SQL Lesson 1: SELECT queries 101

To retrieve data from a SQL database, we need to write **SELECT** statements, which are often colloquially refered to as *queries*. A query in itself is just a statement which declares what data we are looking for, where to find it in the database, and optionally, how to transform it before it is returned. It has a specific syntax though, which is what we are going to learn in the following exercises.

As we mentioned in the introduction, you can think of a table in SQL as a type of an entity (ie. Dogs), and each row in that table as a specific *instance* of that type (ie. A pug, a beagle, a different colored pug, etc). This means that the columns would then represent the common properties shared by all instances of that entity (ie. Color of fur, length of tail, etc).

And given a table of data, the most basic query we could write would be one that selects for a couple columns (properties) of the table with all the rows (instances).

Select query for a specific columns

```
SELECT column, another_column, …
FROM mytable;
```

The result of this query will be a two-dimensional set of rows and columns, effectively a copy of the table, but only with the columns that we requested.

If we want to retrieve absolutely all the columns of data from a table, we can then use the asterisk (*) shorthand in place of listing all the column names individually.

Select query for all columns

```
SELECT * FROM mytable;
```

This query in particular is really useful because it's a simple way inspect a table by dumping all the data at once.

SQL Lesson 2: Queries with constraints (Pt. 1)

Now we know how to select for specific columns of data from a table, but if you had a table with a hundred million rows of data, reading through all the rows would be inefficient and perhaps even impossible.

In order to filter certain results from being returned, we need to use a **WHERE** clause in the query. The clause is applied to each row of data by checking specific column values to determine whether it should be included in the results or not.

Select query with constraints

```
SELECT column, another_column, …
FROM mytable
WHERE condition
AND/OR another_condition
AND/OR …;
```

More complex clauses can be constructed by joining numerous **AND** or **OR** logical keywords (ie. num_wheels >= 4 AND doors <= 2). And below are some useful operators that you can use for numerical data (ie. integer or floating point):

Operator	Condition	SQL Example
=, !=, < <=, >, >=	Standard numerical operators	col_name **!=** 4
BETWEEN ... AND ...	Number is within range of two values (inclusive)	col_name **BETWEEN** 1.5 **AND** 10.5
NOT BETWEEN ... AND ...	Number is not within range of two values (inclusive)	col_name **NOT BETWEEN** 1 **AND** 10
IN (...)	Number exists in a list	col_name **IN** (2, 4, 6)
NOT IN (...)	Number does not exist in a list	col_name **NOT IN** (1, 3, 5)

In addition to making the results more manageable to understand, writing clauses to constrain the set of rows returned also allows the query to run faster due to the reduction in unnecessary data being returned.

SQL Lesson 3: Queries with constraints (Pt. 2)

When writing **WHERE** clauses with columns containing text data, SQL supports a number of useful operators to do things like case-insensitive string comparison and wildcard pattern matching. We show a few common text-data specific operators below:

Operator	Condition	Example
=	Case sensitive exact string comparison (**notice the single equals**)	col_name **=** "abc"
!= or <>	Case sensitive exact string inequality comparison	col_name **!=** "abcd"
LIKE	Case insensitive exact string comparison	col_name **LIKE** "ABC"
NOT LIKE	Case insensitive exact string inequality comparison	col_name **NOT LIKE** "ABCD"
%	Used anywhere in a string to match a sequence of zero or more characters (only with LIKE or NOT LIKE)	col_name **LIKE** "%AT%" (matches "AT", "ATTIC", "CAT" or even "BATS")
_	Used anywhere in a string to match a single character (only with LIKE or NOT LIKE)	col_name **LIKE** "AN_" (matches "AND", but not "AN")
IN (...)	String exists in a list	col_name **IN** ("A", "B", "C")
NOT IN (...)	String does not exist in a list	col_name **NOT IN** ("D", "E", "F")

Did you know?

All strings must be quoted so that the query parser can distinguish words in the string from SQL keywords.

We should note that while most database implementations are quite efficient when using these operators, full-text search is best left to dedicated libraries like Apache Lucene or Sphinx. These libraries are designed specifically to do full text search, and as a result are more efficient and can support a wider variety of search features including internationalization and advanced queries.

SQL Lesson 4: Filtering and sorting Query results

Even though the data in a database may be unique, the results of any particular query may not be – take our Movies table for example, many different movies can be released the same year. In such cases, SQL provides a convenient way to discard rows that have a duplicate column value by using the **DISTINCT** keyword.

Select query with unique results

```
SELECT DISTINCT column,
another_column, …
FROM mytable
WHERE condition(s);
```

Since the **DISTINCT** keyword will blindly remove duplicate rows, we will learn in a future lesson how to discard duplicates based on specific columns using grouping and the **GROUP BY** clause.

Ordering results

Unlike our neatly ordered table in the last few lessons, most data in real databases are added in no particular column order. As a result, it can be difficult to read through and understand the results of a query as the size of a table increases to thousands or even millions rows.

To help with this, SQL provides a way to sort your results by a given column in ascending or descending order using the **ORDER BY** clause.

Select query with ordered results

```
SELECT column, another_column, …
FROM mytable
WHERE condition(s)
ORDER BY column ASC/DESC;
```

When an **ORDER BY** clause is specified, each row is sorted alpha-numerically based on the specified column's value. In some databases, you can also specify a collation to better sort data containing international text.

Limiting results to a subset

Another clause which is commonly used with the **ORDER BY** clause are the **LIMIT** and **OFFSET**clauses, which are a useful optimization to indicate to the database the subset of the results you care about. The **LIMIT** will reduce the number of rows to return, and the optional **OFFSET** will specify where to begin counting the number rows from.

Select query with limited rows

```
SELECT column, another_column, …
FROM mytable
WHERE condition(s)
ORDER BY column ASC/DESC
LIMIT num_limit OFFSET num_offset;
```

If you think about websites like Reddit or Pinterest, the front page is a list of links sorted by popularity and time, and each subsequent page can be represented by sets of links at different offsets in the database. Using these clauses, the database can then execute queries faster and more efficiently by processing and returning only the requested content.

Did you know?

If you think about websites like Reddit or Pinterest, the front page is a list of links sorted by popularity and time, and each subsequent page can be represented by sets of links at different offsets in the database. Using these clauses, the database can then execute queries faster and more efficiently by processing and returning only the requested content.

SQL Lesson 6: Multi-table queries with JOINs

Up to now, we've been working with a single table, but entity data in the real world is often broken down into pieces and stored across multiple orthogonal tables using a process known as *normalization*[1].

Database normalization

Database normalization is useful because it minimizes duplicate data in any single table, and allows for data in the database to grow independently of each other (ie. Types of car engines can grow independent of each type of car). As a trade-off, queries get slightly more complex since they have to be able to find data from different parts of the database, and performance issues can arise when working with many large tables.

In order to answer questions about an entity that has data spanning multiple tables in a normalized database, we need to learn how to write a query that can combine all that data and pull out exactly the information we need.

Multi-table queries with JOINs

Tables that share information about a single entity need to have a *primary key* that identifies that entity *uniquely* across the database. One common primary key type is an auto-incrementing integer (because they are space efficient), but it can also be a string, hashed value, so long as it is unique.

Using the **JOIN** clause in a query, we can combine row data across two separate tables using this unique key. The first of the joins that we will introduce is the **INNER JOIN**.

Select query with INNER JOIN on multiple tables

```
SELECT column, another_table_column, …
FROM mytable INNER JOIN another_table
     ON mytable.id = another_table.id
WHERE condition(s)
ORDER BY column, … ASC/DESC
LIMIT num_limit OFFSET num_offset;
```

The **INNER JOIN** is a process that matches rows from the first table and the second table which have the same key (as defined by the **ON** constraint) to create a result row with the combined columns from both tables. After the tables are joined, the other clauses we learned previously are then applied.

Did you know?

You might see queries where the **INNER JOIN** is written simply as a **JOIN**. These two are equivalent, but we will continue to refer to these joins as inner-joins because they make the query easier to read once you start using other types of joins, which will be introduced in the following lesson.

SQL Lesson 7: OUTER JOINs

Depending on how you want to analyze the data, the **INNER JOIN** we used last lesson might not be sufficient because the resulting table only contains data that belongs in both of the tables.

If the two tables have asymmetric data, which can easily happen when data is entered in different stages, then we would have to use a **LEFT JOIN**, **RIGHT JOIN** or **FULL JOIN** instead to ensure that the data you need is not left out of the results.

Select query with LEFT/RIGHT/FULL JOINs on multiple tables

```
SELECT column, another_column, …
FROM mytable INNER/LEFT/RIGHT/FULL JOIN another_table
     ON mytable.id = another_table.matching_id
WHERE condition(s)
ORDER BY column, … ASC/DESC
LIMIT num_limit OFFSET num_offset;
```

Like the **INNER JOIN** these three new joins have to specify which column to join the data on. When joining table A to table B, a **LEFT JOIN** simply includes rows from A regardless of whether a matching row is found in B. The **RIGHT JOIN** is the same, but reversed, keeping rows in B regardless of whether a match is found in A. Finally, a **FULL JOIN** simply means that rows from both tables are kept, regardless of whether a matching row exists in the other table.

When using any of these new joins, you will likely have to write additional logic to deal with **NULL**s in the result and constraints (more on this in the next lesson).

Did you know?

You might see queries written these joins written as **LEFT OUTER JOIN**, **RIGHT OUTER JOIN**, or **FULL OUTER JOIN**, but the **OUTER** keyword is really kept for SQL-92 compatibility and these queries are simply equivalent to **LEFT JOIN**, **RIGHT JOIN**, and **FULL JOIN** respectively.

SQL Lesson 8: A short note on NULLs

As promised in the last lesson, we are going to quickly talk about **NULL** values in an SQL database. It's always good to reduce the possibility of **NULL** values in databases because they require special attention when constructing queries, constraints (certain functions behave differently with null values) and when processing the results.

An alternative to **NULL** values in your database is to have **data-type appropriate default values**, like 0 for numerical data, empty strings for text data, etc. But if your database needs to store incomplete data, then **NULL** values can be appropriate if the default values will skew later analysis (for example, when taking averages of numerical data).

Sometimes, it's also not possible to avoid **NULL** values, as we saw in the last lesson when outer-joining two tables with asymmetric data. In these cases, you can test a column for **NULL** values in a **WHERE** clause by using either the **IS NULL** or **IS NOT NULL** constraint.

Select query with constraints on NULL values

```
SELECT column, another_column, … FROM mytable WHERE column IS/IS NOT NULL
AND/OR another_condition AND/OR …;
```

SQL Lesson 9: Queries with expressions

In addition to querying and referencing raw column data with SQL, you can also use *expressions* to write more complex logic on column values in a query. These expressions can use mathematical and string functions along with basic arithmetic to transform values when the query is executed, as shown in this physics example.

Example query with expressions

```
SELECT particle_speed / 2.0 AS half_particle_speed
FROM physics_data
WHERE ABS(particle_position) * 10.0 > 500;
```

Each database has its own supported set of mathematical, string, and date functions that can be used in a query, which you can find in their own respective docs.

The use of expressions can save time and extra post-processing of the result data, but can also make the query harder to read, so we recommend that when expressions are used in the **SELECT**part of the query, that they are also given a descriptive *alias* using the **AS** keyword.

Select query with expression aliases

```
SELECT col_expression AS expr_description, …
FROM mytable;
```

In addition to expressions, regular columns and even tables can also have aliases to make them easier to reference in the output and as a part of simplifying more complex queries.

Example query with both column and table name aliases

```
SELECT column AS better_column_name, …
FROM a_long_widgets_table_name AS mywidgets
INNER JOIN widget_sales
     ON mywidgets.id = widget_sales.widget_id;
```

SQL Lesson 10: Queries with aggregates (Pt. 1)

In addition to the simple expressions that we introduced last lesson, SQL also supports the use of aggregate expressions (or functions) that allow you to summarize information about a group of rows of data. With the Pixar database that you've been using, aggregate functions can be used to answer questions like, "How many movies has Pixar produced?", or "What is the highest grossing Pixar film each year?".

Select query with aggregate functions over all rows

```
SELECT AGG_FUNC(column_or_expression) AS aggregate_description, … FROM mytable
WHERE constraint_expression;
```

Without a specified grouping, each aggregate function is going to run on the whole set of result rows and return a single value. And like normal expressions, giving your aggregate functions an alias ensures that the results will be easier to read and process.

Common aggregate functions

Here are some common aggregate functions that we are going to use in our examples:

Function	Description
COUNT(*****), **COUNT(***column***)**	A common function used to counts the number of rows in the group if no column name is specified. Otherwise, count the number of rows in the group with non-NULL values in the specified column.
MIN(*column***)**	Finds the smallest numerical value in the specified column for all rows in the group.
MAX(*column***)**	Finds the largest numerical value in the specified column for all rows in the group.
AVG(*column***)**	Finds the average numerical value in the specified column for all rows in the group.
SUM(*column***)**	Finds the sum of all numerical values in the specified column for the rows in the group.

Grouped aggregate functions

In addition to aggregating across all the rows, you can instead apply the aggregate functions to individual groups of data within that group (ie. box office sales for Comedies vs Action movies).
This would then create as many results as there are unique groups defined as by the **GROUP BY**clause.

Select query with aggregate functions over groups

```
SELECT AGG_FUNC(column_or_expression) AS aggregate_description, …
FROM mytable
WHERE constraint_expression GROUP BY column;
```

The **GROUP BY** clause works by grouping rows that have the same value in the column specified.

SQL Lesson 11: Queries with aggregates (Pt. 2)

Our queries are getting fairly complex, but we have nearly introduced all the important parts of a**SELECT** query.
One thing that you might have noticed is that if the **GROUP BY** clause is executed after the **WHERE** clause (which filters the rows which are to be grouped), then how exactly do we filter the grouped rows?

Luckily, SQL allows us to do this by adding an additional **HAVING** clause which is used specifically with the **GROUP BY** clause to allow us to filter grouped rows from the result set.

Select query with HAVING constraint

```
SELECT group_by_column, AGG_FUNC(column_expression) AS aggregate_result_alias,
FROM mytable
WHERE condition
GROUP BY column HAVING group_condition;
```

The **HAVING** clause constraints are written the same way as the **WHERE** clause constraints, and are applied to the grouped rows. With our examples, this might not seem like a particularly useful construct, but if you imagine data with millions of rows with different properties, being able to apply additional constraints is often necessary to quickly make sense of the data.

Did you know?
If you aren't using the `GROUP BY` clause, a simple `WHERE` clause will suffice.

SQL Lesson 12: Order of execution of a Query

Now that we have an idea of all the parts of a query, we can now talk about how they all fit together in the context of a complete query.

Complete SELECT query

```
SELECT DISTINCT column, AGG_FUNC(column_or_expression), …
FROM mytable
JOIN another_table
    ON mytable.column = another_table.column
WHERE constraint_expression
GROUP BY column
HAVING constraint_expression
ORDER BY column ASC/DESC
LIMIT count OFFSET COUNT;
```

Each query begins with finding the data that we need in a database, and then filtering that data down into something that can be processed and understood as quickly as possible. Because each part of the query is executed sequentially, it's important to understand the order of execution so that you know what results are accessible where.

Query order of execution

1. **FROM** and **JOIN**s

The **FROM** clause, and subsequent **JOIN**s are first executed to determine the total working set of data that is being queried. This includes subqueries in this clause, and can cause temporary tables to be created under the hood containing all the columns and rows of the tables being joined.

2. **WHERE**

Once we have the total working set of data, the first-pass **WHERE** constraints are applied to the individual rows, and rows that do not satisfy the constraint are discarded. Each of the constraints can only access columns directly from the tables requested in the **FROM** clause. Aliases in the **SELECT** part of the query are not accessible in most databases since they may include expressions dependent on parts of the query that have not yet executed.

3. **GROUP BY**

The remaining rows after the **WHERE** constraints are applied are then grouped based on common values in the column specified in the **GROUP BY** clause. As a result of the grouping, there will only be as many rows as there are unique values in that column. Implicitly, this means that you should only need to use this when you have aggregate functions in your query.

4. **HAVING**

If the query has a **GROUP BY** clause, then the constraints in the **HAVING** clause are then applied to the grouped rows, discard the grouped rows that don't satisfy the constraint. Like the **WHERE** clause, aliases are also not accessible from this step in most databases.

5. **SELECT**

Any expressions in the **SELECT** part of the query are finally computed.

6. **DISTINCT**

Of the remaining rows, rows with duplicate values in the column marked as **DISTINCT** will be discarded.

7. **ORDER BY**

If an order is specified by the **ORDER BY** clause, the rows are then sorted by the specified data in either ascending or descending order. Since all the expressions in the **SELECT** part of the query have been computed, you can reference aliases in this clause.

8. **LIMIT** / **OFFSET**

Finally, the rows that fall outside the range specified by the **LIMIT** and **OFFSET** are discarded, leaving the final set of rows to be returned from the query.

Conclusion

Not every query needs to have all the parts we listed above, but a part of why SQL is so flexible is that it allows developers and data analysts to quickly manipulate data without having to write additional code, all just by using the above clauses.

SQL Lesson 13: Inserting rows

We've spent quite a few lessons on how to query for data in a database, so it's time to start learning a bit about SQL schemas and how to add new data.

What is a Schema?

We previously described a table in a database as a two-dimensional set of rows and columns, with the columns being the properties and the rows being instances of the entity in the table. In SQL, the *database schema* is what describes the structure of each table, and the datatypes that each column of the table can contain.

Example: Correlated subquery

For example, in our **Movies** table, the values in the **Year** column must be an Integer, and the values in the **Title** column must be a String.

This fixed structure is what allows a database to be efficient, and consistent despite storing millions or even billions of rows.

Inserting new data

When inserting data into a database, we need to use an **INSERT** statement, which declares which table to write into, the columns of data that we are filling, and one or more rows of data to insert. In general, each row of data you insert should contain values for every corresponding column in the table. You can insert multiple rows at a time by just listing them sequentially.

Insert statement with values for all columns

```
INSERT INTO mytable
VALUES (value_or_expr, another_value_or_expr, …), (value_or_expr_2,
another_value_or_expr_2, …),        …;
```

In some cases, if you have incomplete data and the table contains columns that support default values, you can insert rows with only the columns of data you have by specifying them explicitly.

Insert statement with specific columns

```
INSERT INTO mytable
(column, another_column, …)
VALUES (value_or_expr, another_value_or_expr, …), (value_or_expr_2,
another_value_or_expr_2, …),       …;
```

In these cases, the number of values need to match the number of columns specified. Despite this being a more verbose statement to write, inserting values this way has the benefit of being forward compatible. For example, if you add a new column to the table with a default value, no hardcoded **INSERT** statements will have to change as a result to accommodate that change.

In addition, you can use mathematical and string expressions with the values that you are inserting.
This can be useful to ensure that all data inserted is formatted a certain way.

Example Insert statement with expressions

```
INSERT INTO boxoffice
(movie_id, rating, sales_in_millions)
VALUES (1, 9.9, 283742034 / 1000000);
```

SQL Lesson 14: Updating rows

In addition to adding new data, a common task is to update existing data, which can be done using an **UPDATE** statement. Similar to the **INSERT** statement, you have to specify exactly which table, columns, and rows to update. In addition, the data you are updating has to match the data type of the columns in the table schema.

Update statement with values

```
UPDATE mytable
SET column = value_or_expr, other_column = another_value_or_expr, …
WHERE condition;
```

The statement works by taking multiple column/value pairs, and applying those changes to each and every row that satisfies the constraint in the **WHERE** clause.

Taking care

Most people working with SQL **will** make mistakes updating data at one point or another. Whether it's updating the wrong set of rows in a production database, or accidentally leaving out the **WHERE** clause (which causes the update to apply to *all* rows), you need to be extra careful when constructing **UPDATE** statements.

One helpful tip is to always write the constraint first and test it in a **SELECT** query to make sure you are updating the right rows, and only then writing the column/value pairs to update.

SQL Lesson 15: Deleting rows

When you need to delete data from a table in the database, you can use a **DELETE** statement, which describes the table to act on, and the rows of the table to delete through the **WHERE**clause.

Delete statement with condition

```
DELETE FROM mytable WHERE condition;
```

If you decide to leave out the **WHERE** constraint, then *all* rows are removed, which is a quick and easy way to clear out a table completely (if intentional).

Taking extra care

Like the **UPDATE** statement from last lesson, it's recommended that you run the constraint in a **SELECT** query first to ensure that you are removing the right rows. Without a proper backup or test database, it is downright easy to irrevocably remove data, so always read your **DELETE** statements twice and execute once.

SQL Lesson 16: Creating tables

When you have new entities and relationships to store in your database, you can create a new database table using the **CREATE TABLE** statement.

Create table statement w/ optional table constraint and default value

```
CREATE TABLE IF NOT EXISTS mytable (     column DataType TableConstraint
DEFAULT default_value,     another_column DataType TableConstraint DEFAULT
default_value,     … );
```

The structure of the new table is defined by its *table schema*, which defines a series of columns. Each column has a name, the type of data allowed in that column, an *optional* table constraint on values being inserted, and an optional default value.

If there already exists a table with the same name, the SQL implmentation will usually throw an error, so to suppress the error and skip creating a table if one exists, you can use the **IF NOT EXISTS** clause.

Table data types

Different databases support different data types, but the common types support numeric, string, and other miscellaneous things like dates, booleans, or even binary data. Here are some examples that you might use in real code.

Data type	Description
INTEGER, **BOOLEAN**	The integer datatypes can store whole integer values like the count of a number or an age. In some implementations, the boolean value is just represented as an integer value of just 0 or 1.
FLOAT, **DOUBLE**, **REAL**	The floating point datatypes can store more precise numerical data like measurements or fractional values. Different types can be used depending on the floating point precision required for that value.
CHARACTER (num_chars), **VARCHAR(num_chars)**, **TEXT**	The text based datatypes can store strings and text in all sorts of locales. The distinction between the various types generally amount to underlaying efficiency of the database when working with these columns. Both the CHARACTER and VARCHAR (variable character) types are specified with the max number of characters that they can store (longer values may be truncated), so can be more efficient to store and query with big tables.
DATE, **DATETIME**	SQL can also store date and time stamps to keep track of time series and event data. They can be tricky to work with especially when manipulating data across timezones.
BLOB	Finally, SQL can store binary data in blobs right in the database. These values are often opaque to the database, so you usually have to store them with the right metadata to requery them.

Table constraints

We aren't going to dive too deep into table constraints in this lesson, but each column can have additional table constraints on it which limit what values can be inserted into that column. This is not a comprehensive list, but will show a few common constraints that you might find useful.

Constraint	Description
PRIMARY KEY	This means that the values in this column are unique, and each value can be used to identify a single row in this table.
AUTOINCREMENT	For integer values, this means that the value is automatically filled in and incremented with each row insertion. Not supported in all databases.

UNIQUE	This means that the values in this column have to be unique, so you can't insert another row with the same value in this column as another row in the table. Differs from the `PRIMARY KEY` in that it doesn't have to be a key for a row in the table.
NOT NULL	This means that the inserted value can not be `NULL`.
CHECK (expression)	This is allows you to run a more complex expression to test whether the values inserted are value. For example, you can check that values are positive, or greater than a specific size, or start with a certain prefix, etc.
FOREIGN KEY	This is a consistency check which ensures that each value in this column corresponds to another value in a column in another table. For example, if there are two tables, one listing all Employees by ID, and another listing their payroll information, the `FOREIGN KEY` can ensure that every row in the payroll table corresponds to a valid employee in the master Employee list.

Here's an example schema for the **Movies** table that we've been using in the lessons up to now.

Movies table schema
```
CREATE TABLE movies (
      id INTEGER PRIMARY KEY,
      title TEXT,
      director TEXT,
      year INTEGER,
      length_minutes INTEGER
);
```

SQL Lesson 17: Altering tables

As your data changes over time, SQL provides a way for you to update your corresponding tables and database schemas by using the **ALTER TABLE** statement to add, remove, or modify columns and table constraints.

Adding columns

The syntax for adding a new column is similar to the syntax when creating new rows in the **CREATE TABLE** statement. You need to specify the data type of the column along with any potential table constraints and default values to be applied to both existing *and* new rows. In some databases like MySQL, you can even specify where to insert the new column using the **FIRST** or **AFTER** clauses, though this is not a standard feature.

Altering table to add new column(s)
```
ALTER TABLE mytable
ADD column DataType OptionalTableConstraint
      DEFAULT default_value;
```

Removing columns

Dropping columns is as easy as specifying the column to drop, however, many databases (including Postgres, and SQLite) don't support this feature. Instead you may have to create a new table and migrate the data over.

Altering table to remove column(s)

```
ALTER TABLE mytable
DROP column_to_be_deleted;
```

Renaming the table

If you need to rename the table itself, you can also do that using the **RENAME TO** clause of the statement.

Altering table name
```
ALTER TABLE mytable
RENAME TO new_table_name;
```

SQL Lesson 18: Dropping tables

In some rare cases, you may want to remove an entire table including all of its data and metadata, and to do so, you can use the **DROP TABLE** statement, which differs from the **DELETE** statement in that it also removes the table schema from the database entirely.

Drop table statement
```
DROP TABLE IF EXISTS mytable;
```

Like the **CREATE TABLE** statement, the database may throw an error if the specified table does not exist, and to suppress that error, you can use the **IF EXISTS** clause.

In addition, if you have another table that is dependent on columns in table you are removing (for example, with a **FOREIGN KEY** dependency) then you will have to either update all dependent tables first to remove the dependent rows or to remove those tables entirely.