Course Notes on SQL

WITH A FOCUS ON T-SQL

Daniel Vidali Fryer

New Zealand Social Statistics Network

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*

I lied, we won't actually unpack the quote. We don't truly 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 help you understand problems in the way that the Gods understood them when they developed SQL.

Possibly you: "hey listen, 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.

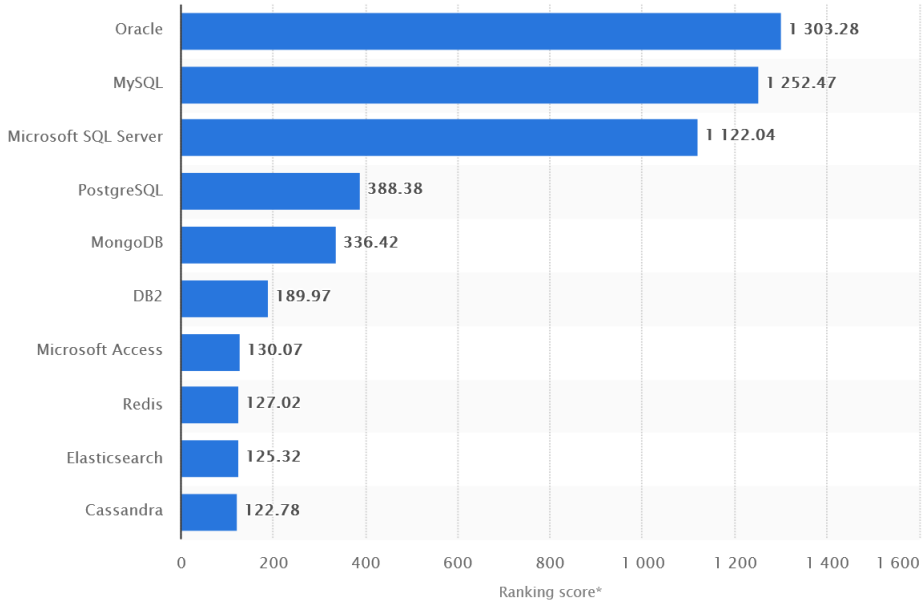


Figure 1.1: DBMS Rankings by popularity. Source: statista.com

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 (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 (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.

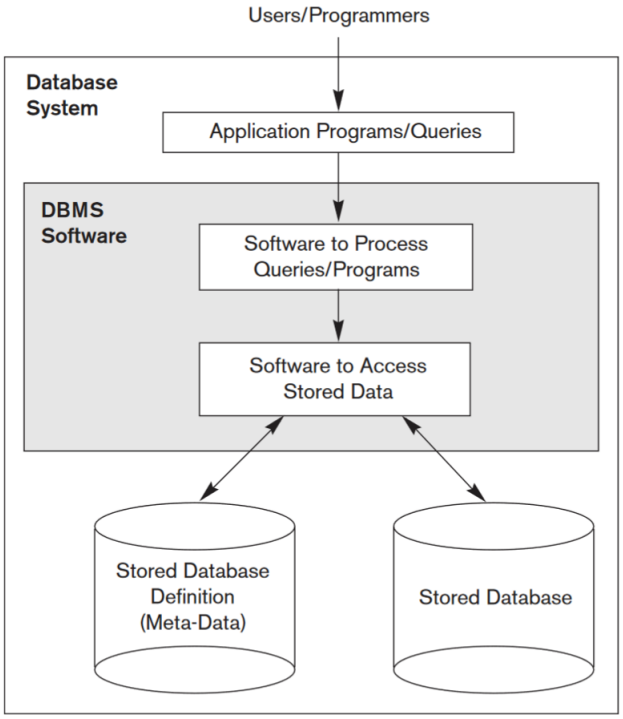


Figure 1.2: A simplified database system. Source: [1].

A database aims to contain data that is 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 update 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 (e.g., insert, update or delete), and manipulate (e.g., select parts of) the data.

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. However, dear reader, I apologise for assuming that you're not here to learn the cold hard math, like:

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

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. These representations, using tables, are very relevant for SQL users. In fact, from now on, we will think of the relational model as a collection of tables, and a collection of operations on tables. To begin, here is a table [drum roll]:

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 (tables)

In the relational model, the data is thought of as belonging to a collection of tables. Each piece of data (e.g., a person's name, a phone number, a date, etc) belongs to some row and column of some table, somewhere. To keep things organised, each table is thought of as a collection of rows, where each row (i.e., **tuple** or record) is one realisation of the abstract entity that the table represents.

For example, a table named Friends may be chosen to help me keep track of all of my friends' favourite colours. In this case, each row of the Friends table represents one record on one friend, i.e., each row belongs to one, and only one, friend. Before I put any data in the table, I need to make a decision on what columns (i.e., **attributes**) the table should have. A reasonable decision might be to include each friend's first name (FirstName), their last name (LastName), and their favourite colour (FavColour). For good practice, I'll add a fourth column, and use it to assign each friend their own unique ID number (FriendID). Before long, we will see why these ID numbers are useful. At the very least, the ID will help me distinguish between any two friends who happen to share the same name. One can never be too prepared. 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**. 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 would

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 prove useful to us later, when we want to describe the important operations that an SQL user 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 viscious 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., "give me a list of all my 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. 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.

CORRECT

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

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

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.

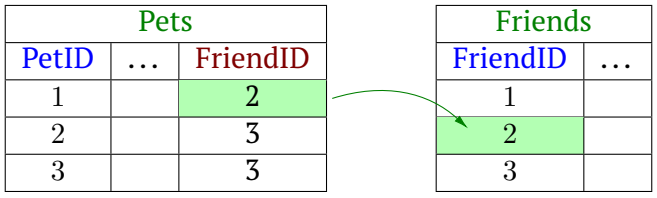


Figure 1.8: Finding the details of the friend who has the pet with PetID of 1.

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.

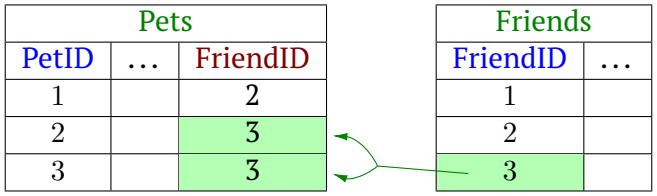


Figure 1.9: Finding the details of all pets who belong to the friend with FriendID of 3.

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 many-to-one relationship between pets and something else (like pet toys), by creating a new table (e.g., PetToys) with an attribute (e.g., PetID), that corresponds to the PetID column of the Pets table (and indicates which pet each toy belongs to). Try this yourself, as an exercise.

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 Frien-

dID 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. A table can only have one primary key, at most. It is good practice to give every table a primary key.

A foreign key is any column (or collection of columns) such that each foreign key entry is equal to one, and only one, primary key entry in some table. Thus, given an entry in the foreign key, we can identify the unique primary key entry that it is equal to. For this reason, we say that the foreign key is "pointing at", or that it *references*, the primary key. A table can have zero or more foreign keys, and each of the foreign keys must reference exactly one primary key.

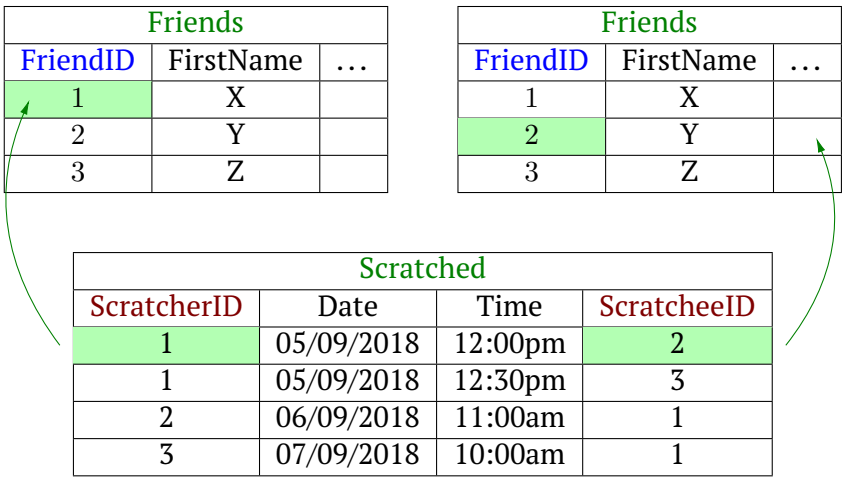
We will get plenty of practice with primary and foreign key pairs as we go. So, don't be too concerned if your head is spinning. The important thing is to go and have another look at the one-to-many relation between Friends and Pets in Section 1.4.1 now, to remind yourself which column plays the role 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.

1.4.3 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 (again, presumably at different times). 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 also references the primary key FriendID from the Friends table. This lets us know whose back was being scratched.



Friends

FriendID	FirstName	...
1	X	
2	Y	
3	Z	

Friends

FriendID	FirstName	...
1	X	
2	Y	
3	Z	

Scratched

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

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

In this example, both foreign keys reference the primary key from the Friends table. However, 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 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.

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

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

WRONG

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

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

Assuming that each friend has only one passport (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. 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. If we want to make absolutely sure that each friend can have only one passport, so that the relationship is strictly one-to-one, then we can place a constraint on the FriendID column, demanding that each entry in this column be unique.

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 whom the corresponding passport belongs to. Hmmmm, 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* 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

The basics of SQL

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 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 rather 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 wonderous fact that SQL is a standardised language, and these standards are maintained by the American National Standards Institute (ANSI), and the International Organisation for Standardisation (ISO).

2.1 Basic queries

Our database is full of tables. A query is designed to go into the database and chop tables up, join them together, 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.

2.1.1 SELECT and FROM

The FROM clause lets you specify a table that you want to chop up. For this reason, you're going to use FROM in almost every query you 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*. In other words, SELECT lets you *select* certain columns of the table. For example:

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

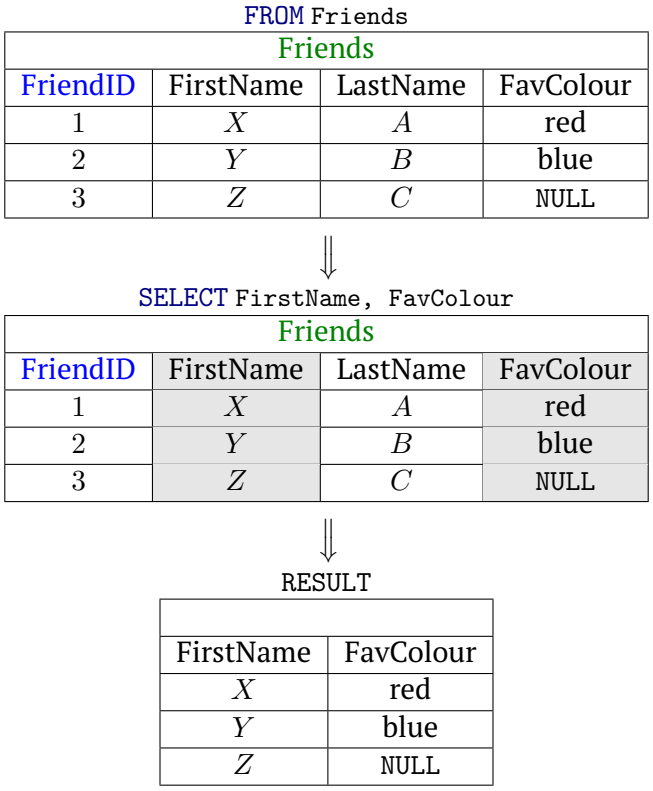


Figure 2.1: The SELECT and FROM clauses.

In SQL, the sytax attempts 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 off 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;
```

2.1.2 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 2.4.3, and more on search conditions in Section 2.4.4.

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

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

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

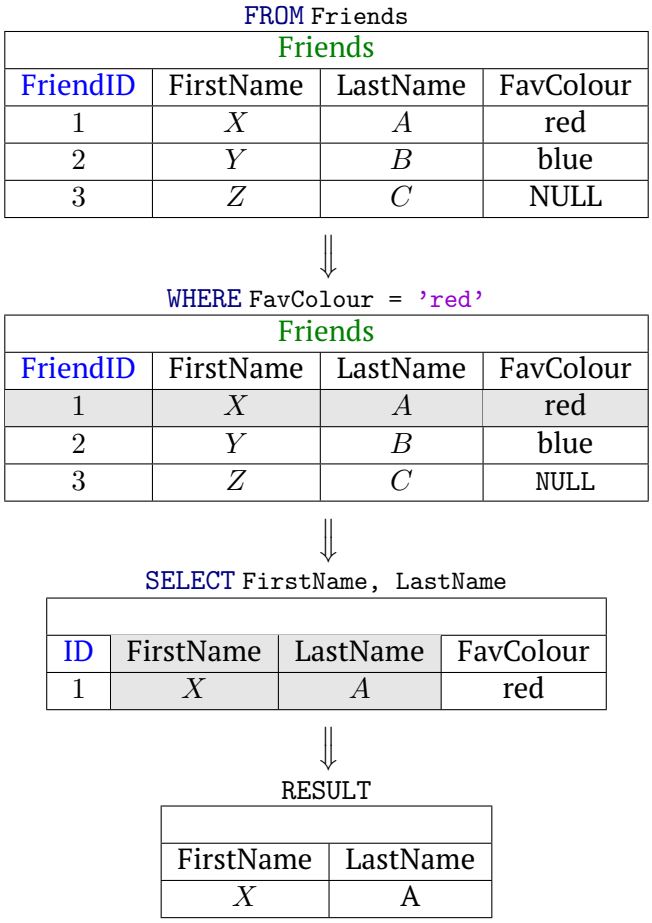


Figure 2.2: The `WHERE` clause.

2.1.3 `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;
```

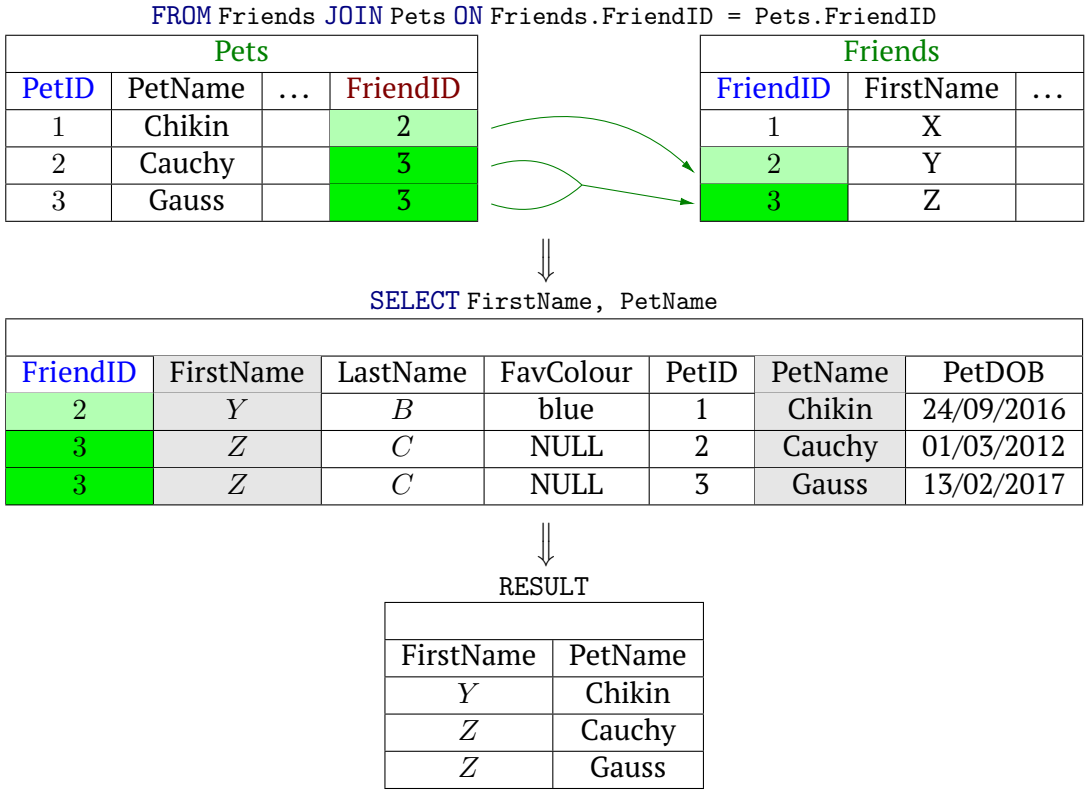


Figure 2.3: The `JOIN` clause

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

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

2.2 Aggregating queries

We have progressed to the section on aggregating queries!

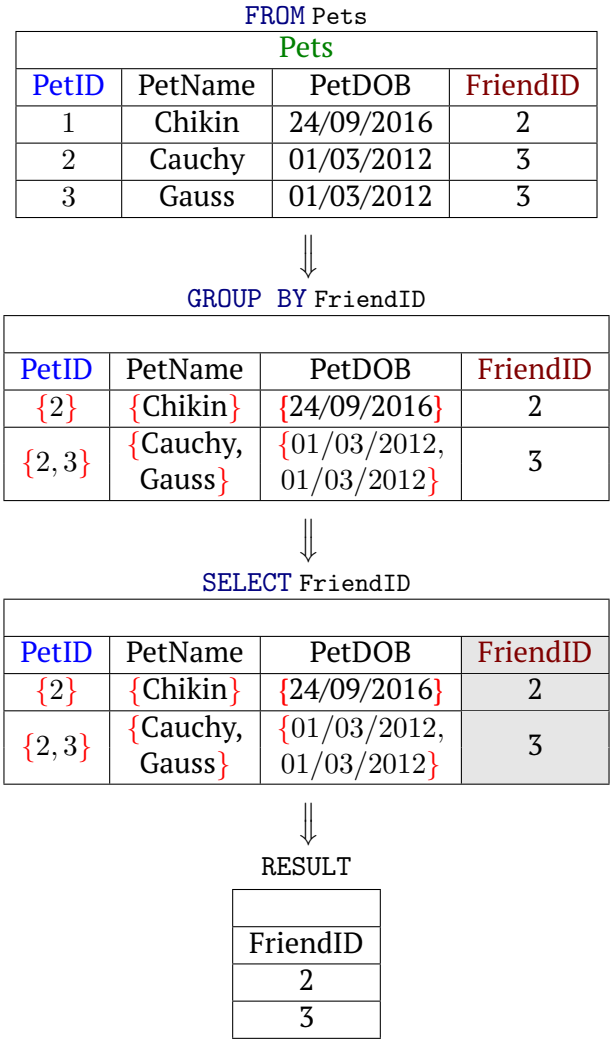
An aggregating query is the SQL way to divide the rows of a table into groups and somehow return *a single value* for each group. This is especially useful when we want to avoid extracting very large datasets that we can instead summarise. The `GROUP BY` clause determines how the groups are decided, then the `HAVING` clause decides which (if any) groups to discard, and finally an aggregating function can be used to get basic summary statistics (like the average or the standard deviation) within the groups.

2.2.1 GROUP BY

The `GROUP BY` clause does exactly what it says on the tin. It groups the rows of a table according to 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:



After grouping, 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. SQL knew that we chose 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.

We were able to execute `SELECT FriendID` after that with no issues. However, if we had of chosen to `SELECT` any of the other columns, then SQL would have produced an error. This is because SQL cannot 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 can't be sure that the lists contain only one value in them. In summary, when we use `GROUP BY`, we can't select any columns that will end up with lists in them.

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

It just happened that the two pets that have FriendID equal to 3, also shared the same birthday, so the groups were unchanged. Since we now have no lists in the PetDOB column, we could `SELECT` that column in our query as well. Keep in mind that we can't execute `GROUP BY` on its own without a `SELECT` statement; the above is just an illustration of 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`, 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.

2.2.2 Aggregation functions

In the previous section, when applying `GROUP BY`, we faced some pesky lists, indicated with red curly brackets, whose columns could not be selected with `SELECT`. We learned that 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, selected completely randomly:

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

Figure 2.4: 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;
```

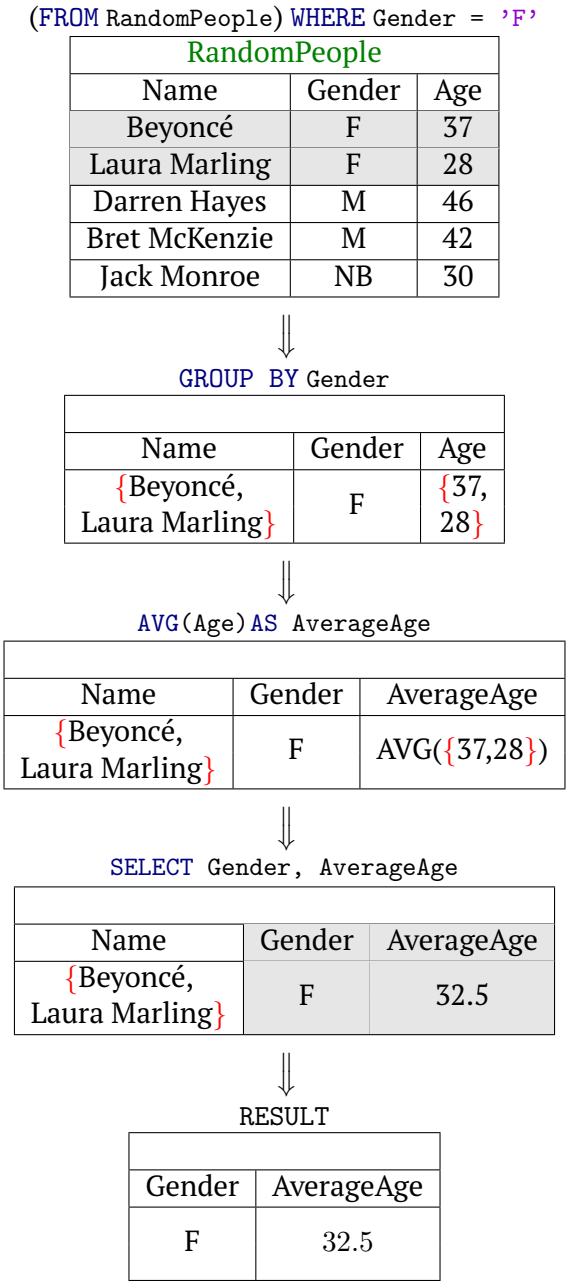


Figure 2.5: Notice that `WHERE` is executed before `GROUP BY`

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`.

We call `AVG` an **aggregation function**. There are a host of other aggregation functions available in most implementations of SQL. In Section 2.4, 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 2.1: Some of the aggregation functions in Microsoft's Transact-SQL

2.2.3 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.

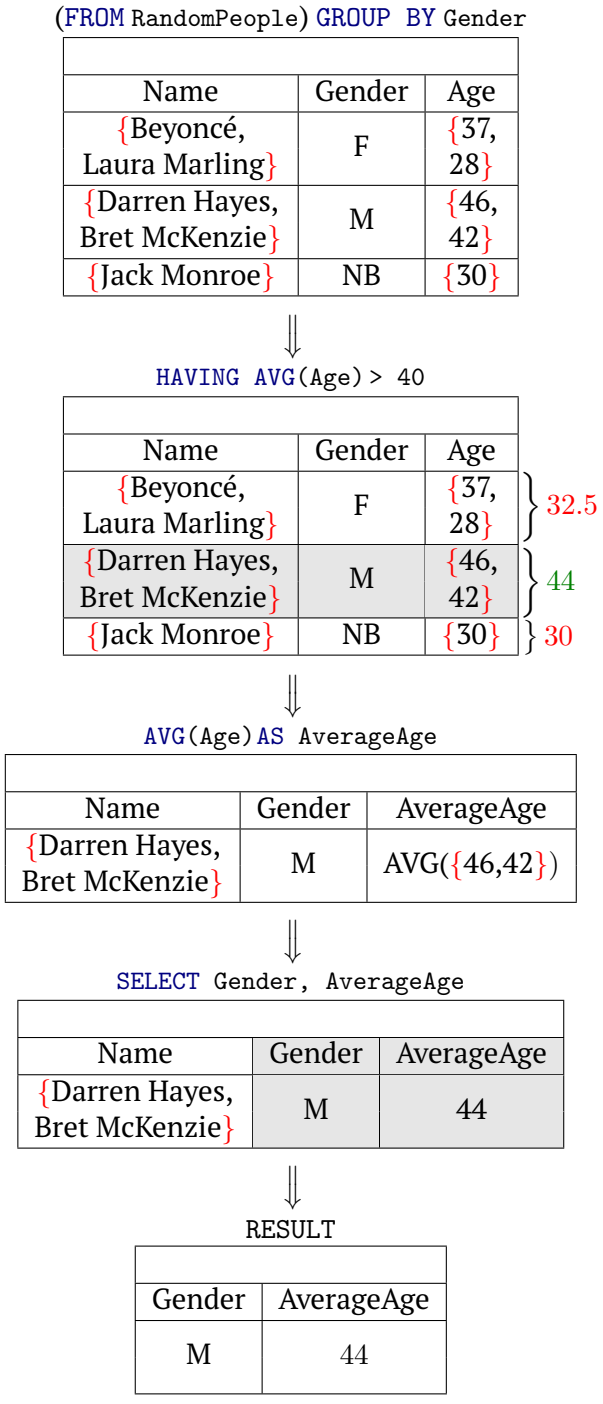


Figure 2.6: The `HAVING` clause

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 groups 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 2.1). 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 2.4.4.

2.3 Basic nested queries

We have come a long way now, and can rest with the knowledge that we've covered the basics of SQL. However, hot shot, ask yourself, how can I extract a table containing all of the people's names from RandomPeople (Figure 2.4), who belong to a gender with average age greater than 40?

I can see in the execution diagram (Figure 2.6) that these people's names are Darren Hayes and Bret McKenzie, but I can't `SELECT` the Name column because it contains the pesky red lists in it after we `GROUP BY Gender`. It won't help me to group by Name and Gender using `GROUP BY Gender, Name`, because if I do then the groups will be separated according to Gender *and* Name, so that when I subsequently run `HAVING AVG(Age) > 40`, the averages won't be calculated within whole genders, but instead only 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 are you talking about, buddy?" And you think carefully and say, "the names whose gender is present in the `RESULT` table of my previous query (Figure 2.6), please."

So, to wrap our heads around this, let's for a moment use the word `RESULT` to denote the previous query. Then 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 2.4.3.

2.4 Reading the docs

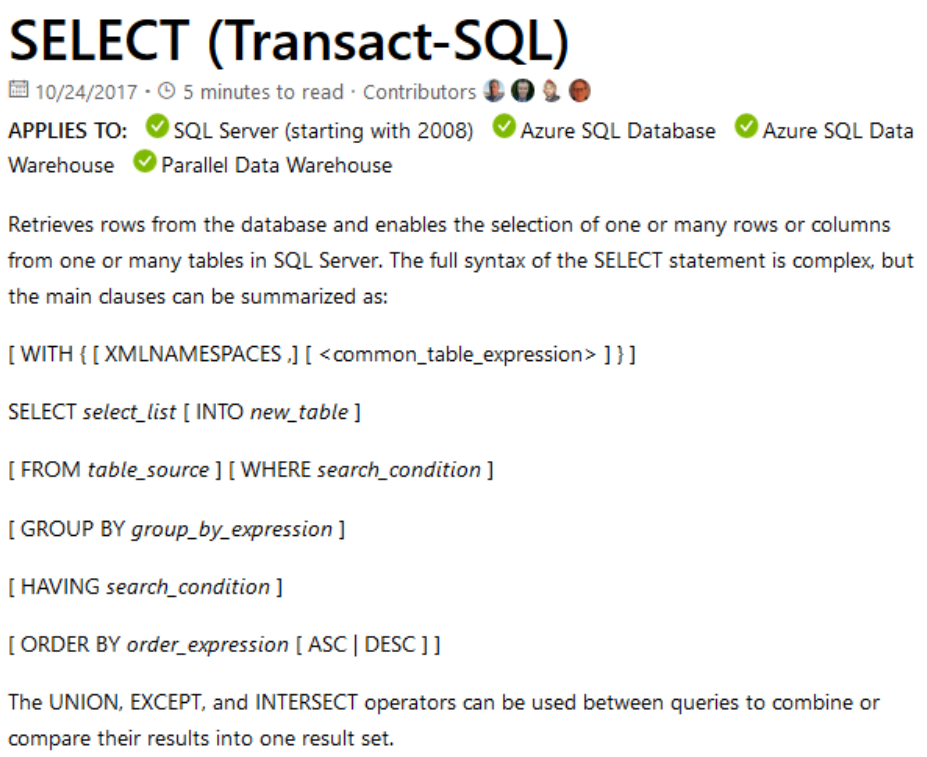
There are two secrets to being an effective programmer at any skill level. The first is that you will borrow a lot of code from other people. Often, somebody else has already done what you want to do, or at least something very similar. Online forums like StackExchange and specialty discussion boards 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. I could include a bunch of links here but I would just be typing terms like "beginner SQL tutorial" into a search engine and giving you my results. We'll leave that as an exercise to the reader. If you must take my recommendation: Code School do a 10 day free trial on their very entertaining Try SQL online course. Make the best of it!

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.

2.4.1 How to read the docs

The first thing you'll need is a reference sheet of the T-SQL Syntax Conventions. If you can't click the link here, then have a look at the screenshots in Appendix A.1. The reference sheet allows you to interpret what the damn hell 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 the link here, or in the footnote[1]. In the navigation menu to the left of that page, click on `SELECT`. This takes you to the `SELECT` documentation page. 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]]

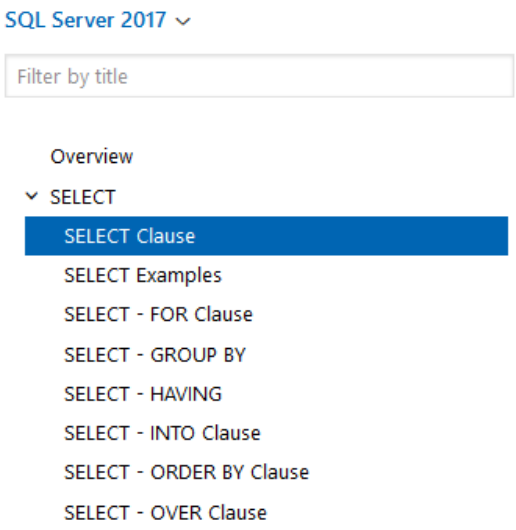
The UNION, EXCEPT, and INTERSECT operators can be used between queries to combine or compare their results into one result set.

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.

[1] https://docs.microsoft.com/en-us/sql/t-sql/queries/queries?view=sql-server-2017

[ ] (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! I'll tell you a secret: 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. 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. It's like:

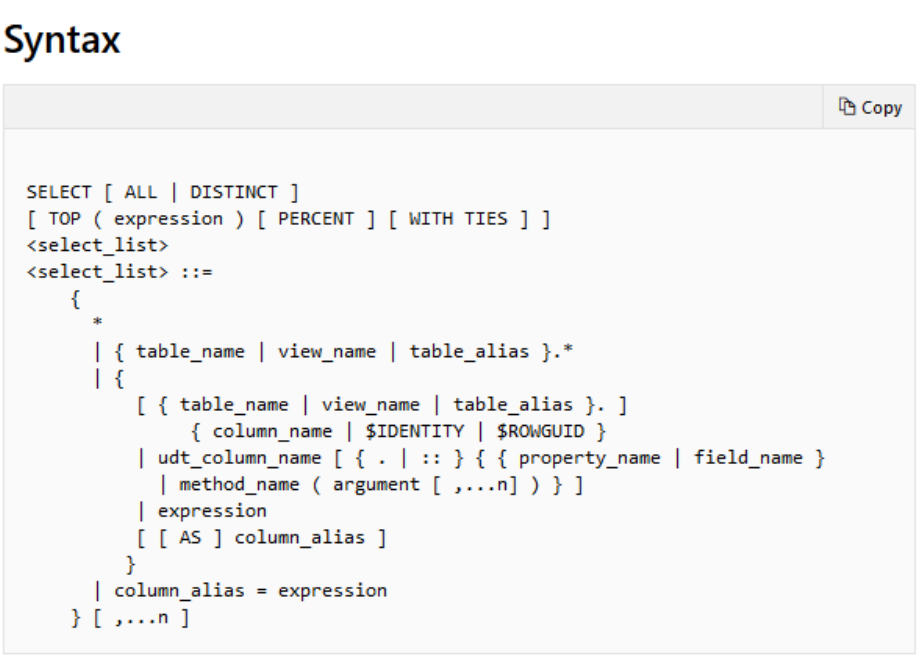


On that page, under the heading **Arguments**, you can find ⟨*`select_list`*⟩ and an explanation for what it does:

> < 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:



Now, this is a mess. So, in your mind's eye 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:

$$\langle greeting \rangle ::= \text{Hello. [Do you \{love | hate\} reading the docs?]}$$

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. You have come a long way, but the journey ahead is long and perilous-full. Sadly, I cannot go with you, for I have limited resources and this is outside the scope of these notes. I encourage you to read the docs frequently and with patience. My parting advice: if something that you read in the docs doesn't make sense to you, then do a quick online search for help using a search engine.

2.4.2 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 2.2.2 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*. 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;
```

2.4.3 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 2.1.2, 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 2.4.4).

A table of comparison operators and a table of logical operators can both be found under the language elements section of the T-SQL docs. Both ta-

bles are included in these notes in Appendix A.2. We will have more practice with logical and comparison operators during the exercises and in Section 2.4.4.

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` and/or the T-SQL built-in function `ISNULL`.

2.4.4 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 2.3:

```
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.

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`.

Chapter 3

Exercises

3.1 A note on style

SQL is not sensitive to empty spaces, upper-case/lower-case letters, or new lines. So, you can write any query on one long horrifying line in lower-case, or you can use no indentation, if you like! However, this is a bit of a curse for people who need to read SQL code written by beginner programmers (including the beginner programmers themselves). Be thoughtful about how you structure and present your code. If you write carelessly, then you might vaguely understand it today, but chances are you'll go off and think about something else for a while and come back to your SQL script tomorrow, only to beat yourself to death with a keyboard for not using upper-case letters for keywords and including some comments, line breaks, and indentations to make your own code more readable.

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 (and it makes it clear where one query ends a new one begins).

3.2 Exercises

Open SQL Server Management Studio (SSMS) and connect to the database server as directed by your course instructor. You may have difficulty completing some of the below exercises. Check your solutions with the instructor and ask for help when needed. Discuss solutions with your neighbours!

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

Introductory exercises

1. Search online for your own beginner SQL tutorial or use this excellent Learn SQL tutorial from codeschool. Start the tutorial and see if you like it. Follow the tutorial for around 15 minutes and come back to it later if you like it!

2. Search online for a simple syntax guide. I like the one from dofactory. 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.

3. Use the Object Explorer pane in SSMS to begin investigating PlayPen and the other databases. If you click on the PlayPen database and press F7 it will open an Object Explorer Details window which is easier to browse in. 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. Right click on the Notes.Friends table in the Object Explorer pane and click "Design." In the new window that opens, you can see the name of each column, the *data type*, and 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. Don't spend too long on this! Come back to it later.

5. Right click on the Notes.Friends table in the object explorer pane and click "Select Top 1000 Rows." 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.

6. Read the T-SQL documentation for `TOP`. 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.

Using the `SELECT` and `WHERE` clauses

7. Right-click on the PlayPen database icon in the Object Explorer 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.

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

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

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

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

11. Write a query that selects all columns of the Notes.Scratched table and only returns records in which the ScratchTime was before 12 PM. You will need to use the < comparison operator, and enter times in the format `'HH:MM[AM][PM]'`. So, if you wanted all rows with ScratchTime before 3:45 PM then you would enter

```
WHERE ScratchTime < '03:35PM'
```

    For more information on time and date formats, refer to the T-SQL documentation on time and date.

12. 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 enter dates in the format `'YYYYMMDD'`. So, if you wanted to use the 12th of February 2016 then you would write `'20160212'`.

13. 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`.

14. The solution to Question 13 can be achieved in a more readable way using the `BETWEEN` clause. Read about it on the T-SQL docs then try to use it to replace your solution to Question 13.

15. 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.

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

17. 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 41 of these notes, or in the T-SQL docs).

18. Edit your query from Question 17 (still using the `LIKE` logical operator and the % wild card) so that it displays only rows where the letters 'ar' appear together *anywhere* in the PersonName.

19. Read about the other wild cards (`_`, `[ ]` and `[^]`) in the T-SQL docs for `LIKE`. Run the following query and explain what it does.

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

20. 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 doesn't it work?

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

    If you're confused, then read "**A note on** `NULL`" on Page 40. Then, read the documentation for the SQL keyword `IS NULL` and the T-SQL built-in function `ISNULL`. Try to use one of those to solve the problem.

21. Edit your solution from Question 20 so that it only returns rows where FavColour is *not* `NULL`. There are various ways to do this, you could use a variant of `IS NULL`, or you could use the "not equal" comparison operator (<>) with `ISNULL`, or you could use the `NOT` keyword.

Joining tables implicitly or with the `JOIN` clause

22. In the Object Explorer pane, 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).

23. 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?

24. Execute the following query and explain what it does. Is the result any different to the result from Question 23? Why or why not?

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

25. 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 23 is for an implicit join and the syntax in Question 24 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?

26. 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).

27. Edit the query from Question 26 so that it only returns entries where the favourite colour is blue. Use the logical operator `AND`.

28. 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 16 of these notes. Use the Object Explorer in SSMS 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 the link or view the screenshot in Appendix A.1), 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 what the syntax should be for joining the 3 tables. Ask for help if you're stuck.

29. There were duplicate columns present in the result for Question 28. To produce a cleaner result after a join, select only the columns that you need (instead of using `*`). Edit your solution to Question 28 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`.

30. 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.

Using the `GROUP BY` clause

31. 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;
```

32. 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 2.2.1, and in particular Page 28.

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

33. In the query below, we've edited Question 32 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.

34. 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.

35. 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.

Aggregation functions and the `HAVING` clause

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

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

37. Retrieve the sample standard deviation of the widths of banana trees in Ape.BananaTree. You may wish to consult the T-SQL aggregate function docs.

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

39. 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 32.

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

40. If we add `GROUP BY TreeID` to the query from Question 39, 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;
```

41. 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.

42. Use the aggregation function `COUNT` 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.

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

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

45. 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;
```

46. 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;
```

47. 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.

48. 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.

49. Edit the query from Question 48 so that it only returns results from the tree with TreeID = 5.

Putting it all together and using nested queries

50. The solution to Question 42 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.

51. Extend your solution to Question 50 so that it excludes all colours that appear less than twice.

52. Extend your solution to question 51 so that it excludes all colours whose name starts with 'b'.

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

54. 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;
```

55. Change the query from Question 54 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.

56. 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 54 so that it includes every TasteRank, but orders the results according to TasteRank.

57. Referring to the documentation for `ORDER BY` again, change your solution to Question 56 so that it lists the TasteRank in descending rather than ascending order (Hint: Use the keyword `DESC`).

58. 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.

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

60. 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 2.3).

61. 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 4

Using SQL with R

There are essentially 3 ways to pass data back and forth from SQL to R:

(a) Import SQL results into R manually.

(b) Run SQL code from R.

(c) Run R code from SQL.

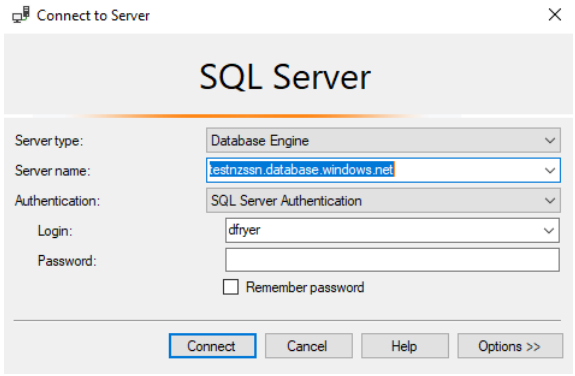
Option (c) can be tricky to set up, and it only works if your DBMS supports the feature. So, we will just look at options (a) and (b) here. For more on option (c) in SQL Server, read the Microsoft T-SQL docs.

4.1 Import SQL results into R manually

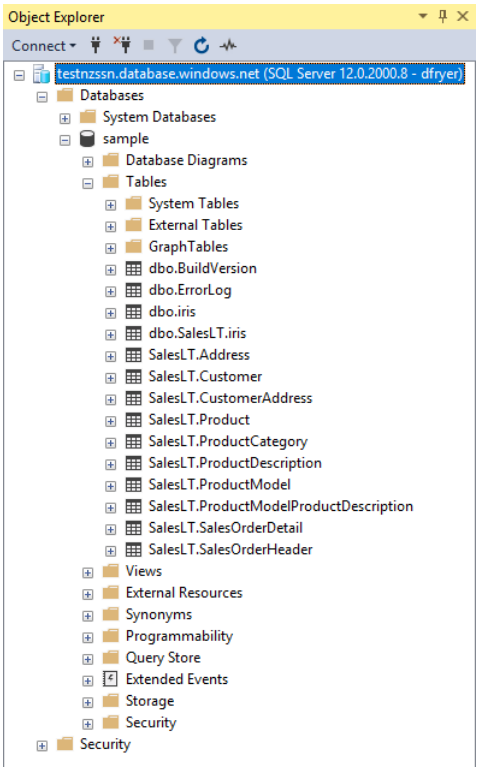
This approach requires the least amount of initial set-up and is fairly uncomplicated. The downside is that you will need to work in two different editors at once. Typically, these editors will be R Studio and Microsoft SQL Server Management Studio (SSMS). The idea is that you execute a query in SSMS, then save the results in a suitable file format, then open R and import the saved data. This means that each time you want to refresh the data you will need to open SSMS and run the query again. However, there are some advantages. The biggest advantage is that you will be using an editor that is built for SQL, so all settings and features will be geared towards writing and executing SQL queries.

4.1.1 Connecting to server

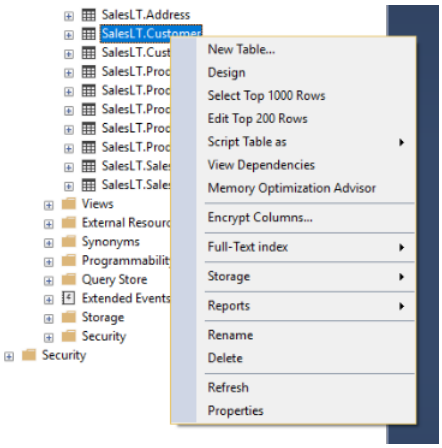
When you open SQL Server Management Studio (SSMS), a Connect to Server box will appear:



Type the server name (given to you by a database administrator or similar person), choose "Database Engine" for the server type, and choose "SQL Server Authentication". Enter your login details, and click "Connect." You should now see some details in the SSMS Object Explorer pane:



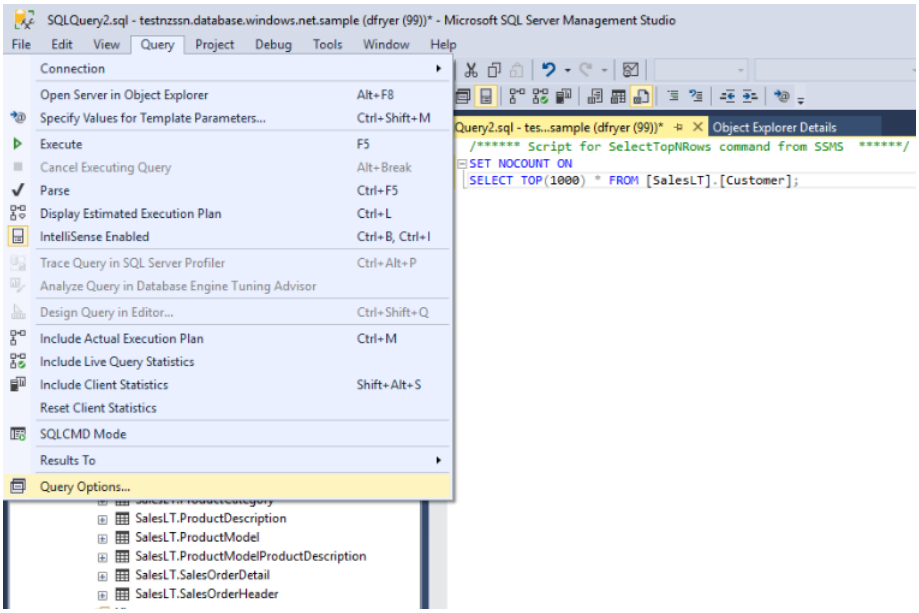
You can use the object explorer to get an idea of what is inside each database that you have access to. In the image above, there is one database called "sample," and we have expanded the tables folder. Notice that there are a lot of tables starting with the word `SalesLT`. They all share the same prefix because they belong to the same **schema**. A schema is a collection of tables in a database that are grouped together conceptually to help keep the database organised. Right-clicking on any of the tables will open a menu. Below, we have right-clicked the `SalesLT.Customer` table.



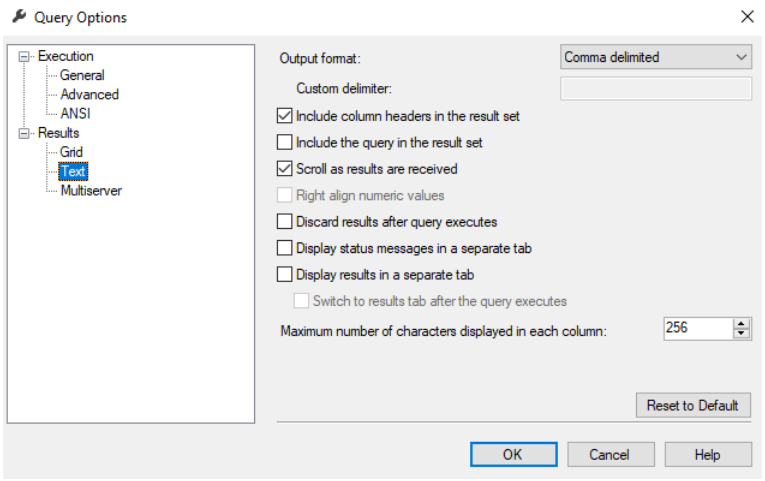
In this menu, you can click to "Select Top 1000 Rows". This will open a query editor pane and automatically generate an SQL query that returns the top 1000 rows from the table. Congratulations if this is the first SQL query that you have ever executed! The output of the query should appear as a spreadsheet table in the results tab below the query editor.

4.1.2 Getting results from SQL to R

It is easy to copy and paste data from the results tab into excel. Just press CTRL + A to highlight all the query results, then CTRL + C to copy them. However, if the results table is very large, or if you are planning to generate lots different results tables, then it is better to export the results as a file directly.

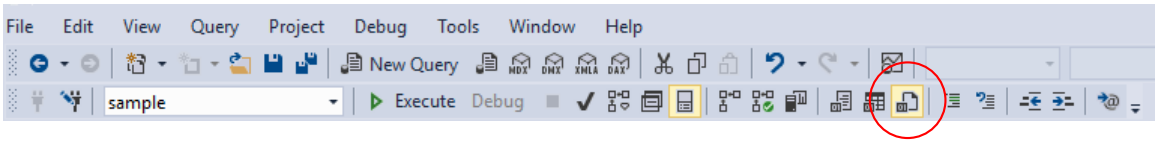


To export the results as a file, first click on the "Query" menu, and select "Query Options", as in the above image. The Query Options menu opens:



Under "Results" ⟶ "Text," choose whatever options you prefer for your output file. You can choose a custom delimiter if your data includes commas, though this is rare. Otherwise, comma delimited should be fine. Click "OK."

Now, in the SQL Editor Toolbar there is a tiny little icon you can press to change the output mode to "Results to File" (see the icon within the red circle in the image below). Alternatively, just press CTRL + SHIFT + F.



One thing to keep in mind is that SQL Server may add an annoying line of text to the output file specifying the number of rows selected. This can confuse whatever program we use to import the data later, so you should remove it by including `SET NOCOUNT ON` above your SQL queries before execution. For example, the following query will extract the first 1000 rows of the SalesLT.Customer table, using the built-in T-SQL function `TOP`.

```
SET NOCOUNT ON
SELECT TOP(1000) *
FROM SalesLT.Customer;
```

When the query is executed, you will be prompted to choose a filename, and the file will be saved as a `.rpt` file. Once this is done, open R Studio and execute the following, replacing "my_file_name.rpt" with the name you chose (and the path if it is not in the current working directory).

```
filename <- "my_file_name.rpt"
top1000 <- read.csv(filename, header = TRUE)
```

In the above, we have assumed that the file is comma delimited, so we have used `read.csv`, which is equivalent to using `read.table` with the following parameter choices:

```
filename <- "test.rpt"
top1000 <- read.table(filename, sep = ",", header = TRUE,
                      fill = TRUE, comment.char = "")
```

If your file is not comma delimited, you can play with the options in `read.table` until you get the result that you want. For example, you can change the `sep` parameter from a comma to something suitable. To see details on all of the `read.table` parameters, execute `?read.table` in R.

4.2 Run SQL code from R

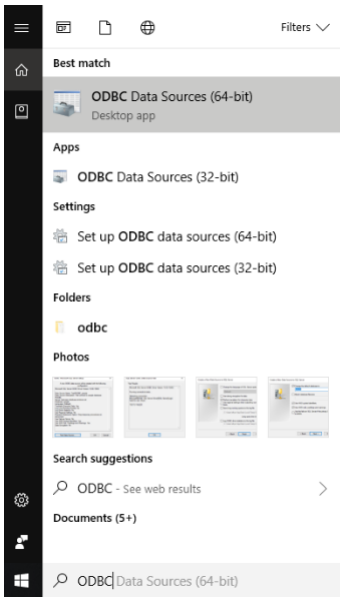
As of early 2018, there is a fresh new R package, called `odbc`[1], that is designed to help you connect to SQL databases. This lets you write SQL queries within R, send the queries to an SQL server, and then rerieve the result as a `data.frame`. There are other similar R packages (e.g., the packages `RSQLServer`, `RODBC` and `RODBCDBI`), but `odbc` is generally much more efficient than these. For a comparison using quick benchmarks, go here. We cover the basics in these notes, and there are some useful R Studio guides available at db.rstudio.com.

4.2.1 Setting up an ODBC Data Source in Windows 10

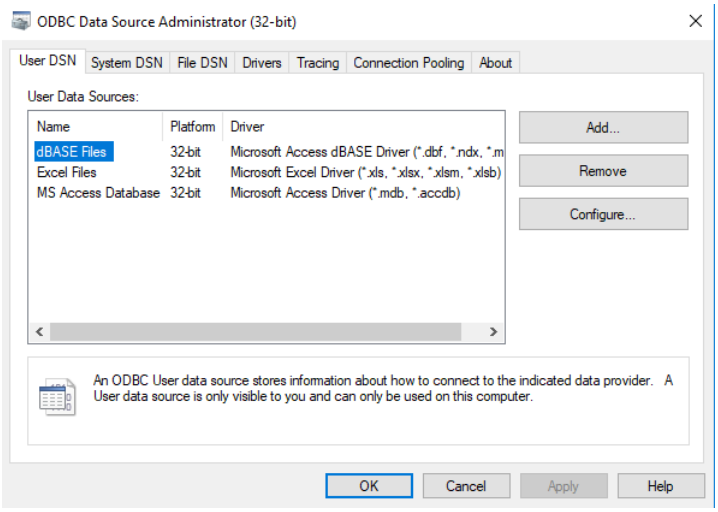
If you are planning to use the same computer to connect to a particular database over and over again, then it might be worth setting up an ODBC Data Source, where ODBC stands for Open Database Connectivity. If you have some configuration details provided by a database administrator, and you want to jump straight into connecting to the database, then skip this section. We'll give a quick walkthrough on how to create a Data Source Name (DSN) in Windows 10. If you're doing this on a work computer then you may need assistance from a system administrator or IT support.

Assuming you already have Microsoft SQL Server Management Studio installed, click on the Cortana search bar and type "ODBC Data Sources". You should see this:

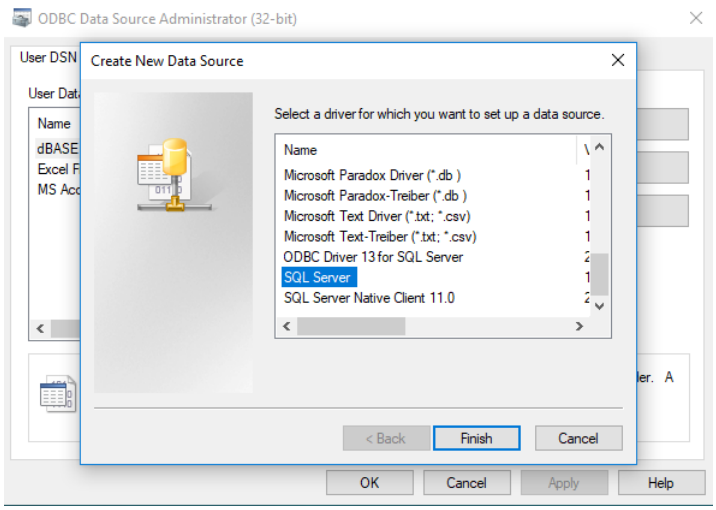
[1]The R package `odbc` (lower case), is not to be confused with the application programming interface ODBC (upper case) that `odbc` relies on.



Click on ODBC Data Sources (64-bit), and a dialogue box with some Data Sources will appear:



Click “Add” and the Create New Data Source dialogue box opens:



Select “SQL Server” and click “Finish”. The Create New Data Source to SQL Server dialogue box appears:

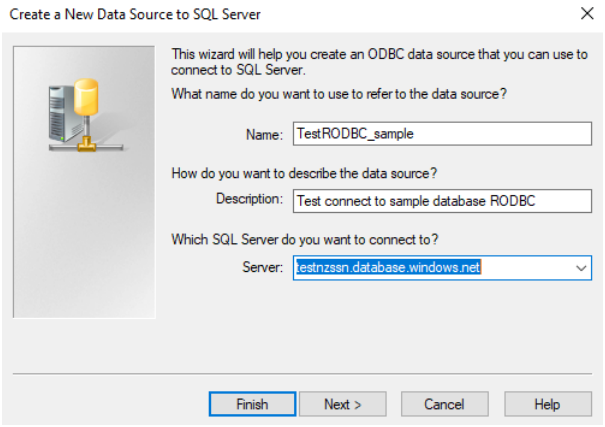
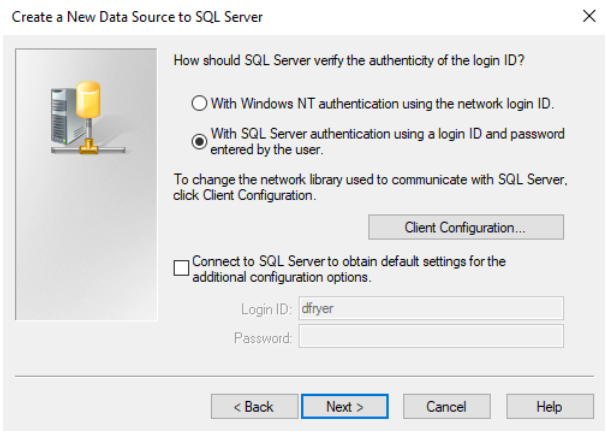
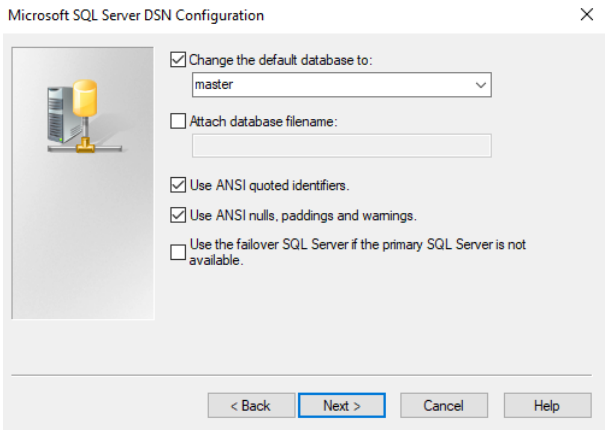


Figure 4.1: Choosing a Data Source Name

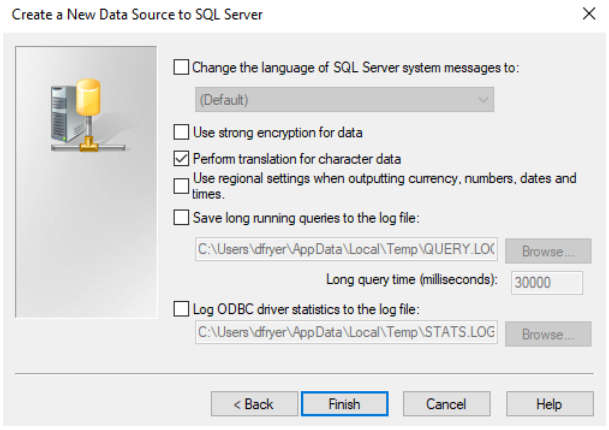
Choose any name (but write it down for later since this name is the DSN), write any short Description, and specify the server address. The server address is much like a website address, it helps us locate and access the server. It should be given to you by someone who administers the database you are trying to connect to. Once this information is inserted, click “Next”.



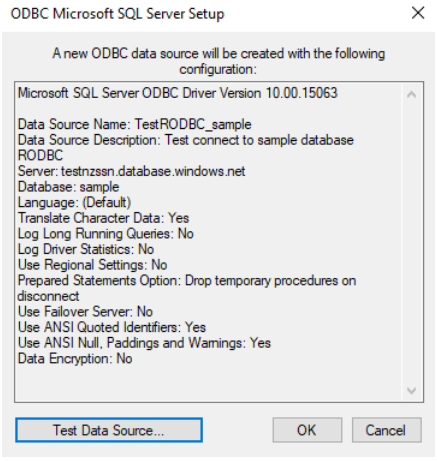
Under "How should SQL Server verify the authority of the login ID?" choose "With SQL Server authentication ..." You don't need to connect to SQL Server to obtain default settings. Click "Next."



Make sure you choose a default database. If you're not sure, try 'master.' Come back to this step as a likely culprit if you have problems later. Click "Next" again.



Leave the defaults as they are. Click “Finish”.



Do not test the data source (the test will fail), click “OK”.

4.2.2 Using `odbc` to connect to SQL from R

It's time to connect to SQL Server in R. Frist, download and install the latest version of R Studio from rstudio.com. At the time of writing, the latest stable release was version 1.1.463, but anything later than that should be fine. Now choose one of Method 1 or Method 2 below.

- **Method 1:** If you skipped the above guide on setting up a DSN, or nobody has given you a DSN for the computer you are using, then you can connect to an SQL server provided that you know some configuration details. Just fill in the correct details in the code below.

```
#example: driver <- 'ODBC Driver 13 for SQL Server'
driver <- "insert the driver name here"
#example: server <- 'testnzssn.database.windows.net'
server <- "insert the server name here"
#example: database <- 'sample'
database <- "insert the database name here"
#your user ID should be given to you
user <- "insert your user ID here"
#replace the number 1433 below with the correct port number
port <- 1433
con <- DBI::dbConnect(odbc::odbc(),
                Driver = driver,
                Server = server,
                Database = database,
                UID = user,
                PWD = rstudioapi::askForPassword("User
                    Password"),
                Port = port)
```

- **Method 2:** On the other hand, if you've set up an ODBC Data Source (or been given the DSN by someone else who set it up for you), open a new script in R Studio and run the following code, with correct choices for "your user name," "your data source name" (which we chose in Figure 4.1), and "the database to connect to". Do not edit the words "User Password".

```
# Connect to the database
uid <- "your user name"
DSN <- "your data source name"
database <- "the database to connect to"
con <- DBI::dbConnect(drv = odbc::odbc(), DSN = DSN, uid = uid,
                    pwd = rstudioapi::askForPassword("User
                        Password"),
                    database = database)
```

Now, regardless of whether you chose Method 1 or Method 2 above, you just created an object that manages the database connection, and stored that object in the variable named `con`. We can now use `con` for a few things. The most important of which, for us, is to send queries to SQL Server and retrieve the results. Before we do that, have a look at the R Studio connections pane. It will look something like this:

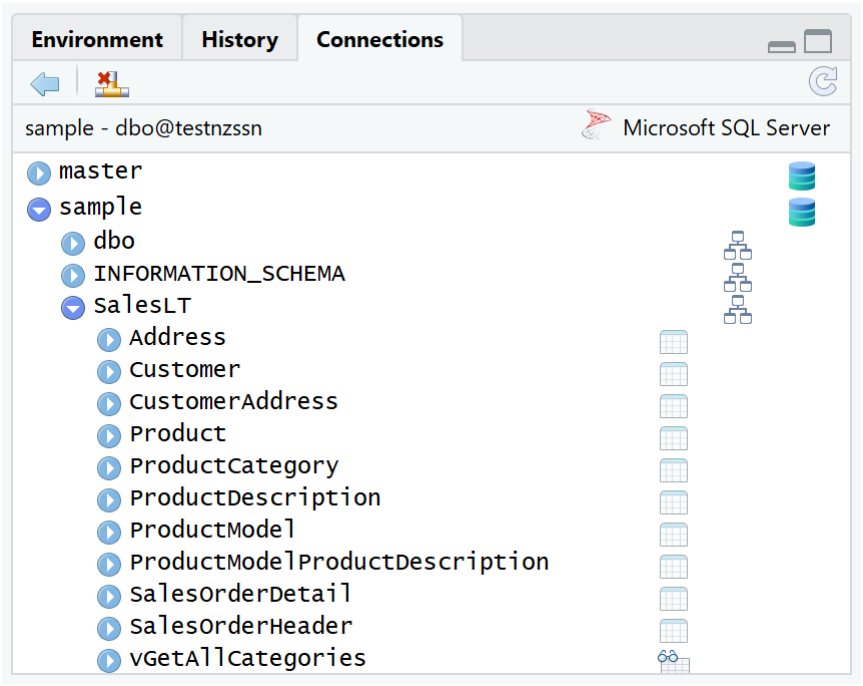
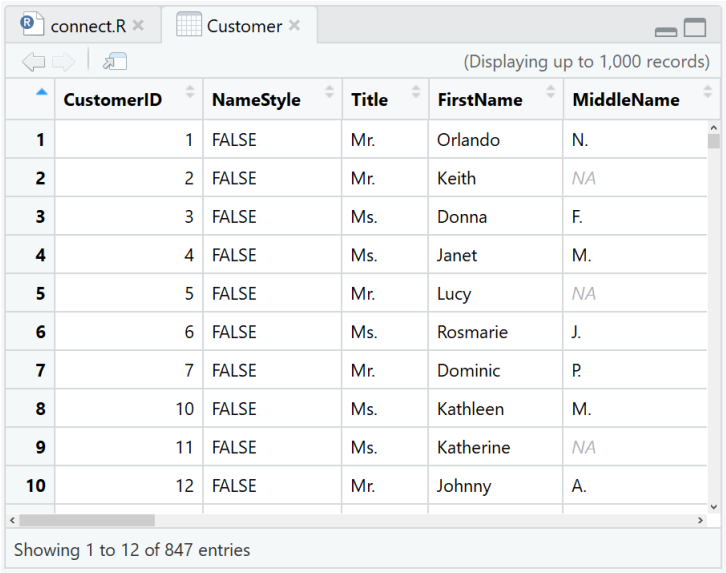


Figure 4.2: The R Studio connections pane, after a database is connected.

The connections pane has one or more database names displayed in it, which can be clicked to open lists of the schemas in each database, which can each be clicked to open lists of the tables in each schema. To the right of the table names are little white table icons that can be clicked to display the first 1000 rows of the data in each table. The display is scrollable and looks like this in R Studio:



connect.R | Customer

(Displaying up to 1,000 records)

	CustomerID	NameStyle	Title	FirstName	MiddleName
1	1	FALSE	Mr.	Orlando	N.
2	2	FALSE	Mr.	Keith	NA
3	3	FALSE	Ms.	Donna	F.
4	4	FALSE	Ms.	Janet	M.
5	5	FALSE	Mr.	Lucy	NA
6	6	FALSE	Ms.	Rosmarie	J.
7	7	FALSE	Mr.	Dominic	P.
8	10	FALSE	Ms.	Kathleen	M.
9	11	FALSE	Ms.	Katherine	NA
10	12	FALSE	Mr.	Johnny	A.

Showing 1 to 12 of 847 entries

Figure 4.3: Displaying the first 1000 lines of a database table in R Studio

If we want to list just the tables that have a certain string of letters in the name, we can use a wildcard (% symbol) like this:

```
# List all tables beginning with ‘Customer‘
DBI::dbListTables(con, table_name = "Customer%")

# List all tables with with the word ‘cat‘ anywhere in the name
DBI::dbListTables(con, table_name = "%cat%")

# List all tables ending with ‘Address‘
DBI::dbListTables(con, table_name = "%Address")
```

We can execute SQL scripts and get the result as a data.frame with the function `DBI::dbGetQuery`. You write the SQL code as a string, and then pass the string to `DBI::dbGetQuery`. For example:

```
# Write the query as a string
myquery <-  "SELECT *
             FROM SalesLT.Customer;"

# Execute the query and return the result as a data.frame
customers <- DBI::dbGetQuery(con, statement = myquery)
```

The SQL code is executed on the database server. So, it is quite efficient and it doesn't consume your R resources. Now that we have a data.frame, we can get the names of all the columns, or we can view the first few rows:

```
# Get the names of all the columns of the Customers table
names(customers)

# View the first few rows of the Customers table
head(customers)
```

Be careful loading entire tables into R as we have done above. If the table is very large, then it may be too big to hold in memory, so the execution will fail. It is better to do the work of summarising or sampling data from the table within the SQL code itself, so that it will be executed on the database server. The next query will use the T-SQL function `TOP` to retrieve only the top 10 rows:

```
# Write the query as a string
myquery <-  "SELECT TOP(10) *
             FROM SalesLT.Customer;"

# Execute the query and return the result as a data.frame
top10_customers <- DBI::dbGetQuery(con, statement = myquery)
```

While `dbGetQuery` sends the query and then retrieves the result in one go, it is possible to do each individually instead, as we do below. This can be useful if the result is very large and you want to fetch it in pieces.

```
myquery <-  "SELECT *
             FROM SalesLT.Customer;"

# Send the query
results <- DBI::dbSendQuery(con, statement = myquery)

# results just holds the connection
results

# Fetch the first next 10 rows
customers <- DBI::dbFetch(results, n = 10)

# Fetch the next 10 rows and bind them to the previous (repeat)
customers <- rbind(customers, DBI::dbFetch(results, n = 10))
```

4.2.3 A note on `dplyr`

Though outside the scope of these notes, it is worth a mention that the R packages `dplyr` and `dbplyr` can be used together to generate SQL queries using `dplyr` syntax. Hadley Wickham's R package `dplyr` provides a simple and powerful grammar of data manipulation. Along with the rest of the `tidyverse` packages, it has an almost cult-like following in the R community. Cult or no cult, in my opinion learning `dplyr` syntax is a great way to complement your SQL knowledge, and vice versa!

4.2.4 A note on other statistical software

Connecting other software to SQL Server via an ODBC driver follows a similar procedure to what we have covered above for R. Here is a list of resources for learning more:

- **SAS:** The following is taken from the VHIN website.

  > Within SAS, you can run SQL code by writing `proc sql;` and then the SQL code, followed by `quit;`. However, this executes that SQL code within SAS, requiring data to be moved from SQL servers to the SAS server. The most efficient way to use SQL within SAS is to use the "passthrough" procedure. This passes the code to the SQL server to be executed there, and exports the results back to SAS. It is therefore more efficient than executing the code on the SAS

server. Here is an example that will access the table called moh_clean.pho_enrolment, and create a table called MyTable in the SAS working directory:

```
proc sql;

connect to odbc (dsn="idi_clean_20171020_srvprd");

create table MyTable as

select * from connection to odbc

(SELECT snz_uid, moh_pho_sex_snz_code as sex
FROM moh_clean.pho_enrolment);

disconnect from odbc;

quit;
```

  - http://support.sas.com/techsup/technote/accessing-microsoft-sql-server-from-sas.pdf
  - For those using the IDI, the MeetaData forumn has a large amount of SAS example code. To access the MeetaData forumn, contact Stats NZ.

- **STATA:** Details coming soon
- **SPSS:** Details coming soon

4.2.5 A note on connecting from an IDI datalab

If you are trying to connect to the IDI from a secure datalab, then it is unlikely that you will have to use a DSN. Instead, use Method 1 on page 63 of these notes, and get the driver name, server name, database name and port number from a Stats NZ representative.

Appendix A

T-SQL Syntax

A.1 T-SQL Syntax Conventions

These are screenshots of the T-SQL Syntax conventions, provided for convenience in the printed notes.

The following table lists and describes conventions that are used in the syntax diagrams in the Transact-SQL Reference.

Convention	Used for
UPPERCASE	Transact-SQL keywords.
italic	User-supplied parameters of Transact-SQL syntax.
bold	Database names, table names, column names, index names, stored procedures, utilities, data type names, and text that must be typed exactly as shown.
underline	Indicates the default value applied when the clause that contains the underlined value is omitted from the statement.
| (vertical bar)	Separates syntax items enclosed in brackets or braces. You can use only one of the items.
`[ ]` (brackets)	Optional syntax items. Do not type the brackets.
{ } (braces)	Required syntax items. Do not type the braces.
[,...*n*]	Indicates the preceding item can be repeated *n* number of times. The occurrences are separated by commas.
[...*n*]	Indicates the preceding item can be repeated *n* number of times. The occurrences are separated by blanks.
;	Transact-SQL statement terminator. Although the semicolon is not required for most statements in this version of SQL Server, it will be required in a future version.
<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>. A set is a collection of expressions, for example <grouping set>; and a list is a collection of sets, for example <composite element list>.

Multipart Names

Unless specified otherwise, all Transact-SQL references to the name of a database object can be a four-part name in the following form:

server_name .[*database_name*].[*schema_name*].*object_name*

| *database_name*.[*schema_name*].*object_name*

| *schema_name*.*object_name*

| *object_name*

server_name
Specifies a linked server name or remote server name.

database_name
Specifies the name of a SQL Server database when the object resides in a local instance of SQL Server. When the object is in a linked server, *database_name* specifies an OLE DB catalog.

schema_name
Specifies the name of the schema that contains the object if the object is in a SQL Server database. When the object is in a linked server, *schema_name* specifies an OLE DB schema name.

object_name
Refers to the name of the object.

When referencing a specific object, you do not always have to specify the server, database, and schema for the SQL Server Database Engine to identify the object. However, if the object cannot be found, an error is returned.

> ⓘ Note
>
> To avoid name resolution errors, we recommend specifying the schema name whenever you specify a schema-scoped object.

A.2 Logical and comparison operators

Logical Operators (Transact-SQL)

03/06/2017 · 2 minutes to read · Contributors

In this article

See Also

APPLIES TO: SQL Server (starting with 2012) Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

Logical operators test for the truth of some condition. Logical operators, like comparison operators, return a **Boolean** data type with a value of TRUE, FALSE, or UNKNOWN.

Operator	Meaning
ALL	TRUE if all of a set of comparisons are TRUE.
AND	TRUE if both Boolean expressions are TRUE.
ANY	TRUE if any one of a set of comparisons are TRUE.
BETWEEN	TRUE if the operand is within a range.
EXISTS	TRUE if a subquery contains any rows.
IN	TRUE if the operand is equal to one of a list of expressions.
LIKE	TRUE if the operand matches a pattern.
NOT	Reverses the value of any other Boolean operator.
OR	TRUE if either Boolean expression is TRUE.
SOME	TRUE if some of a set of comparisons are TRUE.

Comparison Operators (Transact-SQL)

03/14/2017 • 2 minutes to read • Contributors all

APPLIES TO: SQL Server (starting with 2012) Azure SQL Database Azure SQL Data Warehouse Parallel Data Warehouse

Comparison operators test whether two expressions are the same. Comparison operators can be used on all expressions except expressions of the **text**, **ntext**, or **image** data types. The following table lists the Transact-SQL comparison operators.

Operator	Meaning
= (Equals)	Equal to
> (Greater Than)	Greater than
< (Less Than)	Less than
>= (Greater Than or Equal To)	Greater than or equal to
<= (Less Than or Equal To)	Less than or equal to
<> (Not Equal To)	Not equal to
!= (Not Equal To)	Not equal to (not ISO standard)
!< (Not Less Than)	Not less than (not ISO standard)
!> (Not Greater Than)	Not greater than (not ISO standard)

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. 31

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). 39

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. 18

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. 11

entry One data value, in one particular row and column of a table. 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`. 21, 39

many-to-many relationship When one record (tuple) in a table can be associated with multiple records (tuples) in another table, and vice versa. 15

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. 33

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. 17

primary key A primary key is any column (or collection of columns) that has (or have, together) been chosen to uniquely identify the rows of the table it belongs to. The entries in a primary key must be unique. 15

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

schema A schema is a collection of tables in a database that are grouped together conceptually to help keep the database organised. 55

search condition A logical statement that evaluates to either True or False, for any given row. 21, 40

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