested query that selects only the rows of `RESULT` that contain `NULL` in the column FavColourID. Do not return all the columns: only return ColourID, ColourName and Comments. What are you looking at?

29. In the Ape schema, return a table of the FirstName, LastName, ColourName and Comments for every ape that either has favourite colour orange, or favourite colour unknown.

30. Produce a table of all the apes, with all their favourite colours, but this time make sure that every ape is included at least once, and every favourite colour is included at least once. Ask for help if you're not sure what to do.

5.2.4 Execution-free join exercises

These join exercises will not require you to execute any code. If you are asked to write SQL code then you can write it on paper, if you like.

31. A house can have many tenants and each tenant lives in one house.

House	
HouseID	Address
⋮	⋮

Tenant	
TenantID	Name
⋮	⋮

    (a) What is likely to be the primary key in each table?

    (b) Choose a name for the foreign key.

    (c) Choose which table to put the foreign key in.

32. The Vacation table lists details of each vacation that a traveller has been on. A traveller can attend multiple vacations and a vacation can be attended by multiple travellers.

Vacation			
TravellerID	City	Country	Language
1	Paris	France	French
1	Paris	France	French
1	Barcelona	Spain	Spanish
2	Paris	France	French
2	Barcelona	Spain	Spanish
2	Berlin	Germany	German
3	Berlin	Germany	German

(a) Is TravellerID a primary key of the Vacation table?

(b) Is the relationship one-to-many or many-to-many?

(c) How can we model this relationship in a better way?

33. We'll now look at a situation where the primary and foreign keys are chained in such a way that you have to go through one table to get to another one. Suppose that we want to link a person's age to the average annual income bracket of the suburb that they live in. We 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	Status_ID
24	Balwyn	3103	1
12	Glen	3146	1
32	Hawthorn	3122	3

Status					
Status_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 Status_ID is the primary key of the Status table. Write a query in SQL that produces a table of people's

birth years, along with the start and end of the average annual income bracket of the suburb that they live in.

5.2.5 Reading the docs

34. Read the T-SQL documentation for `TOP` (click here). Now, make changes to the query from Quesion 5 so that it selects the top 30% of rows from the Notes.Friends table (HINT: the keyword `PERCENT` is described in the `TOP` documentation). Execute the query.

35. Write a query that selects all columns of the Notes.Scratched table but only returns records in which the ScratchTime was strictly before 12 PM. You will need to use the right comparison operator, and, similar to question 10, give the times as a string (also known as a string literal) using a standard format. Use the T-SQL documentation on time and date types (click here) to determine the right format for the string literal.

36. The solution to Question 11 can be achieved in a more readable way using the `BETWEEN` clause. Read about it on the T-SQL docs for `BETWEEN` (click here) then try to use it to replace your solution to Question 11.

37. We've seen the % wildcard. Read about the other wildcard characters (`_`,`[ ]` and `[^]`) in the T-SQL docs for `LIKE` (click here). Run the following query and explain what it does.

```
SELECT *
FROM Notes.RandomPeople
WHERE PersonName LIKE '_[ra]__ %';
```

38. Read the T-SQL documentation for the SQL ***keyword*** `IS NULL` (click here) and then also read the T-SQL documentation for the SQL ***built-in function*** `ISNULL` (click here). In question 16 we used `IS NULL`. Can you solve the problem with `ISNULL` instead?

39. Edit your solution from Question 38 so that it only returns rows where FavColour exists and is not unknown. There are various ways to do this, and it may help to look through some of the documentation that you have already seen.

5.2.6 Using the `GROUP BY` clause

40. Open a query editor linked to the PlayPen database. First we will have a look at the contents of the Notes.Letters table. Write a query that displays all of the columns and rows of the Notes.Letters table. Then, execute the following query and explain what happened:

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

41. Now we will execute a query that fails and returns an error that is very common when using `GROUP BY` (at least amongst SQL newbies). Execute the below query and explain why the error occurred. You may need to revise Section 3.1, and in particular page 32.

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

42. In the query below, we've edited Question 41 so that it uses the aggregation function `MAX`. This prevents the error from occurring. 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 alphanumerically.

43. 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.

44. 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.

5.2.7 Aggregation functions and the `HAVING` clause

45. Run the following query and explain what it does.

```
SELECT AVG(Height) AS AvgTreeHeight
FROM Ape.BananaTree;
```

46. Retrieve the sample standard deviation of the widths of banana trees in Ape.BananaTree. You may wish to consult the T-SQL aggregation function docs (click here).

47. Retrieve the population standard deviation of the widths of banana trees in Ape.BananaTree. When should the population standard deviation be used?

48. Run the following query. It produces an error. Explain why. You may want to compare the error message to the one we got in Question 41.

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

49. If we add `GROUP BY TreeID` to the query from Question 48, then no error will be returned. Execute the below, and explain why it isn't very usefull.

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

50. Use a similar query to the one above to return the average height of banana trees for each YearPlanted in Ape.BananaTree. There should be two columns in your result table: AverageHeight and YearPlanted.

51. Use the aggregation function `COUNT` (click here) to display the number of times that each colour appears in Ape.Friends. There is no need to include a column of colour names, just the FavColourID is fine.

52. Get the maximum height for each MonthPlanted in Ape.BananaTree.

53. Get the average height for each month/year pair in Ape.BananaTree.

54. Execute the following query and explain what it does.

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

55. Execute the query below. It produces an error. Explain why.

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

56. Retrieve the average height and maximum width for each MonthPlanted in Ape.BananaTree, and discard any months where the maximum width of the trees is below 35.

57. 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. Make sure the calculated average has data type Float.

58. Edit the query from Question 57 so that it only returns results from the tree with TreeID = 5.

59. Consider the correlated nested query (explained on page 39):

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

This query would be inefficient if RandomPeople was a large table. Can you rewrite it as a join? Try to do it 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 I have used `R` as an alias for `RESULT`, just like we can do with any table. Note also that you may want to use `GROUP BY` within the nested query, and you may not want to join on a primary and foreign key pair. 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;
```

5.2.8 Putting it all together

60. In Section 1.4.3 I briefly introduced Notes.Houses and Notes.Suburbs in a discussion of informal relationships. Before moving on, investigate the two tables and the relationship between them, in SSMS or Azure Data Studio. There is no specific solution to this question.

61. Suppose we want a table that contains the details of each house, and the suburb to which it belongs. Join the Notes.Houses and Notes.Suburbs tables via their shared attributes (post_code). Are there any problems with this? For each of the following questions, decide if it can be answered using the data in the join result. If it can't, then explain why, and decide whether the problem would be fixed if `Suburbs.post_code`

and `Houses.post_code` were a primary / foreign key pair (that is, a one-to-one or one-to-many formal relationship).

(a) What is the suburb vaccination rate for house `'H0001'`?

(b) What suburb does house `'H0011'` belong to?

(c) How many houses from Notes.Houses are in suburb `'3142'`?

62. The solution to Question 51 was:

```
SELECT COUNT(FavColourID) AS Appearances, FavColourID
FROM Ape.Friends
GROUP BY FavColourID;
```

Extend this query, by joining Ape.Friends with Ape.Colours, so that the result table includes the names of the colours and not FavColourID.

63. Extend your solution to Question 62 so that it excludes all colours that appear less than twice.

64. Extend your solution to question 63 so that it excludes all colours whose name starts with 'b'.

65. Join the Ape.Banana table with the Ape.BananaTree table, and return only rows where the banana has a TasteRank of 5.

66. Read the T-SQL documentation for the built-in function `DATEDIFF`. Run the following query and explain what it does.

```
SELECT *, DATEDIFF(day, B.DatePicked, B.DateEaten)
      AS DateDifference
FROM Ape.Banana B, Ape.BananaTree BT
WHERE B.TreeID = BT.TreeID
AND B.TasteRank = 5;
```

67. Change the query from Question 66 so that it displays the number of days between DatePicked and DateEaten for every row with a TasteRank of 1 or a TasteRank of 2.

68. So far, the rows of the tables that we've produced haven't been ordered in any meaningful way. Read about the keyword `ORDER BY` in the T-SQL docs. Change the query from Question 66 so that it includes every TasteRank, but orders the results according to TasteRank.

69. Referring to the T-SQL documentation for `ORDER BY` again, change your solution to Question 68 so that it lists the TasteRank in descending rather than ascending order (Hint: Use the keyword `DESC`).

70. We have seen that the keyword `TOP` can be used to retrieve the first few rows from a table. In combination with `ORDER BY`, you can use `TOP` to get the leading results for any column you choose. We will now write a query that gives us the TasteRank of all of the bananas produced by the top 5 tallest trees. We will do it in a few steps over the next couple of questions. To start, produce a table of TreeID's from Ape.BananaTree ordered by height in descending order.

71. Now, edit your query from the previous question so that it only returns the TreeID's of the top 5 tallest trees.

72. Finally, create a query that retrieves the TasteRank and Comments from all rows of the Ape.Banana table whose TreeID's are in the results of your query from the previous question. You should also display the TreeID of each tree in your result table. Part of your query should include:

```
WHERE TreeID IN (<result>);
```

    where you should replace `<result>` with your previous query (Hint: In the table of logical operators look for the operator `IN`. Compare the description of `IN` to its use in our query at the end of Section 3.4.1).

73. This query is not small, so you should try to break it down into multiple steps. In the solution, we build it up over 5 steps. The ape who created the Ape database wants to figure out which banana tree is the most popular. The ape did some field work and recorded the FriendID and TreeID, in Ape.EatingFrom, each time she saw one of her friends eating from one of the banana trees. She wants to get a table of TreeID's, and number of visits, for the top 30% most popular trees. However, she only trusts the judgement of apes whose favourite colour is yellow, so she will only count visits from apes that meet this condition. Write the query that the ape needs.

Chapter 6

Solutions

6.
```
SELECT *
FROM Notes.Pets;
```

7.
```
SELECT PetName
FROM Notes.Pets;
```

8. The query selects the PetName and FriendID columns of every row in the Notes.Pets table that has FriendID equal to 3. These are all the pets that belong to my friend whose FriendID is equal to 3.

9.
```
SELECT FirstName, LastName
FROM Notes.Friends
WHERE FavColour = 'red';
```

10.
```
SELECT ScratcherID, ScratcheeID
FROM Notes.Scratched
WHERE ScratchDate <= '20180905';
```

11.
```
-- The brackets that we use here are unnecessary
SELECT *
FROM Notes.Scratched
WHERE (ScratchTime >= '11:00AM') AND (ScratchTime <=
    '12:00PM');
```

12.
```
-- The brackets that we use are unnecessary
-- Pay attendtion to the indentation that helps readability
SELECT *
FROM Notes.Scratched
```

```
WHERE (ScratchTime BETWEEN '11:00AM' AND '12:00PM')
 AND (ScratchDate = '20180905');
```

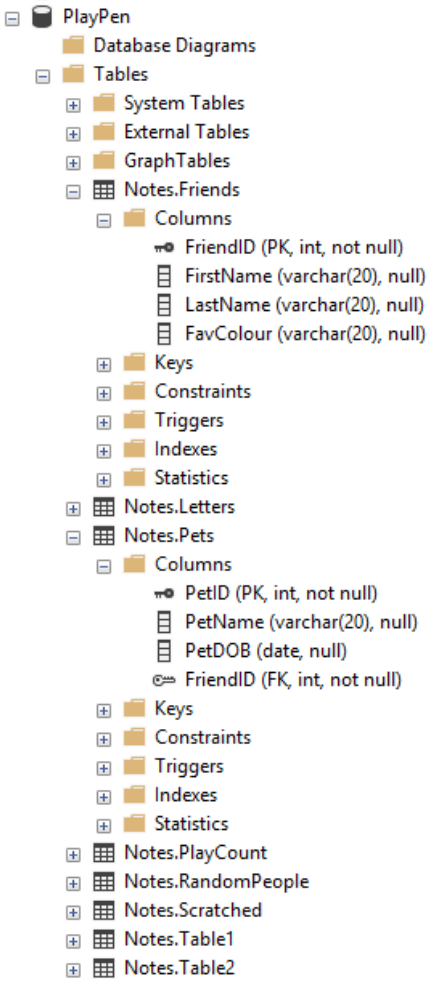
13.
```
SELECT FirstName, LastName
FROM Notes.Friends
WHERE FavColour = 'red' OR FavColour = 'blue';
```

14.
```
SELECT *
FROM Notes.RandomPeople
WHERE PersonName LIKE 'B%';
```

15.
```
SELECT *
FROM Notes.RandomPeople
WHERE PersonName LIKE '%ar%';
```

16.
```
-- Using IS NULL keyword
SELECT *
FROM Notes.Friends
WHERE FavColour IS NULL;
```

17. You should see something like this:



The column names that have PK in brackets after them are primary keys, and those that have FK in brackets after them are foreign keys. So, the primary key of Notes.Friends is FriendID, the primary key of Notes.Pets is PetID, and the only foreign key of Notes.Pets is FriendID.

18. The query joins (`JOIN`) the Notes.Friends table with the Notes.Pets table by comparing the primary/foreign key pair FriendID. My friend 'Z' has two pets: Cauchy and Gauss.

19. There is no difference. One is an explicit join and the other is an implicit join. They both do the same thing and the only difference is the syntax.

20. ```
    -- We used an implicit join
    ```
    ```

```
  -- Join on Table1.A = Table2.A
SELECT *
FROM Notes.Table1 T1, Notes.Table2 T2
WHERE T1.A = T2.A;

-- Join on Table1.A = Table2.E
SELECT *
FROM Notes.Table1 T1, Notes.Table2 T2
WHERE T1.A = T2.E;

-- Join on Table1.C = Table1.C
SELECT *
FROM Notes.Table1 T1, Notes.Table1 T1_again
WHERE T1.C = T1_again.C;
```

Why did this happen? Answer: Every column has 'is' in column C, so every row matches with every other row.

21.
```
-- We used ''AS'' to relabel the column as FavouriteColour,
-- though that was unnecessary
SELECT FirstName, LastName, ColourName AS FavouriteColour
FROM Ape.Friends F, Ape.Colours C
WHERE F.FavColourID = C.ColourID;
```

22.
```
SELECT FirstName, LastName, ColourName AS FavouriteColour
FROM Ape.Friends F, Ape.Colours C
WHERE F.FavColourID = C.ColourID
AND ColourName = 'blue';
```

23.
```
--The brackets that we used are unnecessary and we
-- used an implicit join, but you may prefer to use
-- an explicit join
SELECT *
FROM Notes.Friends F, Notes.PlayCount PC, Notes.Pets P
WHERE (F.FriendID = PC.FriendID) AND (PC.PetID = P.PetID);
```

24.
```
SELECT F.FriendID, F.FirstName, F.LastName, F.FavColour,
       P.PetID, PC.PlayCounter, P.PetName, P.PetDOB
FROM Notes.Friends F, Notes.PlayCount PC, Notes.Pets P
WHERE (F.FriendID = PC.FriendID) AND (PC.PetID = P.PetID);
```

25.
```
--Pay attention to the use of ScratcheeID and ScratcherID in
--the WHERE clause, and its relation to our choice of name
--for F1.FirstName and F2.FirstName
```

```
SELECT F1.FirstName AS ScratcherName,
       F2.FirstName AS ScratcheeName,
       S.ScratchDate, S.ScratchTime
FROM Notes.Friends F1, Notes.Scratched S, Notes.Friends F2
WHERE F1.FriendID = S.ScratcherID AND S.ScratcheeID =
    F2.FriendID;
```

26.
```
SELECT *
FROM Ape.Friends F LEFT JOIN Ape.Colours C
ON F.favColourID = C.colourID;
```

27.
```
SELECT *
FROM Ape.Friends F RIGHT JOIN Ape.Colours C
ON F.favColourID = C.colourID;
```

28.
```
SELECT colourID, colourName, comments
FROM (SELECT *
     FROM Ape.Friends F RIGHT JOIN Ape.Colours C
     ON F.favColourID = C.colourID)
WHERE favColourID IS NULL;
```

29.
```
SELECT F.firstName, F.lastName, C.colourName, C.comments
FROM Ape.Friends F LEFT JOIN Ape.Colours C
ON F.FavColourID = C.ColourID
WHERE C.ColourName = 'orange'
OR C.ColourName IS NULL;
```

30.
```
SELECT *
FROM Ape.Friends F FULL OUTER JOIN Ape.Colours C
ON F.favColourID = C.colourID;
```

31. The HouseID is the primary key of the House table and the TenantID is the primary key of the Tenant table. We create a foreign key called TenantsHouseID. We have to put the foreign key in the many side, which is the Tenant table.

32. TravellerID is not a primary key in the vacation table because the TravellerID is not unique in that table. The relationship is many-to-many, and ideally would be modelled by creating a new table:

Vacation	
TravellerID	LocationID
1	1
1	1
1	2
2	1
2	2
2	3
3	3

Location			
LocationID	City	Country	Language
1	Paris	France	French
2	Barcelona	Spain	Spanish
3	Berlin	Germany	German

33. We can us an implicit join:

```
SELECT P.BirthYr, ST.StartBracket, ST.EndBracket
FROM Person P, Suburb SU, Status ST
WHERE P.S_ID = SU.S_ID
  AND SU.Status_ID = ST.Status_ID;
```

Or, equivalently, we can use an explicit join:

```
SELECT P.BirthYr, ST.StartBracket, ST.EndBracket
FROM Person P
     JOIN Suburb SU ON P.S_ID = SU.S_ID
     JOIN Status ST ON SU.Status\_ID = ST.StatusID;
```

34.

```
-- I left the square brackets in but they aren't necessary
-- I used * instead of naming all columns
SELECT TOP (30) PERCENT *
FROM [Notes].[Friends];
```

35.

```
SELECT *
FROM Notes.Scratched
WHERE ScratchTime < '12:00PM';
```

36.

```
SELECT *
FROM Notes.Scratched
WHERE ScratchTime BETWEEN '11:00AM' AND '12:00PM';
```

37. The query displays only the rows where the PersonName begins with a four letter word, whose second letter is either ‘r’ or ‘a’ (pay attention to the space in between the % symbol and the last underscore).

38.
```
-- Using ISNULL function. We convert any NULL values
-- to the string 'no colour' and then check using '='.
SELECT *
FROM Notes.Friends
WHERE ISNULL(FavColour, 'no colour') = 'no colour';
```

39.
```
-- Approach 1
-- Using IS NOT NULL keyword
SELECT *
FROM Notes.Friends
WHERE FavColour IS NOT NULL;

-- Approach 2
-- Using ISNULL function, and
-- the 'not equal' comparison operator '<>'
SELECT *
FROM Notes.Friends
WHERE ISNULL(FavColour, 'no colour') <> 'no colour';

-- Approach 3
-- Using ISNULL function, and
-- the NOT keyword
SELECT *
FROM Notes.Friends
WHERE NOT (ISNULL(FavColour, 'no colour') = 'no colour');
```

40. The query groups the rows of the Notes.Letters table according as to whether they have the same value in the column named B. Then, it selects only the column named B from the result. Thus the query returned one row for each unique value in the column B. For revision on how this works see Section 3.1, and in particular see Page 32.

41. The query groups the rows of the Notes.Letters table on column B. Then, it selects columns A and B from the result. The error occurs because if a selected column (e.g., column A) is not present in the `GROUP BY` clause, then its values need to be summarised into a single value for each group, e.g. using an aggregation function (see Page 32).

42. Using `MAX(A)` returns a single value for each group formed by `GROUP BY`. This makes it possible to select the result (see full explanation in Section 3.1).

43.
```
SELECT A, B
FROM Notes.Letters
GROUP BY A, B;
```

44.
```
SELECT Gender
FROM Notes.RandomPeople
GROUP BY Gender;
```

45. It retrieves the average height of banana trees in Ape.BananaTree.

46.
```
SELECT STDEV(Width) AS TreeWidthSD
FROM Ape.BananaTree;
```

47.
```
SELECT STDEVP(Width) AS TreeWidthSD
FROM Ape.BananaTree;
```

Population standard deviation should be used if the table represents the entire population and not just a sample of the population.

48. The error is similar to the one in Question 41. The problem is that we have used an aggregate function to return a single value for Height, but there are many values for TreeID.

49.
```
SELECT AVG(Height) AS AvgTreeHeight, TreeID
FROM Ape.BananaTree
GROUP BY TreeID;
```

The query is overcomplicated because every row has a unique TreeID, so there is one group created for each row. We could achieve the same thing with:

```
SELECT Height, TreeID
FROM Ape.BananaTree;
```

50.
```
SELECT AVG(Height) AS AvgTreeHeight, YearPlanted
FROM Ape.BananaTree
GROUP BY YearPlanted;
```

51.
```
SELECT COUNT(FavColourID) AS Appearances, FavColourID
FROM Ape.Friends
GROUP BY FavColourID;
```

52.
```sql
SELECT MAX(Height) AS MaxTreeHeight, MonthPlanted
FROM Ape.BananaTree
GROUP BY MonthPlanted;
```

53.
```sql
SELECT AVG(Height) AS AvgTreeHeight, MonthPlanted, YearPlanted
FROM Ape.BananaTree
GROUP BY MonthPlanted, YearPlanted;
```

54. The query retrieves the average height for each month/year pair in Ape.BananaTree and discards all rows where the average height is not greater than 5.

55. SQL does not allow aggregation functions to be used in the WHERE clause. This is actually the reason why the HAVING clause was created.

56.
```sql
SELECT AVG(Height) AS AvgTreeHeight, MonthPlanted,
       MAX(Width) AS MaxWidth
FROM Ape.BananaTree
GROUP BY MonthPlanted
HAVING MAX(Width) < 35;
```

57.
```sql
-- Don't forget to CAST TasteRank from to float first
SELECT AVG(CAST(TasteRank AS Float)) AS AvgTaste, Ripe
FROM Ape.Banana
GROUP BY Ripe;
```

58.
```sql
-- Here we needed to use WHERE rather than HAVING
-- since 'TreeID = 5' does not involve
-- an agregation function
SELECT AVG(CAST(TasteRank AS Float)) AS AvgTaste, Ripe
FROM Ape.Banana
WHERE TreeID = 5
GROUP BY Ripe;
```

59.
```sql
SELECT 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;
```

60. No solution required.

61. The join query is:

```
SELECT *
FROM Notes.Houses H JOIN Notes.Suburbs S
ON H.post_code = S.post_code;
```

    (a) Cannot be answered. House `'H0001'` has post_code `'3128'`, which is shared by two suburbs with two different vaccination rates. This would be fixed by the formal relationship, since the primary key `Suburbs.post_code` would be guaranteed unique.

    (b) Cannot be answered. The postcode for house `'H0011'` is `NULL`. This would not be fixed by the formal relationship, since the foreign key `Houses.post_code` would still allow `NULL` values. It could be fixed if `post_code` was set to `NOT NULL`.

    (c) This can be answered, but not from the join result. There is no suburb `'3142'` in the suburb table. This would be fixed by the formal relationship, since the the foreign key `Houses.post_code` would be guaranteed to point to a primary key entry that exists in `Suburbs.post_code`.

62.

```
-- Approach 1
-- In this approach we chose to GROUP BY ColourName
-- instead of FavColourID. This works because ColourName
-- is unique to FavColourID
SELECT COUNT(C.ColourName) AS Appearances, C.ColourName
FROM Ape.Friends F, Ape.Colours C
WHERE F.FavColourID = C.ColourID
GROUP BY C.ColourName;

-- Approach 2
-- Here we chose to GROUP BY ColourName
-- as well as FavColourID
SELECT COUNT(F.FavColourID) AS Appearances, C.ColourName
FROM Ape.Friends F, Ape.Colours C
WHERE F.FavColourID = C.ColourID
GROUP BY C.ColourName, F.FavColourID;

-- Approach 3 (using a nested join)
-- In this approach we joined the
-- result of our old query with the Ape.Colours table
SELECT T1.Appearances, T2.ColourName
FROM (SELECT COUNT(FavColourID) AS Appearances, FavColourID
    FROM Ape.Friends
    GROUP BY FavColourID) T1,
```

```
  Ape.Colours T2
WHERE T1.FavColourID = T2.ColourID;
```

63.

```
-- To extend Approach 1 (and Approach 2 is similar):
-- Just add a HAVING clause to the end
SELECT COUNT(C.ColourName) AS Appearances, C.ColourName
FROM Ape.Friends F, Ape.Colours C
WHERE F.FavColourID = C.ColourID
GROUP BY C.ColourName
HAVING COUNT(C.ColourName) >= 2;

-- To extend Approach 3:
-- We need to add the HAVING clause inside
-- the nested query because HAVING must
-- appear after GROUP BY
SELECT T1.Appearances, T2.ColourName
FROM (SELECT COUNT(FavColourID) AS Appearances, FavColourID
    FROM Ape.Friends
    GROUP BY FavColourID
  HAVING COUNT(FavColourID) >= 2) T1,
  Ape.Colours T2
WHERE T1.FavColourID = T2.ColourID;
```

64.

```
-- To extend Approach 1:
SELECT COUNT(C.ColourName) AS Appearances, C.ColourName
FROM Ape.Friends F, Ape.Colours C
WHERE F.FavColourID = C.ColourID
    AND C.ColourName NOT LIKE 'b%'
GROUP BY C.ColourName
HAVING COUNT(C.ColourName) >= 2;

-- To extend Approach 3:
SELECT T1.Appearances, T2.ColourName
FROM (SELECT COUNT(FavColourID) AS Appearances, FavColourID
    FROM Ape.Friends
    GROUP BY FavColourID
  HAVING COUNT(FavColourID) >= 2) T1,
  Ape.Colours T2
WHERE T1.FavColourID = T2.ColourID
   AND T2.ColourName NOT LIKE 'b%';
```

65.

```
SELECT *
FROM Ape.Banana B, Ape.BananaTree BT
WHERE B.TreeID = BT.TreeID
AND B.TasteRank = 5;
```

66. The DATEDIFF function gives the number of days between DatePicked and DateEaten for each row of the result table.

67.
```
-- For this solution, the brackets used around the OR operator
-- are important. Try removing them and see what happens.
SELECT *, DATEDIFF(day, B.DatePicked, B.DateEaten)
       AS DateDifference
FROM Ape.Banana B, Ape.BananaTree BT
WHERE B.TreeID = BT.TreeID
AND (B.TasteRank = 1 OR TasteRank = 2);
```

68.
```
SELECT *, DATEDIFF(day, B.DatePicked, B.DateEaten)
     AS DateDifference
FROM Ape.Banana B, Ape.BananaTree BT
WHERE B.TreeID = BT.TreeID
ORDER BY TasteRank;
```

69.
```
SELECT *, DATEDIFF(day, B.DatePicked, B.DateEaten)
     AS DateDifference
FROM Ape.Banana B, Ape.BananaTree BT
WHERE B.TreeID = BT.TreeID
ORDER BY TasteRank DESC;
```

70.
```
SELECT TreeID
FROM Ape.BananaTree
ORDER BY Height DESC;
```

71.
```
SELECT TOP(5) TreeID
FROM Ape.BananaTree
ORDER BY Height DESC;
```

72.
```
SELECT TreeID, TasteRank, Comments
FROM Ape.Banana
WHERE TreeID IN (SELECT TOP(5) TreeID
                 FROM Ape.BananaTree
                 ORDER BY Height DESC);
```

73.
```
-- Step 1: Get apes with fav colour yellow
SELECT *
FROM Ape.Friends F, Ape.Colours C
WHERE F.FavColourID = C.ColourID
AND C.ColourName = 'yellow'
```

```
-- Step 2: Add the Ape.EatingFrom to the join
SELECT *
FROM Ape.Friends F, Ape.Colours C, Ape.EatingFrom EF
WHERE F.FavColourID = C.ColourID
AND F.FriendID = EF.FriendID
AND C.ColourName = 'yellow';

-- Step 3: Group by TreeID and count the number
-- of times that each row appears
SELECT COUNT(EF.TreeID) AS EatCount, TreeID
FROM Ape.Friends F, Ape.Colours C, Ape.EatingFrom EF
WHERE F.FavColourID = C.ColourID
AND F.FriendID = EF.FriendID
AND C.ColourName = 'yellow'
GROUP BY EF.TreeID;

-- Step 4: Order by EatCount descending
SELECT COUNT(EF.TreeID) AS EatCount, TreeID
FROM Ape.Friends F, Ape.Colours C, Ape.EatingFrom EF
WHERE F.FavColourID = C.ColourID
AND F.FriendID = EF.FriendID
AND C.ColourName = 'yellow'
GROUP BY EF.TreeID
ORDER BY EatCount DESC;

-- Step 5: Get the top 30% of results
SELECT TOP(30) PERCENT COUNT(EF.TreeID) AS EatCount, TreeID
FROM Ape.Friends F, Ape.Colours C, Ape.EatingFrom EF
WHERE F.FavColourID = C.ColourID
AND F.FriendID = EF.FriendID
AND C.ColourName = 'yellow'
GROUP BY EF.TreeID
ORDER BY EatCount DESC;
```

Bibliography

[1] R. Elmasri, *Fundamentals of database systems*. Pearson Education India, 2008.

[2] E. F. Codd, "A relational model of data for large shared data banks," *Communications of the ACM*, vol. 13, no. 6, pp. 377–387, 1970.

Glossary

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. 35

atomic The property of not being subdivisible. Entries in tables in the relational model are referred to as atomic because they consist of units that can/should not be broken into smaller parts. 12

attribute One column of a table. The columns are essentially a set of labels that define, conceptually, the data to be contained in each tuple. 11

comparison operator A symbol used to compare two things and return either `TRUE`, `FALSE` or `NULL` (unknown). 47

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. 20

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. 12

entry One data value, in one particular row and column of a table. 11, 12

foreign key A column whose entries correspond to entries of the primary key in some (usually different) table in the database. 15

logical operator A symbol that denotes a logical operation. A logical operation returns either `TRUE`, `FALSE` or `NULL`. 23, 47

many-to-many relationship When one record (tuple) in a table can be associated with multiple records (tuples) in another table, and vice versa. 17

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. 38

one-to-many relationship When one record (tuple) in a table can be associated with multiple records (tuples) in another table via a primary and foreign key pair. 13

one-to-one relationship When one record (tuple) in a table can be associated with at most one record (tuple) in another table, via a primary and foreign key pair. 18

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. 15

relation A table. The fundamental unit of organisation in a relational database. 11

search condition A logical statement that evaluates to either True or False, for any given row. 23, 47

tuple One row of a table. Each row is one realisation (i.e., one record) in the table. Every row forms one set of related data, labelled by column (i.e., labelled by attribute). 11