This book is provided freely to you by

[EdTechBooks.org]

**CC BY-NC-ND International 4.0**: This work is released under a CC BY-NC-ND International 4.0 license, which means that you are free to do with it as you please as long as you (1) properly attribute it, (2) do not use it for commercial gain, and (3) do not create derivative works.
# Table of Contents

**Introduction** ......................................................................................................................... 6

1. How to Retrieve Data From a Single Table ........................................... 7  
   1.1. The Five Clauses of the **SELECT** Statement ......................... 23  
   1.2. Column Specifications ................................................................. 27  
   1.3. **LIKE** and **REGEXP** Operators ........................................ 28  
   1.4. Arithmetic Operators ........................................................................ 32  
   1.5. Column Aliases ................................................................................. 34  
   1.6. Comparison Operators ...................................................................... 36  
   1.7. IS NULL, BETWEEN, IN Operators ............................................. 38  
   1.8. AND, OR, NOT Logical Operators ............................................. 42  
   1.9. **DISTINCT** Clause ...................................................................... 44

2. How to Retrieve Data from Multiple Tables ........................................ 46  
   2.1. The **JOIN** Clause ...................................................................... 47  
   2.2. Joining More Than Two Tables .................................................. 51  
   2.3. The **OUTER JOIN** Clause ........................................................... 53  
   2.4. How to Code a **UNION** .............................................................. 56

3. Using Functions ...................................................................................................................... 59  
   3.1. Date Functions ................................................................................. 60  
   3.2. Numeric Functions .............................................................................. 67  
   3.3. String Functions ................................................................................. 70

4. How to Insert, Update, Delete Data in Tables ........................................ 77  
   4.1. The **INSERT** Clause With a Column List ............................... 78  
   4.2. The **INSERT** Clause Without a Column List ............................ 81  
   4.4. The **UPDATE** Clause With a Column List .................................. 83  
   4.4. The **DELETE** Clause ................................................................. 85

5. Summary Queries and Aggregate Functions ........................................ 87  
   5.1. Aggregate Functions ........................................................................... 88  
   5.2. Grouping Data .................................................................................. 90  
   5.3. Simple **GROUP BY** Query ............................................................ 91  
   5.4. Improving the **GROUP BY** Query .............................................. 93
Introduction

Before You Begin

This book has examples from two databases that you can download and install in your local MySQL instance. You can practice by copying the code snippets from the chapters and running them in your local environment. In order for this book to be most effective, you must have MySQL installed locally on your machine and have also installed MySQL Workbench.

In a future edition, this book will include SQL design basics and guidance on how to install MySQL and MySQL Workbench. It may also include screencasts that will walk you through the various concepts.

Click here to download the Bikes database

Click here to download the World database
How to Retrieve Data From a Single Table

Mathew Miles

The Five Clauses of the SELECT statement

- SELECT - the columns in the result set
- FROM - names the base table(s) from which results will be retrieved
- WHERE - specifies any conditions for the results set (filter)
- ORDER BY - sets how the result set will be ordered
- LIMIT - sets the number of rows to be returned

The clauses MUST appear in the order shown above.

Code Example:

```sql
1    USE world;
2    SELECT name
3    FROM city
4    WHERE CountryCode = "AFG"
5    ORDER BY name
6    LIMIT 3
```

Results:
Let us break the statement line by line:

**USE world;**

- The **USE** clause sets the database that we will be querying. You typically have more than one database on your database server. You have to specify which database you are working in.
- The semicolon “;” indicates the end of a statement. You can execute multiple statements in sequence by defining each statement with a semicolon

**SELECT name**

- The **SELECT** clause defines the columns and column order that you want to retrieve in your results set. If you want to retrieve all of the columns from the base table you can simply use
SELECT *
- You separate each column name with a comma ‘,’ ex., SELECT name, CountryCode
- There is no trailing comma at the end of a column list

FROM city

- The FROM clause specifies the table that the results will be coming from
- You can specify multiple tables by using a JOIN clause, but we will address that topic at a future time

ORDER BY name

- The ORDER BY clause is not required but when used it defines the sort order of the results
- By default, the sort order is ascending. This is implicit
  However, you can use explicit syntax of ASC. If you want the sort, order to be descending you can use the keyword DESC.
- You can specify more than one column in an Order By statement separated by commas. The sort order DESC, ASC applies to each column individually. Below IS some examples
  - ORDER BY population ASC, name DESC
  - ORDER BY population, name (ASC is always implied if not explicitly stated)

LIMIT 5;

- If you only want to return a specified number of rows from the result set, you can use the LIMIT clause. This can be helpful when you want to test a query for accuracy that could potentially bring back a very large number of rows.
- The semicolon ; defines the end of the statement

Table 1. Column Specifications
LIKE and REGEXP Operators

- The LIKE keyword is used with the WHERE clause.
- The LIKE keyword and can use two symbols as wildcards. The percent ( % ) symbol matches any number of characters and the underscore ( _ ) matches a single character
- REGEXP keyword allows you to do more complex pattern matching than a LIKE keyword/
- Some version of REGEXP exists in many computer languages. Refer to the “LIKE and REGEXP” handout for a full list of examples.

Table 2. LIKE Keyword

<table>
<thead>
<tr>
<th>LIKE Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%</td>
<td>Match any string of characters to the left of the symbol</td>
</tr>
<tr>
<td>_</td>
<td>Match a single character</td>
</tr>
</tbody>
</table>

Code Example:

```sql
USE world;
SELECT name
FROM country
WHERE name LIKE 'A%'
```

Results:
Table 3. REXEXP Keyword

<table>
<thead>
<tr>
<th>REGEXP Characters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>^</td>
<td>Match the pattern to the beginning of the value being tested.</td>
</tr>
<tr>
<td>$</td>
<td>Match the pattern to the end of the value being tested.</td>
</tr>
<tr>
<td>.</td>
<td>Matches any single character.</td>
</tr>
<tr>
<td>[charlist]</td>
<td>Matches any single character listed within the brackets.</td>
</tr>
<tr>
<td>[char1 - char2]</td>
<td>Matches any single character within the given range.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Code Example:**

```
USE world;
SELECT name
FROM country
```
WHERE name REGEXP 'g[o,u]';

Results:

Arithmetic Operators

- Arithmetic operators can be used in the SELECT, WHERE, and ORDER BY clauses.
- Operators are evaluated in the same way as arithmetic in other contexts.

Table 4. Operators and precedence order
<table>
<thead>
<tr>
<th>Operator</th>
<th>Name</th>
<th>Order of Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>Multiplication</td>
<td>1</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
<td>1</td>
</tr>
<tr>
<td>DIV</td>
<td>Integer Division</td>
<td>1</td>
</tr>
<tr>
<td>% (MOD)</td>
<td>Modulo (remainder)</td>
<td>1</td>
</tr>
<tr>
<td>+</td>
<td>Addition</td>
<td>2</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
<td>2</td>
</tr>
</tbody>
</table>

*Code Example:*

```sql
USE world;
SELECT name, population / SurfaceArea
AS "People per square mile"
FROM country;
```

*Results:*
Column Aliases

- A column alias provides a way to create a clean or more descriptive header for a results set.
- A column alias cannot be used in a SELECT, WHERE, GROUP BY or HAVING clause due to the order of execution. You must refer to the original column name.

In the previous example, we created a new column that was a calculated value. The problem is that the column header is now population / SurfaceArea. However we can rename the column header to something cleaner be create a column alias. Look at the code snippet below.

Code Example:
SELECT name, population / SurfaceArea 
   AS "People per square mile"
FROM country;

We used the AS keyword then in quotes we put the new column alias of “People per square mile.” Which changes the column header as seen show below.

Results:
Comparison Operators

- Comparison operators compare two expressions.
- The result of a comparison results to true or false.
- Comparison operators are not case sensitive and are used with text and dates as well as numbers.

Table 5. Comparison Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>Equal</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>Not equal</td>
</tr>
<tr>
<td>!=</td>
<td>Not equal</td>
</tr>
</tbody>
</table>

Code Example:

USE world;
SELECT name, population
FROM country
WHERE population > 1000000;

Results:
IS NULL

- *Null values* indicate an unknown or non-existent value and is different from an empty string (‘ ‘).
- To test for a *null value* you use the IS NULL clause
- The test for a value use IS NOT NULL clause

*Code Example:*

```
SELECT name, IndepYear
FROM country
WHERE IndepYear IS NULL;
```

*Results:*
BETWEEN Operators

- The BETWEEN operator is similar to \( \geq \) and \( \leq \).
- BETWEEN includes everything between the two values indicated.
- BETWEEN works with both text and number.

**Code Example:**

```sql
USE world;
SELECT name, IndepYear
FROM country
WHERE name BETWEEN "Aruba" and "Bahamas";
```

**Results:**
The IN Keyword

- The IN clause tests whether an expression is equal to a value or values in a list of expressions.
- The order of the items in the list does not matter.
- You can use the NOT operator to test for items not in the list.
- The IN clause may be used with a subquery.

**Code Example:**

```sql
USE world;
SELECT name
FROM country
WHERE name BETWEEN "Aruba" and "Bahamas";
```

```sql
1  SELECT name, IndepYear
2  FROM country
3  WHERE name BETWEEN "Aruba" and "Bahamas";
```

<table>
<thead>
<tr>
<th>name</th>
<th>IndepYear</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aruba</td>
<td>NULL</td>
</tr>
<tr>
<td>Australia</td>
<td>1901</td>
</tr>
<tr>
<td>Austria</td>
<td>1918</td>
</tr>
<tr>
<td>Azerbaijan</td>
<td>1991</td>
</tr>
<tr>
<td>Bahamas</td>
<td>1973</td>
</tr>
</tbody>
</table>
Results:

```
USE world;
SELECT name, population
FROM country
WHERE name IN ('Aruba', 'Barbados', 'Cuba', 'Bahamas')
ORDER BY population ASC;
```

AND, OR, NOT Logical Operators

- *Logical operators* are used in the WHERE clause
- You may use multiple *logical operators* in a WHERE clause to create a *compound condition*. The order of evaluation when multiple operators are used is shown in the table above.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Order of Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOT</td>
<td>(a NOT b) - a must be present but b must NOT be present to be included</td>
<td>1</td>
</tr>
<tr>
<td>AND</td>
<td>(a AND b) - If both a and b are present, item is included</td>
<td>2</td>
</tr>
<tr>
<td>OR</td>
<td>(a OR b) - If either a OR b is present item is included</td>
<td>3</td>
</tr>
</tbody>
</table>

Example:

```
USE world;
SELECT name, population
FROM country
```
WHERE region = 'caribbean'
AND population > 100000
ORDER BY population ASC;

Results:

<table>
<thead>
<tr>
<th>name</th>
<th>population</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aruba</td>
<td>103000</td>
</tr>
<tr>
<td>Saint Vincent and the Grenadines</td>
<td>114000</td>
</tr>
<tr>
<td>Saint Lucia</td>
<td>154000</td>
</tr>
<tr>
<td>Netherlands Antilles</td>
<td>217000</td>
</tr>
<tr>
<td>Barbados</td>
<td>270000</td>
</tr>
<tr>
<td>Bahamas</td>
<td>307000</td>
</tr>
<tr>
<td>Martinique</td>
<td>395000</td>
</tr>
</tbody>
</table>

DISTINCT Keyword

- DISTINCT appears directly after the SELECT clause.
- You can specify multiple columns, which means that the combination of columns must be unique.

Table 7. DISTINCT Keyword

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
<th>Order of Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>DISTINCT</td>
<td>Eliminates duplicate rows</td>
<td>1</td>
</tr>
</tbody>
</table>
Example:

SELECT DISTINCT continent, name
FROM country
ORDER BY continent;

Results:
The Five Clauses of the SELECT Statement

- SELECT - the columns in the result set
- FROM - names the base table(s) from which results will be retrieved
- WHERE - specifies any conditions for the results set (filter)
- ORDER BY - sets how the result set will be ordered
- LIMIT - sets the number of rows to be returned

The clauses MUST appear in the order shown above.

Code Example:

```sql
1    USE world;
2    SELECT name
3    FROM city
4    WHERE CountryCode = "AFG"
5    ORDER BY name
6    LIMIT 3
```

Results:
Let us break the statement line by line:

**USE world;**

- The **USE** clause sets the database that we will be querying. You typically have more than one database on your database server. You have to specify which database you are working in.
- The semicolon “;” indicates the end of a statement. You can execute multiple statements in sequence by defining each statement with a semicolon

**SELECT name**

- The **SELECT** clause defines the columns and column order that you want to retrieve in your results set. If you want to retrieve all of the columns from the base table you can simply use
SELECT *
- You separate each column name with a comma “,” ex., SELECT name, CountryCode
- There is no trailing comma at the end of a column list

FROM city
- The FROM clause specifies the table that the results will be coming from
- You can specify multiple tables by using a JOIN clause, but we will address that topic at a future time

ORDER BY name
- The ORDER BY clause is not required but when used it defines the sort order of the results
- By default, the sort order is ascending. This is implicit. However, you can use explicit syntax of ASC. If you want the sort, order to be descending you can use the keyword DESC.
- You can specify more than one column in an Order By statement separated by commas. The sort order DESC, ASC applies to each column individually. Below is some examples
  - ORDER BY population ASC, name DESC
  - ORDER BY population, name (ASC is always implied if not explicitly stated)

LIMIT 5;
- If you only want to return a specified number of rows from the result set, you can use the LIMIT clause. This can be helpful when you want to test a query for accuracy that could potentially bring back a very large number of rows.
- The semicolon; defines the end of the statement.
<table>
<thead>
<tr>
<th>Source</th>
<th>Option</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base Table Value</td>
<td>Show all columns</td>
<td>Comma-separated list of column names</td>
</tr>
<tr>
<td>Base Table Value</td>
<td>Column Name</td>
<td></td>
</tr>
<tr>
<td>Calculated Value</td>
<td>Calculation result</td>
<td>Arithmetic expression</td>
</tr>
<tr>
<td>Calculated Value</td>
<td>Calculation result</td>
<td>Functions</td>
</tr>
</tbody>
</table>
1.2

Column Specifications

<table>
<thead>
<tr>
<th>Source</th>
<th>Option</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base Table Value</td>
<td>Show all columns</td>
<td>*</td>
</tr>
<tr>
<td>Base Table Value</td>
<td>Column Name</td>
<td>Comma separated list of column names</td>
</tr>
<tr>
<td>Calculated Value</td>
<td>Calculation result</td>
<td>Arithmetic expression</td>
</tr>
<tr>
<td>Calculated Value</td>
<td>Calculation result</td>
<td>Functions</td>
</tr>
</tbody>
</table>

Column Specifications
LIKE and REGEXP Operators

- The LIKE keyword is used with the WHERE clause.
- The LIKE keyword and can use two symbols as wildcards. The percent ( % ) symbol matches any number of characters and the underscore ( _ ) matches a single character
- REGEXP keyword allows you to do more complex pattern matching than a LIKE keyword/
- Some version of REGEXP exists in many computer languages. Refer to the “LIKE and REGEXP” handout for a full list of examples.

Table 2. LIKE Keyword

<table>
<thead>
<tr>
<th>LIKE Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%</td>
<td>Match any string of characters to the left of the symbol</td>
</tr>
<tr>
<td>_</td>
<td>Match a single character</td>
</tr>
</tbody>
</table>

Code Example:
USE world;
SELECT name
FROM country
WHERE name LIKE 'A%'

Results:

Table 3. REXEXP Keyword
<table>
<thead>
<tr>
<th>REGEXP Characters</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>^</td>
<td>Match the pattern to the beginning of the value being tested.</td>
</tr>
<tr>
<td>$</td>
<td>Match the pattern to the end of the value being tested.</td>
</tr>
<tr>
<td>.</td>
<td>Matches any single character.</td>
</tr>
<tr>
<td>[charlist]</td>
<td>Matches any single character listed within the brackets.</td>
</tr>
<tr>
<td>[char1 – char2]</td>
<td>Matches any single character within the given range.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Code Example:**

USE world;
SELECT name
FROM country
WHERE name REGEXP 'g[o,u]';

**Results:**
```sql
USE world;
SELECT name
FROM country
WHERE name REGEXP 'g[o,u]';
```

<table>
<thead>
<tr>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Angola</td>
</tr>
<tr>
<td>Anguilla</td>
</tr>
<tr>
<td>Antigua and Barbuda</td>
</tr>
<tr>
<td>Bosnia and Herzegovina</td>
</tr>
<tr>
<td>Congo, The Democratic Republic of the Congo</td>
</tr>
<tr>
<td>Guinea</td>
</tr>
<tr>
<td>Guadeloupe</td>
</tr>
<tr>
<td>Guinea-Bissau</td>
</tr>
<tr>
<td>Equatorial Guinea</td>
</tr>
</tbody>
</table>

---

*Learning MySQL By Example*
1.4

Arithmetic Operators

- Arithmetic operators can be used in the SELECT, WHERE, and ORDER BY clauses.
- Operators are evaluated in the same way as arithmetic in other contexts.

Table 4. Operators and precedence order

<table>
<thead>
<tr>
<th>Operator</th>
<th>Name</th>
<th>Order of Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>Multiplication</td>
<td>1</td>
</tr>
<tr>
<td>/</td>
<td>Division</td>
<td>1</td>
</tr>
<tr>
<td>DIV</td>
<td>Integer Division</td>
<td>1</td>
</tr>
<tr>
<td>% (MOD)</td>
<td>Modulo (remainder)</td>
<td>1</td>
</tr>
<tr>
<td>+</td>
<td>Addition</td>
<td>2</td>
</tr>
<tr>
<td>-</td>
<td>Subtraction</td>
<td>2</td>
</tr>
</tbody>
</table>

Code Example:

USE world;
SELECT name, population / SurfaceArea AS "People per square mile"
FROM country;

Results:

```
<table>
<thead>
<tr>
<th>name</th>
<th>population / SurfaceArea</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aruba</td>
<td>533.678756</td>
</tr>
<tr>
<td>Afghanistan</td>
<td>34.841816</td>
</tr>
<tr>
<td>Angola</td>
<td>10.329670</td>
</tr>
<tr>
<td>Anguilla</td>
<td>83.333333</td>
</tr>
<tr>
<td>Albania</td>
<td>118.310839</td>
</tr>
<tr>
<td>Andorra</td>
<td>166.666667</td>
</tr>
<tr>
<td>Netherlands Antilles</td>
<td>271.250000</td>
</tr>
<tr>
<td>United Arab Emirates</td>
<td>29.198565</td>
</tr>
</tbody>
</table>
```
1.5

Column Aliases

Column Aliases

- A column alias provides a way to create a clean or more descriptive header for a results set.
- A column alias cannot be used in a SELECT, WHERE, GROUP BY or HAVING clause due to the order of execution. You must refer to the original column name.

In the previous example, we created a new column that was a calculated value. The problem is that the column header is now population / SurfaceArea. However, we can rename the column header to something cleaner by creating a column alias. Look at the code snippet below.

Example:

SELECT name, population / SurfaceArea AS “People per square mile”
FROM country;

We used the AS keyword then in quotes we put the new column alias of “People per square mile.” Which changes the column header as seen show below.
Results:

```
1       SELECT name, population / SurfaceArea
2                        AS "People per square mile"
3       FROM world.country;
```

<table>
<thead>
<tr>
<th>name</th>
<th>People per square mile</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aruba</td>
<td>533.678756</td>
</tr>
<tr>
<td>Afghanistan</td>
<td>34.841816</td>
</tr>
<tr>
<td>Angola</td>
<td>10.329670</td>
</tr>
<tr>
<td>Anguilla</td>
<td>83.333333</td>
</tr>
<tr>
<td>Albania</td>
<td>118.310839</td>
</tr>
<tr>
<td>Andorra</td>
<td>166.666667</td>
</tr>
<tr>
<td>Netherlands Antilles</td>
<td>271.250000</td>
</tr>
</tbody>
</table>
1.6

Comparison Operators

- Comparison operators compare two expressions.
- The result of a comparison results to true or false.
- Comparison operators are not case sensitive and are used with text and dates as well as numbers.

Table 5. Comparison Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>Equal</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>Not equal</td>
</tr>
<tr>
<td>!=</td>
<td>Not equal</td>
</tr>
</tbody>
</table>

Example:

USE world;
SELECT name, population
FROM country
WHERE population > 1000000;

Results:
1.7

IS NULL, BETWEEN, IN Operators

IS NULL

- Null values indicate an unknown or non-existent value and is different from an empty string (" ").
- To test for a null value you use the IS NULL clause
- The test for a value use IS NOT NULL clause

Example:

SELECT name, IndepYear
FROM country
WHERE IndepYear IS NULL;

Results:
BETWEEN Operators

- The BETWEEN operator is similar to \( \geq \) and \( \leq \).
- BETWEEN includes everything between the two values indicated.
- BETWEEN works with both text and number.

Example:

```
USE world;
SELECT name, IndepYear
FROM country
WHERE name BETWEEN "Aruba" and "Bahamas";
```

Results:
The IN Keyword

- The IN clause tests whether an expression is equal to a value or values in a list of expressions.
- The order of the items in the list does not matter.
- You can use the NOT operator to test for items not in the list.
- The IN clause may be used with a subquery.

Examples:

USE world;
SELECT name
FROM country
WHERE name BETWEEN "Aruba" and "Bahamas";
ORDER BY population ASC;
Results:

```sql
USE world;
SELECT name
FROM country
WHERE name IN ('Aruba', 'Barbados', 'Cuba', 'Bahamas')
ORDER BY population ASC;
```
1.8

AND, OR, NOT Logical Operators

- **Logical operators** are used in the WHERE clause
- You may use multiple *logical operators* in a WHERE clause to create a Compound condition. The order of evaluation when multiple operators are used is shown in the table above.

Table 6. Logical Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
<th>Order of Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOT</td>
<td>(a NOT b) - a must be present but b must NOT be present to be included</td>
<td>1</td>
</tr>
<tr>
<td>AND</td>
<td>(a AND b) - If both a and b are present, item is included</td>
<td>2</td>
</tr>
<tr>
<td>OR</td>
<td>(a OR b) - If either a OR b is present item is included</td>
<td>3</td>
</tr>
</tbody>
</table>

Example:

USE world;
SELECT name, population
FROM country
WHERE region = 'caribbean'
AND population > 100000
ORDER BY population ASC;

Results:
1.9

DISTINCT Clause

DISTINCT Keyword

- The DISTINCT clause removes duplicate rows from a query.
- DISTINCT appears directly after the SELECT clause.
- You can specify multiple columns, which means that the combination of columns must be unique.

Table 7. DISTINCT Keyword

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Description</th>
<th>Order of Evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>DISTINCT</td>
<td>Eliminates duplicate rows</td>
<td>1</td>
</tr>
</tbody>
</table>

Example:

SELECT DISTINCT continent, name
FROM country
ORDER BY continent;

Results:
```sql
1. SELECT DISTINCT continent
2. FROM country
3. ORDER BY continent;
```
2

How to Retrieve Data from Multiple Tables
2.1 The JOIN Clause

The Join Clause

- A JOIN clause allows you to access data from two or more tables in a query.
- A join links to tables on a common key between the two tables. Usually the primary key on one table is compared to the foreign key on another table using the equals ( = ) sign. This is an equijoin or an inner-join. However, other comparison operators are also valid.
- If column names from each table in the join have the same name, they must be qualified with the table name or a table alias.

Below is a basic example of a SQL statement with an inner join clause using explicit syntax.

```sql
1    USE world;
2    SELECT city.name AS "City Name",
3        country.name AS "Country Name"
4    FROM country
5    JOIN city
```
You could write SQL statements more succinctly with an inner join clause using *table aliases*. Instead of writing out the whole table name to qualify a column, you can use a table alias.

```sql
1    USE world;
2    SELECT ci.name AS "City Name",
3        co.name AS "Country Name"
4    FROM city ci
5        JOIN country co
```

The results of the join query would yield the same results as shown below whether or not table names are completely written out or are represented with table aliases. The table aliases of `co` for country and `ci` for city are defined in the FROM clause and referenced in the SELECT and ON clause:

Results:
Let us break the statement line by line:

**USE world;**

- The **USE** clause sets the database that we will be querying. You typically have more than one database on your database server. You have to specify which database you are working in.
- The semicolon “;” indicates the end of a statement. You can execute multiple statements in sequence by defining each statement with a semicolon

**SELECT ci.name AS “City Name”, co.name AS “Country Name”**

- The **SELECT** clause defines the columns and column order that you want to retrieve in your result set. In this example, we have columns from two separate tables. These columns have the same name, so they MUST be qualified with the full table name or table alias. Otherwise, the column names are ambiguous.
- You separate each column name with a comma “,” including the corresponding table alias if one is provided
- To create a friendlier column name in the output, we assign a column alias to each qualified column name. Instead of ci.name showing in the column header of the report, we assign a friendlier column alias of “City Name” and for co.name “Country Name.”

**FROM city ci**

- The **FROM** clause specifies the table(s) from which results will be returned.
- In a **JOIN** clause, the first table to be joined is specified after the **FROM** clause.

**JOIN country co**

- Use a **JOIN** clause between the two tables.
• Include the alias if desired.


• The **ON** clause specifies the common column from each table (usually a PK in one table and its corresponding foreign key in the other). Each column name is separated with an operator (join condition usually the equals ( = ) sign.)
2.2

Joining More Than Two Tables

How to Join More than Two Tables

- To include more tables in the query, you simply add more additional `JOIN` clauses

*Code Snippet:*

```sql
1    USE world;
2    SELECT ci.name AS "City Name",
3        co.name AS "Country Name",
4        cl.language AS "Country Language"
5    FROM city ci
6        JOIN country co
8        JOIN country language cl
```

*Results:*
JOIN countrylanguage cl.

- The “cl” is the alias for countrylanguage.
- You can refer to tables already specified in a previous join.


- The common column between the two tables being joined is the CountryCode column from the countrylanguage table and the CountryCode column from the city table.
- The “cl” alias previously defined for countrylanguage is used to specify the CountryCode column.
2.3

The OUTER JOIN Clause

The Outer Join Clause

- An outer join will return all the rows from one table and only the rows from the other table that match the join condition
- You can use `LEFT JOIN` or `RIGHT JOIN`. If you use `LEFT JOIN`, all the rows from the table on the left of the equals ( = ) sign will be included in the result set whether the join condition is satisfied or not.
- If you use `RIGHT JOIN`, all the rows from the table on the right of the equals ( = ) sign will be included in the result set whether the join condition is satisfied or not.

Below is a code snippet of a SQL statement with an outer join clause.

```
1 USE world;
2 SELECT c.name, c.continent, cl.language
3 FROM country c LEFT JOIN countrylanguage cl
4 ON c.code = cl.CountryCode
5 ORDER BY cl.language ASC;
```

Results:
SELECT c.name, c.continent, cl.language

- The “c.” pre-pended to name and continent is a table alias to the country table. Therefore, return name and continent from the country table.
- The “cl” prepended to the language table is a table alias to the countrylanguage table. Therefore, return language from the countryLanguage table.

FROM country c LEFT JOIN countrylanguage cl

- “Country c” assigns “c” as an alias for “country” “countrylanguage cl” assigns “cl” as an alias for “countrylanguage”
- LEFT JOIN means that all rows on the left side of the JOIN operator ( = ) are included in the results whether they have a matching key from the table on the
RIGHT side of the operator.

**ON c.code = cl.CountryCode**

- ON is the second part of the JOIN clause. It precedes the JOIN condition.
- `c.code` refers to the code column from the country table and is a primary key. Since the key is on the LEFT side of the join condition, all rows from the country table will be included in the results whether they have a matching key in the countrylanguage table or not.
- `cl.CountryCode` refers to the CountryCode on the countrylanguage table and is a foreign key to the country table. Only the rows that have a matching key in the country table will be included in the results.
2.4

How to Code a UNION

How to Code a UNION

- A **UNION** combines the results of two or more queries into a single result set
- Each result set must have the same number of columns
- The corresponding data types for each column must be compatible. However, the column names may be different from each result set
- A **UNION** removes duplicate rows by default
- You may interfile the results using an **ORDER BY** clause if there is a column with a common name.

*Code Example:*

```
1 USE world;
2 SELECT name, population
3 FROM city WHERE CountryCode = 'AUS'
4 UNION
5 SELECT name, population
6 FROM country
7 WHERE continent = 'Oceania'
8 ORDER BY name;
```
Results:

```
1. USE world;
2. SELECT name, population
3. FROM city WHERE CountryCode = 'AUS'
4. UNION
5. SELECT name, population
6. FROM country
7. WHERE continent = 'Oceania'
8. ORDER BY name;
```

<table>
<thead>
<tr>
<th>name</th>
<th>population</th>
</tr>
</thead>
<tbody>
<tr>
<td>Adelaide</td>
<td>978100</td>
</tr>
<tr>
<td>American Samoa</td>
<td>68000</td>
</tr>
<tr>
<td>Australia</td>
<td>18886000</td>
</tr>
<tr>
<td>Brisbane</td>
<td>1291117</td>
</tr>
<tr>
<td>Cairns</td>
<td>92273</td>
</tr>
<tr>
<td>Canberra</td>
<td>322723</td>
</tr>
<tr>
<td>Central Coast</td>
<td>227657</td>
</tr>
</tbody>
</table>

- The first query returns the name and population from the city table.
- The filter (WHERE CLAUSE) of the query limits the country code to Australia.

**UNION**

- The ‘UNION’ clause will combine this query with the results of the subsequent query.

**SELECT name, population**
FROM country
WHERE continent = 'Oceania'

- The second query returns the name and population from the country table.
- The filter (WHERE CLAUSE) of the query limits the continent code to Oceania.

ORDER BY name;

- It is possible to sort (ORDER BY CLAUSE) and interfile the results of both queries because each query shares a column with the same name. Otherwise, the ORDER BY clause would generate an error.
3

Using Functions
3.1

Date Functions

Current Date/Time Functions

- There are a number of functions that give the current date and time. The DATE() function is a date formatting function, but I include it in the list because it is often confused with the NOW() function.
- CURRENT_DATE, CURRENT_TIME, UTC_DATE, UTC_TIME can be used with the parentheses “()” or not. They accept no parameters.

Table 1. Current Date Functions
<table>
<thead>
<tr>
<th>Function</th>
<th>Type</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOW()</td>
<td>date/time</td>
<td>NOW()</td>
<td>ex. ‘2020-02-24 09:31:31’</td>
</tr>
<tr>
<td>DATE(date)</td>
<td>date/time</td>
<td>DATE('2020-01-01 11:31:31')</td>
<td>‘2020-02-24’</td>
</tr>
<tr>
<td>CURRENT_DATE()</td>
<td>date</td>
<td>CURRENT_DATE</td>
<td>‘2020-02-24’</td>
</tr>
<tr>
<td>CURRENT_TIME()</td>
<td>time</td>
<td>CURRENT_TIME</td>
<td>‘11:52:10’</td>
</tr>
<tr>
<td>UTC_DATE()</td>
<td>date</td>
<td>UTC_DATE</td>
<td>‘2020-02-24’</td>
</tr>
<tr>
<td>UTC_TIME()</td>
<td>time</td>
<td>UTC_TIME</td>
<td>‘18:52:10’</td>
</tr>
</tbody>
</table>

SELECT NOW() AS 'NOW()',
DATE('2020-01-01') AS 'DATE()', date only',
CURRENT_DATE AS 'CURRENT_DATE',
CURRENT_TIME AS 'CURRENT_TIME',
UTC_DATE AS 'UTC_DATE',
UTC_TIME AS 'UTC_TIME';

Results:

<table>
<thead>
<tr>
<th>Function</th>
<th>Type</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATE_ADD(date, interval expression unit)</td>
<td>DATE, DATETIME</td>
<td>DATE_ADD('2020-01-01', INTERVAL 1 DAY)</td>
<td>'2020-02-02'</td>
</tr>
</tbody>
</table>

**DATE_ADD**

- Returns a date with a DATE or DATETIME value equal to the original value plus the specified interval.

Table 2. DATE_ADD Function

**Code Snippet:**

USE bike;
SELECT order_date,
DATE_ADD(order_date, INTERVAL 1 DAY) AS 'ORDER DATE PLUS 1 day',
DATE_ADD(order_date, INTERVAL 6 MONTH) AS 'ORDER DATE PLUS 6 months',
DATE_ADD(order_date, INTERVAL '2 12' DAY_HOUR)
AS 'ORDER DATE PLUS 2 days 1 hour'
FROM cust_order;

Results:

```
USE bike;
SELECT order_date,
    DATE_ADD(order_date, INTERVAL 1 DAY)
    AS 'ORDER DATE PLUS 1 day',
    DATE_ADD(order_date, INTERVAL 6 MONTH)
    AS 'ORDER DATE PLUS 6 months',
    DATE_ADD(order_date, INTERVAL '2 12' DAY_HOUR)
    AS 'ORDER DATE PLUS 2 days 1 hour'
FROM cust_order;
```

<table>
<thead>
<tr>
<th>order_date</th>
<th>ORDER DATE PLUS 1 day</th>
<th>ORDER DATE PLUS 6 months</th>
<th>ORDER DATE PLUS 2 days 1 hour</th>
</tr>
</thead>
<tbody>
<tr>
<td>2016-01-01</td>
<td>2016-01-02</td>
<td>2016-07-01</td>
<td>2016-01-03 12:00:00</td>
</tr>
<tr>
<td>2016-01-01</td>
<td>2016-01-02</td>
<td>2016-07-01</td>
<td>2016-01-03 12:00:00</td>
</tr>
<tr>
<td>2016-01-02</td>
<td>2016-01-03</td>
<td>2016-07-02</td>
<td>2016-01-04 12:00:00</td>
</tr>
<tr>
<td>2016-01-03</td>
<td>2016-01-04</td>
<td>2016-07-03</td>
<td>2016-01-05 12:00:00</td>
</tr>
<tr>
<td>2016-01-01</td>
<td>2016-01-02</td>
<td>2016-07-01</td>
<td>2016-01-03 12:00:00</td>
</tr>
<tr>
<td>2016-01-01</td>
<td>2016-01-02</td>
<td>2016-07-01</td>
<td>2016-01-03 12:00:00</td>
</tr>
<tr>
<td>2016-01-02</td>
<td>2016-01-03</td>
<td>2016-07-02</td>
<td>2016-01-04 12:00:00</td>
</tr>
<tr>
<td>2016-01-03</td>
<td>2016-01-04</td>
<td>2016-07-03</td>
<td>2016-01-05 12:00:00</td>
</tr>
</tbody>
</table>

**DATE_FORMAT**

- Dates must be enclosed in quotes
- You can pass a DATE or DATETIME datatype to DATE_FORMAT
Table 3. DATE_FORMAT Function

<table>
<thead>
<tr>
<th>Function</th>
<th>Type</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATE_FORMAT</td>
<td>DATE</td>
<td>DATE_FORMAT('2020-09-03',</td>
<td>09/03/14</td>
</tr>
<tr>
<td></td>
<td></td>
<td>'%m/%d/%y')</td>
<td></td>
</tr>
</tbody>
</table>

**Code Snippet:**

```
USE world;
SELECT name, continent, DATE_FORMAT('2020-01-28', '%m/%d/%y')
FROM country;
```

**Results:**

Table 4. Format List

*Learning MySQL By Example* 64
<table>
<thead>
<tr>
<th>Specifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%a</td>
<td>Abbreviated weekday name (Sun..Sat)</td>
</tr>
<tr>
<td>%b</td>
<td>Abbreviated month name (Jan..Dec)</td>
</tr>
<tr>
<td>%c</td>
<td>Month, numeric (0..12)</td>
</tr>
<tr>
<td>%D</td>
<td>Day of the month with English suffix (0th, 1st, 2nd, 3rd, ...)</td>
</tr>
<tr>
<td>%d</td>
<td>Day of the month, numeric (0..31)</td>
</tr>
<tr>
<td>%e</td>
<td>Day of the month, numeric (0..31)</td>
</tr>
<tr>
<td>%f</td>
<td>Microseconds (000000..999999)</td>
</tr>
<tr>
<td>%H</td>
<td>Hour (00..23)</td>
</tr>
<tr>
<td>%h</td>
<td>Hour (01..12)</td>
</tr>
<tr>
<td>%I</td>
<td>Hour (01..12)</td>
</tr>
<tr>
<td>%i</td>
<td>Minutes, numeric (00..59)</td>
</tr>
<tr>
<td>%j</td>
<td>Day of year (001..366)</td>
</tr>
<tr>
<td>%k</td>
<td>Hour (0..23)</td>
</tr>
<tr>
<td>%l</td>
<td>Hour (1..12)</td>
</tr>
<tr>
<td>%M</td>
<td>Month name (January..December)</td>
</tr>
<tr>
<td>%m</td>
<td>Month, numeric (00..12)</td>
</tr>
<tr>
<td>%p</td>
<td>AM or PM</td>
</tr>
<tr>
<td>%r</td>
<td>Time, 12-hour (<strong>hh:mm:ss</strong> followed by AM or PM)</td>
</tr>
<tr>
<td>%S</td>
<td>Seconds (00..59)</td>
</tr>
<tr>
<td>%s</td>
<td>Seconds (00..59)</td>
</tr>
<tr>
<td>%T</td>
<td>Time, 24-hour (<strong>hh:mm:ss</strong>)</td>
</tr>
<tr>
<td>%U</td>
<td>Week (00..53), where Sunday is the first day of the week; <strong>WEEK()</strong> mode 0</td>
</tr>
<tr>
<td>%u</td>
<td>Week (00..53), where Monday is the first day of the week; <strong>WEEK()</strong> mode 1</td>
</tr>
<tr>
<td>%V</td>
<td>Week (01..53), where Sunday is the first day of the week; <strong>WEEK()</strong> mode 2; used with %X</td>
</tr>
<tr>
<td>%v</td>
<td>Week (01..53), where Monday is the first day of the week; <strong>WEEK()</strong> mode 3; used with %x</td>
</tr>
<tr>
<td>%W</td>
<td>Weekday name (Sunday..Saturday)</td>
</tr>
<tr>
<td>%W</td>
<td>Day of the week (0=Sunday..6=Saturday)</td>
</tr>
<tr>
<td>%x</td>
<td>Year, numeric, four digits</td>
</tr>
<tr>
<td>%X</td>
<td>Year, numeric, four digits</td>
</tr>
<tr>
<td>%y</td>
<td>Year, numeric (two digits)</td>
</tr>
<tr>
<td>%Y</td>
<td>Year, numeric (two digits)</td>
</tr>
<tr>
<td>%%</td>
<td>A literal % character</td>
</tr>
<tr>
<td>%x</td>
<td>x, for any &quot;x&quot; not listed above</td>
</tr>
</tbody>
</table>
DATEDIFF

- The DATEDIFF function has two parameters. Both are dates.
- The value returned by the function is an integer and is the number of days between the two dates.
- If you provide the latest date, first the results will be positive. Otherwise, it will be negative.

Example:

SELECT DATEDIFF('2018-01-01', '2019-01-01') AS 'Date Difference';

Results:
3.2 Numeric Functions

ROUND

- The ROUND function has two parameters. The first is a number, usually a DECIMAL or a FLOAT. The second defines the number of decimals to which the number will be rounded.
- If no length is provided, the number is rounded to a whole number.

Table 5. ROUND function

<table>
<thead>
<tr>
<th>Function</th>
<th>Type</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROUND(number[, length])</td>
<td>Number</td>
<td>ROUND(13.37, 1)</td>
<td>13.4</td>
</tr>
</tbody>
</table>

Example:

USE world;
SELECT name, LifeExpectancy, ROUND(LifeExpectancy) FROM world.country;

Results:
FLOOR, CEILING, TRUNCATE

- FLOOR() will return the next lowest whole number no matter what the decimal point.
- CEILING() will return the next highest whole number no matter what the decimal point.
- TRUNCATE() will return the number truncated to the precision specified.

Table 6. FLOOR, CEILING, TRUNCATE functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Type</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLOOR(number)</td>
<td>number</td>
<td>FLOOR(7.7)</td>
<td>7</td>
</tr>
<tr>
<td>CEILING(number)</td>
<td>number</td>
<td>CEILING(6.2)</td>
<td>7</td>
</tr>
<tr>
<td>TRUNCATE(NUMBER, length)</td>
<td>number</td>
<td>TRUNCATE(7.9)</td>
<td>7</td>
</tr>
</tbody>
</table>
Example:

```
USE bike;
SELECT list_price, FLOOR(list_price),
     CEILING(list_price),
     TRUNCATE(list_price, 0)
FROM product;
```

Results:

```
<table>
<thead>
<tr>
<th>list_price</th>
<th>FLOOR(list_price)</th>
<th>CEILING(list_price)</th>
<th>TRUNCATE(list_price, 0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>379.99</td>
<td>379</td>
<td>380</td>
<td>379</td>
</tr>
<tr>
<td>749.99</td>
<td>749</td>
<td>750</td>
<td>749</td>
</tr>
<tr>
<td>999.99</td>
<td>999</td>
<td>1000</td>
<td>999</td>
</tr>
<tr>
<td>2899.99</td>
<td>2899</td>
<td>2900</td>
<td>2899</td>
</tr>
<tr>
<td>1320.99</td>
<td>1320</td>
<td>1321</td>
<td>1320</td>
</tr>
</tbody>
</table>
```
3.3

String Functions

**CONCAT**

- Combines a list of strings into a single string.
- Can include column values and literal values.
- In MySQL literal values can be enclosed with either single (‘) or double quotes (").

*Example:*

```sql
USE world;
SELECT CONCAT(name, ', ', continent) FROM country;
```

*Results:*
RIGHT, LEFT

- The RIGHT and LEFT functions have two parameters. The first is a string and the second is the number of characters to be returned.
- The RIGHT function starts counting from the right side of the string. • The LEFT function starts counting from the left side of the string.

Table 7. RIGHT, LEFT functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Type</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>RIGHT(string, num. characters)</td>
<td>string</td>
<td>RIGHT('Salmon', 3)</td>
<td>mon</td>
</tr>
<tr>
<td>LEFT(string, num. characters)</td>
<td>string</td>
<td>LEFT('Salmon', 3)</td>
<td>Sal</td>
</tr>
</tbody>
</table>
Example:

USE bike;
SELECT category_name,
       LEFT(category_name, 8) AS 'First 8 Characters',
       RIGHT(category_name, 8) AS 'Last 8 Characters'
FROM category;

Results:

<table>
<thead>
<tr>
<th>category_name</th>
<th>First 8 Characters</th>
<th>Last 8 Characters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Children Bicycles</td>
<td>Children</td>
<td>Bicycles</td>
</tr>
<tr>
<td>Comfort Bicycles</td>
<td>Comfort</td>
<td>Bicycles</td>
</tr>
<tr>
<td>Cruisers Bicycles</td>
<td>Cruisers</td>
<td>Bicycles</td>
</tr>
<tr>
<td>Cyclocross Bicycles</td>
<td>Cyclocro</td>
<td>Bicycles</td>
</tr>
<tr>
<td>Electric Bikes</td>
<td>Electric</td>
<td>Bikes</td>
</tr>
<tr>
<td>Mountain Bikes</td>
<td>Mountain</td>
<td>in Bikes</td>
</tr>
<tr>
<td>Road Bikes</td>
<td>Road Bikes</td>
<td>ad Bikes</td>
</tr>
</tbody>
</table>

TRIM, LTRIM, RTRIM

- The TRIM function will remove leading and trailing spaces from a string.
- The LTRIM function will remove leading spaces from a string.
- The RTRIM function will remove trailing spaces from a string.
Table 8. TRIM functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Type</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>TRIM(string)</td>
<td>string</td>
<td>TRIM('   Salmon ')</td>
<td>‘salmon’</td>
</tr>
<tr>
<td>LTRIM(string)</td>
<td>string</td>
<td>LEFT('Salmon ')</td>
<td>‘salmon ’</td>
</tr>
<tr>
<td>RTRIM(string)</td>
<td>string</td>
<td>RIGHT(' Salmon')</td>
<td>‘ salmon’</td>
</tr>
</tbody>
</table>

Example:

SELECT LTRIM('  Salmon ') AS "Left Trim",
       RTRIM('  Salmon ') AS "Right Trim",
       TRIM('  Salmon ') AS "Trim";

Results:

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Left Trim</td>
<td>Salmon</td>
</tr>
<tr>
<td>Right Trim</td>
<td>Salmon</td>
</tr>
<tr>
<td>Trim</td>
<td>Salmon</td>
</tr>
</tbody>
</table>
```

FORMAT

- FORMAT() accepts a decimal but returns a comma formatted string.

Table 9. FORMAT functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Type</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>FORMAT(number, decimal)</td>
<td>string</td>
<td>FORMAT(1234.342, 2)</td>
<td>-356</td>
</tr>
</tbody>
</table>
**Code Sample:**

```
SELECT FORMAT(list_price,2)
FROM bike.product;
```

**Results:**

![Result Grid Image]

### LOWER, UPPER

- LOWER() converts all characters to lower case.
- UPPER() converts all characters to upper case.

**Table 9. LOWER, UPPER functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Type</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOWER(string)</td>
<td>string</td>
<td>LOWER('Salmon ')</td>
<td>'salmon'</td>
</tr>
<tr>
<td>UPPER(string)</td>
<td>string</td>
<td>UPPER('Salmon')</td>
<td>'SALMON'</td>
</tr>
</tbody>
</table>

**Example:**

```
SELECT UPPER('Salmon'),
      LOWER('Salmon');
```
LOCATE, LENGTH, SUBSTRING

LOCATE(), and LENGTH() accept a string but return an integer. • SUBSTRING() accepts a string and returns a string.

Table 9. LOCATE. LENGTH, SUBSTRING functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Type</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>LOCATE(find, search[, start])</td>
<td>string</td>
<td>LOCATE('al','salmon',1)</td>
<td>2</td>
</tr>
<tr>
<td>LENGTH(str)</td>
<td>string</td>
<td>LENGTH('salmon')</td>
<td>6</td>
</tr>
<tr>
<td>SUBSTRING(str, start[, length])</td>
<td>string</td>
<td>SUBSTRING('salmon',3,999)</td>
<td>'lmon'</td>
</tr>
</tbody>
</table>

Example:

SELECT LOCATE('al','salmon',1),
    LENGTH('salmon'),
    SUBSTRING('salmon',3,999);

Results:
```sql
SELECT LOCATE("al", "salmon", 1),
       LENGTH("salmon");
```

<table>
<thead>
<tr>
<th>LOCATE(&quot;al&quot;, &quot;salmon&quot;, 1)</th>
<th>LENGTH(&quot;salmon&quot;)</th>
<th>SUBSTRING(&quot;salmon&quot;, 3, 999)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>6</td>
<td>mon</td>
</tr>
</tbody>
</table>
4

How to Insert, Update, Delete Data in Tables
4.1

The INSERT Clause With a Column List

The INSERT Clause With a Column List

- You can INSERT single or multiple rows at a time.
- An INSERT with a column list DOES NOT require you to provide a value for each column. If you do not want to provide a value for a specific column, you do not have to include it in the column list. For columns that allow null values, the system will automatically provide a null value for you.
- If you want a column that provides a default value such as an auto-increment column to be populated with the default value, you do not need to list the column in the column list. The system will automatically provide the default value.
- When coding with a column list, the columns may appear in any order as long as the VALUES list matches the order of the column list.

Below is a basic example of an INSERT statement with a column list:

```sql
1    USE world;
2    INSERT INTO city
3        (name, countryCode, district, population)
```
VALUES
("San Felipe", "CHL", "Valparaiso", 64126);

Results:

Results of the Insert:

INSERT INTO city

- Insert the value into the city table. The INTO keyword is not required.
(name, countryCode, district, population)

- The column list is comma-separated and enclosed in parentheses.

VALUES

- The VALUES keyword is between the column list and the actual values. No commas are necessary.

("San Felipe", "CHL", "Valparaiso", 64126);

- The values order must appear in the corresponding order of the column list.
- You must enclose strings in quotes.
- You must not enclose numbers in quotes.
- You do not have to specify columns that allow null values or default values in the column list. They will automatically get a null or default value.
4.2

The INSERT Clause Without a Column List

The INSERT Clause Without a Column List

- You can INSERT single or multiple rows at a time.
- An INSERT without a column list requires you to provide a value for every column.
- You must list values in the same order that they appear on the table.
- You must explicitly use the keyword “null” for columns that allow for nulls if you do not want to provide a value.
- You must explicitly use the keyword “DEFAULT” for columns that provide a default value if you do not want to provide one.

Code Sample:

```sql
1    USE world;
2    INSERT INTO city
3    VALUES
4    (DEFAULT, "San Felipe", "CHL", "Valparaiso", 64126);
```

Results:
(DEFAULT "San Felipe", "CHL", "Valparaiso", 64126);

- The values order must appear in the same order they exist in the table.
- You must enclose strings in quotes.
- You must NOT enclose numbers in quotes.
- You must specify all column names and provide the keyword “DEFAULT” or a literal value for columns that provide a default option.
- If you do not want to provide a value for columns that allow null values, you must provide the keyword “null”.
4.4

The UPDATE Clause With a Column List

The UPDATE Clause

- You can UPDATE single or multiple rows at a time.
- In a SET clause, you define the column along with its new value that may be a literal value or an expression.
- You can update one or all of the columns in a row.
- You can use a subquery or WHERE clause in an UPDATE statement.

Code Sample:

```
1    USE world;
2    UPDATE city
3    SET Population = 65000, district = 'Aconcagua';
```

Results:
UPDATE city

- You indicate the table you want to UPDATE.

SET Population = 65000, district = 'Aconcagua';

- You indicate the table columns and associated values you want to change them to by using the equals sign (=).
- You must separate each column and value with a comma.
- There is no trailing comma.
4.4

The DELETE Clause

You can delete single or multiple columns with a single statement.
You can use a subquery or a WHERE clause with a DELETE statement.
By default MySQL is in safe update mode which prevents coding a delete statement without a WHERE clause.

Code Example:

```sql
1    USE world;
2    DELETE
3    FROM city
4    WHERE name = 'san felipe' AND countrycode = 'chl';
```

Results:
DELETE

- You begin a delete statement with the DELETE clause.

FROM city

- You must specify the table from which you are deleting rows.

WHERE name = 'san felipe' AND countrycode = 'chl';

- You should use a WHERE clause with a DELETE statement to avoid deleting every row in a table.
Summary Queries and Aggregate Functions
5.1 Aggregate Functions

Aggregate Functions

- Aggregate functions are synonymous with column functions.
- A summary query uses at least one column function.
- AVG, SUM return numeric values.
- MIN, MAX, COUNT can return numeric, date, or string values.
- All values are included in aggregate functions by default unless you specify the DISTINCT keyword.
- Duplicate rows are excluded in all aggregate functions with the exception of COUNT(*).
- ***** IF YOU CODE AN AGGREGATE FUNCTION IN THE SELECT STATEMENT, YOU CANNOT ALSO INCLUDE NON-AGGREGATE FUNCTIONS IN THE SELECT STATEMENT UNLESS THOSE NON-AGGREGATE COLUMNS ARE INCLUDED IN A GROUP BY CLAUSE.

Table 1. Aggregate Functions List
### Aggregate Function

<table>
<thead>
<tr>
<th>Function</th>
<th>Data Type</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVG([DISTINCT] column_values)</td>
<td>numeric</td>
<td>The average of the non-null columns in the expression</td>
</tr>
<tr>
<td>SUM([DISTINCT] column_values)</td>
<td>numeric</td>
<td>The total of the non-null columns in the expression</td>
</tr>
<tr>
<td>MIN([DISTINCT] column_values)</td>
<td>numeric, date, string</td>
<td>The lowest value off the non-null columns in the expression</td>
</tr>
<tr>
<td>MAX([DISTINCT] column_values)</td>
<td>numeric, date, string</td>
<td>The highest value of the non-null columns in the expression</td>
</tr>
<tr>
<td>COUNT([DISTINCT] column_values)</td>
<td>numeric</td>
<td>The number of the non-null columns in the expression</td>
</tr>
<tr>
<td>COUNT(*)</td>
<td>numeric</td>
<td>The number of rows returned by the query</td>
</tr>
</tbody>
</table>

#### Code Sample:

```
USE bike;
SELECT AVG(list_price), SUM(list_price), MIN(list_price),
       MAX(list_price), COUNT(list_price), COUNT(*)
FROM product;
```

#### Output:

![Output](image)
5.2

Grouping Data

Using the GROUP BY Clause

- Group rows based on a column(s) or expression(s).
- If you use an aggregate function with a GROUP BY clause, the aggregation is calculated for each group.

Table 1. GROUP BY Function

<table>
<thead>
<tr>
<th>Aggregate Function</th>
<th>Order of Execution</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GROUP BY</td>
<td>3</td>
<td>Groups rows of a result set based on columns or expressions separated by commas.</td>
</tr>
</tbody>
</table>

Filtering With WHERE And HAVING

- Notice the order of execution. GROUP BY happens before WHERE but after HAVING.
- It is possible to use WHERE and HAVING in the same statement. They are not mutually exclusive.
5.3

Simple GROUP BY Query

Code Example:

USE bike;
SELECT category_id, AVG(list_price)
FROM product
GROUP BY category_id

Results:
USE bike:

- Set the bike database to be the default

SELECT category_id, AVG(list_price):

- Select the category_id from the base table
- Calculate the Average of the list price for all rows in the table

FROM product:

- Product is the base table from which data will be returned

GROUP BY category_id:

- Instead of returning a single value that is the average of all list_price items in the product table, return an average list_price for each category
- Without the GROUP BY clause, we see from our first example only a single row is returned with an average list_price of 1520.591402.
- With the GROUP BY clause, we return an average for each category_id.
Improving the GROUP BY Query

- The report would be nicer if we showed the category name instead of the category_id. This will require joining the product table to the category table.
- We can **ROUND** the **AVG** list price by category to TWO decimals points.
- We can **CONCAT** the dollar sign to the left of the list_price.

*Code Sample:*

```sql
USE bike;
SELECT category_name,
    CONCAT('$', ROUND(AVG(list_price),2)) AS 'Average List Price'
FROM product p
    JOIN category c
        ON p.category_id = c.category_id
GROUP BY category_name
ORDER BY category_name;
```

*Output:*
USE bike:

- Set the bike database to be the default

SELECT category_name,

CONCAT('$', ROUND(AVG(list_price),2)) AS 'Average List Price'

FROM product p
JOIN category c
ON p.category_id = c.category_id
GROUP BY category_name
ORDER BY category_name;
FROM product p

JOIN category c

ON p.category_id = c.category_id

- JOIN the product table to the category table
- Assign a table alias of “p” to product and “c” to category
- The join condition is the primary key of category_id from the category table equal to the foreign key of category_id in the product table.

GROUP BY category_name

- Instead of retrieving a single value with the average price of all products, return a list of average prices by category name.

ORDER BY category_name;

- Sort the results by category_name
5.5

Using the HAVING Clause

Filtering Aggregate Functions With The HAVING Clause

- The HAVING CLAUSE allows you to use an aggregate function as a filter. This is not allowed in a WHERE clause.
- Any columns or expressions you want to use in a HAVING clause, MUST BE DEFINED IN THE SELECT CLAUSE as well.

Code Sample:

USE bike;
SELECT category_id, AVG(list_price)
FROM product
GROUP BY category_id
HAVING AVG(list_price) > 1000

Output:
We previously discussed the preceding lines of code for this query so we will focus solely on the HAVING clause.

```
HAVING AVG(list_price) > 1000
```

- The **HAVING** clause executes after the **GROUP BY** clause but before the **SELECT**
- If you use an aggregate function in the **HAVING** clause, you must include the same aggregate function in the **SELECT**
- If you reference a column or expression in the **HAVING** clause, you must include the same column or expression in the **SELECT**
- You cannot use aggregate functions in a **WHERE** clause
5.5

Using the HAVING and WHERE Clauses Together

Below is an example of a statement that includes both the HAVING and WHERE clause in the same SQL statement.

USE bike;
SELECT category_id, AVG(list_price)
FROM product
WHERE model_year = 2016
GROUP BY category_id
HAVING AVG(list_price) > 1000

Output:
WHERE model_year = 2016

- The **WHERE** clause executes before the **GROUP BY**
- You can refer to columns not defined in the **SELECT**
- You cannot use aggregate functions in the **WHERE**

HAVING AVG(list_price) > 1000

- The **HAVING** clause executes after the **GROUP BY** clause but before the **SELECT**
- If you use an aggregate function in the **HAVING** clause, you must include the same aggregate function in the **SELECT**
- If you reference a column or expression in the **HAVING** clause, you must include the same column or expression in the **SELECT**
- You cannot use aggregate functions in a **WHERE**
COUNT(column_name) and COUNT(*)

How They Are Different

COUNT(column_name) and COUNT(*)

- COUNT(*) is the only aggregate function that counts rows with null values.
- When you specify a count based on a specific column, null values will not be counted.

Code Sample:

USE bike;
SELECT COUNT(phone), COUNT(*)
FROM CUSTOMER

Output:
```sql
1. USE bike;
2. SELECT COUNT(phone), COUNT(*)
3. FROM CUSTOMER
```

<table>
<thead>
<tr>
<th>COUNT(phone)</th>
<th>COUNT(*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>178</td>
<td>1445</td>
</tr>
</tbody>
</table>
5.7

Using the DISTINCT Statement

Removing Duplicate Values With DISTINCT

The DISTINCT keyword allows you to eliminate duplicate rows in aggregate functions.

You may also use the DISTINCT keyword with columns of the base table in a SELECT statement.

COUNT(list_price) counts all the rows in the product table that have a list price.

COUNT(DISTINCT list_price) eliminates duplicate values in the list_price.

Code Sample:

Example
USE bike;
SELECT COUNT(list_price), COUNT(DISTINCT list_price)
FROM product;

Output:
```sql
USE bike;

SELECT COUNT(list_price), COUNT(DISTINCT list_price)
FROM product;
```

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>COUNT(list_price)</td>
<td>COUNT(DISTINCT list_price)</td>
</tr>
<tr>
<td>321</td>
<td>106</td>
</tr>
</tbody>
</table>
6

Working With Subqueries
6.1

The Subquery In a SELECT Statement

The Subquery in a SELECT Statement

- A subquery is a SELECT statement coded within another SELECT statement.
- A subquery can return a single value or a list of values.
- A subquery can return multiple columns.
- A subquery cannot make use of the ORDER BY clause.
- A subquery can be nested within another subquery.
- You can use a subquery in a WHERE, HAVING, FROM and SELECT clause.

Code Sample:

```
1    USE world;
2    SELECT name, population
3    FROM city
4    WHERE CountryCode IN
5        (SELECT code
6        FROM country
7            WHERE region = 'Caribbean')
8    ORDER BY population
```
SELECT name, population
FROM city
WHERE CountryCode IN
(SELECT code
FROM country
WHERE region = 'Caribbean')
ORDER BY population
LIMIT 5
WHERE region = 'Caribbean')

- The subquery shown above returns a result list of all of the codes from the country table that have a region of ‘Caribbean’.
- The subquery must be in parentheses in MySQL.
- Each code (PK in country table) returned by the subquery is checked against CountryCode (FK in city table). If they match, the name and population are retrieved from the city table.
6.2

The Subquery in an UPDATE statement

The Subquery in an UPDATE statement

- Subqueries may be used in an UPDATE statement
- Since it is possible to change many values at once with a subquery, take special care before running an UPDATE statement with a subquery. You might make a copy of the table and data you are trying to change to test with before running your statement on live data.
- It is also possible to run your UPDATE statement inside of a transaction block that allows you to ROLLBACK or undo a statement. We will address the topic of ROLLBACK in a future lesson.

Code Sample:

```sql
1    UPDATE country
2    SET GNPOld = 0.00
3    WHERE Code IN
4    (SELECT CountryCode FROM countrylanguage WHERE population = 0)
```
Results:

```
1. UPDATE country
2. SET GNPOld = 0.00
3. WHERE Code IN
4. (SELECT CountryCode
5.     FROM countrylanguage
6.     WHERE population = 0)
```

**UPDATE country**

- Update the country table

**SET GNPOld = 0.00**

- Set the value of the GNPOld table = 0.00.
- No quotes are required because the GNPOld column is a decimal datatype

**WHERE Code IN**

- Update only the rows where the Code column value is in the results list returned in the subquery show below.

**{(SELECT CountryCode FROM countrylanguage WHERE population = 0)**

- Return a list of values from the CountryCode column from the countrylanguage table that has a population equal to zero.
- If these values match a code in the country table, the row is updated.
6.3

Create a Duplicate Table From An Existing Table

Create a Duplicate Table from an Existing Table with a Select Statement

- It is often helpful to create a duplicate table from an existing table for testing purposes
- You can combine the CREATE TABLE command with a select statement to create a duplicate of a table structure as well as the data in the table.

Code Sample:

1    USE world;
2    CREATE TABLE city_bak AS SELECT * FROM city;

Results:
USE world;

- Select world as the default schema

CREATE TABLE city_bak AS SELECT * FROM city;

- Create a new table named city_bak with the exact same structure as the city table.
- Copy all of the data from the city table to the city_bak table
6.4

The Subquery In a Delete Statement

The Subquery in a DELETE statement

- A subquery can be used in a DELETE statement.
- Always back up your data and test your DELETE statement before running it on live data.

NOTE: Before you can run a DELETE or UPDATE statement without a WHERE clause, you must uncheck “Safe Updates” checkbox in MySQL Preference. Please see below.
Code Sample:

USE world;
DELETE FROM city_bak
WHERE CountryCode IN
    (SELECT code FROM country
     WHERE region = 'Central Africa');

Results:
USE world;

- The tables used in this example are in the world database. Make sure it is selected as the default

DELETE FROM city_bak

- We are going to execute a DELETE statement on the city_bak table

WHERE CountryCode IN

- We are going to use a filter to delete items from the city_bak table where the CountryCode is found in a list of values that we will pass to it.

(SELECT code FROM country

WHERE region = 'Central Africa');

- We will execute a subquery on the country table and return a list of code values (PK to FK in city_bak table) where the region is equal to ‘Central Africa’.
- You could accomplish the same thing by joining the city_bak table to the country table, then filtering on the region column
from the country table.
7

SQL Views
7.1

SQL View Explained

SQL Views

- A SQL view is a SELECT statement that is stored as a database object.
- A SQL view acts as a virtual table but contains no data.
- You can use a view anywhere you would use a table including in a SELECT, INSERT, UPDATE, or DELETE statement.
Benefits of Using Views

- **Design Flexibility**: By using a view instead of a query in an application, it is easier to make changes to the underlying table structure.
- **Improved Security**: By using a view to return data from tables instead of a SELECT, you can hide the WHERE clause or other columns to which you do not want the user to have access.
- **Query Simplification**: You can write simple select statements against views, which handle complex queries and joins.

*Code Sample:*

```sql
USE WORLD;
CREATE VIEW city_country AS
SELECT ci.name AS city_name, co.name AS country_name
FROM city ci
    JOIN country co
```

*Results by selecting from the city_country view:*
CREATE VIEW city_country AS

- Create a new VIEW object and give it the name city_country
- The AS statement precedes the query that will be assigned to the VIEW

SELECT ci.name AS city_name, co.name AS country_name

- Only the columns defined in the SELECT statement will be available to the VIEW
- It is a good idea to provide a column alias in the select because the VIEW will not have access to the underlying table structure.

FROM city ci

JOIN country co


- The JOIN statement of the SELECT.
- Once you have created a VIEW, you can run SQL statements using the VIEW as if it were a table.
- By creating a VIEW, we can run selects that retrieve data from
multiple tables without having to re-code a join.

- Notice how the SELECT * retrieves only the rows defined in the SELECT statement used in the VIEW creation.
- If you want to drop a VIEW, we can run the DROP VIEW statement.
- If you want to modify an existing view you can use the statement CREATE OR REPLACE VIEW. That way you do not have to run a DROP VIEW statement and then a CREATE VIEW statement.
7.3

Views That Allow UPDATE Statements

Creating Views That Can Be Used With an UPDATE Statement

- There are some restrictions to creating a VIEW if you want to be able to run an UPDATE statement against it.
  - SELECT list cannot include a DISTINCT clause.
  - SELECT list cannot contain aggregate functions (SUM, COUNT, MIN, MAX, AVG, COUNT(*))
  - SELECT statement cannot use GROUP BY or HAVING. The VIEW cannot include a UNION operator.
- If you use any of the restricted statements, your view will be read-only.
8

SQL Indexes
8.1

SQL Indexes Explained

SQL Indexes

- You can create SQL indexes from single or multiple columns.
- A SQL index is like the index of a book. It speeds up the retrieval of a record. The relational database management system (RDBMS) can retrieve a record with the index key instead of having to perform a table scan.
- MySQL automatically creates indexes for primary and foreign keys significantly speeding up join performance.
- You should only create indexes on columns used in a join or search because the RDMS must update an index every time you execute an INSERT, UPDATE, or DELETE.

When to Create an Index

- When a column is used frequently in a search or a join.
- When a column contains a large number of distinct values.
- When the column is updated infrequently.
8.2

Clustered vs. Non-clustered Indexes

Clustered vs. Non-clustered Indexes

- Clustered index: The values in the column indexed are physically stored in alphabetical or numeric order.
  - You can only have one clustered index per table.
  - If you assign a primary key, the system automatically creates a clustered index on that column.
  - If no primary key is defined on a table, the first column that has an index defined for it becomes the clustered index.
- Non-clustered index: Column values are not in alphabetical or numeric order
  - You can add as many non-clustered indexes to a table as you want.
  - You should only create additional non-clustered indexes on a table if you need to search or perform a join on that column. When you create a foreign key column, a non-clustered index is automatically created for that column.
8.3

Create an Index in Workbench Using an ERD

- Right-click on the table and select ‘Edit’
- Click on the ‘Indexes’ tab
- Type the name of the index in the ‘Index Name’ field
- Under ‘Type’ select ‘INDEX’ • Click on the column(s) that you want to index.
- Tab to a new line
8.4

How to Manually Add an Index to an Existing Table

- Right-click on the table
- Select ‘Alter Table’

- Click on the ‘Indexes’ tab
• Type the name of the index in the ‘Index Name’ field
• Under ‘Type’ select ‘INDEX’
• Click on the column(s) that you want to index.
• Tab to a new line
Glossary
Index
Book Authors

**CC BY-NC-ND International 4.0**: This work is released under a CC BY-NC-ND International 4.0 license, which means that you are free to do with it as you please as long as you (1) properly attribute it, (2) do not use it for commercial gain, and (3) do not create derivative works.