1. WHERE 1=1

When you code has different WHERE conditions, having a WHERE 1=1 simplifies the logic.

This is because 1=1 is always true and doesn't affect the query's actual result, but it allows you to more easily append additional conditions **without needing special handling for the first condition.**

It improves the overall readability of the code as well.

```
SELECT * FROM users WHERE 1=1
AND age > 25
AND city = 'New York'
AND gender = 'Female'
```

2. QUALIFY

It can be used to filter the results of a query based on the result of a window function.

You don’t need nested queries making the code a lot easier to read.

It is **similar to the HAVING**, but instead of filtering after an aggregation, QUALIFY filters after the application of window functions.

```
SELECT
    id,
    salesperson,
    amount,
    ROW_NUMBER() OVER (PARTITION BY salesperson ORDER BY amount
DESC) AS rank
FROM sales
QUALIFY rank = 1;
```

3. ROW_NUMBER

Incredibly useful when it comes to cleaning data.

ROW_NUMBER ( ) can be used to identifying and removing duplicates, and detecting gaps in data.

It can be used to select a SINGLE row based on conditions such as latest record, highest/lowest value etc.

```
WITH RankedOrders AS (
    SELECT
        customer_id,
        order_id,
        ROW_NUMBER() OVER (PARTITION BY customer_id ORDER BY order
_date DESC) AS rn
    FROM orders
)
SELECT *
FROM RankedOrders
WHERE rn = 1;
```

4. EXCLUDE <COL>

This not standard in SQL but is a feature found in some SQL dialects, such as BigQuery.

It allows you to easily select all columns from a table except one or more specified columns.

This improves readability and reduces repetitive code.

```sql
SELECT col1, col2, col3, col5 -- manually
excluding col4
FROM table;
```

```sql
SELECT * EXCLUDE col4
FROM table;
```

5. EXISTS

Employed when you want to check for the existence of records in a related table or subquery. Helps when you do your EDA.

It returns a TRUE if the subquery returns at least one row.

It is **more performant** than using IN or JOIN.

```sql
SELECT department_name
FROM departments d
WHERE EXISTS (
   SELECT 1
   FROM employees e
   WHERE e.department_id = d.department_id
);
```

6. COALESCE

This function handles NULL values gracefully.

COALESCE allows you to provide a fallback value when encountering NULL.

Rather than using a complex CASE statement or multiple IFNULL/ISNULL functions, COALESCE provides a cleaner syntax.

You can use COALESCE to choose the first non-NULL value across multiple columns.

```
SELECT COALESCE(home_phone, mobile_phone, office_phone)
AS contact_number
FROM contacts;
```

7. TEMP TABLES

Temp tables allow you to break the query into smaller, more manageable parts.

Trying to fit everything into a single, massive nested statement with multiple WITH statements can make your query too complex.

This way you can also avoid repeated calculations & re-use queries.

Also it helps you understand, debug, and optimize each part of the query independently.

8. SYSCAT / SYSINFO

Helps you obtain metadata on the underlying database platform that you are using.

Querying **syscat** or **sysinfo** to find out what schemas, tables, columns, etc are available.

For example, you can query SYS.COLUMNS to get details about all the columns in a particular table.

SYS.KEYS and SYS.CONSTRAINTS can be used to get info about primary keys, foreign keys, and other constraints applied to tables.

```
SELECT * FROM SYS.COLUMNS
WHERE TABLE_NAME = 'employees';
```

9. LAG / LEAD

Extremely useful for performing operations that require accessing data from previous or subsequent rows.

If you are building a KPI dashboard and want to calculate **month-over-month** or **year-over-year**, then this syntax makes the calculation a lot easier.

```
SELECT
    Month,
    Product,
    Sales,
    LAG(Sales, 1) OVER (PARTITION BY Product ORDER BY Month)
AS PreviousMonthSales,
    Sales - LAG(Sales, 1) OVER (PARTITION BY Product ORDER BY
Month) AS SalesDifference
FROM Sales;
```