11/10/2015

Becoming A SQL Guru

Stella Nisenbaum
Stella.Nisenbaum@avant.com

AVANT

The Plan

What will we cover?

- Review Some Basics
- Set Operators
- Subqueries
- Aggregate Filter Clause
- Window Functions Galore
- CTE’s
- Lateral

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

```
[ 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 with_query is:

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

```
VALUES ( expression [, ...] ) [, ...]
    [ ORDER BY sort_expression [ ASC | DESC | USING operator ] [, ...] ]
    [ LIMIT { count | ALL } ]
    [ OFFSET start [ ROW | ROWS ] ]
    [ FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } ONLY ]
```

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

Queries – Basic Examples

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

Column1	Column2
1	one
2	two
3	three

```
INSERT INTO tmp (num, word)
VALUES (1, 'one'), (2, 'two'), (3, 'three')
```

```
SELECT sum(column1)
From
(VALUES (1, 'one'), (2, 'two'), (3, 'three'))A;
```

```
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 left table with each row from the right 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	Jeff Edstrom	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

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, city
, (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	Madrid	0
ExoTerra	Canada	0
Goose Island Beer, Co	USA	2

Subqueries: Correlated – Re-Written using Join

```
SELECT s.supplier_name, s.city
, count(distinct c.id) cust_ct
FROM suppliers s
LEFT JOIN customers c
      ON s.country = c.country
GROUP BY 1,2
```

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

Filtered Aggregates – The Old Way

GOAL: Get a count of all suppliers and a count of suppliers whose revenue is greater than or equal to 4 Million

```
SELECT COUNT (DISTINCT id) as all_suppliers
, COUNT(DISTINCT
        CASE
            WHEN revenue >=400000000
                THEN id
            ELSE NULL
        END) as filtered_suppliers
FROM suppliers s
```

all_suppliers	filtered_suppliers
4	2

Filtered Aggregates – The New Way

AGGREGATE FILTER CLAUSE – GENERAL SYNTAX:

aggregate_name (ALL | DISTINCT *expression* [, ...]) [FILTER (WHERE *filter_clause*)]

```
SELECT COUNT(DISTINCT id) as all_suppliers
, COUNT (DISTINCT id) FILTER (WHERE revenue >=400000000) filtered_suppliers
FROM suppliers s
```

all_suppliers	filtered_suppliers
4	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 ::int 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 ::int 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  ::int 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 ::int 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

Window Functions – nTile

```
SELECT ntile(2) OVER (ORDER BY revenue) as ntile
, supplier_name
, country
, revenue
, sum(revenue) OVER mywindow as sum
, avg(revenue) OVER  mywindow ::int as avg
FROM suppliers
WINDOW mywindow as (PARTITION BY country)
```

rank	supplier_name	country	revenue	sum	avg
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
2	ExoTerra	Canada	400,000,000	400,000,000	400,000,000
2	Bodega Privada	Spain	700000000	700.000.000	700,000,000

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

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 c (country, customer_ct)
as (SELECT country, count(distinct id) as customer_ct
        FROM customers
        GROUP BY country
        )
, 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 c
FULL OUTER JOIN 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	Doors	4
2	Car	Engine	1
3	Car	Wheel	4
4	Car	Steering wheel	1
5	Cylinder head	Screw	14
6	Doors	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 b.whole, a.part, a.count * b.ct as ct FROM list b
JOIN parts a ON b.part = a.whole
-- empty intermediate table and execute recursive term as long as working table
contains any tuple
)
-- produce final result set
SELECT sum(ct) FROM list WHERE part = ‘screw’
```

sum
34

CTE’s – Another Recursive Example

```
SELECT * FROM list
ORDER BY whole, part
```

whole	part	ct
car	cylinder head	1
car	doors	4
car	engine	1
car	screw	20
car	screw	14
car	steering wheel	1
car	wheel	4
car	window	4

CTE’s – Caveats

- Union vs Union All
- Primary query evaluates subqueries defined by WITH only once
- Acts as an Optimization Fence
- Only one recursive self-reference allowed
- Name of the WITH query hides any ‘real’ tables
- No aggregates, GROUP BY, HAVING, ORDER BY, LIMIT, OFFSET allowed in a recursive query

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 – Writable CTE

```
SELECT *
FROM parts_archive
```

whole	part	ct
car	engine	1
car	wheel	4
car	doors	4
car	steering wheel	1

```
SELECT *
FROM parts
```

whole	part	ct
engine	cylinder head	1
cylinder head	screw	14
wheel	screw	5
doors	window	1

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 b.whole, a.part, a.count * b.ct as ct
FROM list b
JOIN parts a ON a.whole = b.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

LATERAL is a new 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);

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
        customers c
 JOIN
         (SELECT *
        FROM suppliers s
        WHERE s.country = c.country
        ORDER BY revenue
        Limit 1) s
        ON true
```

This DOES work:

```
SELECT c.customer_name
, c.country
, s.supplier_name
, s.country
 FROM
        customers c
 JOIN LATERAL
         (SELECT *
        FROM suppliers s
        WHERE s.country = c.country
        ORDER BY revenue
        Limit 1) s
        ON true
```

Lateral – Subquery Example

Customer_name	Country	Supplier_name	Country
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 customers c
JOIN suppliers s
      ON s.id =(SELECT id FROM suppliers
                  WHERE c.country =  country
                  ORDER BY revenue
                  Limit 1)
```

But it’s pretty messy AND less performant!!

Thank you !

Questions?