BECOMING A SQL GURU

Stella Nisenbaum
Stella.Nisenbaum@yello.co

YELLO'S MISSION STATEMENT

> Eliminating gaps in the hiring experience

yello

WHAT MAKES YELLO UNIQUE

CLIENT FIRST CULTURE

Yello is proud to partner with clients ranging from Fortune 500 global enterprises to high-growth early-stage companies

AWARD-WINNING

Yello's Scheduling Solution was named Top HR product of 2015 by Human Resources Executive Magazine.

MARKET EXPERTISE

Yello's leadership team is comprised of many former corporate recruiting and HR technology leaders.

yello

AGENDA

- Syntax Overview
- Join Types
- Set Operators
- Filtered Aggregates
- Grouping Sets, Cube, and Rollup
- Subqueries
- Window Functions
- Common Table Expressions (CTE’s)
- Lateral Join
- Questions

QUERIES – SYNTAX OVERVIEW

When we think of Standard SQL Syntax…

SELECT ***expression***
FROM ***table***
WHERE ***condition***
ORDER BY ***expression***

QUERIES – SYNTAX OVERVIEW

Or maybe we think…

```
SELECT expression
FROM table
[JOIN TYPE] table2
ON join_condition
WHERE condition
ORDER BY expression
```

QUERIES – SYNTAX OVERVIEW

Then we think…

```
SELECT expression
FROM table
JOIN_TYPE table2
ON join_condition
WHERE condition
GROUP BY expression
HAVING condition
ORDER BY expression
```

QUERIES – SYNTAX OVERVIEW

But really …

```
[ WITH [ RECURSIVE ] with_query [, …] ]
SELECT [ ALL | DISTINCT [ ON ( expression [, …] ) ] ]
   [ * | expression [ [ AS ] output_name ] [, …] ]
   [ FROM from_item [, …] ]
   [ WHERE condition ]
   [ GROUP BY expression [, …] ]
   [ HAVING condition [, …] ]
   [ WINDOW window_name AS ( window_definition ) [, …] ]
   [ { UNION | INTERSECT | EXCEPT } [ ALL | DISTINCT ] select ]
   [ ORDER BY expression [ ASC | DESC | USING operator ] [ NULLS { FIRST | LAST } ] [, …] ]
   [ LIMIT { count | ALL } ]
   [ OFFSET start [ ROW | ROWS ] ]
   [ FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } ONLY ]
   [ FOR { UPDATE | NO KEY UPDATE | SHARE | KEY SHARE } [ OF table_name [, …] ] [ NOWAIT ] […] ]
```

QUERIES – SYNTAX OVERVIEW

where from_item can be one of:

```
[ ONLY ] table_name [ * ] [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
[ LATERAL ] ( select ) [ AS ] alias [ ( column_alias [, ...] ) ]
with_query_name [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
[ LATERAL ] function_name ( [ argument [, ...] ] )
          [ WITH ORDINALITY ] [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
[ LATERAL ] function_name ( [ argument [, ...] ] ) [ AS ] alias ( column_definition [, ...] )
[ LATERAL ] function_name ( [ argument [, ...] ] ) AS ( column_definition [, ...] )
[ LATERAL ] ROWS FROM( function_name ( [ argument [, ...] ] ) [ AS ( column_definition [, ...] ) ] [, ...] )
          [ WITH ORDINALITY ] [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
from_item [ NATURAL ] join_type from_item [ ON join_condition | USING ( join_column [, ...] ) ]
```

QUERIES – SYNTAX OVERVIEW

and **grouping_element** can be one of:

( )

expression

(**expression** [, ...])

ROLLUP ({ expression | (expression [, ...]) } [, ...])

CUBE ({ expression | (expression [, ...]) } [, ...])

GROUPING SETS (grouping_element [, ...])

and **with_query** is:

with_query_name [(column_name [, ...])] AS **(select | values | insert | update | delete)**

TABLE [ONLY] **table_name** [*]

QUERIES – BASIC EXAMPLES

```
VALUES (1, 'one'), (2, 'two'), (3, 'three');
```

Column1	Column2
1	one
2	two
3	three

```
TABLE customers;
```

Is equivalent to:

```
SELECT * FROM customers;
```

JOIN TYPES

Inner Join:

Joins each row of the first table with each row from the second table for which the condition matches. Unmatched rows are removed

Outer Join:

Joins each row from the one table with each row from the second table for which the condition matches. Unmatched rows are added to the result set such that:

- Left: All rows from the left table are returned, with null values displayed for the right table
- Right: All rows from the right table are returned, with null values displayed for the left table
- Full: All rows from both tables are returned, with null values displayed for unmatched rows in each table.

Cross Join:

Creates a Cartesian Product of two tables

CROSS JOINS: EXAMPLE

stores

store_id	store_city
1	chicago
2	dallas

products

product_id	product_desc
1	coffee
2	tea

```
SELECT * FROM stores
CROSS JOIN products
```

```
SELECT * FROM stores, products
```

Results:

store_id	store_city	product_id	product_desc
1	chicago	1	coffee
1	chicago	2	tea
2	dallas	1	coffee
2	dallas	2	tea

SET OPERATIONS

customers

ID	customer_name	city	postal_code	country
1	Stella Nisenbaum	Chicago	60605	USA
2	Stephen Frost	New York	10012	USA
3	Luke Daniels	Stockholm	113 50	Sweden
4	Artem Okulik	Minsk	220002	Belarus

suppliers

ID	supplier_name	city	postal_code	country	revenue
1	Herpetoculture, LLC	Meriden	06451	USA	300,000,000
2	Bodega Privada	Madrid	28703	Spain	700,000,000
3	ExoTerra	Montreal	H9X OA2	Canada	400,000,000
4	Goose Island Beer, Co	Chicago	60612	USA	250,000,000

SET OPERATIONS: UNION VS UNION ALL

```
SELECT city FROM customers
UNION ALL
SELECT city FROM suppliers
```

city
Chicago
New York
Stockholm
Minsk
Meriden
Madrid
Montreal
Chicago

```
SELECT city FROM customers
UNION
SELECT city FROM suppliers
```

city
Chicago
New York
Stockholm
Minsk
Meriden
Madrid
Montreal

SET OPERATIONS: EXCEPT VS INTERSECT

```
SELECT city FROM customers
EXCEPT
SELECT city FROM suppliers
```

city
New York
Stockholm
Minsk

```
SELECT city FROM customers
INTERSECT
SELECT city FROM suppliers
```

city
Chicago

FILTERED AGGREGATES (9.4)

Before:

```
SELECT
Sum(revenue) as total_revenue
, Sum(Case
        when country = ‘USA’
                then revenue
        else 0
        End) as USA_revenue
FROM suppliers s
```

Now:

```
SELECT
Sum(revenue) as total_revenue
, Sum(revenue) FILTER (where country = ‘USA’)  as USA_revenue
FROM suppliers s
```

GROUPING SETS, CUBE, ROLLUP(9.5)

Grouping Sets: Allows for the creation of sets wherein a subtotal is calculated for each set

Rollup: Allows for the creation of a hierarchical grouping/subtotals starting with the primary group, then the secondary and so on

Cube: Allows for the creation of subtotals for all possible groups (not only hierarchical)

GROUPING SETS, CUBE, ROLLUP(9.5)

orders

id	customer_id	supplier_id	order_date	order_amt
1	1	1	2016-01-15	100
2	1	3	2016-02-05	250
3	3	2	2016-01-25	85
4	3	4	2016-01-07	125
5	4	4	2016-02-19	65
6	4	1	2016-01-20	150
7	1	3	2016-02-17	300

GROUPING SETS, CUBE, ROLLUP(9.5)

```
SELECT
s.country
, s.supplier_name
, date_trunc('month', o.order_date)::date as order_month
, c.customer_name
, sum(o.order_amt) as sum_amt
, avg(o.order_amt)::int as avg_amt
, count(o.id) as ct
FROM orders o
JOIN customers c
        ON o.customer_id = c.id
JOIN suppliers s
        ON o.supplier_id = s.id
GROUP BY s.country , s.supplier_name ,date_trunc('month', o.order_date), c.customer_name
```

GROUPING SETS, CUBE, ROLLUP(9.5)

Results:

country	supplier_name	order_month	customer_name	sum_amt	avg_amt	ct
Canada	ExoTerra	2016-02-01	Stella Nisenbaum	550	275	2
Spain	Bodega Privada	2016-01-01	Luke Daniels	85	85	1
USA	Goose Island Beer, Co	2016-01-01	Luke Daniels	125	125	1
USA	Goose Island Beer, Co	2016-02-01	Artem Okulik	65	65	1
USA	Herpetoculture, LLC	2016-01-01	Artem Okulik	150	150	1
USA	Herpetoculture, LLC	2016-01-01	Stella Nisenbaum	100	100	1

GROUPING SETS(9.5)

```
SELECT
Case when grouping(supplier_name) = 0
        then s.supplier_name else 'All Suppliers' end as supplier_name
,Case when grouping( date_trunc('month', o.order_date)) = 0
        then date_trunc('month', o.order_date)::date::varchar else 'All Months' end as order_month
, Case when grouping(customer_name) = 0
        then c.customer_name else 'All Customers' end as customer_name
, sum(o.order_amt) as sum_amt
, avg(o.order_amt)::int as avg_amt
, count(o.id) as ct
FROM orders o
JOIN customers c
                ON o.customer_id = c.id
JOIN suppliers s
                ON o.supplier_id = s.id
GROUP BY grouping sets ( s.supplier_name, date_trunc('month', o.order_date),c.customer_name, () )
ORDER BY grouping(supplier_name, customer_name, date_trunc('month', o.order_date))
```

GROUPING SETS(9.5)

Results:

supplier_name	order_month	customer_name	sum_amt	avg_amt	ct
Bodega Privada	All Months	All Customers	85	85	1
ExoTerra	All Months	All Customers	550	275	2
Goose Island Beer, Co	All Months	All Customers	190	95	2
Herpetoculture, LLC	All Months	All Customers	250	125	2
All Suppliers	All Months	Artem Okulik	215	108	2
All Suppliers	All Months	Luke Daniels	210	105	2
All Suppliers	All Months	Stella Nisenbaum	650	217	3
All Suppliers	2016-02-01	All Customers	615	205	3
All Suppliers	2016-01-01	All Customers	460	115	4
All Suppliers	All Months	All Customers	1075	154	7

ROLLUP(9.5)

```
SELECT
Case when grouping(s.country) = 0
        then s.country else 'All Countries' end as supplier_country
, Case when grouping(supplier_name) = 0
                then s.supplier_name else 'All Suppliers' end as supplier_name
, Case when grouping(customer_name) = 0
                then c.customer_name else 'All Customers' end as customer_name
, sum(o.order_amt) as sum_amt
, avg(o.order_amt)::int as avg_amt
, count(o.id) as ct
FROM orders o
JOIN customers c
                ON o.customer_id = c.id
JOIN suppliers s
                ON o.supplier_id = s.id
WHERE s.country in (‘USA’, ‘Spain’)
GROUP BY rollup(s.country ,supplier_name ,customer_name)
```

ROLLUP(9.5)

Results:

supplier_country	supplier_name	customer_name	sum_amt	avg_amt	ct
Spain	Bodega Privada	Luke Daniels	85	85	1
Spain	Bodega Privada	All Customers	85	85	1
Spain	All Suppliers	All Customers	85	85	1
USA	Goose Island Beer, Co	Artem Okulik	65	65	1
USA	Goose Island Beer, Co	Luke Daniels	125	125	1
USA	Goose Island Beer, Co	All Customers	190	95	2
USA	Herpetoculture, LLC	Artem Okulik	150	150	1
USA	Herpetoculture, LLC	Stella Nisenbaum	100	100	1
USA	Herpetoculture, LLC	All Customers	250	125	2
USA	All Suppliers	All Customers	440	110	4
All Countries	All Suppliers	All Customers	1075	154	7

CUBE(9.5)

```
SELECT
Case when grouping(supplier_name) = 0
          then s.supplier_name else 'All Suppliers' end as supplier_name
, Case when grouping(customer_name) = 0
          then c.customer_name else 'All Customers' end as customer_name
, sum(o.order_amt) as sum_amt
, avg(o.order_amt)::int as avg_amt
, count(o.id) as ct
FROM orders o
JOIN customers c
          ON o.customer_id = c.id
JOIN suppliers s
          ON o.supplier_id = s.id
WHERE c.id in (1,3)
GROUP BY cube(supplier_name ,customer_name)
ORDER BY grouping(supplier_name), supplier_name, grouping(customer_name), customer_name
```

CUBE(9.5)

Results:

supplier_name	customer_name	sum_amt	avg_amt	ct
Bodega Privada	Luke Daniels	85	85	1
Bodega Privada	All Customers	85	85	1
ExoTerra	Stella Nisenbaum	550	275	2
ExoTerra	All Customers	550	275	2
Goose Island Beer, Co	Luke Daniels	125	125	1
Goose Island Beer, Co	All Customers	125	125	1
Herpetoculture, LLC	Stella Nisenbaum	100	100	1
Herpetoculture, LLC	All Customers	100	100	1
All Suppliers	Luke Daniels	210	105	2
All Suppliers	Stella Nisenbaum	650	217	3
All Suppliers	All Customers	860	172	5

SUBQUERIES: UNCORRELATED

Uncorrelated subquery:

- Subquery calculates a constant result set for the upper query
- Executed only once

```
SELECT supplier_name, city
FROM suppliers s
WHERE s.country in (SELECT country FROM customers)
```

supplier_name	city
Herpetoculture, LLC	Meriden
Goose Island Beer, Co	Chicago

SUBQUERIES: CORRELATED

Correlated subquery:

- Subquery references variables from the upper query
- Subquery has to be re-executed for each row of the upper query
- Can often be re-written as a join

```
SELECT supplier_name, country
, (SELECT count(distinct id) FROM customers c where c.country=s.country) cust_ct
FROM suppliers s
```

supplier_name	country	cust_ct
Herpetoculture, LLC	USA	2
Bodega Privada	Spain	0
ExoTerra	Canada	0
Goose Island Beer, Co	USA	2

WINDOW FUNCTIONS - BASICS

What is a window function?

A function which is applied to a set of rows defined by a window descriptor and returns a single value for each row from the underlying query

When should you use a window function?

Any time you need to perform calculations or aggregations on your result set while preserving row level detail

WINDOW FUNCTIONS - SYNTAX

```
function_name ([expression [, expression ... ]]) [ FILTER ( WHERE filter_clause ) ] OVER window_name
function_name ([expression [, expression ... ]]) [ FILTER ( WHERE filter_clause ) ] OVER ( window_definition )
function_name ( * ) [ FILTER ( WHERE filter_clause ) ] OVER window_name
function_name ( * ) [ FILTER ( WHERE filter_clause ) ] OVER ( window_definition )

Where window_definition is:

[ existing_window_name ]
[ PARTITION BY expression [, ...] ]
[ ORDER BY expression [ ASC | DESC | USING operator ] [ NULLS { FIRST | LAST } ] [, ...] ]
[ frame_clause ]

{ RANGE | ROWS } frame_start
{ RANGE | ROWS } BETWEEN frame_start AND frame_end
```

WINDOW FUNCTIONS – FRAME CLAUSE

Frame_clause can be one of :

{ RANGE | ROWS } *frame_start*
{ RANGE | ROWS } BETWEEN *frame_start* AND *frame_end*

Where *frame_start* can be one of:

UNBOUNDED PRECEDING
Value PRECEDING
CURRENT ROW

Where *frame_end* can be one of:

UNBOUNDED FOLLOWING
Value FOLLOWING
CURRENT ROW - (default)

When *frame_clause* is omitted, default to RANGE UNBOUNDED PRECEDING

WINDOW FUNCTIONS – BASIC EXAMPLE

```
SELECT
supplier_name , country, revenue
, avg(revenue) OVER (PARTITION BY country)
FROM suppliers
```

supplier_name	country	revenue	avg
ExoTerra	Canada	400,000,000	400,000,000
Bodega Privada	Spain	700,000,000	700,000,000
Herpetoculture, LLC	USA	300,000,000	275,000,000
Goose Island Beer, Co	USA	250,000,000	275,000,000

WINDOW FUNCTIONS – RANGE VS ROWS

With RANGE all duplicates are considered part of the same group and the function is run across all of them, with the same result used for all members of the group.

```
SELECT
supplier_name , country, revenue
, avg(revenue) OVER (ORDER BY country RANGE UNBOUNDED PRECEDING) ::int
FROM suppliers
```

supplier_name	country	revenue	avg
ExoTerra	Canada	400,000,000	400,000,000
Bodega Privada	Spain	700,000,000	550,000,000
Herpetoculture, LLC	USA	300,000,000	412,500,000
Goose Island Beer, Co	USA	250,000,000	412,500,000

WINDOW FUNCTIONS – RANGE VS ROWS

With ROWS, can get a “running” average even across duplicates within the ORDER BY

```
SELECT
supplier_name , country, revenue
, avg(revenue) OVER (ORDER BY country ROWS UNBOUNDED PRECEDING) ::int
FROM suppliers
```

supplier_name	country	revenue	avg
ExoTerra	Canada	400,000,000	400,000,000
Bodega Privada	Spain	700,000,000	550,000,000
Herpetoculture, LLC	USA	300,000,000	466,666,667
Goose Island Beer, Co	USA	250,000,000	412,500,000

WINDOW FUNCTIONS – WINDOW CLAUSE

```
SELECT
supplier_name , country, revenue
, sum(revenue) OVER mywindow as sum
, avg(revenue) OVER  mywindow as avg
FROM suppliers
WINDOW mywindow as (PARTITION BY country)
```

supplier_name	country	revenue	sum	avg
ExoTerra	Canada	400,000,000	400,000,000	400,000,000
Bodega Privada	Spain	700,000,000	700,000,000	700,000,000
Herpetoculture, LLC	USA	300,000,000	550,000,000	275,000,000
Goose Island Beer, Co	USA	250,000,000	550,000,000	275,000,000

WINDOW FUNCTIONS – ROW NUMBER

```
SELECT
Row_number() OVER () as row
,supplier_name , country, revenue
, sum(revenue) OVER mywindow as sum
, avg(revenue) OVER  mywindow as avg
FROM suppliers
WINDOW mywindow as (PARTITION BY country)
```

Row	supplier_name	country	revenue	sum	avg
1	ExoTerra	Canada	400,000,000	400,000,000	400,000,000
2	Bodega Privada	Spain	700,000,000	700,000,000	700,000,000
3	Herpetoculture, LLC	USA	300,000,000	550,000,000	275,000,000
4	Goose Island Beer, Co	USA	250,000,000	550,000,000	275,000,000

WINDOW FUNCTIONS – RANK

```
SELECT
Rank() OVER (ORDER BY country desc) as rank
, supplier_name , country, revenue
, sum(revenue) OVER mywindow as sum
, avg(revenue) OVER  mywindow as avg
FROM suppliers
WINDOW mywindow as (PARTITION BY country)
```

rank	supplier_name	country	revenue	sum	avg
1	Herpetoculture, LLC	USA	300,000,000	550,000,000	275,000,000
1	Goose Island Beer, Co	USA	250,000,000	550,000,000	275,000,000
3	Bodega Privada	Spain	700,000,000	700,000,000	700,000,000
4	ExoTerra	Canada	400,000,000	400,000,000	400,000,000

WINDOW FUNCTIONS – RANK WITH ORDER BY

```
SELECT
Rank() OVER (ORDER BY country desc) as rank
, supplier_name , country, revenue
, sum(revenue) OVER mywindow as sum
, avg(revenue) OVER  mywindow as avg
FROM suppliers
WINDOW mywindow as (PARTITION BY country)
Order by supplier_name
```

rank	supplier_name	country	revenue	sum	avg
3	Bodega Privada	Spain	700,000,000	700,000,000	700,000,000
4	ExoTerra	Canada	400,000,000	400,000,000	400,000,000
1	Goose Island Beer, Co	USA	250,000,000	550,000,000	275,000,000
1	Herpetoculture, LLC	USA	300,000,000	550,000,000	275,000,000

CTE'S – INTRODUCTION

- CTE = Common Table Expression
- Defined by a WITH clause
- Can be seen as a temp table or view which is private to a given query
- Can be recursive/self referencing
- Act as an optimization fence

Syntax:

[WITH [RECURSIVE] with_query [, ...]]

Where *with_query* is:

with_query_name [(column_name [, ...])] AS (select | values | insert | update | delete)

Recursion requires the following syntax within the WITH clause:

non_recursive_term UNION [ALL] recursive_term

CTE'S – NON RECURSIVE EXAMPLE

```
WITH cte_c (country, customer_ct)
as (SELECT country, count(distinct id) as customer_ct
        FROM customers
        GROUP BY country
        )
, cte_s (country, supplier_ct)
as ( SELECT country, count(distinct id) as supplier_ct
FROM  suppliers
GROUP BY country)

SELECT coalesce(c.country, s.country)  as country, customer_ct, supplier_ct
FROM cte_c c
FULL JOIN cte_s  s USING (country)
```

CTE'S – NON RECURSIVE EXAMPLE

Results:

country	customer_ct	supplier_ct
Belarus	1	
Sweden	1	
USA	2	2
Spain		1
Canada		1

CTE'S – RECURSIVE EXAMPLE

List all numbers from 1 to 100:

```
WITH RECURSIVE cte_name(n)
AS
        (VALUES(1)
        UNION
        SELECT n+1
        FROM cte_name
        WHERE n<100)
SELECT * FROM cte_name ORDER by n
```

CTE'S – RECURSIVE QUERY EVALUATION

1. Evaluate the non-recursive term, discarding duplicate rows (for UNION). Include all remaining rows in the result of the recursive query as well as in a temporary *working table.*

2. While the working table is not empty, repeat these steps:

    a. Evaluate the recursive term, substituting the current contents of the working table for the recursive self reference. Discard duplicate rows(for UNION). Include all remaining rows in the result of the recursive query, and also place them in a temporary *intermediate table*.

    b. Replace the contents of the working table with the contents of the intermediate table, then empty the intermediate table.

CTE'S – ANOTHER RECURSIVE EXAMPLE

Parts

Id	Whole	Part	Count
1	Car	Door	4
2	Car	Engine	1
3	Car	Wheel	4
4	Car	Steering wheel	1
5	Cylinder head	Screw	14
6	Door	Window	1
7	Engine	Cylinder head	1
8	Wheel	Screw	5

CTE’S – ANOTHER RECURSIVE EXAMPLE

Goal: Number of screws needed to assemble a car.

```
WITH RECURSIVE list(whole, part, ct)
AS
-- non recursive query, assign results to working table and results table
( SELECT whole, part, count as ct  FROM parts WHERE whole = ‘car’

-- recursive query  with self reference; self reference substituted  by working table
-- assigned to intermediary table , working table and appended to results table
UNION
SELECT cte.whole, a.part, a.count * cte.ct as ct
FROM        list cte
JOIN        parts a
                ON a.whole = cte.part

-- empty intermediate table and execute recursive term as long as working table contains any tuple
)

SELECT sum(ct) FROM list WHERE part = ‘screw’
```

CTE'S – CAVEATS

- Recursive queries actually use iteration
- Union vs Union All
- Only one recursive self-reference allowed
- Primary query evaluates subqueries defined by WITH only once
- Name of the WITH query hides any 'real' table
- No aggregates, GROUP BY, HAVING, ORDER BY, LIMIT, OFFSET allowed in a recursive query
- No mutual recursive WITH queries allowed
- Recursive references must not be part of an OUTER JOIN
- Optimization fence

CTE’S – WRITABLE CTE

Delete from one table and write into another…

```
WITH archive_rows(whole, part, count)
AS
( DELETE FROM parts
WHERE whole = ‘car’
RETURNING *
)
INSERT INTO parts_archive
SELECT * FROM archive_rows;
```

CTE’S – RECURSIVE WRITABLE CTE

```
WITH RECURSIVE list(whole, part, ct)
AS
( SELECT whole, part, count as ct
FROM parts
WHERE whole = ‘car’

UNION
SELECT cte.whole, a.part, a.count * cte.ct as ct
FROM list cte
JOIN parts a ON a.whole = cte.part
)
INSERT INTO car_parts_list
SELECT * FROM list
```

CTE'S – RECURSIVE WRITABLE CTE

```
SELECT * FROM car_parts_list
```

Whole	Part	Ct
car	Engine	1
car	Wheel	4
car	Doors	4
car	Steering wheel	1
car	Cylinder head	1
car	Screw	20
car	window	4
Car	Screw	14

LATERAL(9.3)

LATERAL is a new(ish) JOIN method which allows a subquery in one part of the FROM clause to reference columns from earlier items in the FROM clause

- Refer to earlier table
- Refer to earlier subquery
- Refer to earlier set returning function (SRF)
  - Implicitly added when a SRF is referring to an earlier item in the FROM clause

LATERAL – SET RETURNING FUNCTION EXAMPLE

```
CREATE TABLE numbers
AS
SELECT generate_series as max_num
FROM generate_series(1,10);
-----------------------------------------------------------
SELECT *
 FROM numbers ,
LATERAL generate_series(1,max_num);
```

Same as :

```
SELECT *
FROM numbers ,
generate_series(1,max_num);
```

Results:

Max_num	Generate_series
1	1
2	1
2	2
3	1
3	2
3	3
…	….

LATERAL – SUBQUERY EXAMPLE

This DOES NOT work:

```
SELECT c.customer_name
, c.country
, s.supplier_name
, s.country
 FROM
         (SELECT *
         FROM customers
         WHERE customer_name like ‘S%'
                  ) c
 JOIN
          (SELECT *
         FROM suppliers s
         WHERE s.country = c.country) s
         ON true
```

LATERAL – SUBQUERY EXAMPLE

“ERROR: invalid reference to FROM-clause entry for table "c" Hint: There is an entry for table "c", but it cannot be referenced from this part of the query.”

LATERAL – SUBQUERY EXAMPLE

This DOES NOT work:

```
SELECT c.customer_name
, c.country
, s.supplier_name
, s.country
 FROM
         (SELECT *
         FROM customers
         WHERE customer_name like ‘S%'
                  ) c
 JOIN
          (SELECT *
         FROM suppliers s
         WHERE s.country = c.country) s
         ON true
```

This DOES work:

```
SELECT c.customer_name
, c.country
, s.supplier_name
, s.country
 FROM
         (SELECT *
         FROM customers
         WHERE customer_name like ‘S%'
                  ) c
 JOIN LATERAL
          (SELECT *
         FROM suppliers s
         WHERE s.country = c.country) s
         ON true
```

LATERAL – SUBQUERY EXAMPLE

Results:

Customer_name	Country	Supplier_name	Country
Stephen Frost	USA	Herpetoculture, LLC	USA
Stella Nisenbaum	USA	Herpetoculture, LLC	USA
Stephen Frost	USA	Goose Island Beer, Co	USA
Stella Nisenbaum	USA	Goose Island Beer, Co	USA

LATERAL – SUBQUERY EXAMPLE

We can re-write this logic using a correlated subquery…

```
SELECT
c.customer_name
, c.country
, s.supplier_name
, s.country
FROM (SELECT * FROM customers
          WHERE customer_name like 'S%') c
JOIN suppliers s
     ON s.id =ANY(SELECT id FROM suppliers
               WHERE c.country =  country)
```

But it’s pretty messy.

THANK YOU!

Questions?

REFERENCES

- Join Types :
  - https://www.postgresql.org/docs/9.5/static/queries-table-expressions.html
- Set Operators:
  - https://www.postgresql.org/docs/9.5/static/queries-union.html
- Filtered Aggregates:
  - https://www.postgresql.org/docs/9.5/static/sql-expressions.html#SYNTAX-AGGREGATES
- Grouping Sets, Cube, and Rollup:
  - https://www.postgresql.org/docs/devel/static/queries-table-expressions.html#QUERIES-GROUPING-SETS
- Subqueries:
  - https://momjian.us/main/writings/pgsql/aw_pgsql_book/node80.html
- Window Functions:
  - https://www.postgresql.org/docs/9.5/static/tutorial-window.html
- Common Table Expressions (CTE's):
  - https://www.postgresql.org/docs/9.5/static/queries-with.html
  - https://wiki.postgresql.org/wiki/CTEReadme
- Later Join:
  - https://www.postgresql.org/docs/9.5/static/queries-table-expressions.html#QUERIES-LATERAL