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

Daniel Fryer [†]

Dec, 2020

[†]daniel@vfryer.com

>>> Where are we now?

Day 2

1. Revision of day 1
2. Reading the docs
3. Lots of exercises

   Lunch!

4. The IDI
5. Creating and editing tables
6. Connecting to SQL from R
7. Bonus material?
8. More exercises

   Send me questions and give feedback

Page is hyperlinked: click a topic above to jump to it.

>>> Revision of day 1

Revision Kahoot

How to tell if something is a primary key?

Let's look at some data dictionaries

>>> Revision of day 1

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. The IDI
5. Creating and editing tables
6. Connecting to SQL from R
7. Bonus material?
8. More exercises

   Send me questions and give feedback

Page is hyperlinked: click a topic above to jump to it.

>>> 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.

>>> The fifth Kahoot

. . .

>>> Group practice

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

>>> Where are we now?

Day 2

1. Revision of day 1
2. Reading the docs
3. Lots of exercises

   Lunch!

4. The IDI
5. Creating and editing tables
6. Connecting to SQL from R
7. Bonus material?
8. More exercises

   Send me questions and give feedback

Page is hyperlinked: click a topic above to jump to it.

>>> Exercises

Get started on exercises: 22-61

Click here to find the textbook.

>>> Where are we now?

Day 2

1. Revision of day 1
2. Reading the docs
3. Lots of exercises

   Lunch!

4. The IDI
5. Creating and editing tables
6. Connecting to SQL from R
7. Bonus material?
8. More exercises

   Send me questions and give feedback

Page is hyperlinked: click a topic above to jump to it.

>>> 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

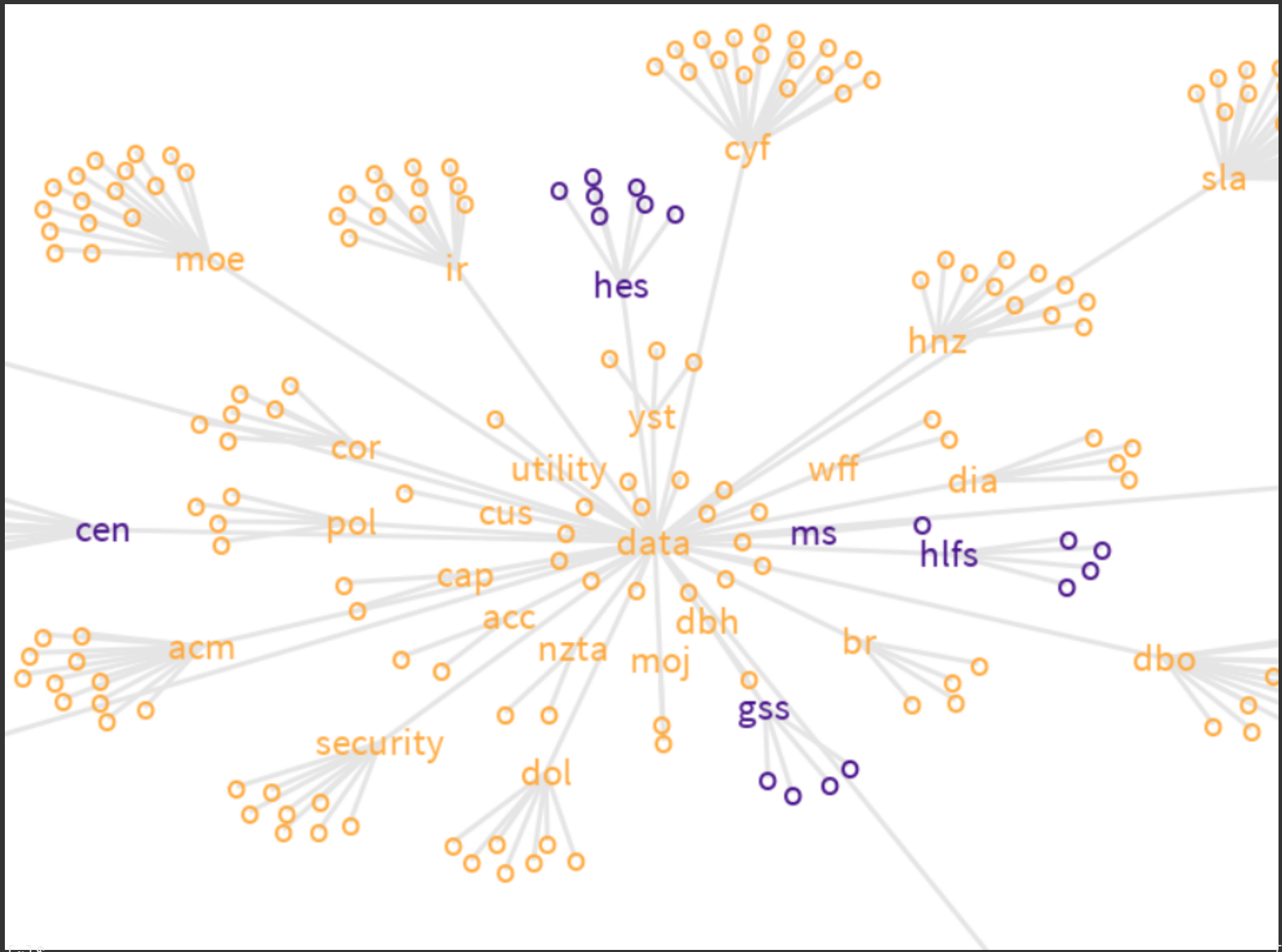
When linkage is created between a schema and the spine, Stats NZ refers to it as a project.

"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. The IDI
5. Creating and editing tables
6. Connecting to SQL from R
7. Bonus material?
8. More exercises

   Send me questions and give feedback

Page is hyperlinked: click a topic above to jump to it.

>>> 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 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. The IDI
5. Creating and editing tables
6. Connecting to SQL from R
7. Bonus material?
8. More exercises

   Send me questions and give feedback

Page is hyperlinked: click a topic above to jump to it.

>>> 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. The IDI
5. Creating and editing tables
6. Connecting to SQL from R
7. Bonus material?
8. More exercises

   Send me questions and give feedback

Page is hyperlinked: click a topic above to jump to it.

>>> 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	…

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	…

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

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

>>> SQL operator: OUTER JOIN

```
SELECT *
FROM Notes.Friends F
     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

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

>>> SQL operator: OUTER JOIN

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

Keep

```sql
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. The IDI
5. Creating and editing tables
6. Connecting to SQL from R
7. Bonus material?
8. More exercises

   Send me questions and give feedback

Page is hyperlinked: click a topic above to jump to it.

>>> Exercises from notes

Finish off exercises: 22-61

Click here to find the textbook.

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.