>>> Introduction to SQL
>>> Featuring MySQL and T-SQL

Daniel Fryer [†]

Feb, 2023

[†]daniel@vfryer.com

>>> Daily schedule

Timetable				
9:00am	-	10:30am	lecture 1	(1.5 hr)
10:30am	-	11:00am	morning tea	(30 min)
11:00am	-	12:30pm	lecture 2	(1.5 hr)
12:30pm	-	1:30pm	lunch	(1 hr)
1:30pm	-	3:00pm	guided exercises	(1.5 hr)
3:00pm	-	5:00pm	one-on-one help	(2 hr)

>>> Where are we now?

Day 2

1. Search conditions
2. Reading the docs
3. Aggregating
4. Subqueries
5. Windowing

   Send me questions and give feedback

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

>>> Search conditions

Search conditions appear in a WHERE clause.
They check which rows match the conditions you specify,
by making use of:

* Comparison operators
* Logical operators
* Other operators

>>> Search conditions

Search conditions appear in a WHERE clause.
They check which rows match the conditions you specify,
by making use of:

* Comparison operators
* Logical operators
* Other operators

Definition
A comparison operator is used to compare two things
and return true, false or NULL.

* Click here for comparison operators in the T-SQL docs.
* Click here for comparison operators in the MySQL docs.

>>> Search conditions

Search conditions appear in a WHERE clause.
They check which rows match the conditions you specify,
by making use of:

- Comparison operators
- Logical operators
- Other operators

Definition
Logical operators compare true, false or NULL
and also return true, false or NULL.

- Click here for logical operators in the T-SQL docs.
- Click here for logical operators in the MySQL docs.

>>> Search conditions

Search conditions appear in a WHERE clause.
They check which rows match the conditions you specify,
by making use of:

- Comparison operators
- Logical operators
- Other operators

Definition
MySQL and T-SQL disagree on where to put this category of operators. I just call them 'other operators'. They include IN, LIKE, BETWEEN, EXISTS and more.

- Click here for other operators in the T-SQL docs.
- Click here for other operators in the MySQL 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

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

>>> Other operators

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

>>> Other operators

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

>>> Other operators

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

>>> Other operators

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

>>> Other 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%'`

>>> Other 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 'b__%'`

>>> Operator precedence

Precedence	Operators
1	Anything in round brackets
2	=,<,>,<=,>=,!=,!<,!> (comparison operators)
3	NOT
4	AND
5	OR, ALL, ANY, SOME, EXISTS, BETWEEN, IN, LIKE

>>> Examples

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

>>> Solution

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

>>> Solution

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

>>> Solution

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

>>> Solution

1. `false`
2. `true OR false`
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)

>>> Solution

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

>>> Solution

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

>>> Solution

1. false
2. true
3. true
4. false

>>> Perils of operator precedence

```
-- this one evaluates to FALSE
1 != 1 AND (2 < 3 OR 3 = 3)

-- but this one evaluates to TRUE
1 != 1 AND 2 < 3 OR 3 = 3
```

>>> Perils of operator precedence

```
-- this one evaluates to FALSE
1 != 1 AND (2 < 3 OR 3 = 3)

-- but this one evaluates to TRUE
1 != 1 AND 2 < 3 OR 3 = 3
```

More concretely, consider the following two:

```
-- matches 50 or 60 year old females only
Gender = 'F' AND (Age = 50 OR Age = 60)

-- matches 50 year old females, or anyone aged 60
Gender = 'F' AND Age = 50 OR Age = 60
```

>>> 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.

Enjoy.

>>> Where are we now?

Day 2

1. Search conditions
2. Reading the docs
3. Aggregating
4. Subqueries
5. Windowing

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, the documentation can sense fear

>>> Data Manipulation Language (DML)

Later, we will learn some Data Definition Language (DDL).

* Click here for MySQL DML docs
* Click here for T-SQL DML docs

>>> Data Manipulation Language (DML)

Later, we will learn some Data Definition Language (DDL).

* Click here for MySQL DML docs
* Click here for T-SQL DML docs

Wait, what have we been learning?

* Click here for MySQL SELECT docs
* Click here for T-SQL SELECT docs

The word 'query' actually refers to SELECT!

>>> The syntax conventions

Why is it so hard to read? Syntax, used to make the documentation clearer and more succinct.

* Click here for MySQL Syntax Conventions
* Click here for T-SQL Syntax Conventions

>>> The important syntax conventions

MySQL	T-SQL	Description
[a]	[a]	a is optional
{a}	{a}	a is not optional
[a|b]	[a|b]	optionally, choose a or b
{a|b}	{a|b}	not optional, choose a or b
a [, a] ...	a [,...n]	optionally repeat a (with comma)
label	<label>	a label/placeholder

>>> The SELECT clause

```
SELECT [ALL | DISTINCT] select_expr [, select_expr] ...
[ INTO <new_table> ]

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

[ WHERE <search_condition> ]

[ GROUP BY {col_name | expr }, ... ]

[ HAVING <search_condition> ]

[ ORDER BY {order_by_expression [ ASC | DESC ]} [,...n] ]
```

>>> The SELECT clause

Square brackets mean content is optional.

```
SELECT [ALL | DISTINCT] select_expr [, select_expr] ...
[ INTO <new_table> ]

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

[ WHERE <search_condition> ]

[ GROUP BY {col_name | expr }, ... ]

[ HAVING <search_condition> ]

[ ORDER BY {order_by_expression [ ASC | DESC ]} [,...n] ]
```

>>> The SELECT clause

Curly brackets mean content is mandatory.

```
SELECT [ALL | DISTINCT] select_expr [, select_expr] ...
[ INTO <new_table> ]

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

[ WHERE <search_condition> ]

[ GROUP BY {col_name | expr }, ... ]

[ HAVING <search_condition> ]

[ ORDER BY {order_by_expression [ ASC | DESC ]} [,...n] ]
```

>>> The SELECT clause

Vertical line means choose one.

```
SELECT [ALL | DISTINCT] select_expr [, select_expr] ...
[ INTO <new_table> ]

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

[ WHERE <search_condition> ]

[ GROUP BY {col_name | expr }, ... ]

[ HAVING <search_condition> ]

[ ORDER BY {order_by_expression [ ASC | DESC ]} [,...n] ]
```

>>> The SELECT clause

must choose one.

```
SELECT [ALL | DISTINCT] select_expr [, select_expr] ...
[ INTO <new_table> ]

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

[ WHERE <search_condition> ]

[ GROUP BY {col_name | expr }, ... ]

[ HAVING <search_condition> ]

[ ORDER BY {order_by_expression [ ASC | DESC ]} [,...n] ]
```

>>> The SELECT clause

optionally choose one.

```
SELECT [ALL | DISTINCT] select_expr [, select_expr] ...
[ INTO <new_table> ]

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

[ WHERE <search_condition> ]

[ GROUP BY {col_name | expr }, ... ]

[ HAVING <search_condition> ]

[ ORDER BY {order_by_expression [ ASC | DESC ]} [,...n] ]
```

>>> The SELECT clause

Repeat with commas (T-SQL)

```
SELECT [ALL | DISTINCT] select_expr [, select_expr] ...
[ INTO <new_table> ]

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

[ WHERE <search_condition> ]

[ GROUP BY {col_name | expr }, ... ]

[ HAVING <search_condition> ]

[ ORDER BY {order_by_expression [ ASC | DESC ]} [,...n] ]
```

>>> The SELECT clause

Repeat with commas (MySQL)

```
SELECT [ALL | DISTINCT] select_expr [, select_expr] ...
[ INTO <new_table> ]

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

[ WHERE <search_condition> ]

[ GROUP BY {col_name | expr }, ... ]

[ HAVING <search_condition> ]

[ ORDER BY {order_by_expression [ ASC | DESC ]} [,...n] ]
```

>>> The SELECT clause

Also repeat with commas (MySQL)

```
SELECT [ALL | DISTINCT] select_expr [, select_expr] ...
[ INTO <new_table> ]

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

[ WHERE <search_condition> ]

[ GROUP BY {col_name | expr }, ... ]

[ HAVING <search_condition> ]

[ ORDER BY {order_by_expression [ ASC | DESC ]} [,...n] ]
```

>>> The SELECT clause

Labels or placeholders

```
SELECT [ALL | DISTINCT] select_expr [, select_expr] ...
[ INTO <new_table> ]

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

[ WHERE <search_condition> ]

[ GROUP BY {col_name | expr }, ... ]

[ HAVING <search_condition> ]

[ ORDER BY {order_by_expression [ ASC | DESC ]} [,...n] ]
```

>>> A discovery...

```
SELECT [ALL|DISTINCT] FirstName, LastName, FavColour
```

From the MySQL SELECT docs (click here):

> The ALL and DISTINCT modifiers specify whether duplicate rows should be returned. ALL (the default), specifies that all matching rows should be returned, including duplicates. DISTINCT specifies removal of duplicate rows from the result set. It is an error to specify both modifiers. DISTINCTROW is a synonym for DISTINCT.

>>> Read the docs: FROM

T-SQL will often use ::= to define placeholders.

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

where

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

>>> Reading the docs: FROM

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

>>> Reading the docs: FROM

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

* { } curly braces group required items

>>> Reading the docs: FROM

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

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

>>> Reading the docs: FROM

```
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

```
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

```
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

```
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

```
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

```
FROM MyTable, MyOtherTable
```

>>> Reading the docs: FROM

```
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

```
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

```
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

>>> Practice

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

* Hello.

>>> Practice

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

* Hello.
* Hi.

>>> Practice

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

* Hello.
* Hi.
* Hello. Do you love reading the docs?

>>> Practice

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

* Hello.
* Hi.
* Hello. Do you love reading the docs?
* Hi. Do you love reading the docs?

>>> Practice

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

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

>>> Practice

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

* 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?

>>> Practice

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

* 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)

```
Example: FavColour IN ('red')
```

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

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

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

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)

Example: FavColour 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)

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)

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

Enjoy.

>>> Where are we now?

Day 2

1. Search conditions
2. Reading the docs
3. Aggregating
4. Subqueries
5. Windowing

   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 return a single value for each group. They can also discard groups.

We will cover:

1. `GROUP BY` clause
2. Aggregation function
3. `HAVING` clause

>>> Aggregating queries

Aggregating queries collect the rows of a table into groups, and return a single value for each group. They can also discard groups.

We will cover:

1. GROUP BY clause - creates the groups
2. Aggregation function
3. HAVING clause

>>> Aggregating queries

Aggregating queries collect the rows of a table into groups, and return a single value for each group. They can also discard groups.

We will cover:

1. GROUP BY clause - creates the groups
2. Aggregation function - computes one value for each group
3. HAVING clause

>>> Aggregating queries

Aggregating queries collect the rows of a table into groups, and return a single value for each group. They can also discard groups.

We will cover:

1. GROUP BY clause - creates the groups
2. Aggregation function - computes one value for each group
3. HAVING clause - discards groups

The HAVING clause discards groups just like the WHERE clause discards 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 examples.

We will look at the execution of this query:

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

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

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

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

```
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

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

Error: SQL can't be sure the entries are atomic.

>>> 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.
```

>>> 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;
```

```
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

```
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

>>> More examples

Watch the curly braces

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 compute one value for each group.
Aggregating frees us to select more columns.

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 compute one value for each group.
Aggregating frees us to select more columns.

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 compute one value for each group.
Aggregating frees us to select more columns.

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		
PersonName	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		
PersonName	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		
PersonName	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		
PersonName	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		
PersonName	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		
PersonName	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		
PersonName	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		
PersonName	Gender	Age
{Beyoncé, Laura Marling}	F	{37, 28}

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

AVG(RP.Age)

Unnamed		
PersonName	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		
PersonName	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.
3.
4.
5.

>>> 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

T-SQL	MySQL	Purpose
AVG	AVG	Average
STDEV	STDDEV_SAMP	Sample standard deviation
STDEVP	STDDEV_POP	Population standard deviation
VAR	VAR_SAMP	Sample variance
VARP	VAR_POP	Population variance
COUNT	COUNT	Count number of rows
MIN	MIN	Minimum
MAX	MAX	Maximum
SUM	SUM	Sum

See the full list in the T-SQL or MySQL docs (click).

>>> More aggregation functions

T-SQL	MySQL	Purpose
AVG	AVG	Average
STDEV	STDDEV_SAMP	Sample standard deviation
STDEVP	STDDEV_POP	Population standard deviation
VAR	VAR_SAMP	Sample variance
VARP	VAR_POP	Population variance
COUNT	COUNT	Count number of rows
MIN	MIN	Minimum
MAX	MAX	Maximum
SUM	SUM	Sum

See the full list in the T-SQL or MySQL docs (click).

>>> Casting - aggregation warning!

We cannot ignore data types! Arithmetic with integers always returns an integer (by rounding down).

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

So we need to use CAST.

>>> Casting - aggregation warning!

We cannot ignore data types! Arithmetic with integers always returns an integer (by rounding down).

AVG({1,2}) = 1

So we need to use CAST. 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 decimal(5,2))) AS AverageAge
FROM Notes.RandomPeople
GROUP BY gender;
```

>>> SQL clause: HAVING

The HAVING clause was created because WHERE is executed before 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
HAVING AVG(RP.age) > 40;
```

The HAVING clause is like WHERE, but it acts on groups.

>>> SQL clause: HAVING

The HAVING clause was created because WHERE is executed before 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
HAVING AVG(RP.age) > 40;
```

Search condition with aggregation function

The HAVING clause is like WHERE, but it acts on groups.

```
FROM Notes.RandomPeople RP
```

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

```
GROUP BY RP.Gender
```

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

AVG(RP.Age)

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

```
HAVING AVG(RP.Age) > 40
```

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

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

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

result

Unnamed	
Gender	AverageAge
M	44

>>> Order of execution

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

>>> 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.

Enjoy.

>>> Where are we now?

Day 2

1. Search conditions
2. Reading the docs
3. Aggregating
4. Subqueries
5. Windowing

   Send me questions and give feedback

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

>>> Basic subquery

Subqueries (next slide) are powerful with 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 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.

It retrieved the details of all friends who have pets.
Let's execute it to experiment.

>>> Challenge

Can we rewrite it as a join?

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

>>> Subquery anywhere!

Quick note: a subquery does not have to be used only in the `WHERE` clause. It can be used almost anywhere, but we will look at this later.

>>> How about this one?

Find all the people whose Gender has AVG(Age) less than 40.

Easy, right?

>>> How about this one?

Find all the people whose Gender has AVG(Age) less than 40.

Easy, right?

```
SELECT PersonName
FROM RandomPeople
GROUP BY Gender
HAVING AVG(Age) < 40;
```

>>> How about this one?

Find all the people whose Gender has AVG(Age) less than 40.

Easy, right?

```
SELECT PersonName
FROM RandomPeople
GROUP BY Gender
HAVING AVG(Age) < 40;
```

Wrong!

Let's try and execute it.

>>> Can you explain why?

FROM RandomPeople GROUP BY Gender

Unnamed		
PersonName	Gender	Age
(Beyoncé, Laura Marling)	F	(37, 28)
(Darren Hayes, Bret McKenzie)	M	(46, 42)
(Jack Monroe)	NB	(30)

```
SELECT PersonName
FROM RandomPeople
GROUP BY Gender
HAVING AVG(Age) < 40;
```

>>> Use a subquery instead

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

Let's execute it and explore.

>>> Correlated subquery

The hardest thing in introductory SQL...

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

>>> Correlated subquery

The hardest thing in introductory SQL...

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

The notes have a good diagram for this...

>>> A subquery in FROM

What does it do?

```
SELECT Gender, AvgAge
FROM (SELECT Gender, AVG(Age) AS AvgAge
      FROM RandomPeople
      GROUP BY Gender)
WHERE AvgAge < 40;
```

>>> A subquery in FROM

What does it do?

```
SELECT Gender, AvgAge
FROM (SELECT Gender, AVG(Age) AS AvgAge
      FROM RandomPeople
      GROUP BY Gender)
WHERE AvgAge < 40;
```

Can we rewrite it with HAVING?

>>> Where are we now?

Day 2

1. Search conditions
2. Reading the docs
3. Aggregating
4. Subqueries
5. Windowing

   Send me questions and give feedback

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

>>> What is windowing?

* Aggregation functions return one value per group.
* Window functions return one value per row.
* Window functions are not scalar functions.

On the next slide: what are windows?

>>> Windows are... groups

The only reason we don't call them groups is because we use (and create) them differently.

>>> Windows are... groups

The only reason we don't call them groups is because we use (and create) them differently.

Sometimes they're called partitions instead of windows.

* 'Groups' when we return one value per group.
* 'Partitions'/'windows' when we return one value per row.

>>> First few window functions

You ready...?

>>> First few window functions

You ready...?

T-SQL	MySQL	Purpose
AVG	AVG	Average
STDEV	STDDEV_SAMP	Sample standard deviation
STDEVP	STDDEV_POP	Population standard deviation
VAR	VAR_SAMP	Sample variance
VARP	VAR_POP	Population variance
COUNT	COUNT	Count number of rows
MIN	MIN	Minimum
MAX	MAX	Maximum
SUM	SUM	Sum

Danny... what are you talking about??
Those are aggregation functions.

>>> Aggregation functions work on windows

What will this do?

```
SELECT PersonName, MIN(Age) AS MinimumAge
FROM RandomPeople;
```

>>> Aggregation functions work on windows

What will this do?

```
SELECT PersonName, MIN(Age) AS MinimumAge
FROM RandomPeople;
```

Error: MIN made the whole table one group

>>> Aggregation functions work on windows

What will this do?

```
SELECT PersonName, MIN(Age) AS MinimumAge
FROM RandomPeople;
```

Error: MIN made the whole table one group

Here we go

```
SELECT PersonName, MIN(Age) OVER() AS MinimumAge
FROM RandomPeople;
```

Success: OVER() made the whole table one partition.

>>> Aggregation functions work on windows

What will this do?

```
SELECT PersonName, MIN(Age) AS MinimumAge
FROM RandomPeople;
```

Error: MIN made the whole table one group

Here we go

```
SELECT PersonName, MIN(Age) OVER() AS MinimumAge
FROM RandomPeople;
```

Success: OVER() made the whole table one partition.

The aggregation still returns one result per partition, but now that single result gets duplicated for each row.

>>> So how do we form the windows?

The minimum age for each person's gender?

```
SELECT PersonName, MIN(Age) OVER() AS MinimumAge
FROM RandomPeople;
```

>>> So how do we form the windows?

The minimum age for each person's gender?

```
SELECT PersonName, MIN(Age) OVER(PARTITION BY Gender) AS MinimumAge
FROM RandomPeople;
```

So what will this do?

```
SELECT PersonName,
       Age - MIN(Age) OVER(PARTITION BY Gender) AS AgeDiff
FROM RandomPeople;
```

>>> Remember this?

Find all the people whose Gender has AVG(Age) less than 40.

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

Can we rewrite it? Let's do it live.

>>> Thinking about order of operations

We know SELECT executes last, so what will this do?

```
SELECT Gender, MAX(Age) OVER()
FROM Notes.RandomPeople
GROUP BY Gender;
```

>>> Thinking about order of operations

We know SELECT executes last, so what will this do?

```
SELECT Gender, MAX(Age) OVER()
FROM Notes.RandomPeople
GROUP BY Gender;
```

```
Error: MAX is a windowed aggregation function now.
```

* MAX((37,28)) = 37
* MAX((37,28)) OVER() = (37,37)

>>> Thinking about order of operations

We know SELECT executes last, so what will this do?

```
SELECT Gender, MAX(MAX(Age)) OVER()
FROM Notes.RandomPeople
GROUP BY Gender;
```

Fixed: we can add a regular aggregation function inside it.

>>> Thinking about order of operations

We know SELECT executes last, so what will this do?

```
SELECT Gender, MAX(Age) OVER()
FROM Notes.RandomPeople
GROUP BY Gender;
```

Fixed: we can add a regular aggregation function inside it.

Shall I go through it on the blackboard?

Enjoy.

>>> Exercises

Do exercises at the end of Chapter 3.

Click here to open the textbook.