**Introduction to SQL**

Welcome to SQLBolt, a series of interactive lessons and exercises designed to help you quickly learn SQL right in your browser.

**What is SQL?**

SQL, or Structured Query Language, is a language designed to allow both technical and non-technical users query, manipulate, and transform data from a relational database. And due to its simplicity, SQL databases provide safe and scalable storage for millions of websites and mobile applications.

Did you know?

There are many popular SQL databases including SQLite, MySQL, Postgres, Oracle and Microsoft SQL Server. All of them support the common SQL language standard, which is what this site will be teaching, but each implementation can differ in the additional features and storage types it supports.

**Relational databases**

Before learning the SQL syntax, it's important to have a model for what a relational database actually is. A relational database represents a collection of related (two-dimensional) tables. Each of the tables are similar to an Excel spreadsheet, with a fixed number of named columns (the attributes or properties of the table) and any number of rows of data.

For example, if the Department of Motor Vehicles had a database, you might find a table containing all the known vehicles that people in the state are driving. This table might need to store the model name, type, number of wheels, and number of doors of each vehicle for example.

Table: Vehicles

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Id** | **Make/Model** | **# Wheels** | **# Doors** | **Type** |
| 1 | Ford Focus | 4 | 4 | Sedan |
| 2 | Tesla Roadster | 4 | 2 | Sports |
| 3 | Kawakasi Ninja | 2 | 0 | Motorcycle |
| 4 | McLaren Formula 1 | 4 | 0 | Race |
| 5 | Tesla S | 4 | 4 | Sedan |

In such a database, you might find additional related tables containing information such as a list of all registered drivers in the state, the types of driving licenses that can be granted, or even driving violations for each driver.

By learning SQL, the goal is to learn how to answer specific questions about this data, like *"What types of vehicles are on the road have less than four wheels?"*, or *"How many models of cars does Tesla produce?"*, to help us make better decisions down the road.

**About the lessons**

Since most users will be learning SQL to interact with an existing database, the lessons begin by introducing you to the various parts of an SQL query. The later lessons will then show you how to alter a table (or schema) and create new tables from scratch.

Each lesson will introduce a different concept and end with an interactive exercise. Go at your pace and don't be afraid to spend time experimenting with the exercises before continuing! If you happen to be familiar with SQL already, you can skip ahead using the links in the top-right, but we would recommend you work through the lessons anyways!

By the end, we hope you will be able to have a strong foundation for using SQL in your own projects and beyond.

Lesson 1

**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 to inspect a table by dumping all the data at once.

**Exercise**

We will be using a database with data about some of Pixar's classic movies for most of our exercises. This first exercise will only involve the **Movies** table, and the default query below currently shows all the properties of each movie. To continue onto the next lesson, alter the query to find the exact information we need for each task.

Lesson 2 – Select Queries with Constraints

**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.

Did you know?

As you might have noticed by now, SQL doesn't *require* you to write the keywords all capitalized, but as a convention, it helps people distinguish SQL keywords from column and tables names, and makes the query easier to read.

**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](http://lucene.apache.org/) or [Sphinx](http://sphinxsearch.com/). 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.

**SQL Review: Simple SELECT Queries**

You've done a good job getting to this point! Now that you've gotten a taste of how to write a basic query, you need to practice writing queries that solve actual problems.

SELECT query

SELECT column, another\_column, …

FROM mytable

WHERE *condition(s)*

ORDER BY column ASC/DESC

LIMIT num\_limit OFFSET num\_offset;

**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]](http://en.wikipedia.org/wiki/Database_normalization).

**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.

**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 with 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

|  |  |
| --- | --- |
| **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. |
| Docs: [MySQL](https://dev.mysql.com/doc/refman/5.6/en/group-by-functions.html), [Postgres](http://www.postgresql.org/docs/9.4/static/functions-aggregate.html), [SQLite](http://www.sqlite.org/lang_aggfunc.html), [Microsoft SQL Server](https://msdn.microsoft.com/en-us/library/ms173454.aspx) | |

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:

**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.

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 JOINs**

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.