Standard SQL Functions Cheat Sheet

LearnSQL.com

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?"**