



# SQL Style Guide

by [Simon Holywell](#) · [@Treffynnon](#)

[Tweet](#)

- [Overview](#)
- [General](#)
  - [Do](#)
  - [Avoid](#)
- [Naming conventions](#)
  - [General](#)
  - [Tables](#)
  - [Columns](#)
  - [Aliasing or correlations](#)
  - [Stored procedures](#)
  - [Uniform suffixes](#)
- [Query syntax](#)
  - [Reserved words](#)
  - [White space](#)
  - [Indentation](#)
  - [Preferred formalisms](#)
- [Create syntax](#)
  - [Choosing data types](#)
  - [Specifying default values](#)
  - [Constraints and keys](#)
  - [Designs to avoid](#)
- [Appendix](#)
  - [Reserved keyword reference](#)
  - [Column data types](#)

## Overview

---

You can use this set of guidelines, [fork them](#) or make your own - the key here is that you pick a style and stick to it. To suggest changes or fix bugs please open an [issue](#) or [pull request](#) on GitHub.

These guidelines are designed to be compatible with Joe Celko's [SQL Programming Style](#) book to make adoption for teams who have already read that book easier. This guide is a little more opinionated in some areas and in others a little more relaxed. It is certainly more succinct where [Celko's book](#) contains anecdotes and reasoning behind each rule as thoughtful prose.

It is easy to include this guide in [Markdown format](#) as a part of a project's code base or reference it here for anyone on the project to freely read—much harder with a physical book.

SQL style guide by [Simon Holywell](#) is licensed under a [Creative Commons Attribution-ShareAlike 4.0 International License](#). Based on a work at

<http://www.sqlstyle.guide>.

## General

---

### ► Do

- Use consistent and descriptive identifiers and names.
- Make judicious use of white space and indentation to make code easier to read.
- Store ISO-8601 compliant time and date information (YYYY-MM-DD HH:MM:SS.SSSSS).
- Try to use only standard SQL functions instead of vendor specific functions for reasons of portability.
- Keep code succinct and devoid of redundant SQL—such as unnecessary quoting or parentheses or `WHERE` clauses that can otherwise be derived.
- Include comments in SQL code where necessary. Use the C style opening `/*` and closing `*/` where possible otherwise precede comments with `--` and finish them with a new line.

```
SELECT file_hash  -- stored ssdeep hash
FROM   file_system
WHERE  file_name = '.vimrc';
```

```
/* Updating the file record after writing to the file */
UPDATE file_system
  SET  file_modified_date = '1980-02-22 13:19:01.00000',
       file_size = 209732
WHERE  file_name = '.vimrc';
```

### ► Avoid

- CamelCase—it is difficult to scan quickly.
- Descriptive prefixes or Hungarian notation such as `sp_` or `tbl`.
- Plurals—use the more natural collective term where possible instead. For example `staff` instead of `employees` or `people` instead of `individuals`.
- Quoted identifiers—if you must use them then stick to SQL92 double quotes for portability (you may need to configure your SQL server to support this depending on vendor).
- Object oriented design principles should not be applied to SQL or database structures.

# Naming conventions

---

## ► General

- Ensure the name is unique and does not exist as a reserved keyword.
- Keep the length to a maximum of 30 bytes—in practice this is 30 characters unless you are using multi-byte character set.
- Names must begin with a letter and may not end with an underscore.
- Only use letters, numbers and underscores in names.
- Avoid the use of multiple consecutive underscores—these can be hard to read.
- Use underscores where you would naturally include a space in the name (first name becomes `first_name`).
- Avoid abbreviations and if you have to use them make sure they are commonly understood.

```
SELECT first_name  
FROM staff;
```

## ► Tables

- Use a collective name or, less ideally, a plural form. For example (in order of preference) `staff` and `employees`.
- Do not prefix with `tbl` or any other such descriptive prefix or Hungarian notation.
- Never give a table the same name as one of its columns and vice versa.
- Avoid, where possible, concatenating two table names together to create the name of a relationship table. Rather than `cars_mechanics` prefer `services`.

## ► Columns

- Always use the singular name.
- Where possible avoid simply using `id` as the primary identifier for the table.
- Do not add a column with the same name as its table and vice versa.
- Always use lowercase except where it may make sense not to such as proper nouns.

## ► Aliasing or correlations

- Should relate in some way to the object or expression they are aliasing.

- As a rule of thumb the correlation name should be the first letter of each word in the object's name.
- If there is already a correlation with the same name then append a number.
- Always include the `AS` keyword—makes it easier to read as it is explicit.
- For computed data (`SUM()` or `AVG()`) use the name you would give it were it a column defined in the schema.

```
SELECT first_name AS fn
FROM staff AS s1
JOIN students AS s2
ON s2.mentor_id = s1.staff_num;
```

```
SELECT SUM(s.monitor_tally) AS monitor_total
FROM staff AS s;
```

## ► Stored procedures

- The name must contain a verb.
- Do not prefix with `sp_` or any other such descriptive prefix or Hungarian notation.

## ► Uniform suffixes

The following suffixes have a universal meaning ensuring the columns can be read and understood easily from SQL code. Use the correct suffix where appropriate.

- `_id`—a unique identifier such as a column that is a primary key.
- `_status`—flag value or some other status of any type such as `publication_status`.
- `_total`—the total or sum of a collection of values.
- `_num`—denotes the field contains any kind of number.
- `_name`—signifies a name such as `first_name`.
- `_seq`—contains a contiguous sequence of values.
- `_date`—denotes a column that contains the date of something.
- `_tally`—a count.
- `_size`—the size of something such as a file size or clothing.
- `_addr`—an address for the record could be physical or intangible such as `ip_addr`.

## Query syntax

---

## ► Reserved words

Always use uppercase for the reserved keywords like `SELECT` and `WHERE`.

It is best to avoid the abbreviated keywords and use the full length ones where available (prefer `ABSOLUTE` to `ABS`).

Do not use database server specific keywords where an ANSI SQL keyword already exists performing the same function. This helps to make code more portable.

```
SELECT model_num
FROM phones AS p
WHERE p.release_date > '2014-09-30';
```

## ► White space

To make the code easier to read it is important that the correct complement of spacing is used. Do not crowd code or remove natural language spaces.

### Spaces

Spaces should be used to line up the code so that the root keywords all end on the same character boundary. This forms a river down the middle making it easy for the readers eye to scan over the code and separate the keywords from the implementation detail. Rivers are bad in typography, but helpful here.

```
(SELECT f.species_name,
      AVG(f.height) AS average_height, AVG(f.diameter) AS average_diameter
FROM flora AS f
WHERE f.species_name = 'Banksia'
      OR f.species_name = 'Sheoak'
      OR f.species_name = 'Wattle'
GROUP BY f.species_name, f.observation_date)

UNION ALL

(SELECT b.species_name,
      AVG(b.height) AS average_height, AVG(b.diameter) AS average_diameter
FROM botanic_garden_flora AS b
WHERE b.species_name = 'Banksia'
      OR b.species_name = 'Sheoak'
      OR b.species_name = 'Wattle'
GROUP BY b.species_name, b.observation_date);
```

Notice that `SELECT`, `FROM`, etc. are all right aligned while the actual column names and implementation specific details are left aligned.

Although not exhaustive always include spaces:

- before and after equals (=)
- after commas (,)
- surrounding apostrophes (') where not within parentheses or with a trailing comma or semicolon.

```
SELECT a.title, a.release_date, a.recording_date
FROM albums AS a
WHERE a.title = 'Charcoal Lane'
OR a.title = 'The New Danger';
```

## Line spacing

Always include newlines/vertical space:

- before AND or OR
- after semicolons to separate queries for easier reading
- after each keyword definition
- after a comma when separating multiple columns into logical groups
- to separate code into related sections, which helps to ease the readability of large chunks of code.

Keeping all the keywords aligned to the righthand side and the values left aligned creates a uniform gap down the middle of query. It makes it much easier to scan the query definition over quickly too.

```
INSERT INTO albums (title, release_date, recording_date)
VALUES ('Charcoal Lane', '1990-01-01 01:01:01.00000', '1990-01-01 01:01:01.00000',
       ('The New Danger', '2008-01-01 01:01:01.00000', '1990-01-01 01:01:01.00000'))
```

```
UPDATE albums
SET release_date = '1990-01-01 01:01:01.00000'
WHERE title = 'The New Danger';
```

```
SELECT a.title,
       a.release_date, a.recording_date, a.production_date -- grouped dates together
FROM albums AS a
WHERE a.title = 'Charcoal Lane'
OR a.title = 'The New Danger';
```

## ► Indentation

To ensure that SQL is readable it is important that standards of indentation are followed.

## Joins

Joins should be indented to the other side of the river and grouped with a new line where necessary.

```
SELECT r.last_name
FROM riders AS r
    INNER JOIN bikes AS b
    ON r.bike_vin_num = b.vin_num
    AND b.engine_tally > 2

    INNER JOIN crew AS c
    ON r.crew_chief_last_name = c.last_name
    AND c.chief = 'Y';
```

## Subqueries

Subqueries should also be aligned to the right side of the river and then laid out using the same style as any other query. Sometimes it will make sense to have the closing parenthesis on a new line at the same character position as its opening partner—this is especially true where you have nested subqueries.

```
SELECT r.last_name,
    (SELECT MAX(YEAR(championship_date))
     FROM champions AS c
     WHERE c.last_name = r.last_name
     AND c.confirmed = 'Y') AS last_championship_year
FROM riders AS r
WHERE r.last_name IN
    (SELECT c.last_name
     FROM champions AS c
     WHERE YEAR(championship_date) > '2008'
     AND c.confirmed = 'Y');
```

## ► Preferred formalisms

- Make use of `BETWEEN` where possible instead of combining multiple statements with `AND`.
- Similarly use `IN()` instead of multiple `OR` clauses.
- Where a value needs to be interpreted before leaving the database use the `CASE` expression. `CASE` statements can be nested to form more complex logical structures.

- Avoid the use of `UNION` clauses and temporary tables where possible. If the schema can be optimised to remove the reliance on these features then it most likely should be.

```
SELECT CASE postcode
  WHEN 'BN1' THEN 'Brighton'
  WHEN 'EH1' THEN 'Edinburgh'
  END AS city
FROM office_locations
WHERE country = 'United Kingdom'
  AND opening_time BETWEEN 8 AND 9
  AND postcode IN ('EH1', 'BN1', 'NN1', 'KW1');
```

## Create syntax

---

When declaring schema information it is also important to maintain human readable code. To facilitate this ensure the column definitions are ordered and grouped where it makes sense to do so.

Indent column definitions by four (4) spaces within the `CREATE` definition.

### ► Choosing data types

- Where possible do not use vendor specific data types—these are not portable and may not be available in older versions of the same vendor's software.
- Only use `REAL` or `FLOAT` types where it is strictly necessary for floating point mathematics otherwise prefer `NUMERIC` and `DECIMAL` at all times. Floating point rounding errors are a nuisance!

### ► Specifying default values

- The default value must be the same type as the column—if a column is declared a `DECIMAL` do not provide an `INTEGER` default value.
- Default values must follow the data type declaration and come before any `NOT NULL` statement.

### ► Constraints and keys

Constraints and their subset, keys, are a very important component of any database definition. They can quickly become very difficult to read and reason about though so it is important that a standard set of guidelines are followed.

## Choosing keys



Deciding the column(s) that will form the keys in the definition should be a carefully considered activity as it will effect performance and data integrity.

1. The key should be unique to some degree.
2. Consistency in terms of data type for the value across the schema and a lower likelihood of this changing in the future.
3. Can the value be validated against a standard format (such as one published by ISO)? Encouraging conformity to point 2.
4. Keeping the key as simple as possible whilst not being scared to use compound keys where necessary.

It is a reasoned and considered balancing act to be performed at the definition of a database. Should requirements evolve in the future it is possible to make changes to the definitions to keep them up to date.

## Defining constraints

Once the keys are decided it is possible to define them in the system using constraints along with field value validation.

### General

- Tables must have at least one key to be complete and useful.
- Constraints should be given a custom name excepting `UNIQUE`, `PRIMARY KEY` and `FOREIGN KEY` where the database vendor will generally supply sufficiently intelligible names automatically.

### Layout and order

- Specify the primary key first right after the `CREATE TABLE` statement.
- Constraints should be defined directly beneath the column they correspond to. Indent the constraint so that it aligns to the right of the column name.
- If it is a multi-column constraint then consider putting it as close to both column definitions as possible and where this is difficult as a last resort include them at the end of the `CREATE TABLE` definition.
- If it is a table level constraint that applies to the entire table then it should also appear at the end.
- Use alphabetical order where `ON DELETE` comes before `ON UPDATE`.
- If it make senses to do so align each aspect of the query on the same character position. For example all `NOT NULL` definitions could start at the same character position. This is not hard and fast, but it certainly makes the code much easier to scan and read.

## Validation

- Use `LIKE` and `SIMILAR TO` constraints to ensure the integrity of strings where the format is known.
- Where the ultimate range of a numerical value is known it must be written as a range `CHECK()` to prevent incorrect values entering the database or the silent truncation of data too large to fit the column definition. In the least it should check that the value is greater than zero in most cases.
- `CHECK()` constraints should be kept in separate clauses to ease debugging.

## Example

```
CREATE TABLE staff (  
    PRIMARY KEY (staff_num),  
    staff_num      INT(5)      NOT NULL,  
    first_name     VARCHAR(100) NOT NULL,  
    pens_in_drawer INT(2)      NOT NULL,  
    CONSTRAINT pens_in_drawer_range  
    CHECK(pens_in_drawer >= 1 AND pens_in_drawer < 100)  
);
```

## ► Designs to avoid

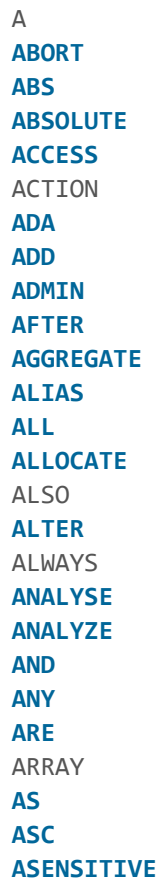
- Object oriented design principles do not effectively translate to relational database designs—avoid this pitfall.
- Placing the value in one column and the units in another column. The column should make the units self evident to prevent the requirement to combine columns again later in the application. Use `CHECK()` to ensure valid data is inserted into the column.
- EAV (Entity Attribute Value) tables—use a specialist product intended for handling such schema-less data instead.
- Splitting up data that should be in one table across many because of arbitrary concerns such as time-based archiving or location in a multi-national organisation. Later queries must then work across multiple tables with `UNION` rather than just simply querying one table.

## Appendix

---

## ► Reserved keyword reference

A list of ANSI SQL (92, 99 and 2003), MySQL 3 to 5.x, PostgreSQL 8.1, MS SQL Server 2000, MS ODBC and Oracle 10.2 reserved keywords.



A  
ABORT  
ABS  
ABSOLUTE  
ACCESS  
ACTION  
ADA  
ADD  
ADMIN  
AFTER  
AGGREGATE  
ALIAS  
ALL  
ALLOCATE  
ALSO  
ALTER  
ALWAYS  
ANALYSE  
ANALYZE  
AND  
ANY  
ARE  
ARRAY  
AS  
ASC  
ASENSITIVE

## ► Column data types

These are some suggested column data types to use for maximum compatibility between database engines.

### Character types:

- CHAR
- CLOB
- VARCHAR

### Numeric types

- Exact numeric types
  - BIGINT
  - DECIMAL
  - DECFLOAT
  - INTEGER
  - NUMERIC
  - SMALLINT
- Approximate numeric types

- DOUBLE PRECISION
- FLOAT
- REAL

## Datetime types

- DATE
- TIME
- TIMESTAMP

## Binary types:

- BINARY
- BLOB
- VARBINARY

## Additional types

- Boolean
- INTERVAL
- XML



Translations: [Deutsch](#) · [English](#) · [日本語](#) · [Português \(BR\)](#) · [Русский](#) · [简体中文](#) · [正體中文](#)

This guide is being discussed on Hacker News [ [1](#), [2](#) ], Reddit [ [1](#), [2](#), [3](#) ], [Lobste.rs](#), [Hatena](#) and of course [Twitter](#) if you want to have your say.

I have attempted to answer most of the frequently asked questions and held misconceptions in a blog post entitled "[SQL style guide misconceptions](#)" please have a read.

Please do [open issues](#) or [pull requests](#) for any errors you may find in the guide and help me to improve it.

SQL style guide by [Simon Holywell](#) is licensed under a [Creative CommonsAttribution-ShareAlike 4.0 International License](#).  
Based on a work at [www.sqlstyle.guide](http://www.sqlstyle.guide).

Star800

Fork362

Follow @treffynnon 887 followers Tweet