Common Table Expressions (CTEs)

CTE stand for Common table Expression. A CTE is mostly a style of SQL query that simplifies using subqueries.

The idea is to name the temporary set of results for a subquery to be able to use that set later in the query.

They are essentially named subqueries aka **auxiliary statements** that can be referenced multiple times within a main query, making complex queries easier to read and manage.

CTEs are useful in scenarios where you want to:

- Break down complex queries into simpler, more manageable parts.
- Refer to a subquery multiple times within a single query.
- Improve query performance by structuring subqueries in a readable way.

and even:

- Handle recursive queries for hierarchical data (such as organizational charts or family trees).

A useful property of WITH queries is that they are normally evaluated only once per execution of the parent query, even if they are referred to more than once by the parent query or sibling WITH queries.

Same output but much cleaner code

- It's very DRY: Don't Repeat Yourself
- makes the final select easy to understand

Advantage of with clause:

- more readable, easier debug
- Avoiding no temporary tables: the temp subset is loaded into memory and not saved to disk. faster.

The overall structure basically follows this query structure:

```sql
WITH subquery_name as (
    Some query
)
-- add other  WITH expressions as needed
-- ...
-- then select, update, insert, etc  from the named subqueries
SELECT *
FROM subquery_name
WHERE ... etc ...
```

Load the data

Let's reload the `WorldHits.csv` data into a newly created `worldhitsdb` database with the tracks table.

Example

Let's start with a simple example where we want to count the number of tracks by artists.

The following query comes to mind

```sql
select artist, count(*) as track_count from tracks group by artist orde
```

But we can also extract the counting query into its own set of result and select from it.

```sql
WITH artist_track_count AS (
    SELECT artist, COUNT(*) as track_count
    FROM tracks
    GROUP BY artist
)
SELECT artist, track_count
FROM artist_track_count
ORDER BY track_count DESC
LIMIT 5;
```

This will output the same result.

The advantage of using the CTE expression in this simple case is not that obvious.

More complex CTE

Imagine :

> You're a music producer. A data driven music producer. You need to understand the music trends so you can decide which artists you want to work with and what types of songs you should produce.
>
> You're curious about how the energy and danceability of hit songs have changed over the years. This could help you understand if there's a trend towards more upbeat, danceable music or if the market is shifting towards more mellow tracks.

Let's take a look at danceability and energy of tracks over the years

```sql
SELECT
    year,
    AVG(energy) AS avg_energy,
    AVG(danceability) AS avg_danceability
FROM tracks
GROUP BY year;
```

You want to reuse these results to see how they evolve year after year. This is a job for the `LAG()` function!

So let's name the above query as `yearly_stats` :

```sql
-- name the data gathering query as: yearly_stats
WITH yearly_stats AS (
    SELECT
        year,
        AVG(energy) AS avg_energy,
        AVG(danceability) AS avg_danceability
    FROM tracks
    GROUP BY year
)
SELECT
    year,
    avg_energy,
    avg_danceability,
    avg_energy - LAG(avg_energy) OVER (ORDER BY year) AS energy_change,
    avg_danceability - LAG(avg_danceability) OVER (ORDER BY year) AS dar
-- reuse the yearly_stats results from the named query
FROM yearly_stats
ORDER BY year DESC
LIMIT 10;
```

Music style versatility per artist

> Next, you want to identify artists who are versatile in their musical style, as they might be open to suggestions and easy to work with. You create a **versatility score** based on the variance (`STDDEV()`) in some of their tracks' features.

This versatility score is defined as the sum of (energy*std + danceability*std + acousticness*std + instrumentalness*std)

Let's first get the raw data, the stats.

We only keep the artists with more than 4 tracks so that the standard deviation makes sense.

```sql
-- get the main sets of results
SELECT
    artist,
    COUNT(*) AS track_count,
    ROUND(CAST(STDDEV(energy) AS numeric), 2) AS energy_std,
    ROUND(CAST(STDDEV(danceability) AS numeric), 2) AS danceability_std
    ROUND(CAST(STDDEV(acousticness) AS numeric), 2) AS acousticness_std
    ROUND(CAST(STDDEV(instrumentalness) AS numeric), 2) AS instrumental
FROM tracks
GROUP BY artist
HAVING COUNT(*) >= 4;
```

Then calculate the versatility_score by resuing the above query as a named subquery

```sql
SQL
WITH artist_stats AS (
    SELECT
        artist,
        COUNT(*) AS track_count,
        ROUND(CAST(STDDEV(energy) AS numeric), 2) AS energy_std,
        ROUND(CAST(STDDEV(danceability) AS numeric), 2) AS danceability_
        ROUND(CAST(STDDEV(acousticness) AS numeric), 2) AS acousticness_
        ROUND(CAST(STDDEV(instrumentalness) AS numeric), 2) AS instrumer
    FROM tracks
    GROUP BY artist
    HAVING COUNT(*) >= 4
)
SELECT
    artist,
    track_count,
    ROUND(
        CAST( (energy_std + danceability_std + acousticness_std + instru
FROM artist_stats;
```

And finally we order by the `versatility_score` by re-using that last query as a named query `artist_versatility`

The final total query is

SQL

```sql
WITH artist_stats AS (
    SELECT
        artist,
        COUNT(*) AS track_count,
        ROUND(CAST(STDDEV(energy) AS numeric), 2) AS energy_std,
        ROUND(CAST(STDDEV(danceability) AS numeric), 2) AS danceability_
        ROUND(CAST(STDDEV(acousticness) AS numeric), 2) AS acousticness_
        ROUND(CAST(STDDEV(instrumentalness) AS numeric), 2) AS instrumer
    FROM tracks
    GROUP BY artist
    HAVING COUNT(*) >= 5
),
artist_versatility AS (
    SELECT
        artist,
        track_count,
        ROUND(CAST( (energy_std + danceability_std + acousticness_std +
    FROM artist_stats
)
SELECT
    artist,
    track_count,
    versatility_score
FROM artist_versatility
ORDER BY versatility_score DESC
LIMIT 10;
```

Imagine what you can do now with CTEs and windows functions !!!

What does this query do ?

SQL

```sql
WITH artist_yearly_stats AS (
    SELECT
        artist,
        year,
        AVG(popularity) AS avg_popularity,
        COUNT(*) AS track_count
    FROM tracks
    -- WHERE year >= 2010
    GROUP BY artist, year
),
artist_growth AS (
    SELECT
        artist,
        SUM(track_count) AS total_tracks,
        MIN(avg_popularity) AS min_popularity,
        MAX(avg_popularity) AS max_popularity,
        MAX(avg_popularity) - MIN(avg_popularity) AS popularity_growth
    FROM artist_yearly_stats
    GROUP BY artist
    HAVING SUM(track_count) >= 3 AND COUNT(DISTINCT year) >= 2
)
SELECT
    artist,
    total_tracks,
    min_popularity,
    max_popularity,
    popularity_growth
FROM artist_growth
-- WHERE max_popularity > 60
ORDER BY popularity_growth DESC
LIMIT 10;
```

More WITH ... AS

We've seen the simple WITH ... AS clause.

> A useful property of WITH queries is that they are normally evaluated only once per execution of the parent query, even if they are referred to more than once by the parent query or sibling WITH queries. Thus, *expensive calculations* that are needed in multiple places can be placed within a WITH query to avoid redundant work.

The downside is that

> The multiply-referenced WITH query will be evaluated as written, without suppression of rows that the parent query might discard afterwards.

The query optimizer (next week) will sometimes choose to store the results of the WITH query in a temporary table, ... or not.

There are cases where we want to enforce or avoid storing the results of the WITH query into a temp table. We can do that ba adding `MATERIALIZED` or `NOT MATERIALIZED` to the WITH clause:

SQL
```sql
WITH w AS (NOT) MATERIALIZED (
    SELECT * FROM big_table
)
SELECT * FROM w AS w1 JOIN w AS w2 ON w1.key = w2.ref
WHERE w2.key = 123;
```

Recursive

Using `RECURSIVE` , a WITH query can refer to its own output. A simple example is this query to sum the integers from 1 through 100:

SQL
```sql
WITH RECURSIVE t(n) AS (
    VALUES (1)
  UNION ALL
    SELECT n+1 FROM t WHERE n < 100
)
SELECT sum(n) FROM t;
```

More on recursive WITH queries in https://www.postgresql.org/docs/current/queries-with.html#QUERIES-WITH-RECURSIVE

Further readings

- Documentation : https://www.postgresql.org/docs/current/queries-with.html
- youtube techFTQ channel : https://www.youtube.com/watch?v=QNfnuK-1YYY