

Practical Application of the SQL SELECT Statement



Using SELECT... FROM...

SELECT... FROM...

So far:

SELECT... FROM...

So far:

- Theory of Relational Databases
- SQL Theory
- Creating SQL Databases and Tables
- Creating SQL Constraints

SELECT... FROM...

So far:

- Theory of Relational Databases
- SQL Theory
- Creating SQL Databases and Tables
- Creating SQL Constraints

In this section:

SELECT... FROM...

So far:

- Theory of Relational Databases
- SQL Theory
- Creating SQL Databases and Tables
- Creating SQL Constraints

In this section:

- Data Manipulation

SELECT... FROM...

- the SELECT statement

SELECT... FROM...

- the SELECT statement

allows you to extract a fraction of the entire data set

SELECT... FROM...

- **the SELECT statement**

allows you to extract a fraction of the entire data set

- used to retrieve data from database objects, like tables

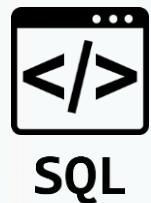
SELECT... FROM...

- **the SELECT statement**

allows you to extract a fraction of the entire data set

- used to retrieve data from database objects, like tables
- used to “*query data from a database*”

SELECT... FROM...



```
SELECT column_1, column_2,... column_n  
FROM table_name;
```

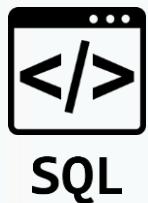
SELECT... FROM...



```
SELECT column_1, column_2,... column_n  
FROM table_name;
```

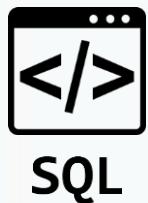
- when extracting information, SELECT goes with FROM

SELECT... FROM...



```
SELECT column_1, column_2,... column_n  
FROM table_name;
```

SELECT... FROM...



```
SELECT column_1, column_2,... column_n  
FROM table_name;
```

```
SELECT first_name, last_name  
FROM employees;
```

SELECT... FROM...

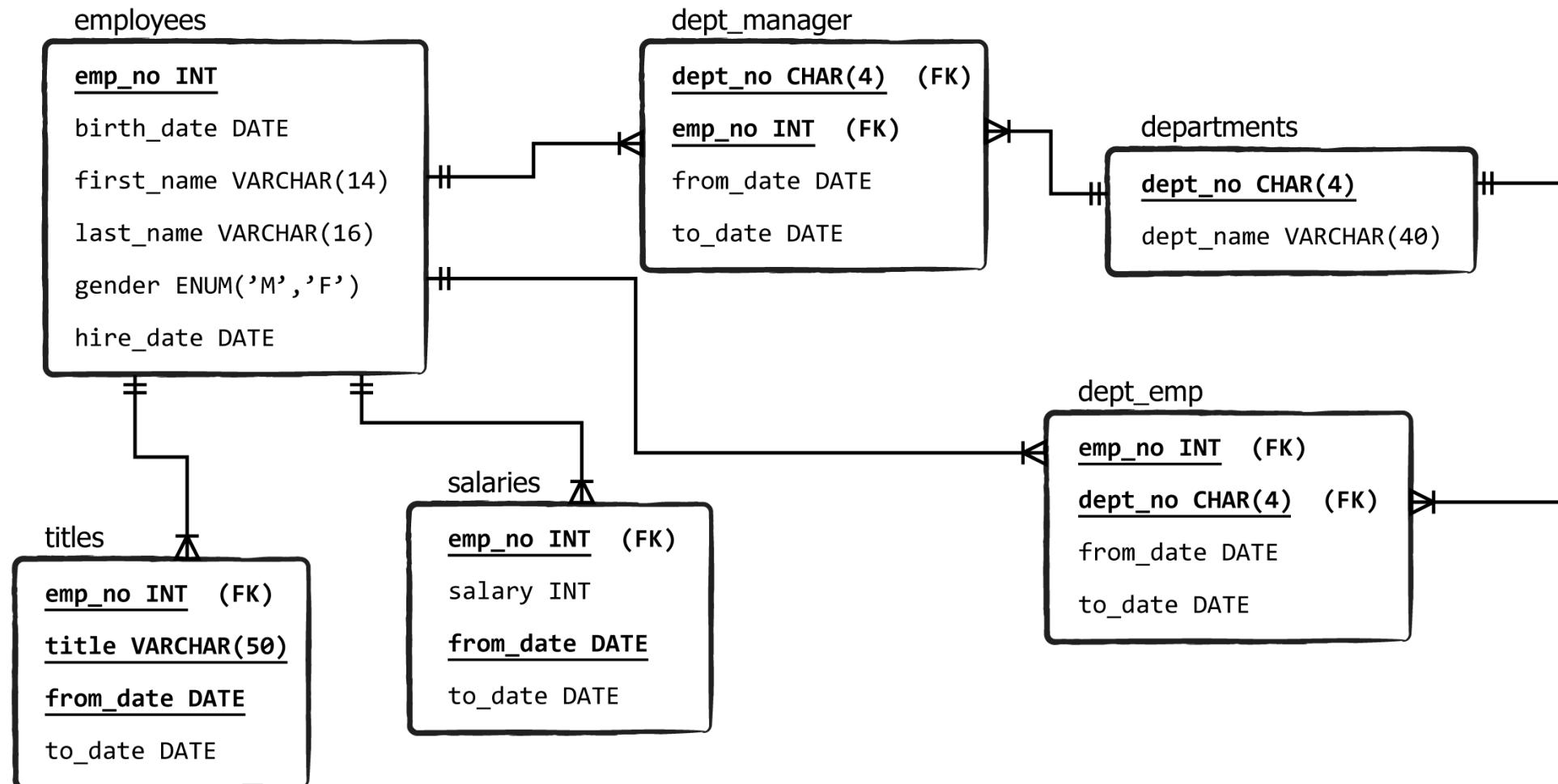


SQL

```
SELECT * FROM employees;
```

* - a wildcard character, means “all” and “everything”

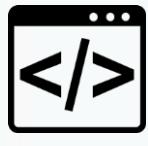
Database: employees





Using WHERE

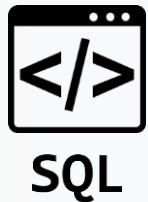
WHERE



SQL

```
SELECT * FROM employees;
```

WHERE



```
SELECT column_1, column_2,... column_n  
FROM table_name;
```

WHERE

- the WHERE clause

WHERE

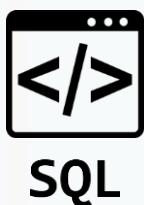
- the WHERE clause

it will allow us to set a condition upon which we will specify what part of the data we want to retrieve from the database

WHERE

- the WHERE clause

it will allow us to set a condition upon which we will specify what part of the data we want to retrieve from the database



```
SELECT column_1, column_2,... column_n  
FROM table_name;
```

WHERE

- the WHERE clause

it will allow us to set a condition upon which we will specify what part of the data we want to retrieve from the database



```
SELECT column_1, column_2,... column_n  
FROM table_name  
WHERE condition;
```



AND

AND

- = equal operator

AND

- **= equal operator**

in SQL, there are many other *Linking keywords and symbols*, called operators, that you can use with the WHERE clause

AND

- **= equal operator**

in SQL, there are many other *Linking keywords and symbols*, called **operators**, that you can use with the WHERE clause

- AND
- OR
- IN - NOT IN
- LIKE - NOT LIKE
- BETWEEN... AND...

AND

- = equal operator

in SQL, there are many other *Linking keywords and symbols*, called operators, that you can use with the WHERE clause

- AND
- OR
- IN - NOT IN
- LIKE - NOT LIKE
- BETWEEN... AND...
- EXISTS - NOT EXISTS
- IS NULL - IS NOT NULL
- comparison operators
- etc.

AND

AND

AND

- **AND**

allows you to logically combine two statements in the condition code block

AND

- **AND**

allows you to logically combine two statements in the condition code block



SQL

```
SELECT column_1, column_2,... column_n  
FROM table_name  
WHERE condition_1 AND condition_2;
```

AND

- AND

allows you to logically combine two statements in the condition code block



SQL

```
SELECT column_1, column_2,... column_n  
FROM table_name  
WHERE condition_1 AND condition_2;
```

- allows us to *narrow* the output we would like to extract from our data



OR

OR

- AND

AND binds SQL to meet both conditions enlisted in the WHERE clause simultaneously



SQL

```
SELECT column_1, column_2,... column_n  
FROM table_name  
WHERE condition_1 AND condition_2;
```

OR

- **AND**

conditions set on *different columns*

OR

- **AND**

conditions set on *different columns*

- **OR**

conditions set on *the same column*



Operator Precedence

Operator Precedence

So far:

Operator Precedence

So far:

- AND
- OR

Operator Precedence

So far:

- AND
- OR

In this lesson:

Operator Precedence

So far:

- AND
- OR

In this lesson:

- the *Logical order* with which you must comply when you use both operators in the same WHERE block

Operator Precedence

- logical operator precedence

Operator Precedence

- **logical operator precedence**

an SQL rule stating that in the execution of the query, the operator **AND** is applied first, while the operator **OR** is applied second

Operator Precedence

- logical operator precedence

an SQL rule stating that in the execution of the query, the operator AND is applied first, while the operator OR is applied second

AND > OR

Operator Precedence

- logical operator precedence

an SQL rule stating that in the execution of the query, the operator AND is applied first, while the operator OR is applied second

AND > OR

regardless of the order in which you use these operators, SQL will always start by reading the conditions around the AND operator



The Three Wildcard Characters

Wildcard Characters

- wildcard characters

%

-

*

you would need a wildcard character whenever you wished to put “anything” on its place

Wildcard Characters

%

- a substitute for a sequence of characters

LIKE ('Mar%')

Mark, Martin, Margaret

_

- helps you match a single character

LIKE ('Mar_')

Mark, Mary, Marl

Wildcard Characters

- * will deliver a list of *all* columns in a table

```
SELECT * FROM employees;
```

- it can be used to count *all* rows of a table

A large conference room with rows of chairs facing a central table. The room has a modern design with a grid-patterned ceiling and large windows overlooking a landscape.

BETWEEN... AND...

BETWEEN... AND...

- **BETWEEN... AND...**

helps us designate the interval to which a given value belongs

BETWEEN... AND...

SELECT

*

FROM

employees

WHERE

hire_date **BETWEEN** '1990-01-01' **AND** '2000-01-01';



SQL

BETWEEN... AND...

SELECT

*

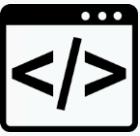
FROM

employees

WHERE

hire_date **BETWEEN** '1990-01-01' **AND** '2000-01-01';

'1990-01-01' AND '2000-01-01' *will be included in the retrieved list of records*



SQL

BETWEEN... AND...

- **NOT BETWEEN... AND...**

BETWEEN... AND...

- **NOT BETWEEN... AND...**

will refer to an interval composed of two parts:

BETWEEN... AND...

- **NOT BETWEEN... AND...**

will refer to an interval composed of two parts:

- an interval below the first value indicated

BETWEEN... AND...

- **NOT BETWEEN... AND...**

will refer to an interval composed of two parts:

- an interval below the first value indicated
- a second interval above the second value

BETWEEN... AND...

SELECT

*

FROM

employees

WHERE

hire_date **NOT BETWEEN** '1990-01-01' **AND** '2000-01-01';



SQL

BETWEEN... AND...

SELECT

*

FROM

employees

WHERE

hire_date NOT BETWEEN '1990-01-01' AND '2000-01-01';

- the hire_date is *before* '1990-01-01'

BETWEEN... AND...



```
SELECT
  *
FROM
  employees
WHERE
  hire_date NOT BETWEEN '1990-01-01' AND '2000-01-01';
```

- the `hire_date` is *before* '1990-01-01'
or
- the `hire_date` is *after* '2000-01-01'

BETWEEN... AND...

SELECT

*

FROM

employees

WHERE

hire_date NOT BETWEEN '1990-01-01' AND '2000-01-01';

'1990-01-01' AND '2000-01-01' are not included in the intervals



SQL

BETWEEN... AND...

- **BETWEEN... AND...**

BETWEEN... AND...

- **BETWEEN... AND...**

- not used only for date values

BETWEEN... AND...

- **BETWEEN... AND...**

- not used only for date values
- could also be applied to strings and numbers

A large, modern conference room is shown from a perspective looking down the length of the room. On both sides, there are long rows of black office chairs with wheels, facing towards a central dark rectangular table. The room has a polished wooden floor and walls made of large glass windows that look out onto a bright, overcast sky. The ceiling is white with a grid of recessed lighting fixtures. The overall atmosphere is professional and spacious.

IS NOT NULL / IS NULL

IS NOT NULL / IS NULL

- **IS NOT NULL**

IS NOT NULL / IS NULL

- **IS NOT NULL**

used to extract values that are not null

IS NOT NULL / IS NULL

- **IS NOT NULL**

used to extract values that are not null



SQL

```
SELECT column_1, column_2,... column_n  
FROM table_name  
WHERE column_name IS NOT NULL;
```

IS NOT NULL / IS NULL

- **IS NULL**

used to extract values that are null



SQL

```
SELECT column_1, column_2,... column_n  
FROM table_name  
WHERE column_name IS NULL;
```



Using Other Comparison Operators

Other Comparison Operators

So far:

Other Comparison Operators

So far:

- BETWEEN... AND...
- LIKE
- IS NOT NULL
- NOT LIKE
- IS NULL

Other Comparison Operators

So far:

- BETWEEN... AND...
- LIKE
- IS NOT NULL

In this lecture:

- NOT LIKE
- IS NULL

Other Comparison Operators

So far:

- BETWEEN... AND...
 - LIKE
 - IS NOT NULL
- NOT LIKE
 - IS NULL

In this lecture:

=, >, >=, =<, <, <>, !=

Other Comparison Operators

SQL

=

equal to

>

greater than

>=

greater than or equal to

<

less than

<=

less than or equal to

Other Comparison Operators

SQL

<>, !=

“Not Equal” operators

not equal, ≠

different from



SELECT DISTINCT

SELECT DISTINCT

- the SELECT statement

SELECT DISTINCT

- the SELECT statement
can retrieve rows from a designated column, given some criteria

SELECT DISTINCT

- **SELECT DISTINCT**

*selects all *distinct*, *different* data values*

SELECT DISTINCT

- **SELECT DISTINCT**

selects all *distinct, different* data values



SQL

```
SELECT DISTINCT column_1, column_2,... column_n  
FROM table_name;
```



Introduction to Aggregate Functions

Introduction to Aggregate Functions

aggregate functions

they are applied on *multiple rows of a single column* of a table and
return an output of a single value

Introduction to Aggregate Functions

- COUNT()

Introduction to Aggregate Functions

- **COUNT()**

counts the number of non-null records in a field

Introduction to Aggregate Functions

- **COUNT()**

counts the number of non-null records in a field

- **SUM()**

Introduction to Aggregate Functions

- **COUNT()**

counts the number of non-null records in a field

- **SUM()**

sums all the non-null values in a column

Introduction to Aggregate Functions

- **COUNT()**

counts the number of non-null records in a field

- **SUM()**

sums all the non-null values in a column

- **MIN()**

Introduction to Aggregate Functions

- **COUNT()**

counts the number of non-null records in a field

- **SUM()**

sums all the non-null values in a column

- **MIN()**

returns the minimum value from the entire list

Introduction to Aggregate Functions

- **COUNT()**

counts the number of non-null records in a field

- **SUM()**

sums all the non-null values in a column

- **MIN()**

returns the minimum value from the entire list

- **MAX()**

Introduction to Aggregate Functions

- **COUNT()**

counts the number of non-null records in a field

- **SUM()**

sums all the non-null values in a column

- **MIN()**

returns the minimum value from the entire list

- **MAX()**

returns the maximum value from the entire list

Introduction to Aggregate Functions

- **COUNT()**

counts the number of non-null records in a field

- **SUM()**

sums all the non-null values in a column

- **MIN()**

returns the minimum value from the entire list

- **MAX()**

returns the maximum value from the entire list

- **AVG()**

Introduction to Aggregate Functions

- **COUNT()**

counts the number of non-null records in a field

- **SUM()**

sums all the non-null values in a column

- **MIN()**

returns the minimum value from the entire list

- **MAX()**

returns the maximum value from the entire list

- **AVG()**

calculates the average of all non-null values belonging to a certain column of a table

Introduction to Aggregate Functions

- **COUNT()**

counts the number of non-null records in a field

Introduction to Aggregate Functions

- **COUNT()**

counts the number of non-null records in a field

- it is frequently used in combination with the reserved word **DISTINCT**

Introduction to Aggregate Functions

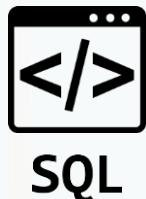
- COUNT()



```
SELECT COUNT(column_name)  
FROM table_name;
```

Introduction to Aggregate Functions

COUNT()



```
SELECT COUNT(column_name)  
FROM table_name;
```

the parentheses after COUNT() must start right after the keyword, not after a whitespace

Introduction to Aggregate Functions

- COUNT(DISTINCT)



```
SELECT COUNT(DISTINCT column_name)  
FROM table_name;
```

Introduction to Aggregate Functions

aggregate functions

they are applied on *multiple rows of a single column* of a table and
return an output of a single value

- they ignore NULL values unless told not to

A large conference room with rows of chairs facing a central table. The room has a modern design with a grid-patterned ceiling and large windows overlooking a landscape.

GROUP BY

GROUP BY

When working in SQL, results can be grouped according to a specific field or fields

GROUP BY

- **GROUP BY**

When working in SQL, results can be grouped according to a specific field or fields

GROUP BY

- **GROUP BY**

When working in SQL, results can be grouped according to a specific field or fields

- **GROUP BY** must be placed immediately after the **WHERE** conditions, if any, and just before the **ORDER BY** clause

GROUP BY

- **GROUP BY**

When working in SQL, results can be grouped according to a specific field or fields

- **GROUP BY** must be placed immediately after the **WHERE** conditions, if any, and just before the **ORDER BY** clause
- **GROUP BY** is one of the most powerful and useful tools in SQL

GROUP BY

- **GROUP BY**



SQL

```
SELECT column_name(s)
FROM table_name
WHERE conditions
GROUP BY column_name(s)
ORDER BY column_name(s);
```

GROUP BY

- **GROUP BY**

in most cases, when you need an aggregate function, you must add a GROUP BY clause in your query, too

Always include the field you have grouped your results by in the SELECT statement!



HAVING



HAVING

- **HAVING**

HAVING

- **HAVING**

- frequently implemented with **GROUP BY**

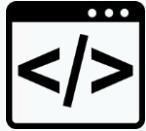
HAVING

- **HAVING**

refines the output from records that do not satisfy a certain condition

- frequently implemented with **GROUP BY**

HAVING



SQL

```
SELECT column_name(s)
FROM table_name
WHERE conditions
GROUP BY column_name(s)
HAVING conditions
ORDER BY column_name(s);
```

HAVING



SQL

```
SELECT column_name(s)
FROM table_name
WHERE conditions
GROUP BY column_name(s)
HAVING conditions
ORDER BY column_name(s);
```

- HAVING is like WHERE but applied to the GROUP BY block

HAVING

- **WHERE vs. HAVING**

after HAVING, you can have a condition with an aggregate function,
while WHERE cannot use aggregate functions within its conditions



WHERE vs HAVING

WHERE vs HAVING

- **WHERE**

WHERE vs HAVING

- **WHERE**

allows us to set conditions that refer to subsets of *individual* rows

WHERE vs HAVING

- **WHERE**

allows us to set conditions that refer to subsets of *individual* rows



applied *before* re-organizing the output into groups

WHERE vs HAVING

1	9/3/2016	1	A_1
2	12/2/2016	2	C_1
3	4/15/2017	3	D_1
4	5/24/2017	1	B_2
5	5/25/2017	4	B_2
6	6/6/2017	2	B_1
7	6/10/2017	4	A_2
8	6/10/2017	3	C_1
9	7/20/2017	1	A_1
10	8/11/2017	2	B_1

WHERE vs HAVING

WHERE



1	9/3/2016	1	A_1
2	12/2/2016	2	C_1
3	4/15/2017	3	D_1
4	5/24/2017	1	B_2
5	5/25/2017	4	B_2
6	6/6/2017	2	B_1
7	6/10/2017	4	A_2
8	6/10/2017	3	C_1
9	7/20/2017	1	A_1
10	8/11/2017	2	B_1

WHERE vs HAVING

WHERE



1	9/3/2016	1	A_1
2	12/2/2016	2	C_1
3	4/15/2017	3	D_1
4	5/24/2017	1	B_2
5	5/25/2017	4	B_2
6	6/6/2017	2	B_1
7	6/10/2017	4	A_2
8	6/10/2017	3	C_1
9	7/20/2017	1	A_1
10	8/11/2017	2	B_1

1	9/3/2016	1	A_1
2	12/2/2016	2	C_1
3	4/15/2017	3	D_1
4	5/24/2017	1	B_2
6	6/6/2017	2	B_1
8	6/10/2017	3	C_1
9	7/20/2017	1	A_1
10	8/11/2017	2	B_1

re-organizing the
output into groups
(GROUP BY)

WHERE vs HAVING

WHERE



1	9/3/2016	1	A_1
2	12/2/2016	2	C_1
3	4/15/2017	3	D_1
4	5/24/2017	1	B_2
5	5/25/2017	4	B_2
6	6/6/2017	2	B_1
7	6/10/2017	4	A_2
8	6/10/2017	3	C_1
9	7/20/2017	1	A_1
10	8/11/2017	2	B_1

1	9/3/2016	1	A_1
2	12/2/2016	2	C_1
3	4/15/2017	3	D_1
4	5/24/2017	1	B_2
6	6/6/2017	2	B_1
8	6/10/2017	3	C_1
9	7/20/2017	1	A_1
10	8/11/2017	2	B_1

re-organizing the
output into groups
(GROUP BY)



the output can be further improved, or *filtered*

WHERE vs HAVING

WHERE



HAVING

1	9/3/2016	1	A_1
2	12/2/2016	2	C_1
3	4/15/2017	3	D_1
4	5/24/2017	1	B_2
5	5/25/2017	4	B_2
6	6/6/2017	2	B_1
7	6/10/2017	4	A_2
8	6/10/2017	3	C_1
9	7/20/2017	1	A_1
10	8/11/2017	2	B_1

1	9/3/2016	1	A_1
2	12/2/2016	2	C_1
3	4/15/2017	3	D_1
4	5/24/2017	1	B_2
6	6/6/2017	2	B_1
8	6/10/2017	3	C_1
9	7/20/2017	1	A_1
10	8/11/2017	2	B_1

re-organizing the
output into groups
(GROUP BY)

WHERE vs HAVING

WHERE



1	9/3/2016	1	A_1
2	12/2/2016	2	C_1
3	4/15/2017	3	D_1
4	5/24/2017	1	B_2
5	5/25/2017	4	B_2
6	6/6/2017	2	B_1
7	6/10/2017	4	A_2
8	6/10/2017	3	C_1
9	7/20/2017	1	A_1
10	8/11/2017	2	B_1

HAVING



1	9/3/2016	1	A_1
2	12/2/2016	2	C_1
3	4/15/2017	3	D_1
4	5/24/2017	1	B_2
6	6/6/2017	2	B_1
8	6/10/2017	3	C_1
9	7/20/2017	1	A_1
10	8/11/2017	2	B_1

1	9/3/2016	1	A_1
3	4/15/2017	3	D_1
4	5/24/2017	1	B_2
6	6/6/2017	2	B_1
10	8/11/2017	2	B_1

re-organizing the
output into groups
(GROUP BY)

WHERE vs HAVING

WHERE



1	9/3/2016	1	A_1
2	12/2/2016	2	C_1
3	4/15/2017	3	D_1
4	5/24/2017	1	B_2
5	5/25/2017	4	B_2
6	6/6/2017	2	B_1
7	6/10/2017	4	A_2
8	6/10/2017	3	C_1
9	7/20/2017	1	A_1
10	8/11/2017	2	B_1

HAVING



1	9/3/2016	1	A_1
2	12/2/2016	2	C_1
3	4/15/2017	3	D_1
4	5/24/2017	1	B_2
6	6/6/2017	2	B_1
8	6/10/2017	3	C_1
9	7/20/2017	1	A_1
10	8/11/2017	2	B_1

ORDER BY...

1	9/3/2016	1	A_1
3	4/15/2017	3	D_1
4	5/24/2017	1	B_2
6	6/6/2017	2	B_1
10	8/11/2017	2	B_1

re-organizing the
output into groups
(GROUP BY)

WHERE vs HAVING

- **HAVING**

- you *cannot* have both an aggregated and a non-aggregated condition in the HAVING clause

WHERE vs HAVING

Aggregate functions - GROUP BY and HAVING

WHERE vs HAVING

Aggregate functions - GROUP BY and HAVING

General conditions - WHERE

WHERE vs HAVING



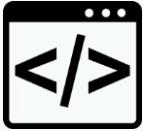
SQL

```
SELECT column_name(s)
FROM table_name
WHERE conditions
GROUP BY column_name(s)
HAVING conditions
ORDER BY column_name(s);
```



LIMIT

LIMIT



SQL

```
SELECT column_name(s)
FROM table_name
WHERE conditions
GROUP BY column_name(s)
HAVING conditions
ORDER BY column_name(s)
LIMIT number ;
```