

```
>>> A crash course in SQL  
>>> New Zealand Social Statistics Network
```

Name: Daniel Fryer[†]

Date: November 25, 2019

[†]d.fryer@latrobe.edu.au

>>> How to pronounce SQL

- * S. Q. L. (Structured Query Language)
- * 'SEQUEL' (Structured English Query Language)

We will be using Microsoft's Transact-SQL (T-SQL)

>>> Overview

Day 1

1. Introductions
2. Conceptual stuff (DBMS and the relational model)
3. Tables and relationships between them
4. Intro to SSMS
5. Basic SQL queries and exercises

LUNCH!

6. Search conditions
7. Join and aggregating queries
8. More exercises!

GO HOME AND READ THE NOTES (AND PRACTICE?)

>>> Overview

Day 2

1. Revision of Day 1
2. Reading the docs
3. Lots of exercises!

LUNCH!

4. Details about the IDI
5. Creating and editing tables in SQL
6. Connecting to SQL from R
7. Time for more SQL?
8. More exercises!

SEND ME QUESTIONS AND GIVE FEEDBACK!

>>> Daily schedule

Session 1 09:00am - 10:30am
morning tea (15min)
Session 2 10:45am - 12:30pm
lunch (1 hour)
Session 3 01:30pm - 03:00pm
afternoon tea (15 min)
Session 4 03:15pm - 04:30pm

>>> A little about yourself

- * What is your name?
- * What is your favourite colour? (or NULL)
- * What would you like to get out of this course?

>>> Show of hands

Past experience

>>> The Kahoots!

Definition

A Kahoot is a fun quiz thing that we'll do at the end of most sessions. Join in to test your skills. Use the same nickname every time if you want to join the leaderboard.

And now for a practice Kahoot...

>>> Let the learning begin

A Relational Database Management System (RDBMS).

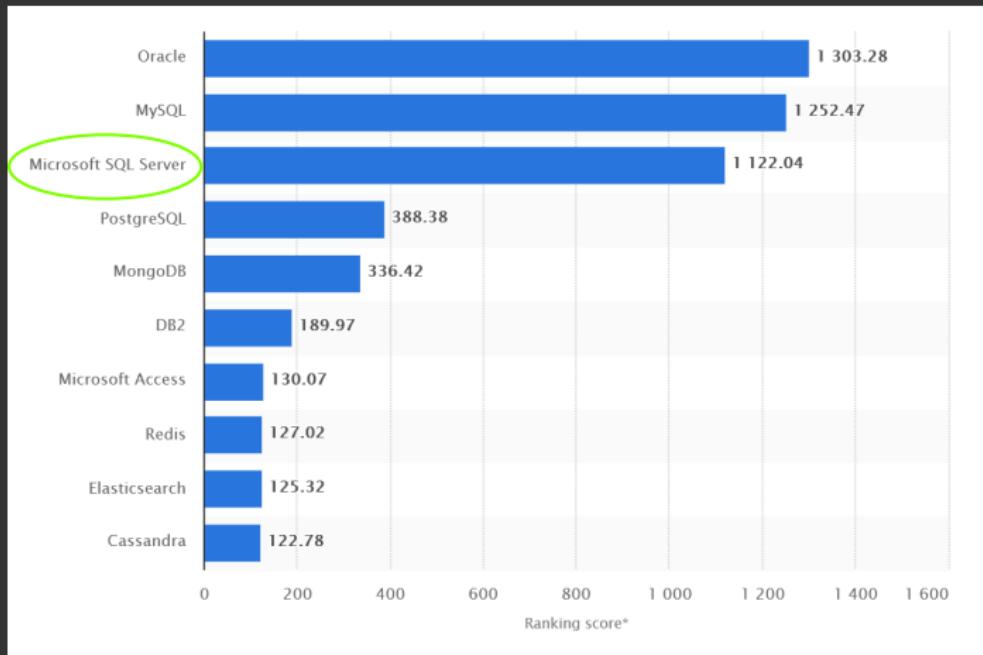
Definition

Kind of complicated. A DBMS is a large collection of interdependent programs all working together to define, construct, manipulate, protect and otherwise manage a database. An RDBMS is the most popular kind of DBMS.

SQL is a programming language for talking to your RDBMS.

>>> RDBMS

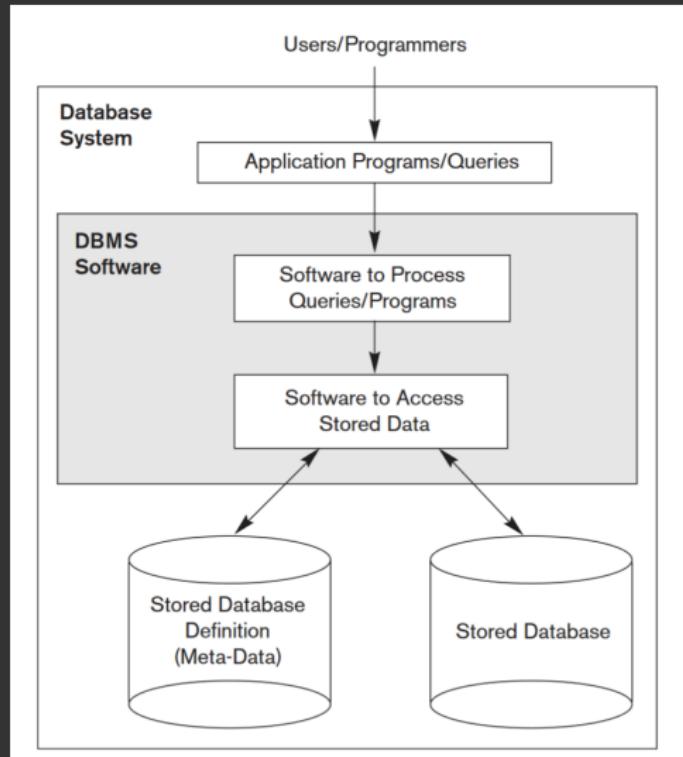
The most popular Relational Database Management Systems



Source: statista.com

>>> RDBMS

A layer of abstraction between human and machine



Grandfather of SQL and RDBMS, in the 1970s:

'Future users of databases should be protected from having to know how the data is organised in the machine.' - Ted Codd (IBM researcher).

To talk to humans and machines, the RDBMS should have a model of the world that is intuitive to both. This model is called the **Relational Model**.

Definition

The Relational Model is the 'common tongue' between the humans and the machines. It has a nice formal mathematical definition, so it is easy for machines to work with.
For the humans, it has a simple intuitive description in terms of tables and relationships between tables!

```
>>> Our very first table
```

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

```
>>> What's the takeaway from all this??
```

When using SQL, you'll always be working with tables. This is (deceptively) simple and intuitive. Underlying that, there is a really powerful system that lets you talk to the machine in a fairly ideal way. This makes SQL **very efficient**.

The tradeoff? Some parts of SQL will be really simple and intuitive. Others can at first be frustrating and confusing. A little practice goes a loooooong way.

```
>>> The anatomy of a table
```

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

>>> The anatomy of a table

Table name

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

>>> The anatomy of a table

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

```
>>> The anatomy of a table
```

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

Column (attribute)

```
>>> The anatomy of a table
```

Column names (attribute names)

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

>>> The anatomy of a table

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

>>> The anatomy of a table

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

- * Every table should have a primary key
- * No two rows can have the same entry
- * There must be no NULL entries

>>> One more thing: The data types of attributes

Friends(FriendID, FirstName, LastName, FavColour)

>>> One more thing: The data types of attributes

Friends(FriendID, FirstName, LastName, FavColour)
int

Definition

An integer is a positive or negative whole number.

>>> One more thing: The data types of attributes

Friends(FriendID, FirstName, LastName, FavColour)
 varchar varchar varchar

Definition

Varchar stands for 'variable length character.'

It is a string of characters of undetermined length.

>>> Where are we now?

Day 1

1. Introductions
2. Conceptual stuff (DBMS and the relational model)
3. Tables and relationships between them
4. Intro to SSMS
5. Basic SQL queries and exercises

LUNCH!

6. Search conditions
7. Join and aggregating queries
8. More exercises!

GO HOME AND READ THE NOTES (AND PRACTICE?)

>>> What are relationships between tables?



>>> Relationships between tables overview

1. One-to-many relationships
2. Primary and foreign keys
3. Many-to-many relationships
4. One-to-one relationships

>>> One-to-many relationships

- * For each car there are *many* wheels.

>>> One-to-many relationships

- * For each car there are *many* wheels.



>>> One-to-many relationships

- * For each car there are *many* wheels.
But each wheel belongs to only *one* car.

>>> One-to-many relationships

- * For each car there are *many* wheels.
But each wheel belongs to only *one* car.
- * One bank can have *many* accounts.
But each account belongs to *one* bank.

>>> One-to-many relationships

- * For each friend there are *many* pets.
But each pet belongs to only *one* friend.

Where do we put the extra pets?

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

>>> One-to-many relationships

- * For each friend there are *many* pets.
But each pet belongs to only *one* friend.

Where do we put the extra pets?

Friends				
FriendID	FirstName	...	PetName ₁	PetName ₂
1	X	...	NULL	NULL
2	Y	...	Chikin	NULL
3	Z	...	Cauchy	Gauss

>>> Problems with putting them in the same table

Ideas?

Friends				
FriendID	FirstName	...	PetName ₁	PetName ₂
1	X	...	NULL	NULL
2	Y	...	Chikin	NULL
3	Z	...	Cauchy	Gauss

>>> Problems with putting them in the same table

- * Have to store NULL in every entry with no pet

>>> Problems with putting them in the same table

- * Have to store NULL in every entry with no pet
- * What if I meet a friend with 3+ pets? Many more NULLs

>>> Problems with putting them in the same table

- * Have to store NULL in every entry with no pet
- * What if I meet a friend with 3+ pets? Many more NULLs
- * New one-to-many relationship between pets and toys?

>>> Problems with putting them in the same table

- * Have to store NULL in every entry with no pet
- * What if I meet a friend with 3+ pets? Many more NULLs
- * New one-to-many relationship between pets and toys?
- * Pets are tied to owners. Delete an owner → delete pets

```
>>> Problems with putting them in the same table
```

- * Have to store NULL in every entry with no pet
- * What if I meet a friend with 3+ pets? Many more NULLs
- * New one-to-many relationship between pets and toys?
- * Pets are tied to owners. Delete an owner → delete pets
- * Ambiguity. Is information related to pets or owners?

>>> So what do we do instead?

Suspense.
The first Kahoot.

```
>>> What we do instead is...
```

Create another table.

```
>>> What we do instead is...
```

Create another table.

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

```
>>> What we do instead is...
```

Create another table.

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

Foreign key

>>> The foreign key 'points at' the primary key

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

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

>>> The foreign key 'points at' the primary key

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

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

Many

>>> The foreign key 'points at' the primary key

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

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

```
>>> Check that we fixed all these problems
```

- * Have to store NULL in every entry with no pet
- * What if I meet a friend with 3+ pets? Many more NULLs
- * New one-to-many relationship between pets and toys?
- * Pets are tied to owners. Delete an owner → delete pets
- * Ambiguity. Is information related to pets or owners?

```
>>> Joining the tables
```

FriendsPets						
PetID	PetName	...	FriendID	FriendID	FirstName	...
1	Chikin	...	2	2	Y	...
2	Cauchy	...	3	3	Z	...
3	Gauss	...	3	3	Z	...

```
>>> Joining the tables
```

FriendsPets						
PetID	PetName	...	FriendID	FriendID	FirstName	...
1	Chikin	...	2	2	Y	...
2	Cauchy	...	3	3	Z	...
3	Gauss	...	3	3	Z	...

Primary/**foreign** key pair

>>> Group practice

1. Come up with a one-to-many relationship between either table (**Friends** or **Pets**) and something new
2. Make up 2 attributes for the new table (aside from the primary and foreign keys)
3. Make up 3 records for the new table
4. Draw up the two tables
5. Join the two tables

6. Challenge: Can you create a one-to-many relationship between **Friends** and **Friends**? How will you model it?

>>> A solution to the challenge question

- * Game in which friends fight to the death. A friend can beat many others, but can only be beaten by one at most.

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

>>> Primary and foreign keys

- * Foreign key 'points at' the primary key
- * Two rows CAN share same foreign key value
- * Two rows CAN NOT share same primary key value
- * Primary key can never be NULL
- * All tables should have a primary key

- * A PK or FK can be made of more than one column.

>>> Primary and foreign keys

- * Foreign key 'points at' the primary key
- * Two rows CAN share same foreign key value
- * Two rows CAN NOT share same primary key value
- * Primary key can never be NULL
- * All tables should have a primary key

- * A PK or FK can be made of more than one column.

For example, a company might sell group holiday packages and the primary key of their **Customer** table might be made of a GroupID and GroupMemberNumber.

>>> Many-to-many relationship

- * A class has many students,
and a student attends many classes
- * A company has many investors,
and an investor invests in many companies
- * A person engages with many government departments,
and a government department engages with many people

>>> Many-to-many relationship

- * Each friend can scratch many backs,
and a back can be scratched by many friends

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

>>> Many-to-many relationship

- * Each friend can scratch many backs,
and a back can be scratched by many friends

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

>>> Many-to-many relationship

- * Each friend can scratch many backs,
and a back can be scratched by many friends

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

>>> Many-to-many relationship

- * Each friend can scratch many backs,
and a back can be scratched by many friends

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

>>> Many-to-many relationship

- * Each friend can scratch many backs,
and a back can be scratched by many friends

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

```
>>> Joining the tables
```

Friend_Scratched_Friend

FrID	FriendName	...	SrID	...	SeID	FrID	FriendName	...
1	X	...	1	...	2	2	Y	...
1	X	...	1	...	3	3	Z	...
2	Y	...	2	...	1	1	X	...
3	Z	...	3	...	1	1	X	...

```
>>> Joining the tables
```

Friend_Scratched_Friend								
FrID	FriendName	...	SrID	...	SeID	FrID	FriendName	...
1	X	...	1	...	2	2	Y	...
1	X	...	1	...	3	3	Z	...
2	Y	...	2	...	1	1	X	...
3	Z	...	3	...	1	1	X	...

Pair 1

```
>>> Joining the tables
```

Friend_Scratched_Friend

FrID	FriendName	...	SrID	...	SeID	FrID	FriendName	...
1	X	...	1	...	2	2	Y	...
1	X	...	1	...	3	3	Z	...
2	Y	...	2	...	1	1	X	...
3	Z	...	3	...	1	1	X	...

Pair 2

```
>>> Group practice (join)
```

- * A friend can play with many pets,
and a pet can play with many friends

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

>>> One-to-one relationship

- * A person can have at most one head,
and each head belongs to only one person
- * A table record has exactly one primary key value,
and each primary key value belongs to exactly one record
- * A user has one set of log-in details,
and each set of log-in details belong to one user

>>> One-to-one relationship

- * One friend can have at most one passport, and each passport belongs to only one friend

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

>>> Why not keep one-to-one relationships in the same table?

- * NULLs (many passport attributes? few people have them?)

>>> Why not keep one-to-one relationships in the same table?

- * NULLs (many passport attributes? few people have them?)
- * Dependence: Delete friend → delete passport.

>>> Goodbye, Mr. X

Friends

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

>>> Solution

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

>>> Solution

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

Mr. X

>>> Any problems with this approach though?

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

>>> Any problems with this approach though?

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

Deleting a friend will delete the owner's name

```
>>> Any idea how to fix this?
```

We should avoid keeping the person's name in both tables,
since otherwise we have **redundant data**.

>>> Any idea how to fix this?

- * Create 'people' table with **binary** variable for friend?

Definition

A binary variable is always either 0, 1 or NULL.
Usually, 0 represents false and 1 represents true.

>>> Any idea how to fix this?

- * Create 'people' table with **binary** variable for friend?
- * Create separate tables for friends, enemies, etc...?

Leave it to the database designers.

>>> How a database design can restrict research

- * Missing information
- * Conflicting information (due to redundancy)
- * Not enough levels of a categorical variable
- * Binary answer when binary is not appropriate
- * Hard to join the tables and connect records
- * Hard to search for information in the database
- * Many more, keep eyes open...

>>> Where are we now?

Day 1

1. Introductions
2. Conceptual stuff (DBMS and the relational model)
3. Tables and relationships between them
4. Intro to SSMS
5. Basic SQL queries and exercises

LUNCH!

6. Search conditions
7. Join and aggregating queries
8. More exercises!

GO HOME AND READ THE NOTES (AND PRACTICE?)

>>> Getting set up with SQL Server

Time for real life SQL action

Go to GearHost.com

>>> Task 1

Use the Object Explorer pane in SSMS to begin investigating the different tables and schemas in your database. If you click on the database name and press F7, it will open an Object Explorer Details window which is easier to browse in. Figure out some table names and column names in the **Notes** schema and **Ape** schema.

>>> Task 2

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 (which we learn about soon).

>>> Task 3

Right click on the `Notes.Friends` table in the object explorer pane and click 'Select Top 1000 Rows.' A 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.

>>> Task 4

Change the query from the previous task so it looks like this.

```
SELECT *
FROM Notes.Friends F;
```

Then, execute it. What does it do?

>>> Task 5

Write the following query.

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

Then, execute it. What does it do?

>>> Task 6

Write the following query.

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

Then, execute it. What has the query done?

```
>>> A note on syntax
```

```
SeLect * froM [N O t e S ].  
[PETS]RIPHarambe20160528;
```

- * Upper/lower-case has no effect
- * Spaces usually have no effect
- * Square brackets can be omitted
- * New lines have no effect
- * Alias can be almost anything

So pay attention to style

>>> Time to learn SQL

What does each clause do?

```
>>> SQL clause: FROM
```

The FROM clause is used to specify the table(s) to access in the SELECT statement (and others).

You'll use FROM in almost every query.

```
>>> SQL clause: SELECT
```

The SELECT clause allows you to choose columns.
You can select *all* columns with SELECT *

We will look at the execution of this query:

```
SELECT F.FirstName, F.FavColour  
FROM Notes.Friends F;
```

```
>>> SELECT clause execution
```

```
FROM Notes.Friends
```

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

```
>>> SELECT clause execution
```

```
SELECT F.FirstName, F.FavColour
```

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

```
>>> SELECT clause execution
```

result

Unnamed	
FirstName	FavColour
X	red
Y	blue
Z	NULL

```
>>> Order of execution
```

Let's go back and see that again

```
>>> Order of execution
```

Let's go back and see that again

- * Syntactic order of execution
- * Logical order of execution
- * Optimal order of execution

```
>>> SQL clause: WHERE
```

The WHERE clause allows you to choose rows,
using a **search condition**.

We will look at the execution of this query:

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

```
>>> WHERE clause execution
```

```
FROM Notes.Friends
```

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

```
>>> WHERE clause execution
```

```
WHERE FavColour = 'red'
```

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

```
SELECT FirstName, LastName
```

Unnamed			
ID	FirstName	LastName	FavColour
1	X	A	red

result

Unnamed	
FirstName	LastName
X	A

>>> Order of execution

1. FROM
2. WHERE
3. SELECT

>>> Where are we now?

Day 1

1. Introductions
2. Conceptual stuff (DBMS and the relational model)
3. Tables and relationships between them
4. Intro to SSMS
5. Basic SQL queries and exercises

LUNCH!

6. Search conditions
7. Join and aggregating queries
8. More exercises!

GO HOME AND READ THE NOTES (AND PRACTICE?)

>>> Search conditions

Search conditions appear in a WHERE clause. They make use of comparison operators and logical operators to check which rows match the conditions you specify.

>>> Search conditions

Search conditions appear in a WHERE clause. They make use of **comparison operators** and logical operators to check which rows match the conditions you specify.

Definition

A comparison operator is used to compare two things and return true, false or NULL. See [examples in the docs](#).

>>> Search conditions

Search conditions appear in a WHERE clause. They make use of comparison operators and logical operators to check which rows match the conditions you specify.

Definition

Logical operators compare a number of things and return true, false or NULL. See examples in the docs.

>>> Comparison operators

* WHERE FavColour = 'blue' (equal)

>>> Comparison operators

- * WHERE FavColour = 'blue' (equal)
- * WHERE FavColour <> 'blue' (not equal)
- * WHERE FavColour != 'blue' (also not equal)

>>> Comparison operators

- * WHERE FavColour = 'blue' (equal)
- * WHERE FavColour <> 'blue' (not equal)
- * WHERE FavColour != 'blue' (also not equal)
- * WHERE Age > 35 (greater than)
- * WHERE Year <= 1995 (less than or equal)

>>> Logical operators

```
* WHERE FavColour IN ('blue', 'red', 'green')
```

>>> Logical operators

- * WHERE FavColour IN ('blue', 'red', 'green')
- * WHERE Age BETWEEN 25 AND 35

>>> Logical operators

- * WHERE FavColour IN ('blue', 'red', 'green')
- * WHERE Age BETWEEN 25 AND 35
- * WHERE FirstName LIKE 'B%'

>>> Logical operators

- * WHERE FavColour IN ('blue', 'red', 'green')
- * WHERE Age BETWEEN 25 AND 35
- * WHERE FirstName LIKE 'B%'
- * WHERE FirstName LIKE '%B'

>>> Logical operators

- * WHERE FavColour IN ('blue', 'red', 'green')
- * WHERE Age BETWEEN 25 AND 35
- * WHERE FirstName LIKE 'B%'
- * WHERE FirstName LIKE '%B'
- * WHERE FirstName LIKE '%b%'

>>> Logical operators

- * WHERE FavColour IN ('blue', 'red', 'green')
- * WHERE Age BETWEEN 25 AND 35
- * WHERE FirstName LIKE 'B%'
- * WHERE FirstName LIKE '%B'
- * WHERE FirstName LIKE '%b%'
- * WHERE FirstName LIKE '%[Bb]%'

>>> Logical operators

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

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

NOT				
NOT	true	=	false	
NOT	false	=	true	

>>> Group practice

1. (1 = 1) AND (2 = 1)
2. ((1 = 1) AND (2 = 1)) OR (1 = 1)
3. ('red' IN ('green', 'red')) AND ('red' LIKE 'r%')
4. NOT ((1 = 1) AND (2 = 2) AND (3 = 3))

>>> Solution

1. $(1 = 1) \text{ AND } (2 = 1)$
2. $((1 = 1) \text{ AND } (2 = 1)) \text{ OR } (1 = 1)$
3. $('red' \text{ IN } ('green', 'red')) \text{ AND } ('red' \text{ LIKE } 'r\%')$
4. $\text{NOT } ((1 = 1) \text{ AND } (2 = 2) \text{ AND } (3 = 3))$

>>> Solution

1. true AND (2 = 1)
2. ((1 = 1) AND (2 = 1)) OR (1 = 1)
3. ('red' IN ('green', 'red')) AND ('red' LIKE 'r%')
4. NOT ((1 = 1) AND (2 = 2) AND (3 = 3))

>>> Solution

1. true AND false
2. ((1 = 1) AND (2 = 1)) OR (1 = 1)
3. ('red' IN ('green', 'red')) AND ('red' LIKE 'r%')
4. NOT ((1 = 1) AND (2 = 2) AND (3 = 3))

>>> Solution

1. **false**
2. **((1 = 1) AND (2 = 1)) OR (1 = 1)**
3. **('red' IN ('green', 'red')) AND ('red' LIKE 'r%')**
4. **NOT ((1 = 1) AND (2 = 2) AND (3 = 3))**

>>> Solution

1. false
2. false OR (1 = 1)
3. ('red' IN ('green', 'red')) AND ('red' LIKE 'r%')
4. NOT ((1 = 1) AND (2 = 2) AND (3 = 3))

>>> Solution

1. false
2. false OR true
3. ('red' IN ('green', 'red')) AND ('red' LIKE 'r%')
4. NOT ((1 = 1) AND (2 = 2) AND (3 = 3))

>>> Solution

1. false
2. true
3. ('red' IN ('green', 'red')) AND ('red' LIKE 'r%')
4. NOT ((1 = 1) AND (2 = 2) AND (3 = 3))

>>> Solution

1. false
2. true
3. true AND ('red' LIKE 'r%')
4. NOT ((1 = 1) AND (2 = 2) AND (3 = 3))

>>> Solution

1. false
2. true
3. true AND true
4. NOT ((1 = 1) AND (2 = 2) AND (3 = 3))

>>> Solution

1. false
2. true
3. true
4. NOT ((1 = 1) AND (2 = 2) AND (3 = 3))

>>> Solution

1. false
2. true
3. true
4. NOT (true AND (3 = 3))

>>> A note on NULL

NULL
anything AND NULL = NULL
anything OR NULL = NULL
anything = NULL = NULL

We will get practice with NULLs during the exercises.

>>> Search condition with subquery

Subqueries are a very valuable addition to search conditions.
In fact, some logical operators only work with subqueries.

- * EXISTS
- * ALL
- * ANY

Subqueries are also known as nested queries.

>>> Class practice

Can anyone guess/figure out what this does?

Note: the subquery is executed first.

```
SELECT *
FROM Notes.Friends F
WHERE F.FriendID IN (SELECT P.FriendID
                      FROM Notes.Pets P);
```

>>> Solution

1. SELECT P.FriendID FROM Notes.Pets P

>>> Solution

1. SELECT P.FriendID FROM Notes.Pets P

Retrieves a table of all the FriendIDs in Notes.Pets.

>>> Solution

1. SELECT P.FriendID FROM Notes.Pets P
Retrieves a table of all the FriendIDs in Notes.Pets.
2. Let's refer to the output of Step 1 as RESULT.

>>> Solution

1. SELECT P.FriendID FROM Notes.Pets P
Retrieves a table of all the FriendIDs in Notes.Pets.
2. Let's refer to the output of Step 1 as RESULT.
3. SELECT * FROM Notes.Friends F WHERE F.FriendID IN RESULT

>>> Solution

1. `SELECT P.FriendID FROM Notes.Pets P`
Retrieves a table of all the FriendIDs in Notes.Pets.
2. Let's refer to the output of Step 1 as `RESULT`.
3. `SELECT * FROM Notes.Friends F WHERE F.FriendID IN RESULT`
Retrieves only the rows of Notes.Friends whose FriendID is in `RESULT`.

>>> Solution

1. `SELECT P.FriendID FROM Notes.Pets P`
Retrieves a table of all the FriendIDs in Notes.Pets.
2. Let's refer to the output of Step 1 as `RESULT`.
3. `SELECT * FROM Notes.Friends F WHERE F.FriendID IN RESULT`
Retrieves only the rows of Notes.Friends whose FriendID is in `RESULT`.

We retrieved the details of all friends who have pets.

>>> Correlated subquery

This query achieves the same thing as the previous one:

```
SELECT *
FROM Notes.Friends F
WHERE EXISTS (SELECT P.FriendID
              FROM Notes.Pets P
              WHERE P.FriendID = F.FriendID);
```

>>> Correlated subquery

This query achieves the same thing as the previous one:

```
SELECT *
FROM Notes.Friends F
WHERE EXISTS (SELECT P.FriendID
              FROM Notes.Pets P
              WHERE P.FriendID = F.FriendID);
```



```
>>> SQL query: JOIN
```

A JOIN (also known as an INNER JOIN) pairs the records from one table with the records from another table, using a primary/foreign key pair.

We will look at the execution of this query:

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

```
>>> SQL query: JOIN
```

Another way to write the same query: explicit JOIN

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

```
>>> SQL query: JOIN
```

Another way to write the same query:

```
SELECT FirstName, PetName  
FROM Notes.Friends AS F  
JOIN Notes.Pets AS P  
ON F.FriendID = P.FriendID;
```

```
>>> SQL query: JOIN
```

Note that JOIN is an operator that is inside the FROM clause.

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

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

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

>>> SQL query: JOIN

SELECT F.FirstName, P.PetName

Unnamed						
PetID	PetName	...	FriendID	FriendID	FirstName	...
1	Chikin	...	2	2	Y	...
2	Cauchy	...	3	3	Z	...
3	Gauss	...	3	3	Z	...

```
>>> SQL query: JOIN
```

result

Unnamed	
PetName	FirstName
Chikin	Y
Cauchy	Z
Gauss	Z

```
>>> Order of execution
```

JOIN is technically an **operator**, not a clause.

1. FROM (and JOIN)
2. WHERE
3. SELECT

>>> Group practice

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

- * SELECT B,C,D FROM Table1 T1, Table2 T2 WHERE T1.A = T2.A
- * SELECT B,C,D FROM Table1 T1, Table2 T2 WHERE T1.A = T2.E

>>> Solutions

* SELECT B,C,D FROM Table1 T1, Table2 T2 WHERE T1.A = T2.A

B	C	D
Ignorance	is	slavery.
War	is	weakness.
Freedom	is	strength.
Friendship	is	peace.

>>> Solutions

* SELECT B,C,D FROM Table1 T1, Table2 T2 WHERE T1.A = T2.A

B	C	D
Ignorance	is	slavery.
War	is	weakness.
Freedom	is	strength.
Friendship	is	peace.

* SELECT B,C,D FROM Table1 T1, Table2 T2 WHERE T1.A = T2.E

B	C	D
Ignorance	is	strength.
War	is	peace.
Freedom	is	slavery.
Friendship	is	weakness.

>>> Where are we now?

Day 1

1. Introductions
2. Conceptual stuff (DBMS and the relational model)
3. Tables and relationships between them
4. Intro to SSMS
5. Basic SQL queries and exercises

LUNCH!

6. Search conditions
7. Join and aggregating queries
8. More exercises!

GO HOME AND READ THE NOTES (AND PRACTICE?)

>>> Aggregating queries

Aggregating queries collect the rows of a table into groups, and somehow return a single value for each group, or act on each group in some way.

We will cover:

1. GROUP BY clause
2. Aggregation functions
3. HAVING clause

The GROUP BY clause creates the groups. An aggregation function returns a single value or summary statistic for each group. The HAVING clause is used to choose groups (much like the WHERE clause chooses rows).

```
>>> SQL clause: GROUP BY
```

The GROUP BY clause 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.

We will look at the execution of this query:

```
SELECT P.FriendID  
FROM Notes.Pets P  
GROUP BY P.FriendID;
```

```
>>> SQL clause: GROUP BY
```

```
FROM Notes.Pets P
```

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

```
>>> SQL clause: GROUP BY
```

```
GROUP BY P.FriendID
```

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

```
>>> SQL clause: GROUP BY
```

```
GROUP BY P.FriendID
```

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

```
>>> SQL clause: GROUP BY
```

```
SELECT P.FriendID
```

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

```
>>> SQL clause: GROUP BY
```

result

Unnamed
FriendID
2
3

```
>>> Can we select any of the other columns?
```

SELECT ???

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

SQL prevents it since it can't be sure that there is only one value in each entry.

>>> What will happen if we run this?

```
SELECT P.FriendID, P.PetDOB  
FROM Notes.Pets P  
GROUP BY P.FriendID;
```



```
>>> Error
```

Msg 8120, Level 16, State 1, Line 1

Column 'Notes.Pets.PetDOB' is invalid in the select list
because it is not contained in either an aggregate function
or the GROUP BY clause.

```
>>> This will fix the error
```

```
SELECT P.FriendID, P.PetDOB  
FROM Notes.Pets P  
GROUP BY P.FriendID, P.PetDOB;
```

```
>>> SQL clause: GROUP BY
```

```
GROUP BY P.FriendID, P.PetDOB
```

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

>>> SQL clause: GROUP BY

GROUP BY P.FriendID, P.PetDOB

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

>>> Group practice

Include the curly braces in your solutions

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

- * GROUP BY B
- * GROUP BY A
- * GROUP BY A, B

>>> Solutions

* GROUP BY B

Unnamed		
A	B	Num
{a, a}	b	{1, 3}
{a, a}	c	{2, 4}

>>> Solutions

* GROUP BY B

Unnamed		
A	B	Num
{a, a}	b	{1, 3}
{a, a}	c	{2, 4}

* GROUP BY A

Unnamed		
A	B	Num
a	{b, c, b, c}	{1, 2, 3, 4}

>>> Solutions

* GROUP BY B

Unnamed		
A	B	Num
{a, a}	b	{1, 3}
{a, a}	c	{2, 4}

* GROUP BY A

Unnamed		
A	B	Num
a	{b, c, b, c}	{1, 2, 3, 4}

* GROUP BY A, B

Unnamed		
A	B	Num
a	b	{1, 3}
a	c	{2, 4}

>>> Aggregation functions

Aggregation functions are able to return a single value for each group. If you use an aggregation function, you can select a column that you haven't included in GROUP BY.

We will look at the execution of this query:

```
SELECT RP.Gender,  
       AVG(RP.Age) AS AverageAge  
FROM Notes.RandomPeople RP  
GROUP BY RP.Gender;
```

>>> Aggregation functions

Aggregation functions are able to return a single value for each group. If you use an aggregation function, you can select a column that you haven't included in GROUP BY.

We will look at the execution of this query:

```
SELECT RP.Gender,  
       AVG(RP.Age) AS AverageAge  
FROM Notes.RandomPeople RP  
GROUP BY RP.Gender;
```

Aggregation function

>>> Aggregation functions

Aggregation functions are able to return a single value for each group. If you use an aggregation function, you can select a column that you haven't included in GROUP BY.

We will look at the execution of this query:

```
SELECT RP.Gender,  
       AVG(RP.Age) AS AverageAge  
FROM Notes.RandomPeople RP  
GROUP BY RP.Gender;
```

Alias

```
>>> Aggregation function: AVG
```

```
FROM Notes.RandomPeople RP
```

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

```
>>> Aggregation function: AVG
```

```
GROUP BY RP.Gender
```

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

```
>>> Aggregation function: AVG
```

```
GROUP BY RP.Gender
```

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

```
>>> Aggregation function: AVG
```

AVG(RP.Age)

Unnamed		
Name	Gender	(unnamed)
{Beyoncé, Laura Marling}	F	AVG({37,28})
{Darren Hayes, Bret McKenzie}	M	AVG({46,42})
{Jack Monroe}	NB	AVG({30})

```
>>> Aggregation function: AVG
```

AVG(RP.Age)

Unnamed		
Name	Gender	(unnamed)
{Beyoncé, Laura Marling}	F	32.5
{Darren Hayes, Bret McKenzie}	M	44
{Jack Monroe}	NB	30

>>> Aggregation function: AVG

```
SELECT RP.Gender, AVG(RP.Age) AS AverageAge
```

Unnamed		
Name	Gender	(unnamed)
{Beyoncé, Laura Marling}	F	32.5
{Darren Hayes, Bret McKenzie}	M	44
{Jack Monroe}	NB	30

```
>>> Aggregation function: AVG
```

result

Unnamed	
Gender	AverageAge
F	32.5
M	44
NB	30

We retrieved the average age for each gender in the table!

```
>>> What happens if we throw in a WHERE clause?
```

We will look at the execution of this query:

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

```
>>> What happens if we throw in a WHERE clause?
```

```
FROM Notes.RandomPeople RP
```

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

```
>>> What happens if we throw in a WHERE clause?
```

```
WHERE RP.Gender = 'F'
```

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

```
>>> What happens if we throw in a WHERE clause?
```

```
GROUP BY RP.Gender
```

Unnamed		
Name	Gender	Age
{Beyoncé, Laura Marling}	F	{37, 28}

```
>>> What happens if we throw in a WHERE clause?
```

```
AVG(RP.Age)
```

Unnamed		
Name	Gender	(unnamed)
{Beyoncé, Laura Marling}	F	32.5

```
>>> What happens if we throw in a WHERE clause?
```

```
SELECT RP.Gender, AVG(RP.Age) AS AverageAge
```

Unnamed		
Name	Gender	(unnamed)
{Beyoncé, Laura Marling}	F	32.5

```
>>> What happens if we throw in a WHERE clause?
```

result

Unnamed	
Gender	AverageAge
F	32.5

We retrieved the average age for females in the table!

>>> Order of execution

1. FROM
2. WHERE
- 3.
- 4.
- 5.

```
>>> Order of execution
```

1. FROM
2. WHERE
3. GROUP BY
- 4.
- 5.

```
>>> Order of execution
```

1. FROM
2. WHERE
3. GROUP BY
4. Aggregation
- 5.

>>> Order of execution

1. FROM
2. WHERE
3. GROUP BY
4. Aggregation
5. SELECT

>>> More 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

See the full list in [the T-SQL docs](#)

>>> More 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

See the full list in the T-SQL docs

```
>>> SQL clause: HAVING
```

The HAVING clause was created because WHERE is executed before GROUP BY. The HAVING clause is like WHERE, but it acts on groups.

We will look at the execution of this query:

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

```
>>> SQL clause: HAVING
```

The HAVING clause was created because WHERE is executed before GROUP BY. The HAVING clause is like WHERE, but it acts on groups.

We will look at the execution of this query:

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

Search condition with aggregation function

```
>>> SQL clause: HAVING
```

```
FROM Notes.RandomPeople RP
```

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

```
>>> SQL clause: HAVING
```

```
GROUP BY RP.Gender
```

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

```
>>> SQL clause: HAVING
```

AVG(RP.Age)

Unnamed		
Name	Gender	(unnamed)
{Beyoncé, Laura Marling}	F	32.5
{Darren Hayes, Bret McKenzie}	M	44
{Jack Monroe}	NB	30

>>> SQL clause: HAVING

HAVING AVG(RP.Age) > 40

Unnamed		
Name	Gender	(unnamed)
{Beyoncé, Laura Marling}	F	32.5
{Darren Hayes, Bret McKenzie}	M	44
{Jack Monroe}	NB	30

```
>>> SQL clause: HAVING
```

```
SELECT RP.Gender, AVG(RP.Age) AS AverageAge
```

Unnamed		
Name	Gender	(unnamed)
{Darren Hayes, Bret McKenzie}	M	44

```
>>> SQL clause: HAVING
```

result

Unnamed	
Gender	AverageAge
M	44

>>> Order of execution

1. FROM
2. WHERE
- 3.
- 4.
- 5.
- 6.

>>> Order of execution

1. FROM
2. WHERE
3. GROUP BY
- 4.
- 5.
- 6.

>>> Order of execution

1. FROM
2. WHERE
3. GROUP BY
4. Aggregation
- 5.
- 6.

>>> Order of execution

1. FROM
2. WHERE
3. GROUP BY
4. Aggregation
5. HAVING
- 6.

>>> Order of execution

1. FROM
2. WHERE
3. GROUP BY
4. Aggregation
5. HAVING
6. SELECT

>>> Group practice

The aggregation function in the HAVING clause does not have to match the one in the SELECT clause.

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

Explain in words what the above query achieves.

>>> Solution

The query finds the sample standard deviation of the ages for each gender that has an average age greater than 40.

>>> Revision of day 1

How to tell if something is a primary key?

Let's look at some data dictionaries

>>> Additional JOIN fun

See JOIN exercise sheet

>>> Where are we now?

Day 2

1. Revision of Day 1

2. Reading the docs

3. Lots of exercises!

LUNCH!

4. Details about the IDI

5. Creating and editing tables in SQL

6. Connecting to SQL from R

7. Time for more SQL?

8. More exercises!

SEND ME QUESTIONS AND GIVE FEEDBACK!

>>> Reading the docs

- * Quickly look something up when you forget syntax
- * Learn new things while you browse and decipher
- * Gain a deeper understanding
- * It actually gets easy pretty quickly

>>> Reading the docs

- * Quickly look something up when you forget syntax
- * Learn new things while you browse and decipher
- * Gain a deeper understanding
- * It actually gets easy pretty quickly

Be brave, computers can sense fear

```
>>> Read the docs: FROM
```

The FROM clause is used to specify the table(s) used in the SELECT statement (and others).

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

where

```
{<table_source>} ::= table_or_view_name [[AS] table_alias]
```

>>> Reading the docs: FROM

The Transact-SQL Syntax Conventions

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

>>> Reading the docs: FROM

The Transact-SQL Syntax Conventions

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

* { } curly braces group required items

```
>>> Reading the docs: FROM
```

The Transact-SQL Syntax Conventions

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

- * { } curly braces group required items
- * <label> placeholder for a block of syntax

>>> Reading the docs: FROM

The Transact-SQL Syntax Conventions

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

- * { } curly braces group required items
- * <label> placeholder for a block of syntax
- * [,...n] means you can repeat with commas between

```
>>> Reading the docs: FROM
```

The Transact-SQL Syntax Conventions

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

- * { } curly braces group required items
- * <label> placeholder for a block of syntax
- * [,...n] means you can repeat with commas between

```
FROM MyTable
```

```
>>> Reading the docs: FROM
```

The Transact-SQL Syntax Conventions

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

- * { } curly braces group required items
- * <label> placeholder for a block of syntax
- * [,...n] means you can repeat with commas between

```
FROM MyTable, MyOtherTable
```

```
>>> Reading the docs: FROM
```

The Transact-SQL Syntax Conventions

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

where

```
<table_source> ::= table_or_view_name [[AS] table_alias]
```

- * { } curly braces group required items
- * <label> placeholder for a block of syntax
- * [,...n] means you can repeat with commas between
- * <label> ::= defining the placeholder

```
>>> Reading the docs: FROM
```

The Transact-SQL Syntax Conventions

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

where

```
<table_source> ::= table_or_view_name [[AS] table_alias]
```

- * { } curly braces group required items
- * <label> placeholder for a block of syntax
- * [,...n] means you can repeat with commas between
- * <label> ::= defining the placeholder

```
>>> Reading the docs: FROM
```

The Transact-SQL Syntax Conventions

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

where

```
<table_source> ::= table_or_view_name [[AS] table_alias]
```

- * { } curly braces group required items
- * <label> placeholder for a block of syntax
- * [,...n] means you can repeat with commas between
- * <label> ::= defining the placeholder
- * [] square brackets indicate optional items

```
>>> Reading the docs: FROM
```

The Transact-SQL Syntax Conventions

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

where

```
<table_source> ::= table_or_view_name [[AS] table_alias]
```

- * { } curly braces group required items
- * <label> placeholder for a block of syntax
- * [,...n] means you can repeat with commas between
- * <label> ::= defining the placeholder
- * [] square brackets indicate optional items

```
FROM MyTable M
```

```
>>> Reading the docs: FROM
```

The Transact-SQL Syntax Conventions

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

where

```
<table_source> ::= table_or_view_name [[AS] table_alias]
```

- * { } curly braces group required items
- * <label> placeholder for a block of syntax
- * [,...n] means you can repeat with commas between
- * <label> ::= defining the placeholder
- * [] square brackets indicate optional items

```
FROM MyTable AS M
```

>>> Feeling confident?

Have a look at the T-SQL FROM documentation

>>> Feeling confident?

Have a look at the T-SQL FROM documentation

- * It really is more of the same
- * It gets easier very quickly with practice
- * Google, StackExchange, etc
- * Beginner tutorial
- * Syntax guides and cheat sheets

>>> One more important syntax convention

- * { } curly braces group required items
- * <label> placeholder for a block of syntax
- * [,...n] means you can repeat with commas between
- * <label> ::= defining the placeholder
- * [] square brackets indicate optional items
- * | vertical bar indicates alternatives (OR)

>>> Group practice

```
<greeting> ::= {{Hello|Hi} [,....n] .}
                  [Do you {love|hate} reading the docs?]
```

- * { } curly braces group required items
- * <label> placeholder for a block of syntax
- * [,....n] means you can repeat with commas between
- * <label> ::= defining the placeholder
- * [] square brackets indicate optional items
- * | vertical bar indicates alternatives (OR)

>>> Solution

- * Hello.
- * Hi.
- * Hello. Do you love reading the docs?
- * Hi. Do you love reading the docs?
- * Hello, Hello, Hi, Hello, Hi, Hi.
- * Hello, Hi, Hello, Hello. Do you love reading the docs?
- * etc.

```
>>> Example from the docs (logical operator IN)
```

```
test_expression [ NOT ] IN ( subquery | expression [ ,...n ] )
```

Don't miss the round brackets!

```
>>> Example from the docs (logical operator IN)
```

```
test_expression [ NOT ] IN ( subquery | expression [ ,...n ] )
```

* test_expression IN (expression)

Don't miss the round brackets!

Example: 'red' IN ('red')

```
>>> Example from the docs (logical operator IN)
```

```
test_expression [ NOT ] IN ( subquery | expression [ ,...n ] )
```

- * test_expression IN (expression)
- * test_expression IN (subquery)

Don't miss the round brackets!

Example: FriendID IN (SELECT FriendID FROM Notes.Pets)

```
>>> Example from the docs (logical operator IN)
```

```
test_expression [ NOT ] IN ( subquery | expression [ ,...n ] )
```

- * test_expression IN (expression)
- * test_expression IN (subquery)
- * test_expression NOT IN (expression)

Don't miss the round brackets!

Example: 'red' NOT IN ('red')

```
>>> Example from the docs (logical operator IN)
```

```
test_expression [ NOT ] IN ( subquery | expression [ ,...n ] )
```

- * test_expression IN (expression)
- * test_expression IN (subquery)
- * test_expression NOT IN (expression)
- * test_expression NOT IN (subquery)

Don't miss the round brackets!

Example: FriendID NOT IN (SELECT FriendID FROM Notes.Pets)

```
>>> Example from the docs (logical operator IN)
```

```
test_expression [ NOT ] IN ( subquery | expression [ ,...n ] )
```

- * test_expression IN (expression)
- * test_expression IN (subquery)
- * test_expression NOT IN (expression)
- * test_expression NOT IN (subquery)
- * test_expression NOT IN (expression, expression,
expression)

Don't miss the round brackets!

Example: 'red' IN ('red', 'blue', 'green')

>>> A big revelation! SELECT

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.

The above is from the SELECT documentation.

>>> A big revelation! SELECT

The others don't take so long to learn

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.

The above is from the SELECT documentation.

>>> A big revelation! SELECT

The others don't take so long to learn

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.

The above is from the SELECT documentation.

>>> A big revelation! SELECT

The others don't take so long to learn

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.

The above is from the SELECT documentation.

>>> Group practice

Find the documentation for ORDER BY. Figure out what it does.
Don't overcomplicate it! We will use it during the exercises.

>>> Exercises

22-61

>>> Where are we now?

Day 2

1. Revision of Day 1
2. Reading the docs
3. Lots of exercises!

LUNCH!

4. Details about the IDI
5. Creating and editing tables in SQL
6. Connecting to SQL from R
7. Time for more SQL?
8. More exercises!

SEND ME QUESTIONS AND GIVE FEEDBACK!

>>> What is the IDI?

A collection of databases and schemas containing deidentified administrative and survey data from people's interactions with many government departments.

The different government departments use different unique identifiers, so interactions have been linked to individuals probabilistically.

>>> What is the IDI?

A collection of databases and schemas containing deidentified administrative and survey data from people's interactions with many government departments.

The different government departments use different unique identifiers, so interactions have been linked to individuals probabilistically.

The schemas (sometimes called nodes) in the main database correspond mostly to different government departments. From a technical perspective, the probabilistic linking allows us to JOIN records between schemas.

>>> The spine

The spine is a derived dataset that we don't have access to.
It is deemed by Stats NZ to be the most ideal for identifying
an **ever-resident population**.

>>> The spine

The spine is a derived dataset that we don't have access to. It is deemed by Stats NZ to be the most ideal for identifying an ever-resident population.

Individuals in the spine are the only ones whose records are linked. All links are made "through the spine." There are 10 mil people in the spine, and 57 mil people in the IDI.

>>> The spine

The spine is a derived dataset that we don't have access to. It is deemed by Stats NZ to be the most ideal for identifying an ever-resident population.

Individuals in the spine are the only ones whose records are linked. All links are made "through the spine." There are 10 mil people in the spine, and 57 mil people in the IDI.

A good spine should include every person in the target population once and only once. It includes tax, births and visa data (not deidentified).

>>> The spine

The spine is a derived dataset that we don't have access to. It is deemed by Stats NZ to be the most ideal for identifying an ever-resident population.

Individuals in the spine are the only ones whose records are linked. All links are made "through the spine." There are 10 mil people in the spine, and 57 mil people in the IDI.

A good spine should include every person in the target population once and only once. It includes tax, births and visa data (not deidentified).

- * Permanent residents
- * Visas to reside, work or study
- * People that live and work here without visas
(e.g., Australians)

>>> More details at these website links

- * Stats NZ prototype spine paper
- * Stats NZ linking methodology paper
- * VHIN spine explainer

>>> Probabilistic linking errors

Probabilistic linking produces a unique identifier, `snz_uid`.
The `snz_uid` can then act as a primary/foreign key pair.

- * Just because two records/interactions are linked, doesn't mean they belong to the same individual.
This is a **false positive**.

>>> Probabilistic linking errors

Probabilistic linking produces a unique identifier, `snz_uid`.
The `snz_uid` can then act as a primary/foreign key pair.

- * Just because two records/interactions are linked, doesn't mean they belong to the same individual.
This is a **false positive**.
- * Just because two records/interactions are *not* linked, doesn't mean they *don't* belong to the same individual.
This is a **false negative**.

>>> Probabilistic linking errors

Probabilistic linking produces a unique identifier, `snz_uid`.
The `snz_uid` can then act as a primary/foreign key pair.

- * Just because two records/interactions are linked, doesn't mean they belong to the same individual.
This is a **false positive**.
- * Just because two records/interactions are *not* linked, doesn't mean they *don't* belong to the same individual.
This is a **false negative**.
- * Just because two records from different refreshes have the same `snz_uid`, definitely doesn't mean they have belong to the same individual.
This is a **silly mistake**.

```
>>> Precision rate
```

The **precision rate** is the proportion of correct links, out of all links made. You can think of it as the probability that a randomly chosen link is correct. You can get the false positive rate from this as:

$$1 - (\text{precision rate}).$$

```
>>> Precision rate
```

The precision rate is the proportion of correct links, out of all links made. You can think of it as the probability that a randomly chosen link is correct. You can get the false positive rate from this as:

$$1 - (\text{precision rate}).$$

Stats NZ measures the precision rate, usually via clerical reviews of random samples of the links.

```
>>> Precision rate
```

The precision rate is the proportion of correct links, out of all links made. You can think of it as the probability that a randomly chosen link is correct. You can get the false positive rate from this as:

$$1 - (\text{precision rate}).$$

Stats NZ measures the precision rate, usually via clerical reviews of random samples of the links.

The priority of Stats NZ is to achieve a high precision rate. This involves a trade-off, with a *higher false negative rate*.

```
>>> Linkage bias
```

Linkage bias examines variables where the false negative rate is particularly high. For example, **bias in year of birth** is expected since older people have lived through longer periods of poor coverage, creating a linking bias. However, it is not easy to look at linked records versus records that didn't link, so estimating linkage bias is difficult.

>>> One-to-one relationship

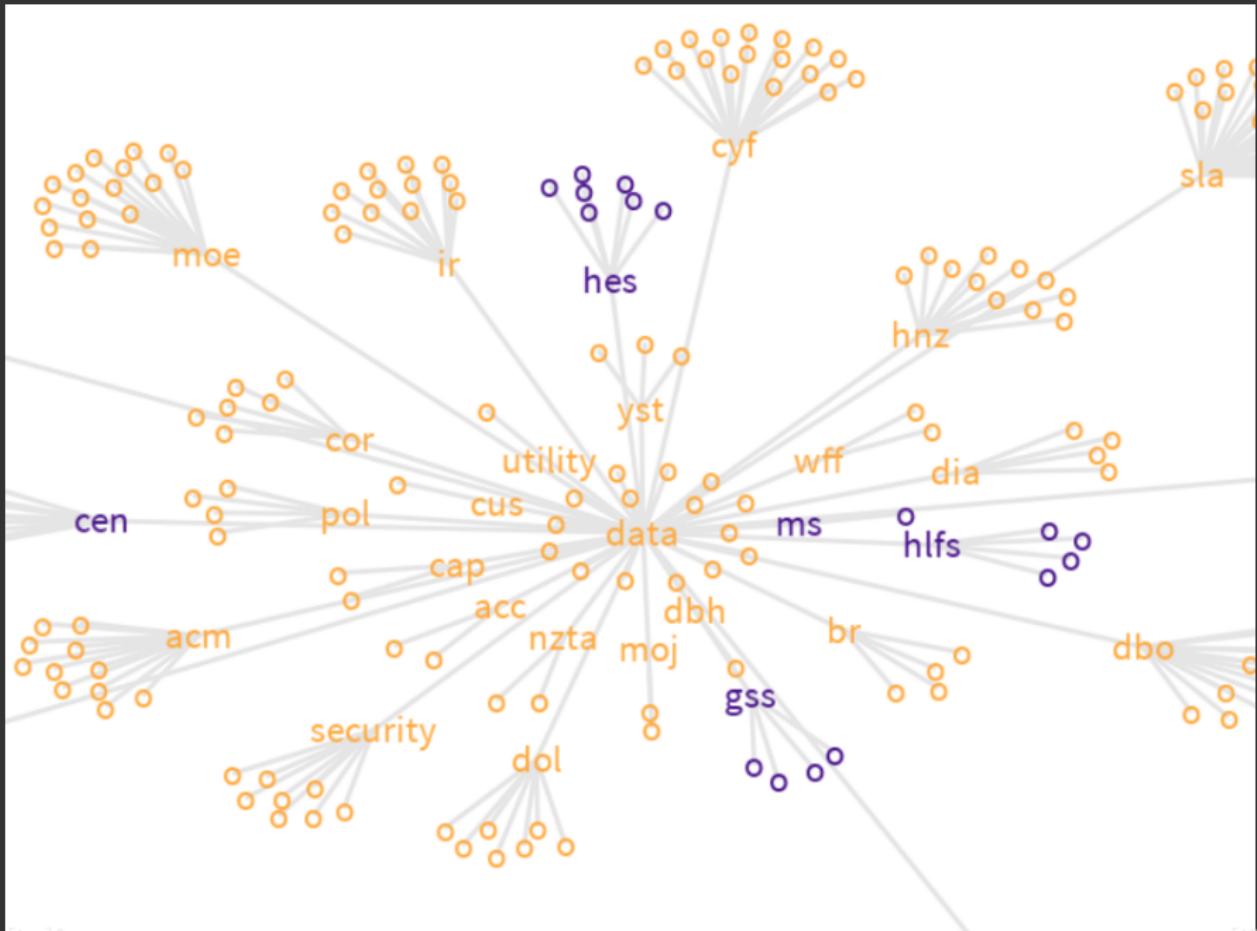
When linkage is created between a schema and the spine, Stats NZ refers to it as a **project**.

"Each project ideally produces **one-to-one links**, where each record on one side links to at most one record on the other side. Duplicates are records which link to more than one record. How these are handled in the IDI depends on the projects."

>>> My favourite website links

- * Stats NZ paper: Use of the IDI "The first part of this report sets out to describe, from a researcher's point of view, what data is available, how it is structured, and the analytical platforms that are available"
- * VHIN guides to getting started "We have created a number of guides to help users to get started with the IDI and to understand some of the different types of data included in it. These are continually evolving and being added to"

```
>>> IDI_Clean schemas (some go off the page)
```



>>> Where are we now?

Day 2

1. Revision of Day 1
2. Reading the docs
3. Lots of exercises!

LUNCH!

4. Details about the IDI
5. Creating and editing tables in SQL
6. Connecting to SQL from R
7. Time for more SQL?
8. More exercises!

SEND ME QUESTIONS AND GIVE FEEDBACK!

>>> We will learn to

- * CREATE SCHEMA
- * CREATE TABLE
- * INSERT INTO table
- * UPDATE entries
- * ALTER (to add columns)
- * SELECT INTO a table

We'll learn all of these as templates rather than look at them as pieces to mix and match like we did with SELECT stuff earlier.

>>> You may need to CREATE SCHEMA first

```
CREATE SCHEMA MySchema;  
GO
```

You should use GO after every chunk of code that creates, updates, or deletes a table.

```
>>> CREATE TABLE
```

```
CREATE TABLE MySchema.Table1 (
    pkey int not null,
    var1 int,
    var2 varchar(50),
    var3 bit,
    var4 char(1),
    PRIMARY KEY (pkey)
);
GO
```

```
>>> CREATE TABLE with foreign key
```

```
CREATE TABLE MySchema.Table2 (
    pkey int not null,
    fkey int,
    PRIMARY KEY (pkey),
    FOREIGN KEY (fkey)
    REFERENCES MySchema.Table1
);
GO
```

Pay attention to the commas!

>>> Exercise

1. On a piece of paper, write down the commands to create a schema in `IDI_Sandpit`. For the name of the schema, use your **first name**.
2. Come up with two tables of your own that have a relationship between them. Give them a few columns. Do not come up with any records for the tables, just the tables themselves. Be creative!
3. Write the commands to create the two tables. Be sure to include primary and foreign keys as needed, and select appropriate data types.
4. Copy your commands into SSMS and run them. Fingers crossed!

```
>>> INSERT INTO table
```

```
INSERT INTO MySchema.Table1  
(pkey, var1, var2, var3, var4)  
VALUES  
(1, 123, 'something' , 0, 'y'),  
(2, 321, 'something else', 1, 'n'),  
(3, 764, 'words here' , 0, 'y');  
GO
```

```
>>> INSERT INTO table
```

```
INSERT INTO MySchema.Table1  
(pkey, var1, var2, var3, var4)  
VALUES  
(1, 123, 'something' , 0, 'y'),  
(2, 321, 'something else', 1, 'n'),  
(3, 764, 'words here' , 0, 'y');  
GO
```

Sometimes it's best to ignore the annoying red underlines in SSMS

```
>>> INSERT INTO table
```

```
    INSERT INTO MySchema.Table2
    (pkey, fkey)
VALUES
    (1, 1),
    (2, 1),
    (3, 1);
GO
```

Be careful: Make sure that each foreign key entry is actually equal to an existing primary key entry in the table that the foreign key points to.

>>> Exercise

Using SSMS, insert at least three rows of data into each of the two tables that you created in the last exercise.

```
>>> UPDATE table
```

```
    UPDATE MySchema.Table1  
    SET var2 = 'updated something'  
    WHERE var2 = 'something';  
    GO
```

You can use **any search condition** in the WHERE clause, as usual.

```
>>> UPDATE table
```

```
UPDATE MySchema.Table1
SET var4 = (CASE
              WHEN (var4 = 'y') THEN 'Y'
              WHEN (var4 = 'n') THEN 'N'
              ELSE (var4)
            END);
GO
```

You can use CASEs in the SET clause, like the above. However, when updating a variable you need to be wary of the data type.

```
>>> ALTER table
```

If you want to UPDATE a table with a new data type then you need to add a new column to the table first, with the data type you want to use.

```
ALTER TABLE MySchema.Table1  
ADD NewVar char(3);  
GO
```

ALTER will fill the new column with NULLs for now.

>>> Now we can do this

```
UPDATE MySchema.Table1
SET NewVar = (CASE
    WHEN (var4 = 'Y') THEN 'yes'
    WHEN (var4 = 'N') THEN 'no'
    ELSE (var4)
END);
GO
```

>>> Exercise

ALTER one of the tables that you created earlier to add a new column with a data type of your choice. Then, UPDATE the table, using CASEs to add values to the new column based on the values in an existing column of your choice.

```
>>> SELECT INTO a table
```

The result of any SELECT query can be stored in
a **new or existing** table using SELECT INTO

```
SELECT FirstName, LastName  
INTO MySchema.MyFriendsNames  
FROM Notes.Friends;  
GO
```

>>> Where are we now?

Day 2

1. Revision of Day 1
2. Reading the docs
3. Lots of exercises!

LUNCH!

4. Details about the IDI
5. Creating and editing tables in SQL
6. Connecting to SQL from R
7. Time for more SQL?
8. More exercises!

SEND ME QUESTIONS AND GIVE FEEDBACK!

```
>>> Connect to SQL from R
```

This will be a walk-through from the notes, if we have time.

>>> Where are we now?

Day 2

1. Revision of Day 1
2. Reading the docs
3. Lots of exercises!

LUNCH!

4. Details about the IDI
5. Creating and editing tables in SQL
6. Connecting to SQL from R
7. Time for more SQL?
8. More exercises!

SEND ME QUESTIONS AND GIVE FEEDBACK!

>>> SQL operator: OUTER JOIN

An OUTER JOIN allows us to join two tables but to **keep all the rows of one of the tables**, even if there are no matching records.

>>> Remember this?

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

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

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

```
>>> The result was...
```

Unnamed						
PetID	PetName	...	FriendID	FriendID	FirstName	...
1	Chikin	...	2	2	Y	...
2	Cauchy	...	3	3	Z	...
3	Gauss	...	3	3	Z	...

>>> SQL operator: OUTER JOIN

If we did an OUTER JOIN instead we would get:

Unnamed						
PetID	PetName	...	FriendID	FriendID	FirstName	...
NULL	NULL	...	NULL	1	X	...
1	Chikin	...	2	2	Y	...
2	Cauchy	...	3	3	Z	...
3	Gauss	...	3	3	Z	...

```
>>> SQL operator: OUTER JOIN
```

If we did an OUTER JOIN instead we would get:

Unnamed						
PetID	PetName	...	FriendID	FriendID	FirstName	...
NULL	NULL	...	NULL	1	X	...
1	Chikin	...	2	2	Y	...
2	Cauchy	...	3	3	Z	...
3	Gauss	...	3	3	Z	...

>>> SQL operator: JOIN

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

>>> SQL operator: OUTER JOIN

```
SELECT *
FROM Notes.Friends F
 $\textcircled{LEFT}$  JOIN Notes.Pets P
ON F.FriendID = P.FriendID;
```

>>> SQL operator: OUTER JOIN

Keep

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

>>> SQL operator: OUTER JOIN

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

>>> SQL operator: OUTER JOIN

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

Keep

>>> SQL operator: OUTER JOIN

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

Since there are no pets that don't belong to friends, the
RIGHT JOIN has no effect here

>>> Where are we now?

Day 2

1. Revision of Day 1
2. Reading the docs
3. Lots of exercises!

LUNCH!

4. Details about the IDI
5. Creating and editing tables in SQL
6. Connecting to SQL from R
7. Time for more SQL?
8. More exercises!

SEND ME QUESTIONS AND GIVE FEEDBACK!

>>> Exercises from notes

22-61

(also get your USB files)

>>> IDI Worksheet

On handouts and USB

>>> IDI group exercise

Write a query (or two) that list(s) the snz_uid and birth year of all people who have registered a civil union with DIA and registered a serious injury with ACC.

```
>>> Solution
```

```
SELECT SI.snz_uid, CU.dia_civ_partnr1_birth_year_nbr
FROM dia_clean.civil_unions CU,
     ACC_Clean.Serious_Injury SI
WHERE CU.partnr1_snz_uid = SI.snz_uid;
```

```
SELECT SI.snz_uid, CU.dia_civ_partnr2_birth_year_nbr
FROM dia_clean.civil_unions CU,
     ACC_Clean.Serious_Injury SI
WHERE CU.partnr2_snz_uid = SI.snz_uid;
```

```
>>> Solution
```

```
SELECT SI.snz_uid, CU.dia_civ_partnr1_birth_year_nbr
FROM dia_clean.civil_unions CU,
     ACC_Clean.Serious_Injury SI
WHERE CU.partnr1_snz_uid = SI.snz_uid
UNION
SELECT SI.snz_uid, CU.dia_civ_partnr2_birth_year_nbr
FROM dia_clean.civil_unions CU,
     ACC_Clean.Serious_Injury SI
WHERE CU.partnr2_snz_uid = SI.snz_uid;
```

```
>>> Another use of CASE
```

Earlier, we used CASE when updating a table. We can also use CASE in a select statement.

```
SELECT FirstName,  
       (CASE  
           WHEN (FavColour = 'red') THEN 'fool'  
           WHEN (FavColour = 'blue') THEN 'smart'  
           WHEN (FavColour = 'green') THEN 'okay'  
           ELSE (FavColour)  
       END)  
FROM Notes.Friends;
```

>>> Many other functions

There are many other functions that allow you to change the values of entries before your select statement returns them.

- * Full collection of them
- * Mathematical functions (see ROUND, ABS, RAND)
- * Date and time functions (see DAY, MONTH, YEAR, DATEDIFF)
- * String functions (see CONCAT and SOUNDEX)

I'll demonstrate these in SSMS now.