SQL tips and tricks

A (somewhat opinionated) list of SQL tips and tricks that I've picked up over the years.

There's so much you can you do with SQL but I've focused on what I find most useful in my day-to-day work as a data analyst and what I wish I had known when I first started writing SQL.

Please note that some of these tips might not be relevant for all RDBMs.

Table of contents

Formatting/readability

Useful features

Avoid pitfalls

Formatting/readability

Use a leading comma to separate fields

Use a leading comma to separate fields in the `SELECT` clause rather than a trailing comma.

- Clearly defines that this is a new column vs code that's wrapped to multiple lines.
- Visual cue to easily identify if the comma is missing or not. Varying line lengths makes it harder to determine.

```
SELECT
employee_id
, employee_name
, job
, salary
FROM employees
;
```

- Also use a leading `AND` in the `WHERE` clause, for the same reasons (following tip demonstrates this).

Use a dummy value in the WHERE clause

Use a dummy value in the `WHERE` clause so you can easily comment out conditions when testing or tweaking a query.

```
-- If I want to comment out the job condition the following query will break.
SELECT *
FROM employees
WHERE
--job IN ('Clerk', 'Manager')
AND dept_no != 5
;

-- With a dummy value there's no issue. I can comment out all the conditions and 1=1 will ensu
SELECT *
FROM employees
WHERE 1=1
-- AND job IN ('Clerk', 'Manager')
AND dept_no != 5
;
```

Indent your code

Indent your code to make it more readable to colleagues and your future self.

Opinions will vary on what this looks like so be sure to follow your company/team's guidelines or, if that doesn't exist, go with whatever works for you.

You can also use an online formatter like poorsql or a linter like sqlfluff.

```
SELECT
-- Bad:
vc.video_id
, CASE WHEN meta.GENRE IN ('Drama', 'Comedy') THEN 'Entertainment' ELSE meta.GENRE END as cont
FROM video_content AS vc
INNER JOIN metadata ON vc.video_id = metadata.video_id
;

-- Good:
SELECT
vc.video_id
, CASE
    WHEN meta.GENRE IN ('Drama', 'Comedy') THEN 'Entertainment'
    ELSE meta.GENRE
END AS content_type
FROM video_content
INNER JOIN metadata
    ON video_content.video_id = metadata.video_id
;
```

Consider CTEs when writing complex queries

For longer than I'd care to admit I would nest inline views, which would lead to queries that were hard to understand, particularly if revisited after a few weeks.

If you find yourself nesting inline views more than 2 or 3 levels deep, consider using common table expressions, which can help you keep your code more organised and readable.

```
/*
The following query doesn't actually need to use an inline view or CTE but I'm just
demonstrating the difference between the two.
*/

-- Using nested inline views.
SELECT
vhs.movie
```

```
, vhs.vhs_revenue
, cs.cinema_revenue
FROM
    (
    SELECT
    movie_id
    , SUM(ticket_sales) AS cinema_revenue
    FROM tickets
    GROUP BY movie_id
    ) AS cs
    INNER JOIN
        (
        SELECT
        movie
        , movie_id
        , SUM(revenue) AS vhs_revenue
        FROM blockbuster
        GROUP BY movie, movie_id
        ) AS vhs
        ON cs.movie_id = vhs.movie_id
;

-- Using CTEs.
WITH cinema_sales AS
    (
        SELECT
        movie_id
        , SUM(ticket_sales) AS cinema_revenue
        FROM tickets
        GROUP BY movie_id
    ),
    vhs_sales AS
    (
        SELECT
        movie
        , movie_id
        , SUM(revenue) AS vhs_revenue
        FROM blockbuster
        GROUP BY movie, movie_id
    )
SELECT
vhs.movie
, vhs.vhs_revenue
, cs.cinema_revenue
FROM cinema_sales AS cs
    INNER JOIN vhs_sales AS vhs
    ON cs.movie_id = vhs.movie_id
;
```

Useful features

Anti-joins will return rows not present in another table

Anti-joins are incredible useful, mostly (in my experience) for when you only want to return rows/ values from one table that aren't present in another table.

- You could instead use a subquery although you might want to experiment as to which method is faster.

```
-- Anti-join.
SELECT
video_content.*
FROM video_content
    LEFT JOIN archive
    on video_content.series_id = archive.series_id
WHERE 1=1
AND archive.series_id IS NULL -- Any rows with no match will have a NULL value.
;

-- Subquery.
SELECT
*
FROM video_content
WHERE 1=1
AND series_id NOT IN (SELECT DISTINCT SERIES_ID FROM archive) -- Be mindful of NULL values.
;

-- Correlated subquery.
SELECT
*
FROM video_content vc
WHERE 1=1
AND NOT EXISTS (
        SELECT 1
        FROM archive a
        WHERE a.series_id = vc.series_id
    )
;
```

Use `QUALIFY` to filter window functions

`QUALIFY` lets you filter the results of a query based on a window function. This is useful for a variety

of reasons, including to reduce the number of lines of code needed.

For example, if I want to return the top 10 markets per product I can use `QUALIFY` rather than an inline view:

```
-- Using QUALIFY:
SELECT
product
, market
, SUM(revenue) AS market_revenue
FROM sales
GROUP BY product, market
QUALIFY DENSE_RANK() OVER (PARTITION BY product ORDER BY SUM(revenue) DESC)  <= 10
ORDER BY product, market_revenue
;

-- Without QUALIFY:
SELECT
product
, market
, market_revenue
FROM
(
SELECT
product
, market
, SUM(revenue) AS market_revenue
, DENSE_RANK() OVER (PARTITION BY product ORDER BY SUM(revenue) DESC) AS market_rank
FROM sales
GROUP BY product, market
)
WHERE market_rank  <= 10
ORDER BY product, market_revenue
;
```

You can (but shouldn't always) `GROUP BY` column position

Instead of using the column name, you can `GROUP BY` or `ORDER BY` using the column position.

- This can be useful for ad-hoc/one-off queries, but for production code you should always refer to a column by its name.

```
SELECT
dept_no
, SUM(salary) AS dept_salary
```

```
FROM employees
GROUP BY 1 -- dept_no is the first column in the SELECT clause.
ORDER BY 2 DESC
;
```

You can create a grand total with `GROUP BY ROLLUP`

Creating a grand total (or sub-totals) is possible thanks to `GROUP BY ROLLUP`.

For example, if you've aggregated a company's employees salary per department you can use `GROUP BY ROLLUP` to create a grand total that sums up the aggregated `dept_salary` column.

```
SELECT
COALESCE(dept_no, 'Total') AS dept_no
, SUM(salary) AS dept_salary
FROM employees
GROUP BY ROLLUP(dept_no)
ORDER BY dept_salary -- Be sure to order by this column to ensure the Total appears last/at th
;
```

Use `EXCEPT` to find the difference between two datasets

`EXCEPT` returns rows from the first query's result set that don't appear in the second query's result set.

```
-- Miles Davis will be returned from this query.
SELECT Name
FROM artist
WHERE name = 'Miles Davis'
EXCEPT
SELECT Name
FROM artist
WHERE name = 'Nirvana'
;

-- Nothing will be returned from this query as 'Miles Davis' appears in both queries' result s
SELECT Name
FROM artist
WHERE name = 'Miles Davis'
EXCEPT
SELECT Name
FROM artist
WHERE name = 'Miles Davis'
;
```

Common pitfalls

Be aware of how `NOT IN` behaves with `NULL` values

`NOT IN` doesn't work if `NULL` is present in the values being checked against. As `NULL` represents Unknown the SQL engine can't verify that the value being checked is not present in the list.

- Instead use `NOT EXISTS`.

```
INSERT INTO departments (id)
VALUES (1), (2), (NULL);

-- Doesn't work due to NULL being present.
SELECT *
FROM employees
WHERE department_id NOT IN (SELECT DISTINCT id from departments)
;

-- Solution.
SELECT *
FROM employees e
WHERE NOT EXISTS (
    SELECT 1
    FROM departments d
```

```
    WHERE d.id = e.department_id
)
;
```

Avoid ambiguity when renaming calculated fields

When creating a calculated field, you might be tempted to rename it to an existing column, but this can lead to unexpected behaviour, such as a window function operating on the wrong field.

```
CREATE TABLE products (
    product VARCHAR(50) NOT NULL,
    revenue INT NOT NULL
)
;

INSERT INTO products (product, revenue)
VALUES
    ('Shark', 100),
    ('Robot', 150),
    ('Alien', 90);

-- The window function will rank the 'Robot' product as 1 when it should be 3.
SELECT
product
, CASE product WHEN 'Robot' THEN 0 ELSE revenue END AS revenue
, RANK() OVER (ORDER BY revenue DESC)
FROM products
;

-- You can instead do this:
SELECT
product
, CASE product WHEN 'Robot' THEN 0 ELSE revenue END AS revenue
, RANK() OVER (ORDER BY CASE product WHEN 'Robot' THEN 0 ELSE revenue END DESC)
FROM products
;
```

Always specify which column belongs to which table

When you have complex queries with multiple joins, it pays to be able to trace back an issue with a value to its source.

Additionally, your RDBMS might raise an error if two tables share the same column name and you

don't specify which column you are using.

```
SELECT
vc.video_id
, vc.series_name
, metadata.season
, metadata.episode_number
FROM video_content AS vc
    INNER JOIN video_metadata AS metadata
    ON vc.video_id = metadata.video_id
;
```

Understand the order of execution

If I had to give one piece of advice to someone learning SQL, it'd be to understand the order of execution (of clauses). It will completely change how you write queries. This blog post is a fantastic resource for learning.

Comment your code!

While in the moment you know why you did something, if you revisit the code weeks, months or years later you might not remember.

- In general you should strive to write comments that explain why you did something, not how.
- Your colleagues and future self will thank you!

```
SELECT
video_content.*
FROM video_content
    LEFT JOIN archive -- New CMS cannot process archive video formats.
    ON video_content.series_id = archive.series_id
WHERE 1=1
AND archive.series_id IS NULL
;
```

Read the documentation (in full)

Using Snowflake I once needed to return the latest date from a list of columns and so I decided to use `GREATEST()`.

What I didn't realise was that if one of the arguments is `NULL` then the function returns `NULL`.

If I'd read the documentation in full I'd have known! In many cases it can take just a minute or less to scan the documentation and it will save you the headache of having to work out why something isn't working the way you expected:

```
-- If I'd read the documentation further I'd also have realised that my solution
--to the NULL problem with GREATEST()...

SELECT COALESCE(GREATEST(signup_date, consumption_date), signup_date, consumption_date);

-- ... could have been solved with the following function:
SELECT GREATEST_IGNORE_NULLS(signup_date, consumption_date);
```

Use descriptive names for your saved queries

There's almost nothing worse than not being able to find a query you need to re-run/refer back to.

Use a descriptive name when saving your queries so you can easily find what you're looking for.

I usually will write the subject of the query, the month the query was ran and the name of the requester (if they exist). For example: `Lapsed users analysis - 2023-09-01 - Olivia Roberts`