

# SQL & Security

Christian Rudder

August 2024

## Contents

<b>1</b>	<b>SQL Basics</b>	<b>3</b>
1.1	Creating Database & Tables . . . . .	3
1.2	Queries . . . . .	5
1.3	Mutating Data & Filtering . . . . .	7
1.4	Joining Tables . . . . .	10
	1.4.1 JOIN Cheat Sheet . . . . .	11
1.5	Aggregate Functions . . . . .	13
1.6	Grouping Data . . . . .	15

*This page is left intentionally blank.*

# 1 SQL Basics

## 1.1 Creating Database & Tables

SQL stands for “*Structured Query Language*,” used to query against databases with tables containing columns of data, which most often relate to each other.

SQL uses key words like `SELECT`, `FROM`, `WHERE`, ignoring case. It’s good practice to use **all caps for SQL keywords**, and **lowercase for table and column names**. Here’s a simple example:

```
1 SELECT * FROM my_table
```

*Selects all (\*) columns from the table `my_table`.*

Now, we are a record company with bands, albums, and songs:

```
1 CREATE DATABASE test; -- creating a test database
2 DROP DATABASE test; -- deleting the test database
3
4
5 CREATE DATABASE concise_records; -- creating our database
6 USE concise_records; -- selecting our database to run commands on it
7
8
9 CREATE TABLE bandds (); -- creating a table for our bands
10 DROP TABLE bandds; -- deleting it because of our typo
11
12
13 -- Create bands table: artist name (at most 255 characters), CANNOT be NULL/EMPTY
14 CREATE TABLE bands (
15     name VARCHAR(255) NOT NULL
16 );
17
18
19 -- Add id column to bands, auto increment, not NULL, make this column important
20 ALTER TABLE bands
21 ADD COLUMN id INT NOT NULL AUTO_INCREMENT PRIMARY KEY;
```

We created the database, `concise_records`, and a table `bands` with two columns: `name` and `id`.

The **PRIMARY KEY** uniquely IDs each row in a table, helping draw a thread of relationships between tables where the ID is present.

Pagination of commands don’t matter, for example:

```
1 -- We could have written the above
2 CREATE DATABASE concise_records; USE concise_records; CREATE TABLE bands (name VARCHAR
3     (255) NOT NULL);
4 ALTER TABLE
5 bands
6 ADD COLUMN id INT NOT NULL
  AUTO_INCREMENT PRIMARY KEY;
```

Which is less readable.

### Definition 1.1: Primary Key

A column which identifies each row in a table. It must be unique, and it cannot be **NULL**.

To create our **albums** table:

```
1  -- Create albums table: album id, album names, release dates (optional)
2  CREATE TABLE albums (
3      id INT NOT NULL AUTO_INCREMENT,
4      name VARCHAR(255) NOT NULL,
5      release_date DATE,
6      PRIMARY KEY (id)
7  );
8
9
10 -- We need a way to link bands to albums
11 -- Create a column in albums pointing to bands' id column
12 ALTER TABLE albums
13 ADD COLUMN band_id INT NOT NULL FOREIGN KEY REFERENCES bands(id);
```

We created the **albums** table with three columns: **id**, **name**, and **release\_date**, and **band\_id**. A **FOREIGN KEY** uses the **PRIMARY KEY** of another table to establish a relationship between them.

### Definition 1.2: Foreign Key

A column that references another table's **PRIMARY KEY**.

So far we have:

```
1  -- DB: concise_records
2  --
3  --      bands
4  --      +-----+-----+
5  --      | id | name          |
6  --      +-----+-----+
7  --      |   |              |
8  --      +-----+-----+
9  --
10 --      albums
11 --      +-----+-----+-----+-----+
12 --      | id | name          | release_date | band_id |
13 --      +-----+-----+-----+-----+
14 --      |   |              |              |         |
15 --      +-----+-----+-----+-----+
```

- The database **concise\_records**
- Tables **bands** and **albums**.
- Column **band\_id** references **id** in **bands**.

Let's begin to add data to our tables:

```
1  -- Insert 'The Beatles', 'The Rolling Stones', 'The Who' into bands
2  INSERT INTO bands (name) VALUES ('The Beatles');
3  INSERT INTO bands (name) VALUES ('The Rolling Stones'), ('The Who');
4
5  -- DB: concise_records
6  --
7  --      bands
8  --      +-----+-----+
9  --      | id | name                |
10 --      +-----+-----+
11 --      | 1 | The Beatles         |
12 --      | 2 | The Rolling Stones  |
13 --      | 3 | The Who             |
14 --      +-----+-----+
15
16 -- Insert 'Abbey Road', 'Let It Bleed', 'Who's Next' into albums
17
18 INSERT INTO albums (name, release_date, band_id) VALUES ('Abbey Road', '1969', 1);
19 INSERT INTO albums (name, band_id) VALUES ('Let Be', '1970', 1), ('Who's Next', 3);
20
21 -- DB: concise_records
22 --
23 --      albums
24 --      +-----+-----+-----+-----+
25 --      | id | name                | release_date | band_id |
26 --      +-----+-----+-----+-----+
27 --      | 1 | Abbey Road         | 1969         | 1       |
28 --      | 2 | Let It Be          | 1970         | 1       |
29 --      | 3 | Who's Next         |              | 3       |
30 --      +-----+-----+-----+-----+
```

**Note:** Single quotes denote [strings](#). Double single quotes in strings act as single quotes, seen in the above with 'Who's next' becoming `Who's next`.

## 1.2 Queries

To retrieve data, we use [SELECT](#):

```
1  -- Retrieve all columns from bands
2  SELECT * FROM bands;
3
4  -- Query Result:
5  --      +-----+-----+
6  --      | id | name                |
7  --      +-----+-----+
8  --      | 1 | The Beatles         |
9  --      | 2 | The Rolling Stones  |
10 --      | 3 | The Who             |
11 --      +-----+-----+
```

Queries to a table return another table.

## Examples:

### 1. `SELECT column`

```
1  -- Retrieve the name column from bands
2  SELECT name FROM bands;
3
4  -- Query Result:
5  -- +-----+
6  -- | name      |
7  -- +-----+
8  -- | The Beatles |
9  -- | The Rolling Stones |
10 -- | The Who    |
11 -- +-----+
```

### 2. `LIMIT`

```
1  -- Retrieve the name column from bands, limit to 1
2  SELECT name FROM bands LIMIT 1;
3
4  -- Query Result:
5  -- +-----+
6  -- | name      |
7  -- +-----+
8  -- | The Beatles |
9  -- +-----+
```

### 3. `AS`

```
1  -- Retrieve and give aliases to id and name columns from bands, limit to 1
2  SELECT id AS 'ID', name AS 'Band Name' LIMIT 1;
3
4  -- Query Result:
5  -- +-----+
6  -- | ID | Band Name |
7  -- +-----+
8  -- | 1  | The Beatles |
9  -- +-----+
```

### 4. `ORDER BY (DESC/ASC)`

```
1  -- Order bands by name in descending order
2  SELECT * FROM bands ORDER BY name DESC;
3
4  -- Query Result:
5  -- +-----+
6  -- | id | name      |
7  -- +-----+
8  -- | 1  | The Who    |
9  -- | 2  | The Rolling Stones |
10 -- | 3  | The Beatles |
11 -- +-----+
12
13 -- Order bands in ascending order
14 SELECT * FROM bands ORDER BY name ASC;
15 -- which can be shortened to
16 SELECT * FROM bands ORDER BY name; -- as ASC is the default
```

Here we used the `LIMIT`, `AS`, and `ORDER BY (ASC/DESC)` commands.

## 5. DISTINCT

```
1  -- Say we had the following table:
2  -- DB: school_table
3  --
4  --      students
5  --      +-----+-----+
6  --      | id | name                |
7  --      +-----+-----+
8  --      | 1  | Joe                  |
9  --      | 2  | Joe                  |
10 --      | 3  | Joe                  |
11 --      | 4  | Alvin                |
12 --      +-----+-----+
13
14 -- Retrieve all unique names from students
15 USE school_table;
16 SELECT DISTINCT name FROM students;
17
18 -- Query Result:
19 --      +-----+-----+
20 --      | name                |
21 --      +-----+-----+
22 --      | Joe                  |
23 --      | Alvin                |
24 --      +-----+-----+
```

## 1.3 Mutating Data & Filtering

Again, visiting our [concise\\_records](#) example:

```
1  -- DB: concise_records
2  --
3  --      bands
4  --      +-----+-----+
5  --      | id | name                |
6  --      +-----+-----+
7  --      | 1  | The Beatles         |
8  --      | 2  | The Rolling Stones  |
9  --      | 3  | The Who             |
10 --      +-----+-----+
11
12 --      albums
13 --      +-----+-----+-----+-----+
14 --      | id | name                | release_date | band_id |
15 --      +-----+-----+-----+-----+
16 --      | 1  | Abbey Road         | 1969         | 1       |
17 --      | 2  | Let It Be          | 1970         | 1       |
18 --      | 3  | Who's Next         |              | 3       |
19 --      +-----+-----+-----+-----+
```

Examples: *(next page.)*

1.

```
1  -- change the release date of Who's Next `1971`:
2  UPDATE albums SET release_date = '1971';
3  -- This would make all albums have the same release date, instead we use WHERE
4  UPDATE albums SET release_date = '1971' WHERE id = 3;
5
6  -- DB: concise_records
7  --
8  --   albums
9  --   +-----+-----+-----+-----+
10 --   | id | name          | release_date | band_id |
11 --   +-----+-----+-----+-----+
12 --   | 1 | Abbey Road   | 1969         | 1       |
13 --   | 2 | Let It Be    | 1970         | 1       |
14 --   | 3 | Who's Next   | 1971         | 3       |
15 --   +-----+-----+-----+-----+
```

We used `id`, as the `name` could change or be duplicated. `WHERE` filters rows based on conditions.

2. `>`, `<`, `>=`, `<=`

```
1  -- Select all albums with a release date greater than 1969
2  SELECT * FROM albums WHERE release_date > 1969;
3
4  -- Query Result:
5  --   +-----+-----+-----+-----+
6  --   | id | name          | release_date | band_id |
7  --   +-----+-----+-----+-----+
8  --   | 2 | Let It Be    | 1970         | 1       |
9  --   | 3 | Who's Next   | 1971         | 3       |
10 --   +-----+-----+-----+-----+
```

3. `LIKE`

```
1  -- Select all albums that contain 'be' in their name
2  SELECT * FROM albums WHERE name LIKE '%be%';
3
4  -- Query Result:
5  --   +-----+-----+-----+-----+
6  --   | id | name          | release_date | band_id |
7  --   +-----+-----+-----+-----+
8  --   | 2 | Let It Be    | 1970         | 1       |
9  --   +-----+-----+-----+-----+
```

'%' wildcard character: E.g. strings `'%x'` end with `'x'`, `'x%'`, start with `'x'`, `'%x%'` contain `'x'`.

4. `OR`

```
1  -- Albums that contain 'x' in their name OR have a release date of 1969
2  SELECT * FROM albums WHERE name LIKE '%x%' OR band_id = 1;
3
4  -- Query Result:
5  --   +-----+-----+-----+-----+
6  --   | id | name          | release_date | band_id |
7  --   +-----+-----+-----+-----+
8  --   | 1 | Abbey Road   | 1969         | 1       |
9  --   | 3 | Who's Next   | 1971         | 3       |
10 --   +-----+-----+-----+-----+
```



## 5. AND

```
1  -- Select all albums that released in 1969 and have an band_id of 1
2  SELECT * FROM albums WHERE release_date = 1969 AND band_id = 1;
3
4  -- Query Result:
5  +-----+-----+-----+-----+
6  | id | name       | release_date | band_id |
7  +-----+-----+-----+-----+
8  | 1  | Abbey Road | 1969         | 1       |
9  +-----+-----+-----+-----+
```

## 6. BETWEEN

```
1  -- Select all albums that released between 1969 and 1971 (inclusive)
2  SELECT * FROM albums WHERE release_date BETWEEN 1969 AND 1971;
3
4  -- Query Result:
5  +-----+-----+-----+-----+
6  | id | name       | release_date | band_id |
7  +-----+-----+-----+-----+
8  | 1  | Abbey Road | 1969         | 1       |
9  | 2  | Let It Be  | 1970         | 1       |
10 | 3  | Who's Next | 1971         | 3       |
11 +-----+-----+-----+-----+
```

## 7a. IS NULL

```
1  -- Add a new album with a NULL release date
2  INSERT INTO albums (name, band_id) VALUES ('The Wall', 2);
3
4  -- Select all albums with a NULL release date
5  SELECT * FROM albums WHERE release_date IS NULL;
6
7  -- Query Result:
8  +-----+-----+-----+-----+
9  | id | name       | release_date | band_id |
10 +-----+-----+-----+-----+
11 | 4  | The Wall   |              | 2       |
12 +-----+-----+-----+-----+
```

## 7b. DELETE

```
1  -- Delete all albums
2  DELETE FROM albums;
3  -- lets not do this though, instead
4
5  -- Delete all albums with name 'The Wall' (added in previous example)
6  DELETE FROM albums WHERE name = 'The Wall';
7
8  -- albums
9  +-----+-----+-----+-----+
10 | id | name       | release_date | band_id |
11 +-----+-----+-----+-----+
12 | 1  | Abbey Road | 1969         | 1       |
13 | 2  | Let It Be  | 1970         | 1       |
14 | 3  | Who's Next | 1971         | 3       |
15 +-----+-----+-----+-----+
```

## 1.4 Joining Tables

We can **JOIN** tables based on common columns, this serves as their **relationship**. We have:

```
1  -- DB: concise_records
2  --
3  --      bands
4  --      +-----+
5  --      | id | name           |
6  --      +-----+
7  --      | 1 | The Beatles      |
8  --      | 2 | The Rolling Stones |
9  --      | 3 | The Who         |
10 --      +-----+
11 --
12 --      albums
13 --      +-----+-----+-----+
14 --      | id | name           | release_date | band_id |
15 --      +-----+-----+-----+
16 --      | 1 | Abbey Road    | 1969         | 1       |
17 --      | 2 | Let It Be     | 1970         | 1       |
18 --      | 3 | Who's Next    | 1971         | 3       |
19 --      +-----+-----+-----+
```

**Examples:**

1a. **JOIN**

```
1  -- Retrieve all columns from bands and albums
2  SELECT * FROM bands JOIN albums;
```

This command doesn't specify how to combine the tables, so the **Cartesian product** is returned.

### Definition 1.3: Cartesian Product

Sets  $A = \{a_1, a_2, \dots, a_n\}$ ,  $B = \{b_1, b_2, \dots, b_m\}$  match in ordered pairs, e.g.,  $(a_2, b_5)$  or  $(a_9, b_3)$ . The set of all ordered pair combinations of  $A$  on  $B$  is the Cartesian product.

Denoted:  $A \times B$ .

1b.

```
1  -- Query Result:
2  -- +-----+-----+-----+-----+
3  -- | id | name           | id | name           | release_date | band_id |
4  -- +-----+-----+-----+-----+
5  -- | 1 | The Beatles      | 1 | Abbey Road    | 1969         | 1       |
6  -- | 1 | The Beatles      | 2 | Let It Be     | 1970         | 1       |
7  -- | 1 | The Beatles      | 3 | Who's Next    | 1971         | 3       |
8  -- | 2 | The Rolling Stones | 1 | Abbey Road    | 1969         | 1       |
9  -- | 2 | The Rolling Stones | 2 | Let It Be     | 1970         | 1       |
10 -- | 2 | The Rolling Stones | 3 | Who's Next    | 1971         | 3       |
11 -- | 3 | The Who         | 1 | Abbey Road    | 1969         | 1       |
12 -- | 3 | The Who         | 2 | Let It Be     | 1970         | 1       |
13 -- | 3 | The Who         | 3 | Who's Next    | 1971         | 3       |
14 -- +-----+-----+-----+-----+
```

## 2. JOIN ON

```
1  -- Retrieve all columns from bands and albums where band_id matches id
2  SELECT * FROM bands JOIN albums ON bands.id = albums.band_id;
3
4  -- Query Result:
5  +-----+-----+-----+-----+-----+-----+
6  -- | id | name           | id | name           | release_date | band_id |
7  -- +-----+-----+-----+-----+-----+-----+
8  -- | 1 | The Beatles    | 1 | Abbey Road     | 1969         | 1       |
9  -- | 1 | The Beatles    | 2 | Let It Be      | 1970         | 1       |
10 -- | 3 | The Who        | 3 | Who's Next     | 1971         | 3       |
11 -- +-----+-----+-----+-----+-----+-----+
```

## 3. INNER JOIN - (default join type)

```
1  -- Retrieve all columns from bands and albums where band_id matches id
2  SELECT * FROM bands INNER JOIN albums ON bands.id = albums.band_id;
3
4  -- Query Result:
5  +-----+-----+-----+-----+-----+-----+
6  -- | id | name           | id | name           | release_date | band_id |
7  -- +-----+-----+-----+-----+-----+-----+
8  -- | 1 | The Beatles    | 1 | Abbey Road     | 1969         | 1       |
9  -- | 1 | The Beatles    | 2 | Let It Be      | 1970         | 1       |
10 -- | 3 | The Who        | 3 | Who's Next     | 1971         | 3       |
11 -- +-----+-----+-----+-----+-----+-----+
```

## 4. LEFT JOIN

```
1  -- Retrieve all columns from bands and albums, include all bands
2  SELECT * FROM bands LEFT JOIN albums ON bands.id = albums.band_id;
3
4  -- Query Result:
5  +-----+-----+-----+-----+-----+-----+
6  -- | id | name           | id | name           | release_date | band_id |
7  -- +-----+-----+-----+-----+-----+-----+
8  -- | 1 | The Beatles    | 1 | Abbey Road     | 1969         | 1       |
9  -- | 1 | The Beatles    | 2 | Let It Be      | 1970         | 1       |
10 -- | 2 | The Rolling Stones | NULL | NULL         | NULL         | NULL    |
11 -- | 3 | The Who        | 3 | Who's Next     | 1971         | 3       |
12 -- +-----+-----+-----+-----+-----+-----+
```

## 5. RIGHT JOIN

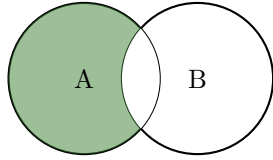
```
1  -- Retrieve all columns from bands and albums, include all albums
2  SELECT * FROM bands RIGHT JOIN albums ON bands.id = albums.band_id;
3
4  -- Query Result:
5  +-----+-----+-----+-----+-----+-----+
6  -- | id | name           | id | name           | release_date | band_id |
7  -- +-----+-----+-----+-----+-----+-----+
8  -- | 1 | The Beatles    | 1 | Abbey Road     | 1969         | 1       |
9  -- | 1 | The Beatles    | 2 | Let It Be      | 1970         | 1       |
10 -- | 3 | The Who        | 3 | Who's Next     | 1971         | 3       |
11 -- +-----+-----+-----+-----+-----+-----+
```

### 1.4.1 JOIN Cheat Sheet

The rest of the [JOINS](#) are shown on the next page.

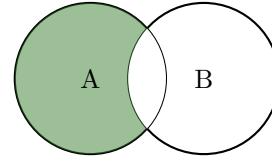
## SQL JOIN Cheat Sheet:

### 1. INNER JOIN



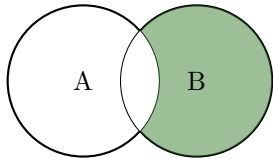
```
SELECT * FROM A  
INNER JOIN B ON A.Key = B.Key
```

### 2. LEFT JOIN (LEFT OUTER JOIN)



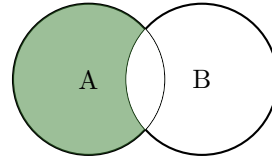
```
SELECT * FROM A  
LEFT JOIN B ON A.Key = B.Key
```

### 3. RIGHT JOIN (RIGHT OUTER JOIN)



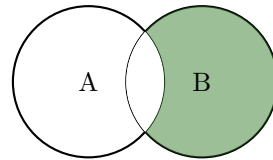
```
SELECT * FROM A  
RIGHT JOIN B ON A.Key = B.Key
```

### 4. LEFT JOIN with NULL check



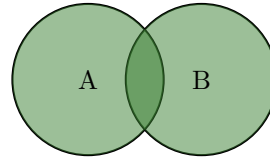
```
SELECT * FROM A  
LEFT JOIN B ON A.Key = B.Key  
WHERE B.Key IS NULL
```

### 5. RIGHT JOIN with NULL check



```
SELECT * FROM A  
RIGHT JOIN B ON A.Key = B.Key  
WHERE A.Key IS NULL
```

### 6. FULL OUTER JOIN



```
SELECT * FROM A  
FULL OUTER JOIN B ON A.Key = B.Key
```

### 7. CROSS JOIN

```
SELECT * FROM A CROSS JOIN B ≡  
SELECT * FROM A JOIN B ≡  
 $A \times B$  The Cartesian product.
```

### Cartesian Product

Sets  $A = \{a_1, a_2, \dots, a_n\}$ ,  $B = \{b_1, b_2, \dots, b_m\}$  can match in ordered pairs, e.g.,  $(a_2, b_5)$  or  $(a_3, b_3)$ . The set of all ordered pair combinations of  $A$  on  $B$  is the Cartesian product.

Denoted:  $A \times B$ .

## 1.5 Aggregate Functions

We can aggregate, i.e., decipher trends in our data via computational functions. Our data:

```
1  -- DB: concise_records
2  --
3  --      bands
4  --      +-----+
5  --      | id | name          |
6  --      +-----+
7  --      | 1 | The Beatles      |
8  --      | 2 | The Rolling Stones |
9  --      | 3 | The Who         |
10 --      +-----+
11 --
12 --      albums
13 --      +-----+-----+-----+
14 --      | id | name          | release_date | band_id |
15 --      +-----+-----+-----+
16 --      | 1 | Abbey Road   | 1969         | 1       |
17 --      | 2 | Let It Be    | 1970         | 1       |
18 --      | 3 | Who's Next   | 1971         | 3       |
19 --      +-----+-----+-----+
```

### Examples:

#### 1. AVG()

```
1  -- Retrieve the average release date of albums
2  SELECT AVG(release_date) FROM albums;
3
4  -- Query Result:
5  --      +-----+
6  --      | AVG(release_date)|
7  --      +-----+
8  --      | 1970              |
9  --      +-----+
```

#### 2. SUM()

```
1  -- Retrieve the sum of release dates of albums
2  SELECT SUM(release_date) FROM albums;
3
4  -- Query Result:
5  --      +-----+
6  --      | SUM(release_date)|
7  --      +-----+
8  --      | 5910              |
9  --      +-----+
```

#### 3. COUNT()

```
1  -- Retrieve the count of albums
2  SELECT COUNT(*) FROM albums;
3
4  -- Query Result:
5  --      +-----+
```

```

6  --      | COUNT(*) |
7  --      +-----+
8  --      | 3       |
9  --      +-----+

```

There are plenty of aggregate functions, which are provided by the [MySQL documentation](#).

NAME	DESCRIPTION
AVG()	Return the average value of the argument
BIT_AND()	Return bitwise AND
BIT_OR()	Return bitwise OR
BIT_XOR()	Return bitwise XOR
COUNT()	Return a count of the number of rows returned
COUNT(DISTINCT)	Return the count of a number of different values
GROUP_CONCAT()	Return a concatenated string
JSON_ARRAYAGG()	Return result set as a single JSON array
JSON_OBJECTAGG()	Return result set as a single JSON object
MAX()	Return the maximum value
MIN()	Return the minimum value
STD()	Return the population standard deviation
STDDEV()	Return the population standard deviation
STDDEV_POP()	Return the population standard deviation
STDDEV_SAMP()	Return the sample standard deviation
SUM()	Return the sum
VAR_POP()	Return the population standard variance
VAR_SAMP()	Return the sample variance
VARIANCE()	Return the population standard variance

*This table is taken from the MySQL documentation.*

## 1.6 Grouping Data

Aggregates can be used with **GROUP BY** to assist in organizing data. Given:

```
1  -- DB: concise_records
2  --
3  --      bands
4  --      +-----+-----+
5  --      | id | name           |
6  --      +-----+-----+
7  --      | 1 | The Beatles      |
8  --      | 2 | The Rolling Stones |
9  --      | 3 | The Who          |
10 --      +-----+-----+
11 --
12 --      albums
13 --      +-----+-----+-----+
14 --      | id | name           | release_date | band_id |
15 --      +-----+-----+-----+
16 --      | 1 | Abbey Road     | 1969         | 1       |
17 --      | 2 | Let It Be      | 1970         | 1       |
18 --      | 3 | Who's Next     | 1971         | 3       |
19 --      +-----+-----+-----+
```

### Examples:

#### 1. GROUP BY

```
1  -- Retrieve the count of albums per band
2  SELECT band_id, COUNT(band_id) FROM albums GROUP BY band_id;
3
4  -- Query Result:
5  --      +-----+-----+
6  --      | band_id | COUNT(band_id) |
7  --      +-----+-----+
8  --      | 1       | 2               |
9  --      | 3       | 1               |
10 --      +-----+-----+
```

Selects **band\_id**, performs **COUNT** on each **band\_ids**. Then group return by **band\_id**.

#### 1b. GROUP BY with JOIN

```
1  -- Retrieve the count of albums per band
2  SELECT bands.name, COUNT(albums.id)
3  FROM bands LEFT JOIN albums ON bands.id = albums.band_id
4  GROUP BY bands.id;
5
6  -- Query Result:
7  --      +-----+-----+
8  --      | name           | COUNT(band_id) |
9  --      +-----+-----+
10 --      | The Beatles      | 2               |
11 --      | The Who          | 1               |
12 --      | The Rolling Stones | 0               |
13 --      +-----+-----+
```

### 1c. GROUP BY with JOIN and AS

```
1 -- Retrieve the count of albums per band
2 SELECT b.name AS 'Band Name', COUNT(a.id) AS 'Album Count'
3 FROM bands AS b
4 LEFT JOIN albums AS a
5 ON b.id = a.band_id
6 GROUP BY b.id;
7
8 -- Query Result:
9
10  +-----+-----+
11  | Band Name          | Album Count |
12  +-----+-----+
13  | The Beatles         | 2           |
14  | The Who             | 1           |
15  | The Rolling Stones  | 0           |
16  +-----+-----+
```

**Note:** We use **LEFT JOIN** to include bands with no albums.

### 2. HAVING

```
1 -- Retrieve the count of albums per band, having more than 1 album
2 SELECT bands.name, COUNT(albums.id)
3 FROM bands
4 LEFT JOIN albums ON bands.id = albums.band_id
5 WHERE COUNT(albums.id) > 1;
6 GROUP BY bands.id
7 -- Error!!!: Cannot use aggregate function in WHERE clause
8 -- We use HAVING instead, which is WHERE, but doesn't filter rows before grouping
9
10
11 -- Retrieve the count of albums per band, having more than 1 album
12 SELECT bands.name, COUNT(albums.id)
13 FROM bands
14 LEFT JOIN albums ON bands.id = albums.band_id
15 GROUP BY bands.id
16 HAVING COUNT(albums.id) > 1;
17
18 -- Query Result:
19
20  +-----+-----+
21  | name          | COUNT(band_id) |
22  +-----+-----+
23  | The Beatles   | 2               |
24  +-----+-----+
```

*This concludes SQL Basics.*