SQL Best Practices & Style Guide

A collection of SQL Best Practices, Style Guide & recommendations for writing SQL

Ben Brumm

www.databasestar.com

SQL Best Practices and Style Guide

SQL is a common language to learn when working with software.

But once you learn the basics, you've probably got a lot of questions about different ways to do things.

How should I format my code? Which alternative of these features is better?

This guide aims to help answer those questions.

Read on to learn how to style your SQL and some recommendations in this "SQL Style Guide and Best Practices".

Introduction

There are several style guides available online for SQL, and many articles that include tips for writing better SQL.

So why write another one? There are a few reasons:

- **More details**: many other guides just list what to do, without any justification. This leads to many comments on Reddit or Twitter about the reasoning. This guide includes the reasoning for all of the choices, many of which you may disagree with.
- **More advice**: there are a lot more suggestions and entries in this list than other Best Practices lists or SQL style guides.
- **Multiple databases**: this guide aims to address many databases including Oracle, SQL Server, MySQL, and PostgreSQL.
- **Alternatives**: I offer alternatives in some of the recommendations, if you or your team disagree with the advice in the guide, and where an alternative makes sense..

If you have any questions or comments, feel free to add them into the comments section at the end of this post. Let me know if you agree or disagree with them.

Finally, when writing SQL or designing a database, **the most important thing in my opinion is to be consistent**. If you choose one approach for something, even if it disagrees with this guide, try to be consistent across your database and your code. This will help you and your team work better together, as well as any future team members.

Naming Conventions

Avoid spaces in object names

Why? To avoid possible errors, and the need to always use quotes.

When you create tables, views, stored procedures, or any other kind of object, you need to give them a name. In many databases, you can add a space in an object name, as long as the name is enclosed in quotes:

```
CREATE TABLE "Product Category" ( …
```

The problem with this approach is that you can't select from a table that has spaces in its name, unless you also use the quotes:

```
SELECT id, category_name
FROM "Product Category";
```

This makes the code more confusing, inconsistent with other queries and objects, and arguably harder to work with.

A better way is to use underscores instead of spaces, which I have detailed below.

Avoid quotes with names

Why? To avoid possible errors, and the need to always use quotes.

When you create a table, you can specify quotes around the table name. This is possible if your table name has a space or not.

```
CREATE TABLE "CustomerOrder" ( …
```

The problem with this is that quotes are required every time you work with the table, as the table name is stored in the exact case you specified.

This means these queries won't work:

```
SELECT * FROM customerorder;
SELECT * FROM CustomerOrder;
```

You'll need to write this query:

```
SELECT * FROM "CustomerOrder";
```

This means quotes are needed every time you refer to the table, making it harder to work with and harder for your team to remember.

The solution? Don't use quotes when creating the table.

Avoid square brackets around your object names

Why? It adds unnecessary characters and makes it harder to read.

In some databases, such as SQL Server, it's common to see queries that have square brackets around their object names, such as tables or columns. This is true for generated code (from the IDE) or examples online.

```
SELECT
[id],
[first_name]
FROM [employee];
```

While the query will work with these square brackets, it adds unnecessary characters to the query and arguably makes it harder to understand.

I recommend removing square brackets from your SQL queries. It's obvious without the brackets that the words refer to columns and tables.

Use underscores instead of camel case

Why? Easier to read and case is often not retained.

When you need to name a table that consists of more than one word, and you can't use spaces, what do you do?

There are two options.

The first is camelCase, where you capitalise the first letter of every word but don't add any characters in between:

```
CREATE TABLE ProductCategoryLookup ( …
```

The second is snake_case, where you add an underscore in between each word:

```
CREATE TABLE product_category_lookup ( …
```

I recommend using snake_case, or adding underscores to your table names, instead of camel case.

There are a couple of reasons for this: case conversion and readability.

In many databases (e.g. Oracle, PostgreSQL), the case of the object name is not kept when stored in the database, so the name ProductCategoryLookup is stored as productcategorylookup or PRODUCTCATEGORYLOOKUP. This defeats the purpose of using camel case, as you can't see the camel case when looking at the name in the object explorer or any list of tables from the data dictionary.

Using underscores arguably makes it easier to read. You can see a clear space between words when you look at it, and it's readable in both upper case and lower case.

So, I recommend using underscores to separate words in your object names rather than camel case.

It does add extra characters to the name, but unless you're on a version of Oracle earlier than 12.2, it's likely not an issue.

Here are the maximum number of characters an object name can have:

Database	**Max Character Length**
Oracle (from 12.2)	128
Oracle (before 12.2)	30
SQL Server	128
MySQL	64
PostgreSQL	63

Avoid prefixes for object names (Hungarian Notation)

Why? It's not needed, it makes it harder to change code.

One tip for naming objects in databases that I see in examples online and database designs is to add a prefix to the name of the object to indicate what type of object it is.

For example:

```
CREATE TABLE tbl_customer ( …

CREATE VIEW vw_emp_salary ( …

CREATE PROCEDURE sp_add_employee ( …

CREATE FUNCTION fn_salary_ex_tax ( …
```

You can see these examples show the object type before the name: tbl_customer for the customer table, and so on. This is called Hungarian Notation.

The reason this is done is that you can see what type of object it is when you see it in your query.

However, I recommend **not adding a prefix** to your object names.

This is because you don't need to know what type of object it is when you see it in your query.

If you select from something called "customer" (without the tbl_ prefix), you don't need to know by looking at it that it's a table and not a view. You can see in your IDE if it's a table (e.g. hover over the name). You just need to know that you select data from it.

```
SELECT
id,
customer_name
FROM customer;
```

Also, if you decide to refactor your code and change the type of object the query uses, it's much easier. You can change the customer table to a view, and still call it customer, and your code will still work:

```
SELECT
id,
customer_name
FROM customer;
```

If it has a prefix, you need to change it:

```
SELECT
```

```
id,
customer_name
FROM tbl_customer;
```

This needs to change to:

```
SELECT
id,
customer_name
FROM vw_customer;
```

The same logic applies to other objects you use in your queries such as functions and stored procedures. You don't need to know what type of object it is.

It also takes up extra characters in the name. While this may not be an issue if your name is short and your maximum length is a lot larger, it adds extra unnecessary characters and can make it a bit harder to read.

So, avoid the prefix and just call the object what it is.

Avoid reserved words for object names

Why? Queries may not run.

There is a range of reserved words in SQL. These words are used by existing objects, functions, and features. The words are similar across different databases.

Some examples of reserved words are:

- User
- Order
- Upper

Try to avoid naming your tables, columns, and other objects as a reserved word. If you use a reserved word, it can cause your queries not to work correctly.

For example, a common table in many databases is one to store users. You may want to call this table "user".

However, user is a reserved word and is used by the database. If you try to call your table user, you may get an error on creation, or you may get an error when you try to query the table.

So, instead of using a reserved word, use something else. Perhaps use the name website_user or app_user instead of user. Use the word customer_order or product_order instead of order.

Use singular names for tables instead of plural

Why? Preference.

When naming tables, you'll come across the question of whether you should name your tables in the singular form or plural form.

This is one of the most commonly debated points on many style guides or recommendations by others.

Should you name tables in a singular form, e.g. "employee" or "product"?

Or should you name tables in a plural form, e.g. "employees" or "products"?

One opinion is that a row in a table is a single instance, so the table should be singular. The other opinion is that a table is a collection of data and should be plural.

My recommendation is to name it in the singular form. This means I would call my tables employee, product, appointment, player, customer_order, and so on.

This is what I recommend and what I try to do with databases I design.

However, there are two other points to make.

Firstly, this is more based on preference than on any issues with performance or errors. If you have the opinion that it's better to have plural names, then go for it.

Secondly, as I mentioned earlier in the guide, it's more important to be consistent. If your team has plural names, then stick with plural. Or if your team has singular names then stick with singular.

So, this is less of a rule than other entries and more of a suggestion.

Don't name columns the same as the table

Why? It can cause confusion and may cause errors.

When you add columns to a table, you need to provide a name.

While it's possible to name a column the same as the table name, I don't recommend it.

For example:

```
CREATE TABLE customer (
id INT,
customer VARCHAR(100)
);
```

There is a column called customer in the table called customer.

I don't recommend this as it can cause confusion when reading and writing queries. It can also cause errors.

Let's consider this query:

```
SELECT
id,
customer
FROM customer;
```

This may run, depending on your IDE and database. Or you might get an error, as the IDE or database might not know which customer is the table and which is the column.

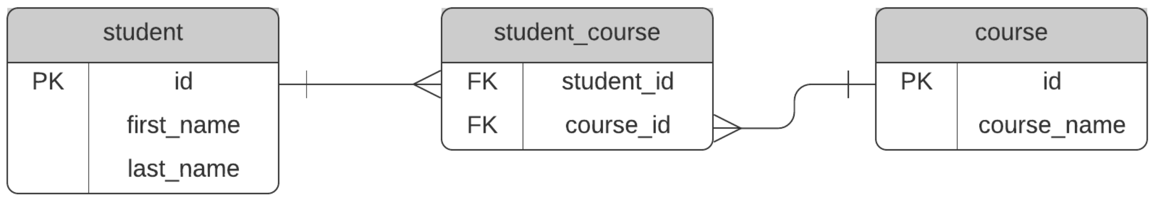
So, don't name your columns the same as the table name.

Name joining tables based on what they represent

Why? It's clearer what they represent.

When you have a many to many relationship in a database, the way to design it is to have a bridging table or joining table. This will let you store valid combinations of the IDs from both tables.

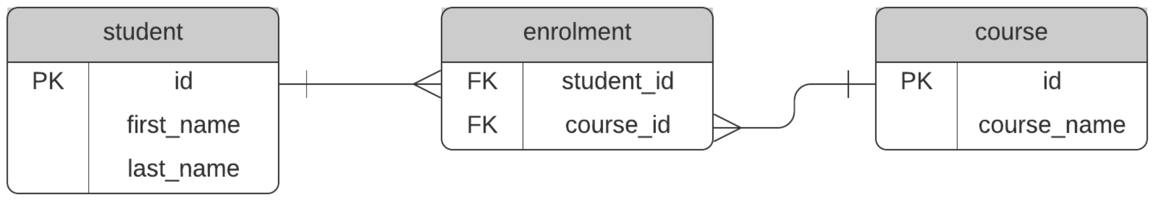
For example, a college database has a course table and a student table. A course has many students, and a student has many courses, so a joining table is needed.



A common way to name this table is the combination of the two tables involved. In the diagram above, the joining table is called student_course.

However, I would recommend that this joining table is given a name that represents this relationship, so it's easier to see what the table is for.

We're not just storing a combination of students and courses here, we are storing student enrolment in courses. So, the table could be called enrolment.



The name enrolment refers to what the joining table represents. This makes it easier to see what the table is for.

If you call it by the two tables that are being joined, it may not be clear, especially if your tables can have multiple ways to be related.

However, if there's no way to come up with a name that represents this, then the combination of the two related tables is OK.

Specify names of constraints instead of the default

Why? It's easier to understand when referring to execution plans.

When you create a constraint for a table, you have the option of creating an inline constraint or an out of line constraint.

An inline constraint means you can just add the constraint type and details after the column name:

```
CREATE TABLE customer (
id INT PRIMARY KEY,
customer_name VARCHAR(100)
);
```

In this example, the word PRIMARY KEY indicates that the id column is the primary key. This will work. However, it means that the primary key constraint will have a name generated by the database with a series of random letters and numbers.

This name makes it hard to identify which table and columns it refers to.

Why does this matter? Because you may see these constraint names in the data dictionary or execution plans.

If you create the constraint using the out of line method, then you'll need to provide name:

```
CREATE TABLE customer (
id INT,
customer_name VARCHAR(100),
CONSTRAINT pk_cust PRIMARY KEY (id)
);
```

This has the same result of creating a primary key on the table. However, the name given to this primary key is pk_cust. When we see this in the data dictionary, we know what it is.

When we see it in an execution plan, we know what it is. We can tell it's a primary key constraint and that it refers to the customer table.

Giving your constraint a name makes it easier and faster to identify the constraint when you see it in your database.

Consider prefixes on constraint names

Why? So you can easily identify it in the database.

In a related point to the above point on naming constraints, I recommend adding a prefix to your constraint names.

This is so you can identify what type it is when you see it in an execution plan, or in the data dictionary.

There may be a column that indicates the type of constraint, but if it's in an execution plan, you won't see that column.

So, a convention of something like this could be useful:

- pk for Primary Key
- fk for Foreign Key
- uc for Unique Constraint
- cc for Check Constraint
- nn for Not Null (if you're able to give it a name)

This goes against the idea earlier in this guide for not giving prefixes to object names. However, this is different, because you can substitute one constraint type for another.

So, consider using a prefix so you can better identify it in an execution plan or data dictionary table.

Database Design

Don't try to apply object-oriented design principles such as inheritance

Why? It's not what they are designed for.

Databases store data. The tables often store the same kind of data that your objects in your code store.

Your application code has likely been designed using principles that improve the quality and maintainability of the code, such as object-oriented design.

Object-oriented design works well for code. But it does not work well for database tables.

One principle that is used in code is inheritance. This is where you can have a generic object type, such as Person, and then more specific object types such as Customer or Employee.

With a database, however, it's not a good idea to use this kind of structure. Databases are not designed for handling inheritance.

So how do you store this kind of data? Store the tables separately. There's no need to develop an inheritance structure in your tables. You can store data in tables separately and let your code handle the object creation.

Use vendor-specific data types

Why? Databases are focused on their data types, and portability is not a reason.

There are several recommendations out there that you should use data types in the SQL standard because they are portable. This will make it easier if you ever want to move database vendors, such as from Oracle to MySQL, or SQL Server to PostgreSQL.

The theory is if you have something stored as a VARCHAR, you don't need to adjust the type when you move databases.

However, I believe that recommendation is wrong. Here's why.

Moving databases is rare. Once applications are built on a specific database platform, it's rare that they move to a different vendor. It's not impossible (a previous client was moving from Oracle to PostgreSQL for many applications), but it's not common.

Moving databases is a big deal. If you ever have to move databases for an application, the data types of columns will be the least of your worries. You can't simply recreate your code on the new database and expect it to work. You need to assess your queries and data structure and test for performance and data quality.

Databases implement specific data types which they optimise for. I recommend using those.

For example, don't use VARCHAR in Oracle because it's the standard, use VARCHAR2.

If your database supports BOOLEAN, use that, don't use CHAR(1).

The database is optimised to use the specific data types that are included, so using a data type for reasons of portability is not ideal.

Prefer the use of surrogate primary keys

Why? To reduce impact of change and to improve relationships between tables.

A primary key in a table is a way to uniquely identify a record. They are used to refer to the record over the life of the database in its table and in other tables that refer to it, using foreign keys.

There are two methods for creating a primary key.

One is to use a field or combination of fields that already exist in your data. This is called a "natural key". Some examples are:

- Social Security Number or Tax File Number
- Phone number
- Email address
- Username

These are fields that have "relevance to the business", which means they are used by the people that use the system. They have relevance. And, this means they can change.

Another method is to use a new field for a primary key. The purpose of this field is solely to identify the record. This is called a "surrogate key". Some examples are:

- Account Number
- Customer ID

This is a single field generated by the system. It has no meaning to the users of the system and cannot change.

Which method should you use?

I strongly recommend the surrogate key approach.

This is so the primary key can be used to uniquely identify the row and not be impacted by any business-relevant fields in the table. These other fields are free to be used how they need to be.

Create a new field for your primary key. Give it a primary key constraint. Use it as foreign keys in your other related tables.

There are two methods you can use to create a primary key in this way:

- auto-increment integer field
- system generated GUID (an alphanumeric value)

Either method works, and the method you use depends on your system. I've used auto-incremented integer fields almost every time, and have seen GUIDs work well too.

Avoid using business-relevant fields for your primary keys.

Add unique constraints where columns must be unique

Why? To improve data quality.

If you have columns in your tables that must be unique, and are not the primary key, then you should add unique constraints to them.

This could be for single columns, or multiple columns.

For example, if usernames need to be unique, then add a unique constraint. You can then easily enforce unique usernames in your system.

A column does not have to be a primary key to be unique. They can be different.

Use Not Null constraints and default values

Why? To improve data quality.

If you have columns in your table that are mandatory, then add Not Null constraints to them. This will ensure values are always used.

You can also use default values for columns. This can be specified for columns in a table, so if a value is not provided, a default is used.

For example, a default status for a customer could be active.

```
CREATE TABLE customer (
id INT,
customer_name VARCHAR(100),
customer_status VARCHAR(10) NOT NULL DEFAULT 'Active'
);
```

This means the value is always populated with something, but if it's not specified in the Insert statement, the default is used. It's easier to understand compared to a null value in a column.

However, don't go overboard with NOT NULL columns. Just because a column should not be empty doesn't mean it has to be NOT NULL. Adding more mandatory fields can mean data entry is harder.

Avoid floating-point data types unless you really understand them

Why? They can cause rounding issues due to the way they are stored.

Floating-point numbers have sme advantages, such as being able to easily work with large numbers.

However, due to the way they are stored in the database, they can result in rounding errors when working with them.

So, unless you truly understand floating-point numbers, don't use them. There are other data types that can be used for storing decimal numbers.

Avoid putting units of measure in separate columns

Why? It makes it harder to work with data.

If you need to store numeric data that can be in different units, it might be tempting to use one column for a value and one for a unit of measure.

For example:

```
CREATE TABLE product (
id INT,
product_name VARCHAR(100),
length INT
length_unit VARCHAR(10)
);
```

You could then store products like this:

id	product_name	length	length_unit
1	Couch	2.1	metres
2	Tallboy	45	centimetres
3	Rug	110	centimetres

This could work. However, it means that your number values are inconsistent. They can't be compared to each other, they can't be displayed in a consistent way, and they can't be sorted.

So, what should you do?

Store the measurement in a column in the same unit.

This means:
- A length_cm column for length in centimetres
- A distance_miles column for the distance in miles
- A weight_lb column for the weight in points

This means the values can be sorted, compared, and displayed in a consistent way. It also means one less column in your table.

Our product table could look like this:

id	**product_name**	**length_cm**
1	Couch	210
2	Tallboy	45
3	Rug	110

Avoid CHAR data type

Why? It takes more space, it's hard to compare, and offers no benefits.

There's a data type available in the SQL standard called CHAR. It's used to store character data. It's available in many databases.

It's a fixed-length data type. This means that a field of CHAR(10) will always have 10 characters stored. Values that are shorter than 10 character will be padded with spaces to make 10 characters.

I suggest avoiding the CHAR data type at all times.

Use your database's variable length data type, such as:
- VARCHAR
- VARCHAR2
- TEXT

These data types are better than CHAR because:

- They take up less space as they don't pad with spaces
- You don't need to trim the fields to remove spaces to compare them
- They shouldn't impact performance

Some arguments can be made for using CHAR for fields that have a fixed length all of the time. For example US states are always two letters.

This may be true. However, if you use VARCHAR in your database in every place except for one or two fields, it's not consistent. VARCHAR and CHAR may operate in the same way for a fixed length, but there's no guarantee that they will always be a fixed length. It could be better to use VARCHAR to be consistent, and add a check constraint.

Use the right date data type

Why? To meet your requirements.

There are many different data types available in different vendors.

There are data types that store date only, time only, date and time, date time with timezone, and with fractional seconds.

It might be tempting to use a simple data type.

However, working with dates is hard. Especially when time zones are involved.

So, choose the most appropriate data type that addresses your requirements and exists in your database vendor.

Each data type in each vendor is different and has different uses.

Use INT(1) for boolean if your database does not have boolean data type

Why? For easier understanding.

Sometimes you may need to store a boolean value (true or false) in your database.

If your database includes a boolean data type, then you can simply use this type.

However, many databases don't. So what should you do?

Some options are:

- INT(1) with 1 for true and 0 for false
- VARCHAR(1) or CHAR(1) with Y for true or N for false
- VARCHAR(1) or CHAR(1) with T for true or F for false
- VARCHAR(5) with either TRUE or FALSE

The approach I recommend is to use an INT column with a check constraint. You can create a column as an INT (or an equivalent integer data type), where the value of 1 is true and 0 is false. You can add a check constraint to ensure the value is one of those two values.

Why shouldn't you use a text field with T/F or Y/N? Because those values aren't as clear. It may not be clear that Y is Yes and N is No, especially for non-English speakers where their words for Yes and No are different. The same can be said of T/F.

But, 0 and 1 are clear. 0 is false and 1 is true.

Avoid the Entity Attribute Value design

Why? It avoids referential integrity and can be slow.

Entity Attribute Value is a type of table design that's occasionally used in databases. It's where you create a table that has three columns:

- Entity: an ID that represents a row in another table
- Attribute: the attribute of that entity
- Value: the value of that attribute for that entity

For example, storing information about employees:

entity_id	**attribute_name**	**attribute_value**
1	first_name	Steve
1	dept	Sales
2	first_name	Michelle
2	dept	Support

This is a simplified example, but during the course of your database design you may come across a scenario where this design makes sense.

It has advantages: it's simple, flexible, and can solve a problem.

However, I believe it's almost never the right solution to the problem.

The data is not normalised, as you just have a range of attributes in a table.

You don't have referential integrity or any way to prevent duplicates, as attributes are in separate rows.

You can't validate the data for attributes, as both attribute and value need to be text values.

A better way to solve this problem is to design proper normalised tables. There may be more than one table, and it may look messier on the ERD, but it will likely function better.

Avoid storing multiple pieces of information in a single field

Why? It reduces data quality and complicates the application.

Sometimes you'll have some data to store in a column that's actually made up of several pieces of information.

An example of this is a street address. An address is actually made up of many pieces of information, such as house number, street, suburb, city, state, country, and postal code.

In most cases, you should avoid storing multiple pieces of information in a single field. This means you would store this information in several fields.

This will allow you to filter based on these separate fields (e.g. addresses in a certain city), order the data, and display it however you like in a report or application.

If it's in a single field, this is harder to do.

So, if you have a column that stores separate pieces of information, consider storing them in separate fields.

Create indexes where needed, but not everywhere

Why? To optimise performance.

Database indexes are a great feature. They allow queries to run faster when accessing data in a certain way.

You specify columns in a table to create an index on.

If indexes speed up access to tables, why not create indexes on all columns?

The reason is that this can slow down all other operations on a table, such as insert, update, and delete.

Indexes should be created on a table to help queries that you have and that are important to run faster. If you create indexes on all columns, many of them won't get used, and your other operations will slow down.

So, analyse which queries are accessing the table, determine which columns should be indexed, and create indexes on those.

Use thoughtful maximum sizes for columns

Why? To improve data quality.

When you create a table, you specify data types for columns. In many cases, you need to specify the maximum size of a value for that data type.

For example, VARCHAR(100) will allow a text string of up to 100 characters.

If you need to specify the maximum, why not just enter the highest possible number every time?

For example, VARCHAR(4000).

This should store all the data you need, right?

The reason I recommend not doing this is because a smaller field can improve data quality. If you know that your City field, for example, can be up to 200 characters, don't create a field for 4,000 characters.

Having a 200 character field will ensure that all values are within this 200 character limit, and prevent other data from being added.

If you have a 4,000 character field, your application may add other data into the field which won't be tested or validated.

You may also reduce the optimisation of the data structure if all fields are at their maximum. This depends on the database, which improves with each version, but it could be impacted.

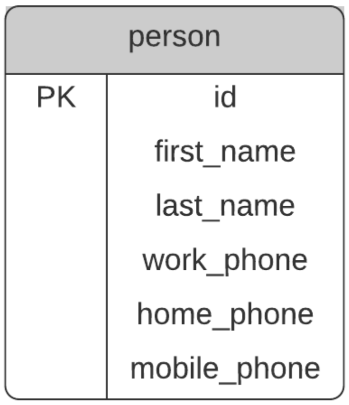
Store multiple optional types of data in a separate table

Why? To improve data quality and reduce complexity.

There may come a time where you need to store multiple different values in a table, and each of them are optional.

A common example of this is a phone number. You want to store a phone number for someone, and the type of phone number. But the phone number could be for work, home, or mobile.

One way to do this is to create three columns:



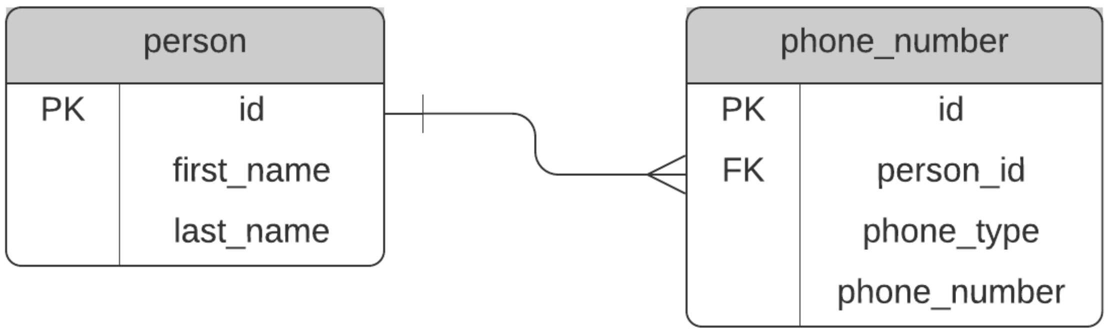
These three columns would be optional, and are populated based on the type of phone number that the user entered.

However, this design is not ideal. It means you have columns that are empty in most cases.

If you have another type of phone number, you need to add another column, which can be hard to do if an application is live.

A better way to approach this is to have a separate table for the optional data. You can relate this to your main table, and your second table stores the record and its type.

For example:



In this example we have the separate phone number table, which has the phone numbers for each person and their type. You could also have a separate phone number type table if you want.

This will ensure the main table has fewer columns and your other table can capture the right data, and additional types if needed.

Don't create large tables just to avoid joins for "performance"

Why? Performance is much more than the number of joins involved.

If you are considering creating a large table that has a lot of columns in order to avoid joining to other tables, as you think it may slow down your query, you may want to reconsider.

Sure, having large tables to avoid joins is one approach, and one that works well in a data warehouse environment that relies heavily on reads instead of writes.

However, joins are not the reason a query may be slow. Adding extra joins likely won't slow down the query that much.

It's more important to have a normalised database that improves data integrity and lets you join to data that's required for each query.

Don't store passwords in plain text

Why? It's bad for security.

This might seem obvious but I wanted to add it in anyway.

You may need to store passwords in your database for an application you're building.

If you do, don't store passwords in plain text, or the way that the user has entered them. You should encrypt them before you store them.

If you store them in plain text, it means anyone with access to viewing that column in your table can see the passwords. They can also be extracted and shared outside of the database. Both of these are a security risk.

A better way is to encrypt the passwords before you store them. There are a few different encryption methods, so research which is more appropriate to use, and use it for your application.

Common Fields

Person Names: two separate text fields

Why? To allow filtering, sorting, and display.

Storing people's names may seem simple but can actually be hard to design. What's the maximum length? Do you store the fields separately or together? Are both required?

There's a great article here that lists a range of falsehoods that programmers believe about names. It explains a lot of things we think are true about names but actually aren't.

I believe a design for handling people's names should cater for many of these scenarios.

A good way to handle names is to use two text fields in your table. These fields should be several hundred characters long, to cater for a wide range of names. They should also be named to allow for different cultures to enter names differently (e.g. "surname" is not common in many cultures).

Your user interface can make the data entry easier, but for the database, you need to handle the storage requirements.

Having separate fields allows you to filter on different parts of the name, and display them as needed in the application or a report.

Email Address: text field of 320 characters

Why? To handle the maximum allowed length.

An email address can be up to 320 characters, which includes:

- 64 characters for the username
- 1 character for the @ symbol
- 255 characters for the domain.

If you have a VARCHAR(320) field, you'll be able to store the maximum size of an email address.

An alternative design could have separate fields for the username and domain, but this would depend on how you use the email address. Having a single field is usually enough.

IP Address (v4 and v6): store as a number

Why? Less space, and can be formatted for display.

An IP address is actually a number that is formatted in a certain way.

It may be easier to store IP addresses as text values (IPv4 as 15 characters and IPv6 as 50 characters), but storing as a number has several benefits.

You will likely take less storage space if you store it as a number.

You don't need to validate the data. If it's text, you allow values to be stored that don't adhere to the right format.

The data can be converted when inserting or selecting, to allow data to be added or displayed easier.

So, consider storing both IPv4 and IPv6 as a number field.

Money: store as a large number with 4 decimals

Why? To improve data quality.

Money or currency data can seem simple to store. It's just a large decimal number.

There are a couple of things to consider.

First, use more decimal places than you think you need. You can always remove precision from your displayed numbers, but can't add precision if it's there.

Second, make it larger than you think you'll need. A common way to store money is up to 15 digits excluding decimals.

This will depend on your requirements, but those are two things to consider.

An alternative is to store the data in the base unit, such as cents instead of dollars.

Addresses: store as separate fields in a separate table

Why? To improve data quality.

An address is a physical location. It contains several pieces of information:

- unit number
- house number
- city
- region or state
- postal code
- country

Many of these may need to be filtered on or sorted or displayed differently in an application, so it's helpful to store them in different fields instead of a single field.

Also, a person's relationship to an address is not an ownership. An address exists even if a person does not live or work there. A person may also have multiple addresses.

If a person moves out of a house, the address still exists, it's just their relationship to that address that changes.

For these reasons, I believe it's better to store addresses in a separate table to people or companies or whatever entity you're storing. You can then relate these to the addresses.

So, with a separate address table that's related to your entity, and separate fields for different components, you can improve your data quality.

Phone Numbers: store as text

Why? To improve data quality.

Phone numbers, whether they are landline numbers or mobile numbers, are a series of digits displayed in a certain way.

However, phone numbers shouldn't be stored as numbers. They don't need to be added together or have other arithmetic performed on them.

Phone numbers are actually identifiers for contacting a phone. They should be stored as text values, not numbers.

Storing them as text will allow the database to store them exactly as they have been entered. Storing them as numbers is a problem as numbers will trim leading zeroes, which exist in many phone numbers.

If you store them as text, you can also allow the application or report to display them as needed. You can display them using whatever combination of brackets, spaces, and dashes that are appropriate.

So, store phone numbers as text instead of number.

Syntax and Formatting

Use meaningful table aliases as your query grows

Why? To improve readability.

Table aliases are a useful feature of SQL. They simplify your query and make it smaller, as you don't need to specify the full table names. They allow you to use autocomplete in your IDE easier, as the columns are often shown after you enter the table alias. They are also required if you need to do a self-join.

I recommend using meaningful table aliases. This means the following aliases are OK:
- One or two letter abbreviations of the table name (e.g. c or cu for customer)
- Several characters for an abbreviation (e.g. cust for customer)

It should be obvious by looking at the table alias which table it refers to.

The following styles of table aliases should be avoided:

- Single letters where there are multiple tables of the same name (e.g. s for staff where there is a staff and student table)
- Adding numbers for extra tables (e.g. t1, t2)

There are so many examples online that use single letters with numbers for table aliases. This defeats the purpose of using a table alias and doesn't show the benefits of them.

So, use meaningful table aliases in your query. Your future self and your team will thank you.

Use AS for column aliases

Why? To prevent errors and improve readability.

Adding a column alias involves using the word AS and then a new name:

```
SELECT
id,
fn AS first_name,
category
FROM employee;
```

The AS keyword is optional when defining a column alias. The following query will work and show the same results and headings:

```
SELECT
id,
fn first_name,
category
FROM employee;
```

However, I recommend using the AS keyword when defining column aliases. It makes it clear that the word coming after the column is an alias. It also prevents issues where you may miss a comma.

For example, this query will work:

```
SELECT
id,
fn
```

```
category
FROM employee;
```

However, you'll only get two columns: the id column, and the fn column. The category column is not shown, as the word category is used as a column alias for the fn column. It's not clear if this is deliberate, or if a comma is missing. Especially if the query is formatted like this:

```
SELECT id, fn category FROM employee;
```

So, use the AS keyword when working with column aliases.

Use a consistent case for keywords

Why? To improve readability.

One of the most common debates about SQL style is whether SQL keywords should be in upper case.

Here's an example of a query with uppercase keywords:

```
SELECT
id,
first_name,
last_name
FROM customer
WHERE active_status = 2
ORDER BY last_name ASC;
```

Here's the same query with lowercase keywords:

```
select
id,
first_name,
last_name
from customer
where active_status = 2
order by last_name asc;
```

Uppercase keywords are preferred by some because they make it clear which part is the SQL keywords and which are the fields in your query. They also stand out from the rest of the query, which is helpful if your query is in a string variable in application code.

Lowercase keywords are preferred by some because they are easier to read, easier to type, and the syntax highlighting in IDEs will show that they are keywords.

Which style should you use?

I recommend being consistent with your code. If that means lowercase because the rest of your queries are lowercase, then stick with that. If you're starting a new project, then make a choice of which one you prefer.

Personally, I prefer uppercase, and my code examples on this site are written in uppercase. But it's up to you. Just be consistent.

Keywords should start on a new line

Why? To improve readability.

Most of the SQL keywords in your query should start a new line. This improves the readability of your query.

For example:

```
SELECT columns
FROM tables
JOIN table ON column = column
WHERE condition
GROUP BY column
HAVING condition
ORDER BY column;
```

Reading a query in this format is easier than if it was all on a single line, or spread over only a couple of lines.

While a query in this style would work, it's harder to read:

```
SELECT columns FROM tables JOIN table ON column = column WHERE
condition GROUP BY column HAVING condition ORDER BY column;
```

Columns in SELECT clause should be on their own lines

Why? To improve readability and make it easier to change.

When you select multiple columns in your query, place each column on its own line.

```
SELECT
id,
first_name,
last_name
FROM employee;
```

This makes it easy to see which columns are being selected. It makes it easy to remove or comment out columns so they are not shown. It's also easy to rearrange columns if you want them in a different order.

Don't group them based on their data type or name or try to fit them to a certain character width per line. Just add them to separate lines. It's much easier to read and work with than this:

```
SELECT id, first_name, last_name
FROM employee;
```

This is even more true when your queries get larger.

Commas for columns should go at the end of a line

Why? It's more natural.

Another common piece of advice is where commas should go in the SELECT clause.

When we first learn SQL, we add commas to the end of the line:

```
SELECT
id,
first_name,
last_name,
active_status,
category
FROM employee;
```

A common alternative is to place commas at the start of the line:

```
SELECT
id
,first_name
```

```
,last_name
,active_status
,category
FROM employee;
```

The reason for this is that it makes it easier to comment out columns and reduce errors. If you want to remove the active_status column for example, just comment out that line. You don't need to remove the comma from the previous line:

```
SELECT
id
,first_name
,last_name
--,active_status
,category
FROM employee;
```

However, this approach is not ideal as it just moves the problem to a different place in the query. If you need to comment out the id column, you need to comment out that row as well as remove the comma on the next line.

```
SELECT
--id
first_name
,last_name
,active_status
,category
FROM employee;
```

So the problem still exists in both formats. Leaving the comma at the end of the line is more natural and easier to read:

```
SELECT
id,
first_name,
last_name,
active_status,
category
FROM employee;
```

Formatting a query to right-align keywords is unnecessary

Why? It adds extra work.

A common suggestion for formatting code is to right align your SQL keywords and left align the criteria.

For example:

```
SELECT     id,
           first_name,
           last_name,
           category
      FROM employee
INNER JOIN department
        ON department.id = employee.dept_id
     WHERE active_status = 2
  ORDER BY last_name ASC;
```

This makes a "river" or a line of white space down the middle between the SQL keyword and the columns or criteria, which some believe is easier to read and looks nicer.

However, I believe this is unnecessary. It also adds extra work into formatting. Even if you have this set up to automatically format in your IDE, it's still an extra step.

So, just leave it as left aligned:

```
SELECT
id,
first_name,
last_name,
category
FROM employee
INNER JOIN department
ON department.id = employee.dept_id
WHERE active_status = 2
ORDER BY last_name ASC;
```

Add a space before and after =

Why? It improves readability.

When you use an equals sign =, or any other operator, it's helpful to add a space before and after it. This makes it easier to read.

Here's an example without a space:

```
SELECT COUNT(*)
FROM employee
WHERE active_status=2;
```

Here's the same example with a space:

```
SELECT COUNT(*)
FROM employee
WHERE active_status = 2;
```

While spaces are not required, they do improve readability.

Treat tables in joins as siblings and don't indent them

Why? It reduces work and improves readability.

Some common advice for formatting your SQL involves indenting your tables in the join clause, like this:

```
SELECT
id,
first_name,
last_name
FROM employee
     INNER JOIN department
          ON employee.dept_id = department.id
     INNER JOIN location
          ON department.loc_id = location.id;
```

However, this doesn't make sense. All of the tables in the FROM clause (including joins) are treated equally and are siblings, so they should have the same level of indentation. It makes it harder to see which tables are included as you need to move left and right to see all the tables.

A more consistent way of indenting is to not indent the JOIN clauses:

```
SELECT
```

```
id,
first_name,
last_name
FROM employee
INNER JOIN department
     ON employee.dept_id = department.id
INNER JOIN location
     ON department.loc_id = location.id;
```

This makes it easier to see the tables involved.

Put multiple join conditions on separate lines

Why? To improve readability

If you're working with a simple query that has one join criteria, you can usually put them on a single line:

```
SELECT
id,
first_name,
last_name
FROM employee e
INNER JOIN department d ON e.dept_id = d.id
INNER JOIN location l ON d.loc_id = l.id;
```

However, if you have multiple join criteria, it can improve readability to have them on separate lines.

```
SELECT
id,
first_name,
last_name
FROM employee e
INNER JOIN department d ON e.dept_id = d.id
INNER JOIN location l
     ON d.loc_id = l.id
     AND d.loc_type = l.loc_type;
```

This prevents your query from getting too wide, and makes it easier to see what each of the criteria are.

Indent subqueries

Why? To improve readability and maintainability.

Subqueries are a helpful feature of SQL. When you write them, it's easier to read the query if the subquery is indented.

Here's an example of a query where the subquery is not indented:

```
SELECT
id,
first_name,
last_name
FROM employee
WHERE start_date > (
SELECT AVG(start_date)
FROM employee
WHERE active_status = 2
);
```

Here's the same query that as the subquery indented:

```
SELECT
id,
first_name,
last_name
FROM employee
WHERE start_date > (
     SELECT AVG(start_date)
     FROM employee
     WHERE active_status = 2
);
```

The second version is more readable and easier to understand. It's more obvious which part is the subquery.

Finish each statement with a semicolon

Why? It reduces errors.

In most database vendors, SQL statements need to end with a semicolon. This makes it clear where the query ends, and is required if you're running a script with multiple queries.

Some vendors don't require a semicolon, but adding a semicolon is a good practice to get into.

Functionality

Use vendor-specific functionality

Why? It's optimised for the database.

There is some advice that mentions you should only use functionality that's available in the SQL standard, so your queries can be moved to other databases easier.

However, I believe the opposite is better: you should use vendor-specific functionality if it's the best choice for your query.

Moving databases is a rare occurrence, and involves much more than just converting functions in a query. I mentioned this earlier in this guide.

If you decide to migrate to a different database, it will likely be a lot of work, and the time savings you get from having the same functionality is likely not as important as the advantage you would get from using vendor-specific functionality.

So, this means you should consider using vendor-specific functions, syntax, and data types, if it's the best choice for your solution. Don't avoid a feature just because it's not in the standard.

Avoid using BETWEEN for dates

Why? To avoid issues with incorrect data.

The BETWEEN keyword in the WHERE clause allows you to filter for data that is between two values. This keyword is inclusive, which means the values you specify are included in the range.

This is OK for numbers:

```
SELECT COUNT(*)
FROM employee
WHERE id BETWEEN 3 AND 6;
```

However, it can be tricky for dates. It includes the start date and end date:

```
SELECT COUNT(*)
FROM employee
WHERE last_updated_date BETWEEN 20200101 AND 20200201;
```

Depending on the data type of your column, you may end up including values where the last_updated_date is equal to the second value: 20200201 at midnight. This may not be correct, as you don't want to include February's date in January's query.

You could change it to be one second before, or the day before, but this still leaves room for error.

A better way to do this is to use multiple conditions in the WHERE clause and use different operators:

```
SELECT COUNT(*)
FROM employee
WHERE last_updated_date >= 20200101
AND last_updated_date < 20200201;
```

You can use combinations of less than, greater than, and equals to get the match you need.

Use IN instead of multiple OR

Why? It simplifies the query and improves readability.

SQL allows you to have multiple OR keywords in a WHERE clause. This can be helpful when looking to match one of a set of values:

```
SELECT
id,
first_name,
last_name
WHERE dept_id = 1
OR dept_id = 3
OR dept_id = 6;
```

A better way to write this query is to use the IN clause. You can specify each of the values to check, and the query will return records that match any of those values.

```
SELECT
```

```
id,
first_name,
last_name
WHERE dept_id IN (1, 3, 6);
```

It's the same as an OR keyword, but shorter and easier to read and modify.

Use CASE instead of other functions such as DECODE

Why? It's more flexible and easier to read.

The CASE statement is a very useful feature of SQL. It lets you perform conditional logic, or IF THEN ELSE functionality, in your query.

Some functions exist in some databases that let you do this in other ways (e.g. IIF in SQL Server, DECODE in Oracle).

However, I believe CASE is better because it's more flexible and is easier to read.

So, aim to use the CASE statement instead of other functions.

Avoid using functions in WHERE clauses if possible

Why? It can cause indexes not to be used, slowing down your query.

A common way to improve the performance of a query is to add an index to any columns in the WHERE clause.

However, if the column has a function applied to it, then the index is not used by the database. This is because the query is using the function on the value and not the value itself.

For example, let's say you had this query, and there was an index on the start_date column:

```
SELECT
id,
first_name,
last_name
WHERE start_date >= 20200101;
```

The index would likely be used. However, if you have this query, the index may not be used:

```
SELECT
id,
first_name,
last_name
WHERE YEAR(start_date) = 2020;
```

It might be more obvious what the query is doing, but it may run slower, as the value of YEAR(start_date) is not indexed.

How can you resolve this?

Either remove the function from the start_date column, or add a function-based index to the column.

Write separate SQL queries for separate tasks

Why? It will likely perform better.

It may be tempting to write a single query to get all of the pieces of data you need. The theory is that you only need to hit the database once, so you should bring back as much as you can.

However, you shouldn't be worried about writing separate queries. Often the performance of the database operation is in the complexity of the query itself, not the number of times you run queries.

Consider writing separate queries for different pieces of data, rather than one query to get everything.

For example, instead of writing a query to get all customer names and all customer types and addresses, consider writing separate queries for them. It may be quicker and easier to do.

Use NOT EXISTS instead of NOT IN

Why? It may perform better and avoid data issues.

There are two common ways for checking that a value for a record is not in a list of other values:

- NOT EXISTS
- NOT IN

The differences are in the way they are processed and how they handle NULL values.

The NOT IN approach will return zero rows if there are any NULL values in the list of data, but the NOT EXISTS approach will still consider other values. This is an issue if you can't be sure you don't have null values.

Also, the NOT EXISTS approach may perform better than NOT IN due to how the records are evaluated. But this may depend on which database you're using.

I recommend using NOT EXISTS instead of NOT IN for these reasons.

Use CASE instead of multiple SELECT with UNION

Why? Better performance, shorter query

Sometimes you may have a query that needs to get data from the same set of tables but has two different conditions.

One way to write this query is to query the tables twice, and use a UNION to join them together.

```
SELECT
columns
FROM tables
WHERE first_condition
UNION
SELECT
columns
FROM tables
WHERE second_condition;
```

This means your overall query is looking at the same set of tables twice, which is like doing double the work.

An alternative to this is to only query from the tables once, and use a CASE statement to show the right data.

```
SELECT
columns
FROM tables
WHERE CASE
     WHEN first_condition THEN 1
     WHEN second_condition THEN 1 ELSE 0
     END = 1
FROM tables;
```

This would likely perform better and still give you the right data.

Use UNION ALL instead of UNION if possible

Why? It performs better.

There are two keywords that let you combine the results of queries: UNION and UNION ALL.

It's common to use UNION to combine the data and leave it at that.

However, UNION ALL is often better.

The difference is that UNION will combine data and eliminate duplicates, and UNION ALL will combine data and keep the duplicates. UNION will often perform worse as it needs to do this DISTINCT or duplicate elimination step.

So, if you know that your data already does not have duplicates, then use UNION ALL.

Avoid DISTINCT to fix duplicate data issues

Why? It slows down your query and doesn't solve the problem.

When you write a query that has many columns and tables, you may get duplicate rows.

You think your query is right, but you don't want duplicates, so you add a DISTINCT keyword.

This will give you the right data, but it actually slows down your query as you need to perform this duplicate removal step. It also doesn't eliminate the cause of the problem.

In many cases, adding a DISTINCT is masking another issue. This issue could be a missing WHERE clause or an incorrect JOIN in one of your tables.

Rather than adding a DISTINCT keyword, look into your query and data, find the issue, and update the query. You'll get the right data and avoid having to take the extra step of removing duplicates, which can speed up your query.

Avoid SELECT *

Why? It's volatile and unnecessary.

The SELECT * keyword allows you to select all columns from a table. This is helpful for quick queries you're running during development to see what's in a table.

However, for anything that goes into your application or other code you use, avoid using SELECT *.

Using SELECT * means the actual columns that are returned are not guaranteed. If you add or remove a column, your column list changes. The order of columns is not guaranteed either.

It's better to specify the exact columns you need. This makes it easier for the application to work with, and easier for other developers.

Avoid implicit conversions

Why? It can slow down your query.

When you write a WHERE clause, you usually specify a column and a value to compare it to.

```
SELECT id
FROM employee
WHERE start_date = 20200101;
```

In this example, a start date is compared to a specific date. The start date is of type "date" and the value of 20200101 is of type "number". The database does an implicit conversion to convert the number to a date, to then return the correct data.

However, this implicit conversion is one extra step the database has to do, which can slow down the query a little.

A better way to do this is to avoid implicit conversions. Specify the data you want to compare in the same data type. This may involve some kind of conversion function:

```
SELECT id
FROM employee
WHERE start_date = TO_DATE(20200101, 'YYYYMMDD');
```

The result will be the same but the performance may be slightly better.

Don't overuse triggers

Why? They can complicate your tables and database.

Triggers are a helpful concept in databases. A trigger is a database object that you can create and attach to a table, for example, which then runs the code you specify on certain events.

A popular use of triggers is for audit tables: if a value in a specific table changes, copy the old row to another table, you can see the history of the row.

However, while triggers are useful, it can be tempting to use them for a lot of things, and even design your system around them.

If you overuse triggers, it can make your system more complicated. You'll start to lose track of how data is updated and can cause other issues when triggers update other tables.

If you want data to be updated in certain situations, consider using application code or stored procedures. They are much simpler and easier to work with.

Use bind variables

Why? Prevent SQL injection and improve performance.

Bind variables are a feature of databases that let you indicate that a value needs to be substituted at a specific place. They are commonly used in WHERE clauses, when you have the same query to run but with different criteria.

Here's a query without bind variables:

```
SELECT
id,
first_name,
last_name
FROM employee
WHERE dept_id = 3;
```

Here's the same query with bind variables:

```
SELECT
id,
first_name,
last_name
```

```
FROM employee
WHERE dept_id = :input_dept_id;
```

The input_dept_id is a bind variable. It's a parameter that can be provided by the application or procedure that runs this query. This helps in two ways.

First, it can help prevent SQL injection. Using bind variables instead of specific values can help prevent invalid characters in your query and issues in the database.

Second, it can help with the performance of your query. Many databases generate new execution plans for queries that are different, even if the only difference is a different value in the WHERE clause. Using bind variables means the execution plan stays the same, and the database has one less thing to do.

Use prepared statements to avoid SQL injection

Why? To improve security.

SQL injection is the act of transforming a valid SQL query into one that can cause damage by adding characters into your query string. It's done by tricking your database to show data that is not otherwise allowed.

This is often done by supplying SQL characters and keywords in a text input box or URL, but can be done in other ways.

A good way to avoid this is to use a concept called prepared statements. This allows the application code to construct the SQL statement before sending it to the database, and is a big help against SQL injection attacks.

For more information on SQL injection, check out this guide.

Only use HAVING with aggregate functions

Why? To improve readability.

The HAVING clause is designed to filter values that are calculated from aggregate functions, such as SUM or COUNT.

It's similar to the WHERE clause. However, the WHERE clause operates on data before the aggregation.

You can use the HAVING clause on data before the aggregation is performed. But this makes the query harder to read. I recommend sticking with the intent of the HAVING clause and only use it for columns that are being aggregated.

Avoid using column numbers in ORDER BY clause

Why? It's prone to errors and harder to read.

The ORDER BY clause allows you to specify the order of your results. You specify the columns you want to order by, and the results will be ordered.

```
SELECT
id,
first_name,
last_name
FROM employee
ORDER BY last_name ASC;
```

Another way to specify the ordering is to specify the number of the column, where the number represents the position of the column in the SELECT clause.

```
SELECT
id,
first_name,
last_name
FROM employee
ORDER BY 3 ASC;
```

This query will order by column 3 in the SELECT clause, which is last_name.

However, this is prone to issues in your results. If you add or remove a column from the SELECT clause, it means the ordering is changed, as column 3 is no longer in position 3. This issue may not be obvious as your query will still run, but will be in a different order.

Also, looking at the ORDER BY clause it's harder to see which column is being ordered. You'll need to go back to the SELECT clause to understand it.

If you order by the column names then you avoid both of these issues.

Only use ORDER BY if needed

Why? To improve performance.

The ORDER BY clause lets you order the results of a SELECT query.

It can be useful to show data in a certain order. However, ordering data is an expensive step, meaning it can really slow down your query.

Do you really need to have data in a certain order? If not, consider not using an ORDER BY.

If you do need to order data, there are ways you can improve the performance still.

Always specify column names in INSERT statements

Why? To reduce errors when inserting data.

When you want to insert data into a table, you specify the table, column names, and values:

```
INSERT INTO employee (id, first_name, last_name)
VALUES (5, 'John', 'Smith');
```

However, the column names are not a required part of the INSERT statement. This query will still work:

```
INSERT INTO employee
VALUES (5, 'John', 'Smith');
```

The values are inserted into the columns in the order they are defined in the data dictionary.

However, this order is not guaranteed. The order may change if new columns are added or in other instances.

Sometimes you'll get an error when you insert data and the columns no longer match. Other times you won't get an error, because the data meets the new column requirements, but it's in the wrong columns.

To avoid this, always specify the column names in your INSERT statement.

Use join keywords instead of WHERE clause

Why? To improve readability and maintenance.

When you join two tables together, you specify a column that they should be joined on.

In SQL, there are two ways to do this. The first way is to use the JOIN keyword (such as INNER JOIN or LEFT JOIN).

```
SELECT
first_name,
last_name,
department_name
FROM employee
INNER JOIN department ON employee.dept_id = department.id;
```

The second way is to use a WHERE clause:

```
SELECT
first_name,
last_name,
department_name
FROM employee,
department
WHERE employee.dept_id = department.id;
```

These queries have the same effect. They are both querying two tables and showing only records where the two columns match.

However, I recommend using the JOIN keyword to join tables instead of the WHERE clause.

The JOIN keyword is designed for joining tables, so it should be used how it's designed. The WHERE clause is designed for filtering data after it has been joined.

With the JOIN keyword, you avoid potential issues of missing a join. The syntax requires you to specify a table then a join criteria.

With the WHERE clause, you can specify all tables then all join criteria, which means you can leave out join criteria and get an incorrect result.

You can also access more functionality with the JOIN keywords than the WHERE clause. You can use outer joins and other join types, which can be done in Oracle with specific syntax (the (+) symbols) but not other databases.

So, I recommend using the JOIN keywords when joining to improve readability, use the features as they are designed, and to reduce the chance of errors in your query.

Don't use the join clause for filtering on conditions

Why? To improve readability and reduce errors.

Similar to the above point, you can filter your data in two ways:

- Use a WHERE clause
- Use a JOIN clause

You can filter data using a WHERE clause like this:

```
SELECT
first_name,
last_name,
department_name
FROM employee
INNER JOIN department
     ON employee.dept_id = department.id
WHERE active_status = 2;
```

Or you can filter it using a JOIN condition like this:

```
SELECT
first_name,
last_name,
department_name
FROM employee
INNER JOIN department
     ON employee.dept_id = department.id
     AND active_status = 2;
```

The issue with the second query is you're combining table joins with table filtering. The active_status check has nothing to do with joining the department table. It should go in a WHERE clause to filter the results after it has been joined.

So, if you need to filter data based on a column in a table, use the WHERE clause and not a join.

Avoid wildcards where possible

Why? To improve performance.

Wildcards are a helpful feature in SQL. They let you find partial matches of strings.

For example, to find all support departments:

```
SELECT
id,
department_name
FROM department
WHERE department_name LIKE '%support%';
```

This will query your table to find the data you need.

However, this query can be quite slow. The database will need to analyse every value in the column against the provided string and see if it matches.

A better way of doing this is to avoid wildcard values. Consider if you can get a definitive list of values and filter on those. Or create a separate table with a type indicator, or whatever works for your query.

For example:

```
SELECT
id,
department_name
FROM department
WHERE department_name IN (
      'Production Support',
      'Customer Support'
);
```

This query would get the same results, and avoids the performance issues of wildcards.

Conclusion

I hope this SQL style guide and best practices has helped you write better SQL.

As mentioned earlier, many of these are strong recommendations and others are personal preferences. Do you agree or disagree with these points? Do you have any suggestions? Leave them in the comments on the blog post: https://www.databasestar.com/sql-best-practices/