SQL & Security

Christian Rudder

August 2024

Contents

This page is left intentionally blank.

1 Introduction

Hackers, Security issues, and exploits all exists, because the software we use isn't perfect. It's written by humans. One or two edge cases from the best programmers might slip through. This doesn't mean just software, but languages, and even hardware.

The languages we use are just solutions we thought would work. Binary is a solution, it's not the intrinsic solution, but it's a solution. So we built assembly on top, then C, Java, and so on abstracting the difficult parts. But the more we abstract, the more we lose control. A vulnerability can quickly cascade up a chain of abstractions.

2 SQL Basics

2.1 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:

```sql
SELECT * FROM my_table
```

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

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

```sql
CREATE DATABASE test; -- creating a test database
DROP DATABASE test; -- deleting the test database


CREATE DATABASE concise_records; -- creating our database
USE concise_records; -- selecting our database to run commands on it


CREATE TABLE bandds (); -- creating a table for our bands
DROP TABLE bandds; -- deleting it because of our typo


-- Create bands table: artist name (at most 255 characters), CANNOT be NULL/EMPTY
CREATE TABLE bands (
    name VARCHAR(255) NOT NULL
);


-- Add id column to bands, auto increment, not NULL, make this column important
ALTER TABLE bands
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`.

<u>Pagination of commands don't matter,</u> for example:

```sql
-- We could have written the above
CREATE DATABASE concise_records; USE concise_records; CREATE TABLE bands (name VARCHAR
    (255) NOT NULL);

ALTER
TABLE
bands
ADD COLUMN id INT NOT NULL
AUTO_INCREMENT PRIMARY KEY;
```

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

Definition 2.1: 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:

```
-- Create albums table: album id, album names, release dates (optional)
CREATE TABLE albums (
    id INT NOT NULL AUTO_INCREMENT,
    name VARCHAR(255) NOT NULL,
    release_date DATE,
    PRIMARY KEY (id)
);


-- We need a way to link bands to albums
-- Create a column in albums pointing to bands' id column
ALTER TABLE albums
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 2.2: Foreign Key

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

So far we have:

```
-- DB: concise_records
--
--      bands
--      +----+----------------+
--      | id | name           |
--      +----+----------------+
--      |    |                |
--      +----+----------------+
--
--      albums
--      +----+----------------+-------------+---------+
--      | id | name           | release_date | band_id |
--      +----+----------------+-------------+---------+
--      |    |                |             |         |
--      +----+----------------+-------------+---------+
```

- The database `concise_records`
- Tables `bands` and `albums`.
- Column band_id references id in bands.

Let's begin to add data to our tables:

```sql
-- Insert 'The Beatles', 'The Rolling Stones', 'The Who' into bands
INSERT INTO bands (name) VALUES ('The Beatles');
INSERT INTO bands (name) VALUES ('The Rolling Stones'), ('The Who');

-- DB: concise_records
--
--     bands
--     +----+----------------------+
--     | id | name                 |
--     +----+----------------------+s
--     | 1  | The Beatles          |
--     | 2  | The Rolling Stones   |
--     | 3  | The Who              |
--     +----+----------------------+

-- Insert 'Abbey Road', 'Let It Bleed', 'Who's Next' into albums

INSERT INTO albums (name, release_date, band_id) VALUES ('Abbey Road', '1969', 1);
INSERT INTO albums (name, band_id) VALUES ('Let Be', '1970', 1), ('Who''s Next', 3);

-- DB: concise_records
--
--     albums
--     +----+----------------+--------------+---------+
--     | id | name           | release_date | band_id |
--     +----+----------------+--------------+---------+
--     | 1  | Abbey Road     | 1969         | 1       |
--     | 2  | Let It Be      | 1970         | 1       |
--     | 3  | Who's Next     |              | 3       |
--     +----+----------------+--------------+---------+
```

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`.

2.2 Queries

To retrieve data, we use `SELECT`:

```sql
-- Retrieve all columns from bands
SELECT * FROM bands;

-- Query Result:
--     +----+----------------------+
--     | id | name                 |
--     +----+----------------------+
--     | 1  | The Beatles          |
--     | 2  | The Rolling Stones   |
--     | 3  | The Who              |
--     +----+----------------------+
```

Queries to a table return another table.

Examples:

1. SELECT *column*

```sql
-- Retrieve the name column from bands
SELECT name FROM bands;

-- Query Result:
--      +----------------------+
--      | name                 |
--      +----------------------+
--      | The Beatles          |
--      | The Rolling Stones   |
--      | The Who              |
--      +----------------------+
```

2. LIMIT

```sql
-- Retrieve the name column from bands, limit to 1
SELECT name FROM bands LIMIT 1;

-- Query Result:
--      +----------------------+
--      | name                 |
--      +----------------------+
--      | The Beatles          |
--      +----------------------+
```

3. AS

```sql
-- Retrieve and give aliases to id and name columns from bands, limit to 1
SELECT id AS 'ID', name AS 'Band Name' LIMIT 1;

-- Query Result:
--      +----+----------------------+
--      | ID | Band Name            |
--      +----+----------------------+
--      | 1  | The Beatles          |
--      +----+----------------------+
```

4. ORDER BY (DESC/ASC)

```sql
-- Order bands by name in descending order
SELECT * FROM bands ORDER BY name DESC;

-- Query Result:
--      +----+----------------------+
--      | id | name                 |
--      +----+----------------------+
--      | 1  | The Who              |
--      | 2  | The Rolling Stones   |
--      | 3  | The Beatles          |
--      +----+----------------------+

-- Order bands in ascending order
SELECT * FROM bands ORDER BY name ASC;
-- which can be shortened to
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

```sql
-- Say we had the following table:
-- DB: school_table
--
--     students
--     +----+----------------------+
--     | id | name                 |
--     +----+----------------------+
--     | 1  | Joe                  |
--     | 2  | Joe                  |
--     | 3  | Joe                  |
--     | 4  | Alvin                |
--     +----+----------------------+

-- Retrieve all unique names from students
USE school_table;
SELECT DISTINCT name FROM students;

-- Query Result:
--     +----------------------+
--     | name                 |
--     +----------------------+
--     | Joe                  |
--     | Alvin                |
--     +----------------------+
```

2.3 Mutating Data & Filtering

Again, visiting our concise_records example:

```sql
-- DB: concise_records
--
--     bands
--     +----+----------------------+
--     | id | name                 |
--     +----+----------------------+
--     | 1  | The Beatles          |
--     | 2  | The Rolling Stones   |
--     | 3  | The Who              |
--     +----+----------------------+
--
--     albums
--     +----+----------------+--------------+---------+
--     | id | name           | release_date | band_id |
--     +----+----------------+--------------+---------+
--     | 1  | Abbey Road     | 1969         | 1       |
--     | 2  | Let It Be      | 1970         | 1       |
--     | 3  | Who's Next     |              | 3       |
--     +----+----------------+--------------+---------+
```

We will begin to update and filter for specific entries.

To change the release date of `Who's Next` to '1971':

```sql
UPDATE albums SET release_date = '1971';
-- But that would result in all albums having the same release date
-- instead we use WHERE

UPDATE albums SET release_date = '1971' WHERE id = 3;

-- DB: concise_records
--
--     albums
--     +----+----------------+--------------+---------+
--     | id | name           | release_date | band_id |
--     +----+----------------+--------------+---------+
--     | 1  | Abbey Road     | 1969         | 1       |
--     | 2  | Let It Be      | 1970         | 1       |
--     | 3  | Who's Next     | 1971         | 3       |
--     +----+----------------+--------------+---------+
```

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

1. `>, <, >=, <=`

```sql
-- Select all albums with a release date greater than 1969
SELECT * FROM albums WHERE release_date > 1969;

-- Query Result:
--     +----+----------------+--------------+---------+
--     | id | name           | release_date | band_id |
--     +----+----------------+--------------+---------+
--     | 2  | Let It Be      | 1970         | 1       |
--     | 3  | Who's Next     | 1971         | 3       |
--     +----+----------------+--------------+---------+
```

2. `LIKE`

```sql
-- Select all albums that contain 'be' in their name
SELECT * FROM albums WHERE name LIKE '%be%';

-- Query Result:
--     +----+----------------+--------------+---------+
--     | id | name           | release_date | band_id |
--     +----+----------------+--------------+---------+
--     | 2  | Let It Be      | 1970         | 1       |
--     +----+----------------+--------------+---------+
```

3. `OR`

```sql
-- Albums that contain 'x' in their name OR have a release date of 1969
SELECT * FROM albums WHERE name LIKE '%x%' OR band_id = 1;

-- Query Result:
--     +----+----------------+--------------+---------+
--     | id | name           | release_date | band_id |
--     +----+----------------+--------------+---------+
--     | 1  | Abbey Road     | 1969         | 1       |
--     | 3  | Who's Next     | 1971         | 3       |
--     +----+----------------+--------------+---------+
```

4. AND

```sql
-- Select all albums that released in 1969 and have an band_id of 1
SELECT * FROM albums WHERE release_date = 1969 AND band_id = 1;

-- Query Result:
--     +----+----------------+--------------+---------+
--     | id | name           | release_date | band_id |
--     +----+----------------+--------------+---------+
--     | 1  | Abbey Road     | 1969         | 1       |
--     +----+----------------+--------------+---------+
```

5. BETWEEN

```sql
-- Select all albums that released between 1969 and 1971 (incluvise)
SELECT * FROM albums WHERE release_date BETWEEN 1900 AND 1980;

-- Query Result:
--     +----+----------------+--------------+---------+
--     | id | name           | release_date | band_id |
--     +----+----------------+--------------+---------+
--     | 1  | Abbey Road     | 1969         | 1       |
--     | 2  | Let It Be      | 1970         | 1       |
--     | 3  | Who's Next     | 1971         | 3       |
--     +----+----------------+--------------+---------+
```

6a. IS NULL

```sql
-- Add a new album with a NULL release date
INSERT INTO albums (name, band_id) VALUES ('The Wall', 2);

-- Select all albums with a NULL release date
SELECT * FROM albums WHERE release_date IS NULL;

-- Query Result:
-- +----+------------+--------------+---------+
-- | id | name       | release_date | band_id |
-- +----+------------+--------------+---------+
-- | 4  | The Wall   |              | 2       |
-- +----+------------+--------------+---------+
```

6b. DELETE

```sql
-- Delete all albums
DELETE FROM albums;
-- lets not do this though, instead

-- Delete all albums with name 'The Wall' (added in previous example)
DELETE FROM albums WHERE name = 'The Wall';

-- albums
-- +----+----------------+--------------+---------+
-- | id | name           | release_date | band_id |
-- +----+----------------+--------------+---------+
-- | 1  | Abbey Road     | 1969         | 1       |
-- | 2  | Let It Be      | 1970         | 1       |
-- | 3  | Who's Next     | 1971         | 3       |
-- +----+----------------+--------------+---------+
```

2.4 Joining Tables

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

```
-- DB: concise_records
--
--     bands
--     +----+----------------------+
--     | id | name                 |
--     +----+----------------------+
--     | 1  | The Beatles          |
--     | 2  | The Rolling Stones   |
--     | 3  | The Who              |
--     +----+----------------------+
--
--     albums
--     +----+----------------+--------------+---------+
--     | id | name           | release_date | band_id |
--     +----+----------------+--------------+---------+
--     | 1  | Abbey Road     | 1969         | 1       |
--     | 2  | Let It Be      | 1970         | 1       |
--     | 3  | Who's Next     | 1971         | 3       |
--     +----+----------------+--------------+---------+
```

Examples:
1a. `JOIN`

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

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

Definition 2.3: Cartesian Product

Sets $A = \{a_1, a_2, ..., a_n\}$, $B = \{b_1, b_2, ..., b_m\}$ 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.

```
-- Query Result:
-- +----+----------------------+----+----------------+--------------+---------+
-- | id | name                 | id | name           | release_date | band_id |
-- +----+----------------------+----+----------------+--------------+---------+
-- |  1 | The Beatles          |  1 | Abbey Road     | 1969         | 1       |
-- |  1 | The Beatles          |  2 | Let It Be      | 1970         | 1       |
-- |  1 | The Beatles          |  3 | Who's Next     | 1971         | 3       |
-- |  2 | The Rolling Stones   |  1 | Abbey Road     | 1969         | 1       |
-- |  2 | The Rolling Stones   |  2 | Let It Be      | 1970         | 1       |
-- |  2 | The Rolling Stones   |  3 | Who's Next     | 1971         | 3       |
-- |  3 | The Who              |  1 | Abbey Road     | 1969         | 1       |
-- |  3 | The Who              |  2 | Let It Be      | 1970         | 1       |
-- |  3 | The Who              |  3 | Who's Next     | 1971         | 3       |
-- +----+----------------------+----+----------------+--------------+---------+
```

2. JOIN ON

```sql
-- Retrieve all columns from bands and albums where band_id matches id
SELECT * FROM bands JOIN albums ON bands.id = albums.band_id;

-- Query Result:
-- +----+----------------------+----+----------------+--------------+---------+
-- | id | name                 | id | name           | release_date | band_id |
-- +----+----------------------+----+----------------+--------------+---------+
-- |  1 | The Beatles          |  1 | Abbey Road     | 1969         | 1       |
-- |  1 | The Beatles          |  2 | Let It Be      | 1970         | 1       |
-- |  3 | The Who              |  3 | Who's Next     | 1971         | 3       |
-- +----+----------------------+----+----------------+--------------+---------+
```

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

```sql
-- Retrieve all columns from bands and albums where band_id matches id
SELECT * FROM bands INNER JOIN albums ON bands.id = albums.band_id;

-- Query Result:
-- +----+----------------------+----+----------------+--------------+---------+
-- | id | name                 | id | name           | release_date | band_id |
-- +----+----------------------+----+----------------+--------------+---------+
-- |  1 | The Beatles          |  1 | Abbey Road     | 1969         | 1       |
-- |  1 | The Beatles          |  2 | Let It Be      | 1970         | 1       |
-- |  3 | The Who              |  3 | Who's Next     | 1971         | 3       |
-- +----+----------------------+----+----------------+--------------+---------+
```

4. LEFT JOIN

```sql
-- Retrieve all columns from bands and albums, include all bands
SELECT * FROM bands LEFT JOIN albums ON bands.id = albums.band_id;

-- Query Result:
-- +----+----------------------+------+----------------+--------------+---------+
-- | id | name                 | id   | name           | release_date | band_id |
-- +----+----------------------+------+----------------+--------------+---------+
-- |  1 | The Beatles          |  1   | Abbey Road     | 1969         | 1       |
-- |  1 | The Beatles          |  2   | Let It Be      | 1970         | 1       |
-- |  2 | The Rolling Stones   | NULL | NULL           | NULL         | NULL    |
-- |  3 | The Who              |  3   | Who's Next     | 1971         | 3       |
-- +----+----------------------+------+----------------+--------------+---------+
```

5. RIGHT JOIN

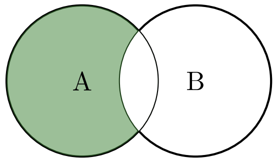
```sql
-- Retrieve all columns from bands and albums, include all albums
SELECT * FROM bands RIGHT JOIN albums ON bands.id = albums.band_id;

-- Query Result:
-- +----+----------------------+------+----------------+--------------+---------+
-- | id | name                 | id   | name           | release_date | band_id |
-- +----+----------------------+------+----------------+--------------+---------+
-- |  1 | The Beatles          |  1   | Abbey Road     | 1969         | 1       |
-- |  1 | The Beatles          |  2   | Let It Be      | 1970         | 1       |
-- |  3 | The Who              |  3   | Who's Next     | 1971         | 3       |
-- +----+----------------------+------+----------------+--------------+---------+
```

2.4.1 JOIN Cheat Sheet

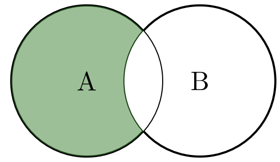
The rest of the JOINs are shown on the next page.

1. INNER JOIN



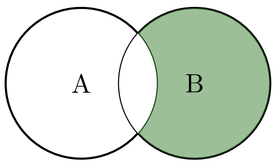
SELECT * FROM A
INNER JOIN B ON A.Key = B.Key

2. LEFT JOIN (LEFT OUTER JOIN)



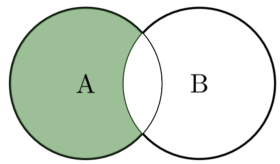
SELECT * FROM A
LEFT JOIN B ON A.Key = B.Key

3. RIGHT JOIN (RIGHT OUTER JOIN)



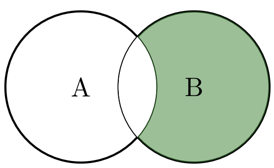
SELECT * FROM A
RIGHT JOIN B ON A.Key = B.Key

4. LEFT JOIN with NULL check



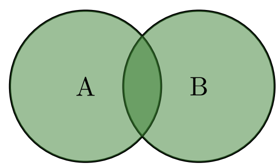
SELECT * FROM A
LEFT JOIN B ON A.Key = B.Key
WHERE B.Key IS NULL

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 $\equiv$
SELECT * FROM A JOIN B $\equiv$
$A \times B$ The **Cartesian product**.

Cartesian Product

Sets $A = \{a_1, a_2, ..., a_n\}, B = \{b_1, b_2, ..., b_m\}$ 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: $A \times B$.