# A Comprehensive Guide to SQL Style and Best Practices: A Consensus-Driven Approach

## Introduction: The Philosophy of Readable and Maintainable SQL

Structured Query Language (SQL) is the lingua franca of data. While its syntax allows for immense flexibility, this freedom can lead to code that is difficult to read, debug, and maintain. A well-defined style guide addresses this challenge by establishing a set of conventions for writing clear, consistent, and portable SQL. The primary goal is not to impose rigid, arbitrary rules, but to foster a shared language and structure that enhances collaboration and reduces cognitive load for developers and analysts.

The single most important principle of any style guide is **consistency**. Consistency within a specific project is more critical than adherence to any external standard. When code is stylistically consistent, developers can focus on the logic of a query rather than deciphering its structure. This guide synthesizes the recommendations from top technology companies, influential data communities, and renowned SQL experts to provide a robust, consensus-driven starting point for teams looking to establish or refine their own standards.

The recommendations herein are optimized for human readability, recognizing that code is read far more often than it is written. The objective is clarity and long-term maintainability, not merely writing the fewest lines of code. Well-formatted SQL, like well-structured prose, is easier to parse, understand, and trust. This philosophy extends to advocating for standard ANSI SQL functions and syntax wherever possible, which ensures code is portable across different database systems and less prone to vendor-specific quirks. Similarly, the guide promotes explicitness—such as using INNER JOIN instead of the ambiguous JOIN or always including the AS keyword for aliasing—to remove ambiguity and make the author's intent clear.

The evolution of SQL style is no longer a purely manual or academic exercise. It is now intrinsically linked to the rise of automated linters and formatters, such as SQLFluff, and the features integrated into modern Integrated Development Environments (IDEs). This has a profound effect on how style guides are created and used. It transforms recommendations into enforceable rules, dramatically increasing adoption and consistency across a team or organization. Furthermore, this automation reframes long-standing debates. Arguments based on the manual inconvenience of a particular style—for instance, the difficulty of managing the final comma in a list—are weakened when a tool can automatically correct the syntax with a single command. This shifts the focus of the debate from what is easiest to type by hand to what is most inherently readable and logically sound. Consequently, an effective modern style guide must be designed with its potential implementation in a linter's configuration file in mind, featuring rules that are clear, unambiguous, and logically consistent to facilitate this automation.

## Section 1: Naming Conventions - The Foundation of Clarity

Clear and consistent naming is the most critical element of readable SQL. Ambiguous or inconsistent names for tables, columns, and other database objects are a primary source of confusion, bugs, and maintenance overhead. The following conventions represent a broad consensus for creating a clear and predictable data landscape.

### 1.1 General Identifier Rules: The Unifying Principles

These rules apply to all database identifiers, including tables, views, columns, and aliases.

* **Casing: The snake\_case Standard** The overwhelming consensus is to use lowercase snake\_case for all identifiers. This convention, where words are separated by underscores, enhances readability by creating clear visual separation between words. More importantly, it avoids common issues with case-sensitivity across different database systems. PostgreSQL, for example, automatically folds unquoted identifiers to lowercase, meaning a camelCase or PascalCase name like productCategory would be stored and retrieved as productcategory, defeating the purpose of the capitalization. For these reasons, camelCase and PascalCase are strongly discouraged.
* **Language: Full Words, Not Abbreviations** Identifiers should be descriptive and use full English words. Avoid abbreviations and acronyms unless the abbreviation is significantly more common than the full term (e.g., i18n for "internationalization"). While sid might seem like a convenient shorthand for student\_id to the original author, it creates a barrier to understanding for new team members or future maintainers.
* **Characters: Letters, Numbers, and Underscores Only** Names should begin with a letter and contain only alphanumeric characters and underscores. The use of multiple consecutive underscores (\_\_) should be avoided as they can be difficult to read and distinguish from a single underscore.
* **Quoting: Avoid If Possible** Identifiers should be named in a way that does not require them to be quoted. This means avoiding spaces, special characters, and reserved keywords in names. While most databases allow for such names if they are enclosed in double quotes (e.g., "All Employees"), this practice is cumbersome. It forces the use of quotes in every query that references the object, making the code harder to write and more prone to errors.
* **Reserved Keywords: Never Use for Names** Under no circumstances should a database object be named after a reserved SQL keyword. Words like user, order, group, table, and select have special meaning to the SQL parser. Using them as identifiers can lead to syntax errors or unexpected behavior, and often requires quoting, which should be avoided. Instead of user, prefer a more specific name like app\_user or website\_user. Instead of order, use customer\_order or product\_order.

### 1.2 The Great Debate: Singular vs. Plural Table Names

The debate over whether to use singular (e.g., customer) or plural (e.g., customers) table names is one of the oldest and most contentious in SQL styling. This is not merely a matter of preference; it reflects a fundamental disagreement on the conceptual model of a table. One camp views a table as a representation of an *entity type*—a template for a single customer. The other camp views it as a *collection* of those entities—a set of customers. This is a classic clash between the purism of relational theory and the intuitive pragmatism of application development.

* **The Case for Singular Names** Proponents of singular naming argue that a table defines the structure for a single instance of an entity. A row in the customer table represents one customer. This aligns well with object-oriented programming, where a class is named Customer, not Customers. This approach also simplifies naming conventions for primary and foreign keys (e.g., customer.id and address.customer\_id) and avoids the complexities of English pluralization (e.g., person becomes people, octopus becomes octopi).
* **The Case for Plural Names** Advocates for plural names argue that it is more intuitive, as a table is a container for multiple records. This makes SQL queries read more like natural language: SELECT name FROM customers feels more natural than SELECT name FROM customer. The most compelling pragmatic argument for plural names is that they are far less likely to conflict with reserved SQL keywords. Naming a table users or orders immediately avoids conflicts with the USER and ORDER keywords. Furthermore, many popular web development frameworks, such as Ruby on Rails, default to plural table names, so using them promotes consistency between the database and application layers.
* **The Consensus Recommendation** While both sides present valid arguments, the most critical rule is to **choose one convention and apply it consistently** throughout a project. For new projects, this guide recommends using **plural table names**. The tangible, practical benefit of avoiding reserved keyword conflicts provides a stronger advantage than the theoretical purity of singular names.

| Aspect | Argument for Singular (customer) | Argument for Plural (customers) | Recommendation |
| --- | --- | --- | --- |
| **Conceptual Model** | Represents a single entity type, aligning with relational theory and OOP class naming. | Represents a collection of records, which is intuitive. | Plural |
| **Readability** | customer.name can feel more logical when referring to a single record's attribute. | SELECT name FROM customers reads more like natural language. | Plural |
| **Reserved Words** | Prone to conflicts (e.g., user, order, constraint). | Less likely to conflict with reserved keywords. | Plural |
| **Consistency** | Avoids irregular pluralization rules (person -> people). | Many frameworks default to plural, promoting application-database consistency. | Plural |

### 1.3 Column Naming Conventions

* **Names:** Columns should always have singular, descriptive names. For example, a column in the customers table should be named name or first\_name, not customer\_name, as the table context makes the "customer" part redundant.
* **Ambiguous Names:** General names like id, name, type, or date must be prefixed to provide context and avoid ambiguity when joined with other tables. For example, use customer\_id, account\_name, order\_type, and created\_date.
* **Primary Keys (PKs):** The most common and recommended convention is to name the primary key column simply id. It is concise and unambiguous within the context of its table. When queried, it can be aliased for clarity (e.g., SELECT id AS customer\_id FROM customers).
* **Foreign Keys (FKs):** The standard convention for a foreign key column is {referenced\_table\_singular}\_id. For example, a product\_orders table that references the customers table would have a foreign key column named customer\_id.
* **Booleans:** Boolean columns should be prefixed with is\_, has\_, or does\_ to make their binary nature immediately obvious. Examples include is\_active, has\_sla, and does\_exist.
* **Temporal Fields:** To distinguish between different temporal data types, timestamps (date and time) should end with the suffix \_at (e.g., created\_at, updated\_at), and dates (date only) should end with \_date (e.g., order\_date, birth\_date). All timestamps should be stored in a consistent time zone, preferably UTC, to avoid ambiguity.

### 1.4 Naming Other Database Objects

* **Aliases:** Aliases must be descriptive and should relate to the object or expression they represent. Avoid meaningless single-letter aliases like a, b, and c, as they obscure the query's logic. For a table like product\_orders, an alias like po or orders is acceptable. For CTEs, the name should describe the logical step it performs, such as customers\_with\_recent\_orders. The AS keyword should always be used when creating an alias for explicitness and readability.
* **Constraints & Indexes:** All constraints (PRIMARY KEY, FOREIGN KEY, UNIQUE, CHECK) and indexes should be explicitly and descriptively named. This is crucial for debugging, as database error messages will reference these names. A standard naming pattern is {table\_name}\_{column\_names}\_{suffix}, where the suffix is one of \_pk, \_fk, \_uk (for unique), or \_ix (for index).
* **Join Tables (Many-to-Many):** For tables that resolve a many-to-many relationship, avoid simply concatenating the two table names (e.g., products\_users). Instead, use a name that describes the relationship itself, such as purchases, enrollments, or services.

### 1.5 Naming Anti-Patterns

* **Hungarian Notation:** Do not use type-descriptive prefixes like tbl\_ for tables, vw\_ for views, or sp\_ for stored procedures. The type of a database object is readily available from its context or from system metadata catalogs, making these prefixes redundant clutter.
* **Redundant Suffixes:** Similarly, avoid adding suffixes like \_table or \_column to object names.

## Section 2: Formatting and Layout - Structuring for Readability

Visual presentation directly impacts how easily code can be scanned, understood, and modified. A consistent layout reduces cognitive friction and allows developers to focus on the query's logic.

### 2.1 Keyword Casing: The UPPERCASE Convention

* **The Rule:** The overwhelming consensus is that all reserved SQL keywords (SELECT, FROM, WHERE, JOIN, etc.) and built-in functions (COUNT, SUM, DATE\_TRUNC, etc.) should be written in UPPERCASE. This practice creates a strong visual distinction between the structural components of the query and the business-specific logic contained within the lowercase table and column names.
* **The Counter-Argument:** It is worth acknowledging a vocal minority that advocates for lowercase keywords. The arguments for this position are that UPPERCASE can feel like "shouting," is considered harder to read in long blocks of text due to the uniform height of the letters (a concept related to "bouma shapes"), and is inconsistent with the conventions of most other modern programming languages. While these points have merit, this guide adheres to the UPPERCASE convention due to its deep-rooted history in the SQL community and the undeniable clarity it provides, especially in editors or environments that lack syntax highlighting.

### 2.2 Indentation and Whitespace

* **Spaces, Not Tabs:** Always use spaces for indentation. The width of a tab character can vary between different editors and environments, which can completely destroy carefully crafted alignment. A standard of 2 or 4 spaces per indentation level is recommended; this guide prefers 4 spaces for enhanced clarity.
* **Logical Clause Separation:** Each major clause of a query (SELECT, FROM, WHERE, GROUP BY, HAVING, ORDER BY) should begin on a new line. This cleanly delineates the distinct logical parts of the statement.
* **Use of Vertical Space:** Use blank lines to separate and group related sections of a complex query, such as between CTEs or before a complex JOIN block. This is analogous to using paragraphs in prose to group related ideas and improve readability.

### 2.3 The Other Great Debate: Leading vs. Trailing Commas

The placement of commas in a SELECT list is another topic of fervent debate, representing a classic tension between developer ergonomics and established community convention.

* **The Case for Trailing Commas (Recommended):** This is the dominant, de-facto standard across the industry. Code written with trailing commas will look familiar to the widest possible audience. This style also aligns with standard punctuation in natural languages. Most importantly, modern code formatters and IDEs are built with the expectation of trailing commas, and many SQL dialects now permit a "hanging" trailing comma on the last item in a list, which negates the primary ergonomic argument against them.  
  -- Good: Trailing commas  
  SELECT  
   customer\_id,  
   order\_date,  
   order\_total  
  FROM  
   orders;
* **The Case for Leading Commas (Contextual):** The argument for leading commas is almost entirely based on ergonomics for manual editing. It makes commenting out the last line in a list simpler, as it does not leave a syntactically invalid hanging comma on the preceding line. It also produces cleaner diffs in version control systems when adding or removing items from the end of a list, as the change is confined to a single line.  
  -- Not Recommended: Leading commas  
  SELECT  
   customer\_id  
   , order\_date  
   , order\_total  
  FROM  
   orders;
* **The Consensus Recommendation:** Use **trailing commas**. The argument for consistency with the vast majority of existing SQL code and tooling is paramount. The ergonomic benefits of leading commas have been significantly diminished by advances in modern SQL engines and automated formatters, which can manage trailing commas effortlessly.

### 2.4 Alignment and Line Structure

* **One Item Per Line:** In SELECT lists and other lists, each item should be on its own line and indented one level from the parent clause.
* **Line Length:** To avoid horizontal scrolling and improve readability, lines should be wrapped if they exceed a reasonable length. A common standard is 80 to 100 characters.
* **Alignment Styles:**
  + **Left-Align (Recommended):** All root keywords (SELECT, FROM, WHERE) begin at the same indentation level (the far left). The contents of each clause are then indented. This style is simple, easy to maintain both manually and with auto-formatters, and is the most common convention.
  + **"River" Align (Alternative):** In this style, keywords are right-aligned to create a vertical "river" of whitespace that separates the keywords from the expressions. While visually appealing to some, it is very difficult to maintain manually and is not well-supported by most automated formatters. This guide presents it as a classic but less practical alternative.
* **Parentheses:** For multi-line constructs enclosed in parentheses (such as a subquery or a long list in an IN clause), the opening parenthesis should terminate the line. The contents should be indented one level. The closing parenthesis should be on its own new line, aligned vertically with the first character of the line that contains the opening parenthesis.

## Section 3: Query Syntax - Writing Clean and Performant SQL

Beyond aesthetics, the logical structure of a query is crucial for clarity, performance, and maintainability.

### 3.1 The SELECT Clause

The use of SELECT \* in production code is a dangerous anti-pattern and should be strictly avoided. Always explicitly list the columns required by the query. There are three primary reasons for this rule:

1. **Performance:** Requesting all columns from a table can lead to significant unnecessary data transfer from storage and across the network, slowing down query execution.
2. **Stability:** Queries using SELECT \* are brittle. If columns are added to or removed from the source table, the query may break downstream processes or return unexpected data without warning.
3. **Clarity:** Explicitly listing columns makes the query self-documenting. It clearly states the data dependencies of the query, making it far easier to understand and debug.

The only acceptable use for SELECT \* is in ad-hoc, interactive exploratory analysis, and even then, it should always be paired with a LIMIT clause to prevent accidentally fetching a massive result set.

### 3.2 JOIN Operations

* **Be Explicit:** Always specify the full join type: INNER JOIN, LEFT JOIN, RIGHT JOIN, or FULL OUTER JOIN. Do not rely on the shorthand JOIN, which defaults to INNER JOIN. Being explicit removes any ambiguity about the query's intent.
* **Formatting:** The JOIN clause should be at the same indentation level as the FROM clause. The ON condition should start on a new, indented line. For multiple join conditions, each subsequent AND should also start on a new line, aligned under the ON clause.
* **Platform-Specific Considerations:** Be aware that some SQL dialects have quirks. For example, some sources indicate that Snowflake can produce inaccurate results when using the USING keyword in joins, making the explicit ON condition preferable.

### 3.3 Structuring Complex Logic with CTEs

Common Table Expressions (CTEs), defined using the WITH clause, are strongly preferred over nested subqueries for structuring complex logic.

The rationale is compelling:

* **Readability:** CTEs are read linearly from top to bottom, which is more natural than the "inside-out" reading required for deeply nested subqueries. This makes the flow of data transformations much easier to follow.
* **Modularity:** CTEs allow a complex problem to be broken down into discrete, logical steps. Each CTE can be given a descriptive name that explains its purpose (e.g., filtered\_users, aggregated\_monthly\_sales).
* **Reusability (DRY Principle):** A CTE can be referenced multiple times within the same query, eliminating the need to repeat the same subquery logic. This "Don't Repeat Yourself" (DRY) approach makes the code cleaner and easier to maintain.

When structuring CTEs, it is good practice to use them at the top of the query to declare all table dependencies, similar to import statements in other programming languages. Each CTE should ideally perform a single, logical unit of work.

### 3.4 Filtering and Grouping

* **WHERE vs. HAVING:** A fundamental rule is to use the WHERE clause to filter rows *before* any grouping or aggregation occurs. The HAVING clause should only be used to filter groups *after* aggregation has been performed. Using HAVING for a condition that could have been placed in the WHERE clause is inefficient and semantically incorrect.
* **Compact Expressions:** For improved readability and conciseness, prefer IN () over a series of OR conditions, and use BETWEEN for inclusive range checks instead of a compound AND condition (e.g., date >= '2023-01-01' AND date <= '2023-01-31').
* **UNION vs. UNION ALL:** It is critical to understand the difference between these two operators. UNION implicitly performs a DISTINCT operation on the combined result set to remove duplicate rows. This can be computationally expensive. UNION ALL simply concatenates the result sets and is much more performant. Therefore, UNION ALL should be the default choice unless duplicate removal is explicitly required.
* **GROUP BY:** While grouping by ordinal position (GROUP BY 1, 2) is supported by many SQL dialects and can be concise, it is also brittle. If the order of columns in the SELECT list changes, the grouping logic will break silently, leading to incorrect results. For maximum clarity and maintainability, it is strongly recommended to explicitly list the column names in the GROUP BY clause.

### 3.5 Conditional Logic with CASE

CASE statements are a powerful tool for implementing conditional logic within a query. To maintain readability, they should be formatted cleanly: the CASE statement begins the block, each WHEN... THEN... pair is on its own indented line, the ELSE clause follows, and the END keyword, aligned with CASE, concludes the block. For simple binary logic, some dialects offer more concise functions like IFF(condition, true\_value, false\_value), which should be preferred over a CASE statement for brevity.

## Section 4: Documentation and Schema Definition

Code quality extends beyond the query itself to the supporting documentation and the underlying database structure.

### 4.1 Commenting Your Code

Comments are essential for maintaining complex SQL over time.

* **Syntax:** Use -- for single-line comments. Use /\*... \*/ for multi-line or block comments, which is useful for commenting out large sections of code during debugging or for writing detailed header comments.
* **Purpose:** The purpose of a comment is to explain the *why*, not the *what*. The code itself should clearly state what it is doing. Comments should be used to explain complex or non-obvious business logic, justify a particular technical approach, or warn future developers about a potential pitfall.
* **Headers:** For complex models or scripts, it is good practice to include a block comment at the top of the file. This header can provide a high-level overview of the model's purpose, define the grain of the output data (i.e., what a single row represents), and list the author or maintainer.

### 4.2 Data Definition Language (DDL) Best Practices

The design of the database schema itself is the ultimate foundation for clean data and queries.

* **Data Types:** For maximum portability, use standard ANSI SQL data types (e.g., VARCHAR, INTEGER, TIMESTAMP, DECIMAL) instead of vendor-specific aliases (e.g., STRING, INT, DATETIME, NUMBER). To optimize storage and performance, always use the smallest possible data type and column size that can safely accommodate the data.
* **Constraints:** All tables should have explicitly defined and named constraints (PRIMARY KEY, FOREIGN KEY, UNIQUE, CHECK). Named constraints are critical because when a data integrity rule is violated, the database will return an error message that includes the constraint's name, making debugging significantly faster and easier.
* **Primary Keys:** Every table must have a primary key to be considered a true relational table. This key uniquely identifies each row and is essential for creating relationships. Common choices for surrogate keys (keys with no business meaning) are auto-incrementing integers (SERIAL or BIGSERIAL in PostgreSQL) or universally unique identifiers (UUIDs).
* **Defaults and Nullability:** Use NOT NULL constraints by default for all columns unless a NULL value is explicitly required and meaningful for that attribute. Where appropriate, define a DEFAULT value for a column to ensure data consistency upon insertion.

## Appendix: Quick Reference and Tooling

### A.1 The Consolidated Style Checklist

This table provides a high-density summary of the key recommendations in this guide, serving as a quick reference for developers and analysts.

| Category | Rule | Consensus Level | Recommended Practice & Example |
| --- | --- | --- | --- |
| **Naming** | Identifier Casing | Strong Consensus | Use lowercase snake\_case for all names. SELECT first\_name FROM customer\_orders; |
|  | Table Names | Debated | Use **plural** nouns. customers, products, orders. |
|  | Column Names | Strong Consensus | Use **singular** nouns. name, price, order\_date. |
|  | Primary Keys | Majority Consensus | Name the column id. CREATE TABLE customers (id INT PRIMARY KEY,...); |
|  | Foreign Keys | Strong Consensus | Use {table\_singular}\_id. customer\_id, product\_id. |
|  | Booleans | Strong Consensus | Prefix with is\_, has\_, or does\_. is\_active, has\_sla. |
|  | Aliases | Strong Consensus | Use descriptive aliases with AS. FROM customer\_orders AS co. |
| **Formatting** | Keyword Casing | Strong Consensus | Use UPPERCASE for all keywords and functions. SELECT, COUNT(), LEFT JOIN. |
|  | Indentation | Strong Consensus | Use 4 spaces, not tabs. |
|  | Commas | Debated | Use **trailing** commas. SELECT a, b, c FROM... |
|  | Line Structure | Strong Consensus | Place each major clause and SELECT column on a new line. |
| **Query Syntax** | SELECT \* | Strong Consensus | **Never** use in production code. Always list columns explicitly. |
|  | Joins | Strong Consensus | Be explicit: INNER JOIN, LEFT JOIN. |
|  | Complex Logic | Strong Consensus | Use CTEs (WITH clauses) instead of nested subqueries. |
|  | Set Operators | Strong Consensus | Default to UNION ALL. Use UNION only when DISTINCT is required. |

### A.2 Automated Formatting and Linting

Manually enforcing a style guide is tedious and prone to error. Modern data teams leverage automated tools to ensure consistency.

* **SQLFluff:** A highly configurable, dialect-aware linter and formatter that is rapidly becoming an industry standard. It can be integrated into CI/CD pipelines to automatically check and format code, ensuring all contributions adhere to the established style. Below is a sample .sqlfluff configuration file that implements many of the rules from this guide.  
  [sqlfluff]  
  dialect = postgres  
  max\_line\_length = 100  
    
  [sqlfluff:rules]  
  # Naming conventions  
  capitalisation.keywords = upper  
  capitalisation.identifiers = lower  
  capitalisation.functions = upper  
    
  [sqlfluff:rules:capitalisation.keywords]  
  capitalisation\_policy = upper  
    
  [sqlfluff:rules:capitalisation.identifiers]  
  capitalisation\_policy = lower  
    
  [sqlfluff:rules:aliasing.table]  
  aliasing = explicit  
    
  [sqlfluff:rules:aliasing.column]  
  aliasing = explicit  
    
  [sqlfluff:rules:layout.commas]  
  # trailing (default) or leading  
  comma\_style = trailing  
    
  [sqlfluff:rules:ambiguous.join]  
  # Enforces explicit INNER JOIN  
  fully\_qualify\_join\_type = True  
    
  [sqlfluff:rules:convention.select\_star]  
  # Disallows SELECT \*  
  enabled = True
* **IDE Integration:** Many modern IDEs and SQL clients, such as DataGrip, DBeaver, and Visual Studio Code (with extensions), have powerful built-in code formatters that can be configured to match a team's style guide.

### A.3 Glossary of Key Terms

* **ANSI SQL:** A standard for the SQL language published by the American National Standards Institute. Adhering to this standard promotes portability between different database systems.
* **camelCase:** A naming convention where the first word is lowercase and subsequent words are capitalized, with no spaces (e.g., firstName).
* **Common Table Expression (CTE):** A temporary, named result set, defined using a WITH clause, that can be referenced within a SELECT, INSERT, UPDATE, or DELETE statement.
* **DDL (Data Definition Language):** The subset of SQL commands used to define and manage database structures, such as CREATE TABLE, ALTER TABLE, and DROP TABLE.
* **DML (Data Manipulation Language):** The subset of SQL commands used to manipulate data within tables, such as SELECT, INSERT, UPDATE, and DELETE.
* **Hungarian Notation:** A naming convention where a prefix is added to an identifier to indicate its data type or purpose (e.g., tbl\_users). This is considered an anti-pattern in modern SQL.
* **PascalCase:** A naming convention where every word is capitalized, with no spaces (e.g., FirstName). Also known as UpperCamelCase.
* **"River" Alignment:** A formatting style where keywords are right-aligned to create a vertical "river" of whitespace, separating them from expressions.
* **snake\_case:** A naming convention where words are separated by underscores, typically in all lowercase (e.g., first\_name).

#### Works cited

1. SQL Style Guide - Mozilla Data Documentation, https://docs.telemetry.mozilla.org/concepts/sql\_style 2. RootSoft/Database-Naming-Convention: Database Naming ... - GitHub, https://github.com/RootSoft/Database-Naming-Convention 3. SQL Style Guide | Meltano Handbook, https://handbook.meltano.com/data-team/sql-style-guide 4. SQL style guide by Simon Holywell, https://www.sqlstyle.guide/ 5. SQL Best Practices and Style Guide - AWS, https://dbshostedfiles.s3-us-west-2.amazonaws.com/dbs/sql\_best\_practices\_style.pdf 6. SQL Best Practices. (Excerpt from my course at… | by Brandon @ AnalyticsMentor.io | Medium, https://medium.com/@analyticsmentor/sql-best-practices-e1c61e96ee27 7. SQL Style Guide - enable data union, https://enabledataunion.org/docs/manage\_extend/guides/sql-style-guide/ 8. content/handbook/enterprise-data/platform/sql-style-guide.md - GitLab, https://gitlab.com/gitlab-com/content-sites/handbook/-/blob/03493b869d08279e5584cd33747b8aedbdb37ff1/content/handbook/enterprise-data/platform/sql-style-guide.md 9. Markdown format - GitHub, https://raw.githubusercontent.com/treffynnon/sqlstyle.guide/gh-pages/\_includes/sqlstyle.guide.md 10. AWS Redshift View Formating, https://repost.aws/questions/QU1OsGzzCEShi51cecSp6kNw/aws-redshift-view-formating 11. Best Practices to Write Clean SQL Queries With Examples - DataLemur, https://datalemur.com/sql-tutorial/best-practices-for-writing-sql-queries 12. T-SQL style guide by John McCall, https://lowlydba.github.io/tsqlstyle.guide/ 13. SQL Style Guide — Data 101 Course Notes, https://data101.org/notes/appendix/sql-style.html 14. Modern SQL Style Guide - GitHub Gist, https://gist.github.com/mattmc3/38a85e6a4ca1093816c08d4815fbebfb 15. General question -- how do you name tables? Singular or Plural? : r/SQLServer - Reddit, https://www.reddit.com/r/SQLServer/comments/1dvx7g8/general\_question\_how\_do\_you\_name\_tables\_singular/ 16. Singular Vs. Plural Database Table Names - Tandem, https://madeintandem.com/blog/singular-vs-plural-database-table-names/ 17. The table naming dilemma: singular vs. plural | by Fabien Lasserre - Medium, https://medium.com/@fbnlsr/the-table-naming-dilemma-singular-vs-plural-dc260d90aaff 18. SQL style guide by Simon Holywell : r/programming - Reddit, https://www.reddit.com/r/programming/comments/1gr2jpg/sql\_style\_guide\_by\_simon\_holywell/ 19. Database Naming Standards - DEV Community, https://dev.to/ovid/database-naming-standards-2061 20. Singular vs plural table names : r/dataengineering - Reddit, https://www.reddit.com/r/dataengineering/comments/125wlvm/singular\_vs\_plural\_table\_names/ 21. Joe Celko's SQL Programming Style - 1st Edition | Elsevier Shop, https://shop.elsevier.com/books/joe-celkos-sql-programming-style/celko/978-0-12-088797-2 22. FGRibreau/sql-convention: :ok\_hand: The only SQL conventions you will ever need - GitHub, https://github.com/FGRibreau/sql-convention 23. Formatting SQL Code - Part the Second - Simple Talk - Redgate Software, https://www.red-gate.com/simple-talk/databases/sql-server/t-sql-programming-sql-server/formatting-sql-code-part-second/ 24. Software-Standards/SQL Server Coding Standards.md at master - GitHub, https://github.com/CA-CST-SII/Software-Standards/blob/master/SQL%20Server%20Coding%20Standards.md 25. The Pragmatic SQL Style Guide - kvz.io, https://kvz.io/blog/sql-formatting.html 26. SQL style - where do the commas go? - blog.harterrt.com, https://blog.harterrt.com/commas-sql-style.html 27. Prevalence of leading commas vs. trailing commas style, as analyzed via SQL code found on Github - Reddit, https://www.reddit.com/r/SQL/comments/6pazgm/prevalence\_of\_leading\_commas\_vs\_trailing\_commas/ 28. SQL Comma Dilemmas: Should You Trail Behind or Lead Ahead? | by Matteo Bennio | Medium | Google Cloud, https://medium.com/google-cloud/sql-comma-dilemmas-should-you-trail-behind-or-lead-ahead-61988b324c48 29. Leading vs trailing commas on new lines - General - Posit Community, https://forum.posit.co/t/leading-vs-trailing-commas-on-new-lines/6744 30. Marton Trencseni – How I write SQL code - Bytepawn, https://bytepawn.com/how-i-write-sql-code.html 31. SQL Pipe Syntax Style Guide & Best Practices | by Axel Thevenot | Google Cloud, https://medium.com/google-cloud/sql-pipe-syntax-style-guide-best-practices-791d978c3bf2 32. SQL commands - Amazon Redshift - AWS Documentation, https://docs.aws.amazon.com/redshift/latest/dg/c\_SQL\_commands.html 33. Joe Celko's SQL Programming Style 0120887975, 9780080478838, 0080478832, https://dokumen.pub/joe-celkos-sql-programming-style-0120887975-9780080478838-0080478832.html 34. SQL style guide : r/learnSQL - Reddit, https://www.reddit.com/r/learnSQL/comments/1j9gjjc/sql\_style\_guide/