

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

Daniel Fryer [†]

Nov, 2020

[†]daniel@vfryer.com

>>> Where are we now?

Day 2

1. Revision of day 1
2. Reading the docs
3. Aggregating
4. Big reveal and exercises

Lunch!

5. Creating and editing tables
6. The Integrated Data Infrastructure
7. Putting it all together
8. Connecting and exporting

Send me questions and give feedback

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

```
>>> Revision of day 1
```

Revision Kahoot

```
>>> Revision of day 1 JOIN
```

Do Exercises Section 5.2.4

[Click here to find the textbook.](#)

>>> Where are we now?

Day 2

1. Revision of day 1
 2. Reading the docs
 3. Aggregating
 4. Big reveal and exercises
- Lunch!
5. Creating and editing tables
 6. The Integrated Data Infrastructure
 7. Putting it all together
 8. Connecting and exporting
- 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')

>>> Exercises

Do Exercises Section 5.2.5

[Click here to find the textbook.](#)

>>> Where are we now?

Day 2

1. Revision of day 1
2. Reading the docs
3. Aggregating
4. Big reveal and exercises

Lunch!

5. Creating and editing tables
6. The Integrated Data Infrastructure
7. Putting it all together
8. Connecting and exporting

Send me questions and give feedback

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

>>> 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		
<i>A</i>	<i>B</i>	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		
<i>A</i>	<i>B</i>	Num
{a, a}	b	{1, 3}
{a, a}	c	{2, 4}

* GROUP BY A

Unnamed		
<i>A</i>	<i>B</i>	Num
a	{b, c, b, c}	{1, 2, 3, 4}

>>> Solutions

* GROUP BY B

Unnamed		
<i>A</i>	<i>B</i>	Num
{a, a}	b	{1, 3}
{a, a}	c	{2, 4}

* GROUP BY A

Unnamed		
<i>A</i>	<i>B</i>	Num
a	{b, c, b, c}	{1, 2, 3, 4}

* GROUP BY A, B

Unnamed		
<i>A</i>	<i>B</i>	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;
```

Column name 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.

>>> Correlated subquery group exercise

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

Two questions:

1. What does the query do?
2. Why don't we use HAVING?

>>> Correlated subquery group exercise

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

Two questions:

1. What does the query do?

Returns the name of every person whose age is greater than the average for their own gender.

2. Why don't we use HAVING?

>>> Correlated subquery group exercise

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

Two questions:

1. What does the query do?

Returns the name of every person whose age is greater than the average for their own gender.

2. Why don't we use HAVING?

There is no aggregation function in the search condition.

>>> Correlated subquery group exercise

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

Two questions:

1. What does the query do?

Returns the name of every person whose age is greater than the average for their own gender.

2. Why don't we use HAVING?

There is no aggregation function in the search condition.

We will come back to this during the exercises.

>>> Where are we now?

Day 2

1. Revision of day 1
2. Reading the docs
3. Aggregating
4. Big reveal and exercises

Lunch!

5. Creating and editing tables
6. The Integrated Data Infrastructure
7. Putting it all together
8. Connecting and exporting

Send me questions and give feedback

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

>>> A big reveal! 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.

>>> A big reveal! 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 others don't take so long to learn

>>> A big reveal! 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 others don't take so long to learn

>>> A big reveal! 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 others don't take so long to learn

>>> Group practice

Find the T-SQL documentation for `ORDER BY` ([click here](#)).
Figure out what it does, but try to keep it simple!

>>> Group practice

Find the T-SQL documentation for [INTO](#) (click here). Figure out what it does, but try to keep it simple!

>>> Group practice

Find the T-SQL documentation for [UNION](#) (click here). Figure out what it does, but try to keep it simple!

>>> Casting - aggregation warning!

Up to this point we have largely ignored data types. This can go on no longer. Arithmetic with integers always returns an integer (by rounding down).

$$\text{AVG}(\{1,2\}) = 1$$

So we need to use CAST in such cases.

>>> Casting - aggregation warning!

Up to this point we have largely ignored data types. This can go on no longer. Arithmetic with integers always returns an integer (by rounding down).

$$\text{AVG}(\{1,2\}) = 1$$

So we need to use CAST in such cases. For example, in Notes.RandomPeople, the data type for Age is integer.

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

Must be changed to:

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

>>> Exercises

Do Exercises Sections 5.2.6 and 5.2.7

[Click here to find the textbook.](#)

>>> Where are we now?

Day 2

1. Revision of day 1
2. Reading the docs
3. Aggregating
4. Big reveal and exercises

Lunch!

5. Creating and editing tables
6. The Integrated Data Infrastructure
7. Putting it all together
8. Connecting and exporting

Send me questions and give feedback

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

```
>>> We will learn to
```

- * CREATE SCHEMA to store your tables

```
>>> We will learn to
```

- * CREATE SCHEMA to store your tables
- * SELECT INTO to store result in a table

```
>>> We will learn to
```

- * CREATE SCHEMA to store your tables
- * SELECT INTO to store result in a table
- * ALTER to add columns to a stored table

>>> We will learn to

- * CREATE SCHEMA to store your tables
- * SELECT INTO to store result in a table
- * ALTER to add columns to a stored table
- * UPDATE to change the entries in a table

>>> We will learn to

- * CREATE SCHEMA to store your tables
- * SELECT INTO to store result in a table
- * ALTER to add columns to a stored table
- * UPDATE to change the entries in a table

- * CREATE TABLE to make one from scratch

```
>>> We will learn to
```

- * CREATE SCHEMA to store your tables
- * SELECT INTO to store result in a table
- * ALTER to add columns to a stored table
- * UPDATE to change the entries in a table
- * INSERT INTO to create a whole record
- * CREATE TABLE to make one from scratch

```
>>> CREATE SCHEMA to store your tables
```

```
CREATE SCHEMA MySchema;  
GO
```

Use `GO` after every chunk of code that creates, updates, or deletes a table. It tells SQL Server to execute the code.

```
>>> SELECT INTO to store result in a table
```

Any `SELECT` result will be stored - creates a table!

```
SELECT friendID, firstName, lastName  
INTO MySchema.MyFriends  
FROM Notes.Friends;  
GO
```

```
>>> ALTER to add columns to a stored table
```

We can create a column to hold new or transformed data.

```
ALTER TABLE MySchema.MyFriends  
ADD initials char(2);  
GO
```

ALTER fills the column with NULLs for now.

>>> Now we can do this

```
UPDATE MySchema.MyFriends
SET initials = CONCAT(
    SUBSTRING(firstName, 1, 1),
    SUBSTRING.lastName, 1, 1)
)
WHERE firstName IS NOT NULL
AND lastName IS NOT NULL;
GO
```

See the [SUBSTRING docs \(click here\)](#)
and the [CONCAT docs \(click here\)](#)

```
>>> INSERT INTO to create a whole record
```

Hold on! What about NULL values?

It's hard to tell from the docs exactly what SUBSTRING and CONCAT do with NULL values. So, let's experiment.

```
>>> INSERT INTO to create a whole record
```

Hold on! What about NULL values?

It's hard to tell from the docs exactly what SUBSTRING and CONCAT do with NULL values. So, let's experiment.

```
INSERT INTO MySchema.MyFriends  
(firstName, lastName, initials)  
VALUES  
(NULL,NULL,NULL);
```

```
>>> INSERT INTO to create a whole record
```

Hold on! What about NULL values?

It's hard to tell from the docs exactly what SUBSTRING and CONCAT do with NULL values. So, let's experiment.

```
INSERT INTO MySchema.MyFriends  
(firstName, lastName, initials)  
VALUES  
(NULL,NULL,NULL);
```

```
UPDATE MySchema.MyFriends  
SET initials = CONCAT(  
    SUBSTRING(firstName, 1, 1),  
    SUBSTRING(lastName, 1, 1)  
    );  
GO
```

```
>>> INSERT INTO to create a whole record
```

Hold on! What about NULL values?

It's hard to tell from the docs exactly what SUBSTRING and CONCAT do with NULL values. So, let's experiment.

```
INSERT INTO MySchema.MyFriends  
(firstName, lastName, initials)  
VALUES  
(NULL,NULL,NULL);
```

```
UPDATE MySchema.MyFriends  
SET initials = CONCAT(  
    SUBSTRING(firstName, 1, 1),  
    SUBSTRING(lastName, 1, 1)  
    );  
GO
```

```
SELECT *  
FROM MySchema.MyFriends;
```

```
>>> CREATE TABLE to make one from scratch
```

```
CREATE TABLE MySchema.MyTable(  
    pkey int not null ,  
    var1 Float ,  
    var2 varchar(50) ,  
    var3 bit not null ,  
    var4 char(1) ,  
    initials int ,  
    PRIMARY KEY (pkey) ,  
    FOREIGN KEY (initials) REFERENCES MySchema.MyFriends (initials)  
);  
GO
```

>>> Solution?

Check the [documentation \(click here\)](#)

```
ALTER TABLE MySchema.MyFriends
ADD CONSTRAINT PK_MyFriends_initials
PRIMARY KEY CLUSTERED (initials);
```

Anything wrong with this?

>>> Better solution

```
ALTER TABLE MySchema.MyFriends  
ADD CONSTRAINT PK_MyFriends_initials  
PRIMARY KEY CLUSTERED (friendID);
```

```
>>> Class exercise
```

Change `MySchema.MyFriends` so that its `friendID` column can be referenced by the foreign key of `MySchema.MyTable`. There are a number of ways to do this. You may decide to:

```
DROP TABLE MySchema.MyFriends;
```

>>> 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!

>>> 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.

```
>>> 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 `CASE` in the `SET` clause, like the above. However, when updating a variable you need to be wary of the data type. `CASE` can also be used in the `SELECT` clause.

>>> Where are we now?

Day 2

1. Revision of day 1
2. Reading the docs
3. Aggregating
4. Big reveal and exercises

Lunch!

5. Creating and editing tables
6. The Integrated Data Infrastructure
7. Putting it all together
8. Connecting and exporting

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 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

Very beginner friendly.

- * Data in the IDI

Visual summary of available data as at September 2020.

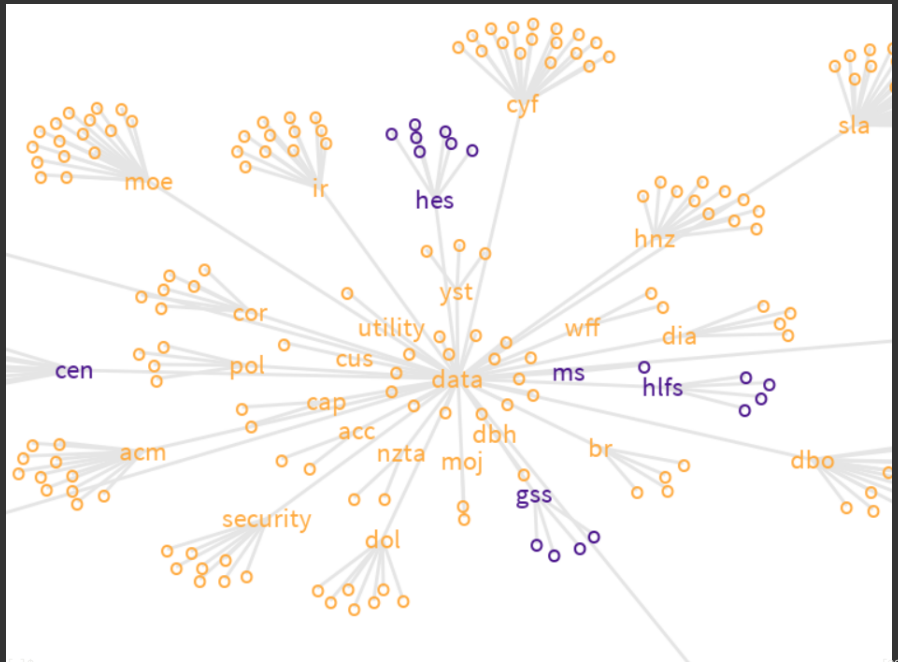
- * Current StatsNZ website

Extensive and introductory material.

- * Older StatsNZ website (may be moving soon)

Contains data dictionaries.

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



```
>>> Primary and foreign key pairs?
```

Well, yes and no

Let's look at some data dictionaries ([click here](#))

```
>>> Primary and foreign key pairs?
```

Well, yes and no

Let's look at some data dictionaries ([click here](#))

ACC Injury Claims Data: Serious Injury

- * `snz_uid`

Global, refreshed, not unique in table

- * `snz_acc_uid`

- * `snz_acc_claim_uid`

```
>>> Primary and foreign key pairs?
```

Well, yes and no

Let's look at some data dictionaries ([click here](#))

ACC Injury Claims Data: Serious Injury

- * `snz_uid`

Global, refreshed, not unique in table

- * `snz_acc_uid`

Local, not refreshed, not unique in table

- * `snz_acc_claim_uid`


```
>>> Primary and foreign key pairs?
```

Well, yes and no

Let's look at some data dictionaries ([click here](#))

ACC Injury Claims Data: Serious Injury

- * `snz_uid`

Global, refreshed, not unique in table

- * `snz_acc_uid`

Local, not refreshed, not unique in table

- * `snz_acc_claim_uid`

Local, 'event ID', unique in table? maybe not.

```
>>> Primary and foreign key pairs?
```

Well, yes and no

Let's look at some data dictionaries ([click here](#))

ACC Injury Claims Data: Serious Injury

- * `snz_uid`

Global, refreshed, not unique in table

- * `snz_acc_uid`

Local, not refreshed, not unique in table

- * `snz_acc_claim_uid`

Local, 'event ID', unique in table? maybe not.

The above 3 together form the primary key

```
>>> Identifying a primary / foreign key pair?
```

A foreign key is any column (or collection of columns) whose each entry is **guaranteed to be** equal to one, and only one, primary key entry in some other table that the foreign key is designated to 'point at'.

If the database administrator has not defined foreign keys, then you need to find natural relationships yourself. Then, write your own tests.

>>> An example to contemplate

Suppose the **Houses** table has a column PostCode,
but no column SuburbName.

Suppose the **Suburbs** table has two columns,
SuburbName and PostCode.

>>> An example to contemplate

Suppose the **Houses** table has a column PostCode,
but no column SuburbName.

Suppose the **Suburbs** table has two columns,
SuburbName and PostCode.

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

>>> An example to contemplate

Suppose the **Houses** table has a column **PostCode**,
but no column **SuburbName**.

Suppose the **Suburbs** table has two columns,
SuburbName and **PostCode**.

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

We will come back to this during the exercises.

>>> Where are we now?

Day 2

1. Revision of day 1
2. Reading the docs
3. Aggregating
4. Big reveal and exercises

Lunch!

5. Creating and editing tables
6. The Integrated Data Infrastructure
7. Putting it all together
8. Connecting and exporting

Send me questions and give feedback

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

>>> Exercises

Do Exercises Section 5.2.8

[Click here to find the textbook.](#)

>>> Where are we now?

Day 2

1. Revision of day 1
2. Reading the docs
3. Aggregating
4. Big reveal and exercises

Lunch!

5. Creating and editing tables
6. The Integrated Data Infrastructure
7. Putting it all together
8. Connecting and exporting

Send me questions and give feedback

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

```
>>> Connect to SQL
```

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

```
>>> 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.