SQL Basics Cheat Sheet

SQL

SQL, or *Structured Query Language*, is a language to talk to databases. It allows you to select specific data and to build complex reports. Today, SQL is a universal language of data. It is used in practically all technologies that process data.

SAMPLE DATA

COUNTRY

id	name	population	area
1	France	66600000	640680
2	Germany	80700000	357000
...	...	...	...

CITY

id	name	country_id	population	rating
1	Paris	1	2243000	5
2	Berlin	2	3460000	3
...	...	...	...	...

QUERYING SINGLE TABLE

Fetch all columns from the `country` table:

```
SELECT *
FROM country;
```

Fetch `id` and `name` columns from the `city` table:

```
SELECT id, name
FROM city;
```

Fetch city names sorted by the `rating` column in the default ASCending order:

```
SELECT name
FROM city
ORDER BY rating [ASC];
```

Fetch city names sorted by the `rating` column in the DESCending order:

```
SELECT name
FROM city
ORDER BY rating DESC;
```

ALIASES

COLUMNS

```
SELECT name AS city_name
FROM city;
```

TABLES

```
SELECT co.name, ci.name
FROM city AS ci
JOIN country AS co
  ON ci.country_id = co.id;
```

FILTERING THE OUTPUT

COMPARISON OPERATORS

Fetch names of cities that have a rating above 3:

```
SELECT name
FROM city
WHERE rating > 3;
```

Fetch names of cities that are neither Berlin nor Madrid:

```
SELECT name
FROM city
WHERE name != 'Berlin'
  AND name != 'Madrid';
```

TEXT OPERATORS

Fetch names of cities that start with a 'P' or end with an 's':

```
SELECT name
FROM city
WHERE name LIKE 'P%'
   OR name LIKE '%s';
```

Fetch names of cities that start with any letter followed by 'ublin' (like Dublin in Ireland or Lublin in Poland):

```
SELECT name
FROM city
WHERE name LIKE '_ublin';
```

OTHER OPERATORS

Fetch names of cities that have a population between 500K and 5M:

```
SELECT name
FROM city
WHERE population BETWEEN 500000 AND
5000000;
```

Fetch names of cities that don't miss a rating value:

```
SELECT name
FROM city
WHERE rating IS NOT NULL;
```

Fetch names of cities that are in countries with IDs 1, 4, 7, or 8:

```
SELECT name
FROM city
WHERE country_id IN (1, 4, 7, 8);
```

QUERYING MULTIPLE TABLES

INNER JOIN

`JOIN` (or explicitly **`INNER JOIN`**) returns rows that have matching values in both tables.

```
SELECT city.name, country.name
FROM city
[INNER] JOIN country
  ON city.country_id = country.id;
```

CITY			COUNTRY	
id	name	country_id	id	name
1	Paris	1	1	France
2	Berlin	2	2	Germany
3	Warsaw	4	3	Iceland

FULL JOIN

`FULL JOIN` (or explicitly **`FULL OUTER JOIN`**) returns all rows from both tables – if there's no matching row in the second table, **`NULL`**s are returned.

```
SELECT city.name, country.name
FROM city
FULL [OUTER] JOIN country
  ON city.country_id = country.id;
```

CITY			COUNTRY	
id	name	country_id	id	name
1	Paris	1	1	France
2	Berlin	2	2	Germany
3	Warsaw	4	NULL	NULL
NULL	NULL	NULL	3	Iceland

LEFT JOIN

`LEFT JOIN` returns all rows from the left table with corresponding rows from the right table. If there's no matching row, **`NULL`**s are returned as values from the second table.

```
SELECT city.name, country.name
FROM city
LEFT JOIN country
  ON city.country_id = country.id;
```

CITY			COUNTRY	
id	name	country_id	id	name
1	Paris	1	1	France
2	Berlin	2	2	Germany
3	Warsaw	4	NULL	NULL

CROSS JOIN

`CROSS JOIN` returns all possible combinations of rows from both tables. There are two syntaxes available.

```
SELECT city.name, country.name
FROM city
CROSS JOIN country;

SELECT city.name, country.name
FROM city, country;
```

CITY			COUNTRY	
id	name	country_id	id	name
1	Paris	1	1	France
1	Paris	1	2	Germany
2	Berlin	2	1	France
2	Berlin	2	2	Germany

RIGHT JOIN

`RIGHT JOIN` returns all rows from the right table with corresponding rows from the left table. If there's no matching row, **`NULL`**s are returned as values from the left table.

```
SELECT city.name, country.name
FROM city
RIGHT JOIN country
  ON city.country_id = country.id;
```

CITY			COUNTRY	
id	name	country_id	id	name
1	Paris	1	1	France
2	Berlin	2	2	Germany
NULL	NULL	NULL	3	Iceland

NATURAL JOIN

`NATURAL JOIN` will join tables by all columns with the same name.

```
SELECT city.name, country.name
FROM city
NATURAL JOIN country;
```

CITY			COUNTRY	
country_id	id	name	name	id
6	6	San Marino	San Marino	6
7	7	Vatican City	Vatican City	7
5	9	Greece	Greece	9
10	11	Monaco	Monaco	10

`NATURAL JOIN` used these columns to match rows: **`city.id, city.name, country.id, country.name`**
`NATURAL JOIN` is very rarely used in practice.

AGGREGATION AND GROUPING

`GROUP BY` **groups** together rows that have the same values in specified columns.
It computes summaries (aggregates) for each unique combination of values.

CITY		
id	name	country_id
1	Paris	1
101	Marseille	1
102	Lyon	1
2	Berlin	2
103	Hamburg	2
104	Munich	2
3	Warsaw	4
105	Cracow	4

→

CITY	
country_id	count
1	3
2	3
4	2

AGGREGATE FUNCTIONS

- **`avg(`**`expr`**`)`** – average value for rows within the group
- **`count(`**`expr`**`)`** – count of values for rows within the group
- **`max(`**`expr`**`)`** – maximum value within the group
- **`min(`**`expr`**`)`** – minimum value within the group
- **`sum(`**`expr`**`)`** – sum of values within the group

EXAMPLE QUERIES

Find out the number of cities:

```
SELECT COUNT(*)
FROM city;
```

Find out the number of cities with non-null ratings:

```
SELECT COUNT(rating)
FROM city;
```

Find out the number of distinctive country values:

```
SELECT COUNT(DISTINCT country_id)
FROM city;
```

Find out the smallest and the greatest country populations:

```
SELECT MIN(population), MAX(population)
FROM country;
```

Find out the total population of cities in respective countries:

```
SELECT country_id, SUM(population)
FROM city
GROUP BY country_id;
```

Find out the average rating for cities in respective countries if the average is above 3.0:

```
SELECT country_id, AVG(rating)
FROM city
GROUP BY country_id
HAVING AVG(rating) > 3.0;
```

SUBQUERIES

A subquery is a query that is nested inside another query, or inside another subquery. There are different types of subqueries.

SINGLE VALUE

The simplest subquery returns exactly one column and exactly one row. It can be used with comparison operators =, <, <=, >, or >=.

This query finds cities with the same rating as Paris:

```
SELECT name FROM city
WHERE rating = (
    SELECT rating
    FROM city
    WHERE name = 'Paris'
);
```

MULTIPLE VALUES

A subquery can also return multiple columns or multiple rows. Such subqueries can be used with operators `IN`, `EXISTS`, `ALL`, or `ANY`.

This query finds cities in countries that have a population above 20M:

```
SELECT name
FROM city
WHERE country_id IN (
    SELECT country_id
    FROM country
    WHERE population > 20000000
);
```

CORRELATED

A correlated subquery refers to the tables introduced in the outer query. A correlated subquery depends on the outer query. It cannot be run independently from the outer query.

This query finds cities with a population greater than the average population in the country:

```
SELECT *
FROM city main_city
WHERE population > (
    SELECT AVG(population)
    FROM city average_city
    WHERE average_city.country_id = main_city.country_id
);
```

This query finds countries that have at least one city:

```
SELECT name
FROM country
WHERE EXISTS (
    SELECT *
    FROM city
    WHERE country_id = country.id
);
```

SET OPERATIONS

Set operations are used to combine the results of two or more queries into a single result. The combined queries must return the same number of columns and compatible data types. The names of the corresponding columns can be different.

CYCLING		
id	name	country
1	YK	DE
2	ZG	DE
3	WT	PL
...	...	...

SKATING		
id	name	country
1	YK	DE
2	DF	DE
3	AK	PL
...	...	...

UNION

`UNION` combines the results of two result sets and removes duplicates.
`UNION ALL` doesn't remove duplicate rows.

This query displays German cyclists together with German skaters:

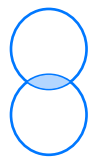
```
SELECT name
FROM cycling
WHERE country = 'DE'
UNION / UNION ALL
SELECT name
FROM skating
WHERE country = 'DE';
```

INTERSECT

`INTERSECT` returns only rows that appear in both result sets.

This query displays German cyclists who are also German skaters at the same time:

```
SELECT name
FROM cycling
WHERE country = 'DE'
INTERSECT
SELECT name
FROM skating
WHERE country = 'DE';
```



EXCEPT

`EXCEPT` returns only the rows that appear in the first result set but do not appear in the second result set.

This query displays German cyclists unless they are also German skaters at the same time:

```
SELECT name
FROM cycling
WHERE country = 'DE'
EXCEPT / MINUS
SELECT name
FROM skating
WHERE country = 'DE';
```

SQL CHEAT SHEET

QUERYING DATA FROM A TABLE

```
SELECT c1, c2 FROM t;
```
Query data in columns c1, c2 from a table

```
SELECT * FROM t;
```
Query all rows and columns from a table

```
SELECT c1, c2 FROM t
WHERE condition;
```
Query data and filter rows with a condition

```
SELECT DISTINCT c1 FROM t
WHERE condition;
```
Query distinct rows from a table

```
SELECT c1, c2 FROM t
ORDER BY c1 ASC [DESC];
```
Sort the result set in ascending or descending order

```
SELECT c1, c2 FROM t
ORDER BY c1
LIMIT n OFFSET offset;
```
Skip *offset* of rows and return the next n rows

```
SELECT c1, aggregate(c2)
FROM t
GROUP BY c1;
```
Group rows using an aggregate function

```
SELECT c1, aggregate(c2)
FROM t
GROUP BY c1
HAVING condition;
```
Filter groups using HAVING clause

QUERYING FROM MULTIPLE TABLES

```
SELECT c1, c2
FROM t1
INNER JOIN t2 ON condition;
```
Inner join t1 and t2

```
SELECT c1, c2
FROM t1
LEFT JOIN t2 ON condition;
```
Left join t1 and t1

```
SELECT c1, c2
FROM t1
RIGHT JOIN t2 ON condition;
```
Right join t1 and t2

```
SELECT c1, c2
FROM t1
FULL OUTER JOIN t2 ON condition;
```
Perform full outer join

```
SELECT c1, c2
FROM t1
CROSS JOIN t2;
```
Produce a Cartesian product of rows in tables

```
SELECT c1, c2
FROM t1, t2;
```
Another way to perform cross join

```
SELECT c1, c2
FROM t1 A
INNER JOIN t2 B ON condition;
```
Join t1 to itself using INNER JOIN clause

USING SQL OPERATORS

```
SELECT c1, c2 FROM t1
UNION [ALL]
SELECT c1, c2 FROM t2;
```
Combine rows from two queries

```
SELECT c1, c2 FROM t1
INTERSECT
SELECT c1, c2 FROM t2;
```
Return the intersection of two queries

```
SELECT c1, c2 FROM t1
MINUS
SELECT c1, c2 FROM t2;
```
Subtract a result set from another result set

```
SELECT c1, c2 FROM t1
WHERE c1 [NOT] LIKE pattern;
```
Query rows using pattern matching %, _

```
SELECT c1, c2 FROM t
WHERE c1 [NOT] IN value_list;
```
Query rows in a list

```
SELECT c1, c2 FROM t
WHERE  c1 BETWEEN low AND high;
```
Query rows between two values

```
SELECT c1, c2 FROM t
WHERE  c1 IS [NOT] NULL;
```
Check if values in a table is NULL or not

SQL CHEAT SHEET

MANAGING TABLES

```
CREATE TABLE t (
    id INT PRIMARY KEY,
    name VARCHAR NOT NULL,
    price INT DEFAULT 0
);
```
Create a new table with three columns

```
DROP TABLE t ;
```
Delete the table from the database

```
ALTER TABLE t ADD column;
```
Add a new column to the table

```
ALTER TABLE t DROP COLUMN c ;
```
Drop column c from the table

```
ALTER TABLE t ADD constraint;
```
Add a constraint

```
ALTER TABLE t DROP constraint;
```
Drop a constraint

```
ALTER TABLE t1 RENAME TO t2;
```
Rename a table from t1 to t2

```
ALTER TABLE t1 RENAME c1 TO c2 ;
```
Rename column c1 to c2

```
TRUNCATE TABLE t;
```
Remove all data in a table

USING SQL CONSTRAINTS

```
CREATE TABLE t(
   c1 INT, c2 INT, c3 VARCHAR,
   PRIMARY KEY (c1,c2)
);
```
Set c1 and c2 as a primary key

```
CREATE TABLE t1(
   c1 INT PRIMARY KEY,
   c2 INT,
   FOREIGN KEY (c2) REFERENCES t2(c2)
);
```
Set c2 column as a foreign key

```
CREATE TABLE t(
   c1 INT, c1 INT,
   UNIQUE(c2,c3)
);
```
Make the values in c1 and c2 unique

```
CREATE TABLE t(
  c1 INT, c2 INT,
  CHECK(c1> 0 AND c1 >= c2)
);
```
Ensure c1 > 0 and values in c1 >= c2

```
CREATE TABLE t(
    c1 INT PRIMARY KEY,
    c2 VARCHAR NOT NULL
);
```
Set values in c2 column not NULL

MODIFYING DATA

```
INSERT INTO t(column_list)
VALUES(value_list);
```
Insert one row into a table

```
INSERT INTO t(column_list)
VALUES (value_list),
       (value_list), ....;
```
Insert multiple rows into a table

```
INSERT INTO t1(column_list)
SELECT column_list
FROM t2;
```
Insert rows from t2 into t1

```
UPDATE t
SET c1 = new_value;
```
Update new value in the column c1 for all rows

```
UPDATE t
SET c1 = new_value,
    c2 = new_value
WHERE condition;
```
Update values in the column c1, c2 that match the condition

```
DELETE FROM t;
```
Delete all data in a table

```
DELETE FROM t
WHERE condition;
```
Delete subset of rows in a table

SQL CHEAT SHEET

MANAGING VIEWS

```
CREATE VIEW v(c1,c2)
AS
SELECT c1, c2
FROM t;
```
Create a new view that consists of c1 and c2

```
CREATE VIEW v(c1,c2)
AS
SELECT c1, c2
FROM t;
WITH [CASCADED | LOCAL] CHECK OPTION;
```
Create a new view with check option

```
CREATE RECURSIVE VIEW v
AS
select-statement -- anchor part
UNION [ALL]
select-statement; -- recursive part
```
Create a recursive view

```
CREATE TEMPORARY VIEW v
AS
SELECT c1, c2
FROM t;
```
Create a temporary view

```
DROP VIEW view_name;
```
Delete a view

MANAGING INDEXES

```
CREATE INDEX idx_name
ON t(c1,c2);
```
Create an index on c1 and c2 of the table t

```
CREATE UNIQUE INDEX idx_name
ON t(c3,c4);
```
Create a unique index on c3, c4 of the table t

```
DROP INDEX idx_name;
```
Drop an index

SQL AGGREGATE FUNCTIONS

AVG returns the average of a list

COUNT returns the number of elements of a list

SUM returns the total of a list

MAX returns the maximum value in a list

MIN returns the minimum value in a list

MANAGING TRIGGERS

```
CREATE OR MODIFY TRIGGER trigger_name
WHEN EVENT
ON table_name TRIGGER_TYPE
EXECUTE stored_procedure;
```
Create or modify a trigger

WHEN
- **BEFORE** – invoke before the event occurs
- **AFTER** – invoke after the event occurs

EVENT
- **INSERT** – invoke for INSERT
- **UPDATE** – invoke for UPDATE
- **DELETE** – invoke for DELETE

TRIGGER_TYPE
- **FOR EACH ROW**
- **FOR EACH STATEMENT**

```
CREATE TRIGGER before_insert_person
BEFORE INSERT
ON person FOR EACH ROW
EXECUTE stored_procedure;
```
Create a trigger invoked before a new row is inserted into the person table

```
DROP TRIGGER trigger_name;
```
Delete a specific trigger

Standard SQL Functions Cheat Sheet

TEXT FUNCTIONS

CONCATENATION

Use the `||` operator to concatenate two strings:

```
SELECT 'Hi ' || 'there!';
-- result: Hi there!
```

Remember that you can concatenate only character strings using `||`. Use this trick for numbers:

```
SELECT '' || 4 || 2;
-- result: 42
```

Some databases implement non-standard solutions for concatenating strings like `CONCAT()` or `CONCAT_WS()`. Check the documentation for your specific database.

LIKE OPERATOR – PATTERN MATCHING

Use the `_` character to replace any single character. Use the `%` character to replace any number of characters (including 0 characters).

Fetch all names that start with any letter followed by `'atherine'`:

```
SELECT name
FROM names
WHERE name LIKE '_atherine';
```

Fetch all names that end with `'a'`:

```
SELECT name
FROM names
WHERE name LIKE '%a';
```

USEFUL FUNCTIONS

Get the count of characters in a string:

```
SELECT LENGTH('LearnSQL.com');
-- result: 12
```

Convert all letters to lowercase:

```
SELECT LOWER('LEARNSQL.COM');
-- result: learnsql.com
```

Convert all letters to uppercase:

```
SELECT UPPER('LearnSQL.com');
-- result: LEARNSQL.COM
```

Convert all letters to lowercase and all first letters to uppercase (not implemented in MySQL and SQL Server):

```
SELECT INITCAP('edgar frank ted cODD');
-- result: Edgar Frank Ted Codd
```

Get just a part of a string:

```
SELECT SUBSTRING('LearnSQL.com', 9);
-- result: .com
SELECT SUBSTRING('LearnSQL.com', 0, 6);
-- result: Learn
```

Replace part of a string:

```
SELECT REPLACE('LearnSQL.com', 'SQL',
'Python');
-- result: LearnPython.com
```

NUMERIC FUNCTIONS

BASIC OPERATIONS

Use `+`, `-`, `*`, `/` to do some basic math. To get the number of seconds in a week:

```
SELECT 60 * 60 * 24 * 7; -- result: 604800
```

CASTING

From time to time, you need to change the type of a number. The `CAST()` function is there to help you out. It lets you change the type of value to almost anything (`integer`, `numeric`, `double precision`, `varchar`, and many more).

Get the number as an integer (without rounding):

```
SELECT CAST(1234.567 AS integer);
-- result: 1234
```

Change a column type to `double precision`

```
SELECT CAST(column AS double precision);
```

USEFUL FUNCTIONS

Get the remainder of a division:

```
SELECT MOD(13, 2);
-- result: 1
```

Round a number to its nearest integer:

```
SELECT ROUND(1234.56789);
-- result: 1235
```

Round a number to three decimal places:

```
SELECT ROUND(1234.56789, 3);
-- result: 1234.568
```

PostgreSQL requires the first argument to be of the type `numeric` – cast the number when needed.

To round the number **up**:

```
SELECT CEIL(13.1); -- result: 14
SELECT CEIL(-13.9); -- result: -13
```

The `CEIL(x)` function returns the **smallest** integer **not less** than `x`. In SQL Server, the function is called `CEILING()`.

To round the number **down**:

```
SELECT FLOOR(13.8); -- result: 13
SELECT FLOOR(-13.2); -- result: -14
```

The `FLOOR(x)` function returns the **greatest** integer **not greater** than `x`.

To round towards 0 irrespective of the sign of a number:

```
SELECT TRUNC(13.5); -- result: 13
SELECT TRUNC(-13.5); -- result: -13
```

`TRUNC(x)` works the same way as `CAST(x AS integer)`. In MySQL, the function is called `TRUNCATE()`.

To get the absolute value of a number:

```
SELECT ABS(-12); -- result: 12
```

To get the square root of a number:

```
SELECT SQRT(9); -- result: 3
```

NULLs

To retrieve all rows with a missing value in the `price` column:

```
WHERE price IS NULL
```

To retrieve all rows with the `weight` column populated:

```
WHERE weight IS NOT NULL
```

Why shouldn't you use `price = NULL` or `weight != NULL`? Because databases don't know if those expressions are true or false – they are evaluated as `NULL`s.

Moreover, if you use a function or concatenation on a column that is `NULL` in some rows, then it will get propagated. Take a look:

domain	LENGTH(domain)
LearnSQL.com	12
LearnPython.com	15
NULL	*NULL*
vertabelo.com	13

USEFUL FUNCTIONS

COALESCE(x, y, ...)

To replace `NULL` in a query with something meaningful:

```
SELECT
  domain,
  COALESCE(domain, 'domain missing')
FROM contacts;
```

domain	coalesce
LearnSQL.com	LearnSQL.com
NULL	domain missing

The `COALESCE()` function takes any number of arguments and returns the value of the first argument that isn't `NULL`.

NULLIF(x, y)

To save yourself from *division by 0* errors:

```
SELECT
  last_month,
  this_month,
  this_month * 100.0
    / NULLIF(last_month, 0)
    AS better_by_percent
FROM video_views;
```

last_month	this_month	better_by_percent
723786	1085679	150.0
0	178123	*NULL*

The `NULLIF(x, y)` function will return `NULL` if `x` is the same as `y`, else it will return the `x` value.

CASE WHEN

The basic version of `CASE WHEN` checks if the values are equal (e.g., if `fee` is equal to 50, then `'normal'` is returned). If there isn't a matching value in the `CASE WHEN`, then the `ELSE` value will be returned (e.g., if `fee` is equal to 49, then `'not available'` will show up.

```
SELECT
  CASE fee
    WHEN 50 THEN 'normal'
    WHEN 10 THEN 'reduced'
    WHEN  0 THEN 'free'
    ELSE 'not available'
  END AS tariff
FROM ticket_types;
```

The most popular type is the **searched CASE WHEN** – it lets you pass conditions (as you'd write them in the `WHERE` clause), evaluates them in order, then returns the value for the first condition met.

```
SELECT
  CASE
    WHEN score >= 90 THEN 'A'
    WHEN score >  60 THEN 'B'
    ELSE 'F'
  END AS grade
FROM test_results;
```

Here, all students who scored at least 90 will get an A, those with the score above 60 (and below 90) will get a B, and the rest will receive an F.

TROUBLESHOOTING

Integer division

When you don't see the decimal places you expect, it means that you are dividing between two integers. Cast one to `decimal`:

```
CAST(123 AS decimal) / 2
```

Division by 0

To avoid this error, make sure that the denominator is not equal to 0. You can use the `NULLIF()` function to replace 0 with a `NULL`, which will result in a `NULL` for the whole expression:

```
count / NULLIF(count_all, 0)
```

Inexact calculations

If you do calculations using `real` (floating point) numbers, you'll end up with some inaccuracies. This is because this type is meant for scientific calculations such as calculating the velocity. Whenever you need accuracy (such as dealing with monetary values), use the `decimal` / `numeric` type (or `money` if available).

Errors when rounding with a specified precision

Most databases won't complain, but do check the documentation if they do. For example, if you want to specify the rounding precision in PostgreSQL, the value must be of the `numeric` type.

AGGREGATION AND GROUPING

- **`COUNT(expr)`** – the count of values for the rows within the group
- **`SUM(expr)`** – the sum of values within the group
- **`AVG(expr)`** – the average value for the rows within the group
- **`MIN(expr)`** – the minimum value within the group
- **`MAX(expr)`** – the maximum value within the group

To get the number of rows in the table:

```
SELECT COUNT(*)
FROM city;
```

To get the number of non-NULL values in a column:

```
SELECT COUNT(rating)
FROM city;
```

To get the count of unique values in a column:

```
SELECT COUNT(DISTINCT country_id)
FROM city;
```

GROUP BY

CITY	
name	**country_id**
Paris	1
Marseille	1
Lyon	1
Berlin	2
Hamburg	2
Munich	2
Warsaw	4
Cracow	4

→

CITY	
country_id	**count**
1	3
2	3
4	2

The example above – the count of cities in each country:

```
SELECT name, COUNT(country_id)
FROM city
GROUP BY name;
```

The average rating for the city:

```
SELECT city_id, AVG(rating)
FROM ratings
GROUP BY city_id;
```

Common mistake: `COUNT(*)` and `LEFT JOIN`

When you join the tables like this: `client LEFT JOIN project`, and you want to get the number of projects for every client you know, `COUNT(*)` will return **1** for each client even if you've never worked for them. This is because, they're still present in the list but with the `NULL` in the fields related to the project after the `JOIN`. To get the correct count (**0** for the clients you've never worked for), count the values in a column of the other table, e.g., `COUNT(project_name)`. Check out this exercise to see an example.

DATE AND TIME

There are 3 main time-related types: `date`, `time`, and `timestamp`. Time is expressed using a 24-hour clock, and it can be as vague as just hour and minutes (e.g., `15:30` – 3:30 p.m.) or as precise as microseconds and time zone (as shown below):

```
2021-12-31 14:39:53.662522-05
  date     time
  timestamp
YYYY-mm-dd HH:MM:SS.ssssss±TZ
```

`14:39:53.662522-05` is almost 2:40 p.m. CDT (e.g., in Chicago; in UTC it'd be 7:40 p.m.). The letters in the above example represent:

In the date part:

- `YYYY` – the 4-digit year.
- `mm` – the zero-padded month (`01`—January through `12`—December).
- `dd` – the zero-padded day.

In the time part:

- `HH` – the zero-padded hour in a 24-hour clock.
- `MM` – the minutes.
- `SS` – the seconds. *Omissible.*
- `ssssss` – the smaller parts of a second – they can be expressed using 1 to 6 digits. *Omissible.*
- `±TZ` – the timezone. It must start with either `+` or `-`, and use two digits relative to UTC. *Omissible.*

What time is it?

To answer that question in SQL, you can use:

- `CURRENT_TIME` – to find what time it is.
- `CURRENT_DATE` – to get today's date. (`GETDATE()` in SQL Server.)
- `CURRENT_TIMESTAMP` – to get the timestamp with the two above.

Creating values

To create a `date`, `time`, or `timestamp`, simply write the value as a string and cast it to the proper type.

```
SELECT CAST('2021-12-31' AS date);
SELECT CAST('15:31' AS time);
SELECT CAST('2021-12-31 23:59:29+02' AS
timestamp);
SELECT CAST('15:31.124769' AS time);
```

Be careful with the last example – it will be interpreted as 15 minutes 31 seconds and 124769 microseconds! It is always a good idea to write `00` explicitly for hours: `'00:15:31.124769'`.

You might skip casting in simple conditions – the database will know what you mean.

```
SELECT airline, flight_number,
departure_time
FROM airport_schedule
WHERE departure_time < '12:00';
```

INTERVALs

Note: In SQL Server, intervals aren't implemented – use the `DATEADD()` and `DATEDIFF()` functions.

To get the simplest interval, subtract one time value from another:

```
SELECT CAST('2021-12-31 23:59:59' AS
timestamp) - CAST('2021-06-01 12:00:00' AS
timestamp);
-- result: 213 days 11:59:59
```

To define an interval: `INTERVAL '1' DAY`

This syntax consists of three elements: the `INTERVAL` keyword, a quoted value, and a time part keyword (in singular form.) You can use the following time parts: `YEAR`, `MONTH`, `WEEK`, `DAY`, `HOUR`, `MINUTE`, and `SECOND`. In MySQL, omit the quotes. You can join many different `INTERVAL`s using the `+` or `-` operator:

```
INTERVAL '1' YEAR + INTERVAL '3' MONTH
```

In some databases, there's an easier way to get the above value. And it accepts plural forms! `INTERVAL '1 year 3 months'`

There are two more syntaxes in the Standard SQL:

Syntax	What it does
`INTERVAL 'x-y' YEAR TO MONTH`	`INTERVAL 'x year y month'`
`INTERVAL 'x-y' DAY TO SECOND`	`INTERVAL 'x day y second'`

In MySQL, write `year_month` instead of `YEAR TO MONTH` and `day_second` instead of `DAY TO SECOND`.

To get the last day of a month, add one month and subtract one day:

```
SELECT CAST('2021-02-01' AS date)
       + INTERVAL '1' MONTH
       - INTERVAL '1' DAY;
```

To get all events for next three months from today:

```
SELECT event_date, event_name
FROM calendar
WHERE event_date BETWEEN CURRENT_DATE AND
CURRENT_DATE + INTERVAL '3' MONTH;
```

To get part of the date:

```
SELECT EXTRACT(YEAR FROM birthday)
FROM artists;
```

One of possible returned values: 1946. In SQL Server, use the `DATEPART(part, date)` function.

TIME ZONES

In the SQL Standard, the `date` type can't have an associated time zone, but the `time` and `timestamp` types can. In the real world, time zones have little meaning without the date, as the offset can vary through the year because of **daylight saving time**. So, it's best to work with the `timestamp` values.

When working with the type `timestamp with time zone` (abbr. `timestamptz`), you can type in the value in your local time zone, and it'll get converted to the UTC time zone as it is inserted into the table. Later when you select from the table it gets converted back to your local time zone. This is immune to time zone changes.

AT TIME ZONE

To operate between different time zones, use the `AT TIME ZONE` keyword.

If you use this format: `{timestamp without time zone} AT TIME ZONE {time zone}`, then the database will read the time stamp in the specified time zone and convert it to the time zone local to the display. It returns the time in the format `timestamp with time zone`.

If you use this format: `{timestamp with time zone} AT TIME ZONE {time zone}`, then the database will convert the time in one time zone to the target time zone specified by `AT TIME ZONE`. It returns the time in the format `timestamp without time zone`, in the target time zone.

You can define the time zone with popular shortcuts like `UTC`, `MST`, or `GMT`, or by `continent/city` such as: `America/New_York`, `Europe/London`, and `Asia/Tokyo`.

Examples

We set the local time zone to `'America/New_York'`.

```
SELECT TIMESTAMP '2021-07-16 21:00:00' AT
TIME ZONE 'America/Los_Angeles';
-- result: 2021-07-17 00:00:00-04
```

Here, the database takes a timestamp without a time zone and it's told it's in Los Angeles time, which is then converted to the local time – New York for displaying. This answers the question **"At what time should I turn on the TV if the show starts at 9 PM in Los Angeles?"**

```
SELECT TIMESTAMP WITH TIME ZONE '2021-06-
20 19:30:00' AT TIME ZONE
'Australia/Sydney';
-- result: 2021-06-21 09:30:00
```

Here, the database gets a timestamp specified in the local time zone and converts it to the time in Sydney (note that it didn't return a time zone.) This answers the question **"What time is it in Sydney if it's 7:30 PM here?"**