BECOMING A SQL GURU

Stella Nisenbaum
Stella.Nisenbaum@yello.co
Eliminating gaps in the hiring experience
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.
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
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
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 ] [ ... ] ]
BECOMING A SQL GURU

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 [, ...] ) ]
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 [ * ]
VALUES (1, 'one'), (2, 'two'), (3, 'three');

<table>
<thead>
<tr>
<th>Column1</th>
<th>Column2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>one</td>
</tr>
<tr>
<td>2</td>
<td>two</td>
</tr>
<tr>
<td>3</td>
<td>three</td>
</tr>
</tbody>
</table>

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
BECOMING A SQL GURU

CROSS JOINS: EXAMPLE

<table>
<thead>
<tr>
<th>store_id</th>
<th>store_city</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>chicago</td>
</tr>
<tr>
<td>2</td>
<td>dallas</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>product_id</th>
<th>product_desc</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>coffee</td>
</tr>
<tr>
<td>2</td>
<td>tea</td>
</tr>
</tbody>
</table>

SELECT * FROM stores
CROSS JOIN products

Results:

<table>
<thead>
<tr>
<th>store_id</th>
<th>store_city</th>
<th>product_id</th>
<th>product_desc</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>chicago</td>
<td>1</td>
<td>coffee</td>
</tr>
<tr>
<td>1</td>
<td>chicago</td>
<td>2</td>
<td>tea</td>
</tr>
<tr>
<td>2</td>
<td>dallas</td>
<td>1</td>
<td>coffee</td>
</tr>
<tr>
<td>2</td>
<td>dallas</td>
<td>2</td>
<td>tea</td>
</tr>
</tbody>
</table>
### SET OPERATIONS

#### customers

<table>
<thead>
<tr>
<th>ID</th>
<th>customer_name</th>
<th>city</th>
<th>postal_code</th>
<th>country</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Stella Nisenbaum</td>
<td>Chicago</td>
<td>60605</td>
<td>USA</td>
</tr>
<tr>
<td>2</td>
<td>Stephen Frost</td>
<td>New York</td>
<td>10012</td>
<td>USA</td>
</tr>
<tr>
<td>3</td>
<td>Luke Daniels</td>
<td>Stockholm</td>
<td>113 50</td>
<td>Sweden</td>
</tr>
<tr>
<td>4</td>
<td>Artem Okulik</td>
<td>Minsk</td>
<td>220002</td>
<td>Belarus</td>
</tr>
</tbody>
</table>

#### suppliers

<table>
<thead>
<tr>
<th>ID</th>
<th>supplier_name</th>
<th>city</th>
<th>postal_code</th>
<th>country</th>
<th>revenue</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Herpetoculture, LLC</td>
<td>Meriden</td>
<td>06451</td>
<td>USA</td>
<td>300,000,000</td>
</tr>
<tr>
<td>2</td>
<td>Bodega Privada</td>
<td>Madrid</td>
<td>28703</td>
<td>Spain</td>
<td>700,000,000</td>
</tr>
<tr>
<td>3</td>
<td>ExoTerra</td>
<td>Montreal</td>
<td>H9X OA2</td>
<td>Canada</td>
<td>400,000,000</td>
</tr>
<tr>
<td>4</td>
<td>Goose Island Beer, Co</td>
<td>Chicago</td>
<td>60612</td>
<td>USA</td>
<td>250,000,000</td>
</tr>
</tbody>
</table>
## SET OPERATIONS: UNION VS UNION ALL

<table>
<thead>
<tr>
<th>city</th>
<th>city</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chicago</td>
<td>Chicago</td>
</tr>
<tr>
<td>New York</td>
<td>New York</td>
</tr>
<tr>
<td>Stockholm</td>
<td>Stockholm</td>
</tr>
<tr>
<td>Minsk</td>
<td>Minsk</td>
</tr>
<tr>
<td>Meriden</td>
<td>Meriden</td>
</tr>
<tr>
<td>Meriden</td>
<td>Meriden</td>
</tr>
<tr>
<td>Madrid</td>
<td>Madrid</td>
</tr>
<tr>
<td>Montreal</td>
<td>Montreal</td>
</tr>
</tbody>
</table>

**Union All**
```
SELECT city FROM customers
UNION ALL
SELECT city FROM suppliers
```

**Union**
```
SELECT city FROM customers
UNION
SELECT city FROM suppliers
```
### SET OPERATIONS: EXCEPT VS INTERSECT

<table>
<thead>
<tr>
<th>city</th>
<th>city</th>
</tr>
</thead>
<tbody>
<tr>
<td>New York</td>
<td>Chicago</td>
</tr>
<tr>
<td>Stockholm</td>
<td></td>
</tr>
<tr>
<td>Minsk</td>
<td></td>
</tr>
</tbody>
</table>

- SELECT city FROM customers EXCEPT SELECT city FROM suppliers
- SELECT city FROM customers INTERSECT SELECT city FROM suppliers
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

**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

### orders

<table>
<thead>
<tr>
<th>id</th>
<th>customer_id</th>
<th>supplier_id</th>
<th>order_date</th>
<th>order_amt</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2016-01-15</td>
<td>100</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>3</td>
<td>2016-02-05</td>
<td>250</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>2</td>
<td>2016-01-25</td>
<td>85</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>4</td>
<td>2016-01-07</td>
<td>125</td>
</tr>
<tr>
<td>5</td>
<td>4</td>
<td>4</td>
<td>2016-02-19</td>
<td>65</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>1</td>
<td>2016-01-20</td>
<td>150</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>3</td>
<td>2016-02-17</td>
<td>300</td>
</tr>
</tbody>
</table>
GROUPING SETS, CUBE, ROLLUP

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
Results:

<table>
<thead>
<tr>
<th>country</th>
<th>supplier_name</th>
<th>order_month</th>
<th>customer_name</th>
<th>sum_amt</th>
<th>avg_amt</th>
<th>ct</th>
</tr>
</thead>
<tbody>
<tr>
<td>Canada</td>
<td>ExoTerra</td>
<td>2016-02-01</td>
<td>Stella Nisenbaum</td>
<td>550</td>
<td>275</td>
<td>2</td>
</tr>
<tr>
<td>Spain</td>
<td>Bodega Privada</td>
<td>2016-01-01</td>
<td>Luke Daniels</td>
<td>85</td>
<td>85</td>
<td>1</td>
</tr>
<tr>
<td>USA</td>
<td>Goose Island Beer, Co</td>
<td>2016-01-01</td>
<td>Luke Daniels</td>
<td>125</td>
<td>125</td>
<td>1</td>
</tr>
<tr>
<td>USA</td>
<td>Goose Island Beer, Co</td>
<td>2016-02-01</td>
<td>Artem Okulik</td>
<td>65</td>
<td>65</td>
<td>1</td>
</tr>
<tr>
<td>USA</td>
<td>Herpetoculture, LLC</td>
<td>2016-01-01</td>
<td>Artem Okulik</td>
<td>150</td>
<td>150</td>
<td>1</td>
</tr>
<tr>
<td>USA</td>
<td>Herpetoculture, LLC</td>
<td>2016-01-01</td>
<td>Stella Nisenbaum</td>
<td>100</td>
<td>100</td>
<td>1</td>
</tr>
</tbody>
</table>
GROUPING SETS

```sql
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:**

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

<table>
<thead>
<tr>
<th>supplier_country</th>
<th>supplier_name</th>
<th>customer_name</th>
<th>sum_amt</th>
<th>avg_amt</th>
<th>ct</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spain</td>
<td>Bodega Privada</td>
<td>Luke Daniels</td>
<td>85</td>
<td>85</td>
<td>1</td>
</tr>
<tr>
<td>Spain</td>
<td>Bodega Privada</td>
<td>All Customers</td>
<td>85</td>
<td>85</td>
<td>1</td>
</tr>
<tr>
<td>Spain</td>
<td>All Suppliers</td>
<td>All Customers</td>
<td>85</td>
<td>85</td>
<td>1</td>
</tr>
<tr>
<td>USA</td>
<td>Goose Island Beer, Co</td>
<td>Artem Okulik</td>
<td>65</td>
<td>65</td>
<td>1</td>
</tr>
<tr>
<td>USA</td>
<td>Goose Island Beer, Co</td>
<td>Luke Daniels</td>
<td>125</td>
<td>125</td>
<td>1</td>
</tr>
<tr>
<td>USA</td>
<td>Goose Island Beer, Co</td>
<td>All Customers</td>
<td>190</td>
<td>95</td>
<td>2</td>
</tr>
<tr>
<td>USA</td>
<td>Herpetoculture, LLC</td>
<td>Artem Okulik</td>
<td>150</td>
<td>150</td>
<td>1</td>
</tr>
<tr>
<td>USA</td>
<td>Herpetoculture, LLC</td>
<td>Stella Nisenbaum</td>
<td>100</td>
<td>100</td>
<td>1</td>
</tr>
<tr>
<td>USA</td>
<td>Herpetoculture, LLC</td>
<td>All Customers</td>
<td>250</td>
<td>125</td>
<td>2</td>
</tr>
<tr>
<td>USA</td>
<td>All Suppliers</td>
<td>All Customers</td>
<td>440</td>
<td>110</td>
<td>4</td>
</tr>
<tr>
<td>All Countries</td>
<td>All Suppliers</td>
<td>All Customers</td>
<td>1075</td>
<td>154</td>
<td>7</td>
</tr>
</tbody>
</table>
**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
### Results:

<table>
<thead>
<tr>
<th>supplier_name</th>
<th>customer_name</th>
<th>sum_amt</th>
<th>avg_amt</th>
<th>ct</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bodega Privada</td>
<td>Luke Daniels</td>
<td>85</td>
<td>85</td>
<td>1</td>
</tr>
<tr>
<td>Bodega Privada</td>
<td>All Customers</td>
<td>85</td>
<td>85</td>
<td>1</td>
</tr>
<tr>
<td>ExoTerra</td>
<td>Stella Nisenbaum</td>
<td>550</td>
<td>275</td>
<td>2</td>
</tr>
<tr>
<td>ExoTerra</td>
<td>All Customers</td>
<td>550</td>
<td>275</td>
<td>2</td>
</tr>
<tr>
<td>Goose Island Beer, Co</td>
<td>Luke Daniels</td>
<td>125</td>
<td>125</td>
<td>1</td>
</tr>
<tr>
<td>Goose Island Beer, Co</td>
<td>All Customers</td>
<td>125</td>
<td>125</td>
<td>1</td>
</tr>
<tr>
<td>Herpetoculture, LLC</td>
<td>Stella Nisenbaum</td>
<td>100</td>
<td>100</td>
<td>1</td>
</tr>
<tr>
<td>Herpetoculture, LLC</td>
<td>All Customers</td>
<td>100</td>
<td>100</td>
<td>1</td>
</tr>
<tr>
<td>All Suppliers</td>
<td>Luke Daniels</td>
<td>210</td>
<td>105</td>
<td>2</td>
</tr>
<tr>
<td>All Suppliers</td>
<td>Stella Nisenbaum</td>
<td>650</td>
<td>217</td>
<td>3</td>
</tr>
<tr>
<td>All Suppliers</td>
<td>All Customers</td>
<td>860</td>
<td>172</td>
<td>5</td>
</tr>
</tbody>
</table>
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)

<table>
<thead>
<tr>
<th>supplier_name</th>
<th>city</th>
</tr>
</thead>
<tbody>
<tr>
<td>Herpetoculture, LLC</td>
<td>Meriden</td>
</tr>
<tr>
<td>Goose Island Beer, Co</td>
<td>Chicago</td>
</tr>
</tbody>
</table>
**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

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

<table>
<thead>
<tr>
<th>supplier_name</th>
<th>country</th>
<th>cust_ct</th>
</tr>
</thead>
<tbody>
<tr>
<td>Herpetoculture, LLC</td>
<td>USA</td>
<td>2</td>
</tr>
<tr>
<td>Bodega Privada</td>
<td>Spain</td>
<td>0</td>
</tr>
<tr>
<td>ExoTerra</td>
<td>Canada</td>
<td>0</td>
</tr>
<tr>
<td>Goose Island Beer, Co</td>
<td>USA</td>
<td>2</td>
</tr>
</tbody>
</table>
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
SELECT supplier_name, country, revenue, avg(revenue) OVER (PARTITION BY country) FROM suppliers

<table>
<thead>
<tr>
<th>supplier_name</th>
<th>country</th>
<th>revenue</th>
<th>avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>ExoTerra</td>
<td>Canada</td>
<td>400,000,000</td>
<td>400,000,000</td>
</tr>
<tr>
<td>Bodega Privada</td>
<td>Spain</td>
<td>700,000,000</td>
<td>700,000,000</td>
</tr>
<tr>
<td>Herpetoculture, LLC</td>
<td>USA</td>
<td>300,000,000</td>
<td>275,000,000</td>
</tr>
<tr>
<td>Goose Island Beer, Co</td>
<td>USA</td>
<td>250,000,000</td>
<td>275,000,000</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>supplier_name</th>
<th>country</th>
<th>revenue</th>
<th>avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>ExoTerra</td>
<td>Canada</td>
<td>400,000,000</td>
<td>400,000,000</td>
</tr>
<tr>
<td>Bodega Privada</td>
<td>Spain</td>
<td>700,000,000</td>
<td>550,000,000</td>
</tr>
<tr>
<td>Herpetoculture, LLC</td>
<td>USA</td>
<td>300,000,000</td>
<td>412,500,000</td>
</tr>
<tr>
<td>Goose Island Beer, Co</td>
<td>USA</td>
<td>250,000,000</td>
<td>412,500,000</td>
</tr>
</tbody>
</table>
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
```

<table>
<thead>
<tr>
<th>supplier_name</th>
<th>country</th>
<th>revenue</th>
<th>avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>ExoTerra</td>
<td>Canada</td>
<td>400,000,000</td>
<td>400,000,000</td>
</tr>
<tr>
<td>Bodega Privada</td>
<td>Spain</td>
<td>700,000,000</td>
<td>550,000,000</td>
</tr>
<tr>
<td>Herpetoculture, LLC</td>
<td>USA</td>
<td>300,000,000</td>
<td>466,666,667</td>
</tr>
<tr>
<td>Goose Island Beer, Co</td>
<td>USA</td>
<td>250,000,000</td>
<td>412,500,000</td>
</tr>
</tbody>
</table>
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)

<table>
<thead>
<tr>
<th>supplier_name</th>
<th>country</th>
<th>revenue</th>
<th>sum</th>
<th>avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>ExoTerra</td>
<td>Canada</td>
<td>400,000,000</td>
<td>400,000,000</td>
<td>400,000,000</td>
</tr>
<tr>
<td>Bodega Privada</td>
<td>Spain</td>
<td>700,000,000</td>
<td>700,000,000</td>
<td>700,000,000</td>
</tr>
<tr>
<td>Herpetoculture, LLC</td>
<td>USA</td>
<td>300,000,000</td>
<td>550,000,000</td>
<td>275,000,000</td>
</tr>
<tr>
<td>Goose Island Beer, Co</td>
<td>USA</td>
<td>250,000,000</td>
<td>550,000,000</td>
<td>275,000,000</td>
</tr>
</tbody>
</table>
### Window Functions – Row Number

```sql
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)
```

<table>
<thead>
<tr>
<th>Row</th>
<th>supplier_name</th>
<th>country</th>
<th>revenue</th>
<th>sum</th>
<th>avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ExoTerra</td>
<td>Canada</td>
<td>400,000,000</td>
<td>400,000,000</td>
<td>400,000,000</td>
</tr>
<tr>
<td>2</td>
<td>Bodega Privada</td>
<td>Spain</td>
<td>700,000,000</td>
<td>700,000,000</td>
<td>700,000,000</td>
</tr>
<tr>
<td>3</td>
<td>Herpetoculture, LLC</td>
<td>USA</td>
<td>300,000,000</td>
<td>550,000,000</td>
<td>275,000,000</td>
</tr>
<tr>
<td>4</td>
<td>Goose Island Beer, Co</td>
<td>USA</td>
<td>250,000,000</td>
<td>550,000,000</td>
<td>275,000,000</td>
</tr>
</tbody>
</table>
BECOMING A SQL GURU

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)

<table>
<thead>
<tr>
<th>rank</th>
<th>supplier_name</th>
<th>country</th>
<th>revenue</th>
<th>sum</th>
<th>avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Herpetoculture, LLC</td>
<td>USA</td>
<td>300,000,000</td>
<td>550,000,000</td>
<td>275,000,000</td>
</tr>
<tr>
<td>1</td>
<td>Goose Island Beer, Co</td>
<td>USA</td>
<td>250,000,000</td>
<td>550,000,000</td>
<td>275,000,000</td>
</tr>
<tr>
<td>3</td>
<td>Bodega Privada</td>
<td>Spain</td>
<td>700,000,000</td>
<td>700,000,000</td>
<td>700,000,000</td>
</tr>
<tr>
<td>4</td>
<td>ExoTerra</td>
<td>Canada</td>
<td>400,000,000</td>
<td>400,000,000</td>
<td>400,000,000</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>rank</th>
<th>supplier_name</th>
<th>country</th>
<th>revenue</th>
<th>sum</th>
<th>avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Bodega Privada</td>
<td>Spain</td>
<td>700,000,000</td>
<td>700,000,000</td>
<td>700,000,000</td>
</tr>
<tr>
<td>4</td>
<td>ExoTerra</td>
<td>Canada</td>
<td>400,000,000</td>
<td>400,000,000</td>
<td>400,000,000</td>
</tr>
<tr>
<td>1</td>
<td>Goose Island Beer, Co</td>
<td>USA</td>
<td>250,000,000</td>
<td>550,000,000</td>
<td>275,000,000</td>
</tr>
<tr>
<td>1</td>
<td>Herpetoculture, LLC</td>
<td>USA</td>
<td>300,000,000</td>
<td>550,000,000</td>
<td>275,000,000</td>
</tr>
</tbody>
</table>
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:

```
onrecursive_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:**

<table>
<thead>
<tr>
<th>country</th>
<th>customer_ct</th>
<th>supplier_ct</th>
</tr>
</thead>
<tbody>
<tr>
<td>Belarus</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>Sweden</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>USA</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Spain</td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Canada</td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Id</th>
<th>Whole</th>
<th>Part</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Car</td>
<td>Door</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>Car</td>
<td>Engine</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>Car</td>
<td>Wheel</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>Car</td>
<td>Steering wheel</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>Cylinder head</td>
<td>Screw</td>
<td>14</td>
</tr>
<tr>
<td>6</td>
<td>Door</td>
<td>Window</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>Engine</td>
<td>Cylinder head</td>
<td>1</td>
</tr>
<tr>
<td>8</td>
<td>Wheel</td>
<td>Screw</td>
<td>5</td>
</tr>
</tbody>
</table>
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'
BECOMING A SQL GURU

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
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;
BECOMING A SQL GURU

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

```sql
SELECT * FROM car_parts_list
```

<table>
<thead>
<tr>
<th>Whole</th>
<th>Part</th>
<th>Ct</th>
</tr>
</thead>
<tbody>
<tr>
<td>car</td>
<td>Engine</td>
<td>1</td>
</tr>
<tr>
<td>car</td>
<td>Wheel</td>
<td>4</td>
</tr>
<tr>
<td>car</td>
<td>Doors</td>
<td>4</td>
</tr>
<tr>
<td>car</td>
<td>Steering wheel</td>
<td>1</td>
</tr>
<tr>
<td>car</td>
<td>Cylinder head</td>
<td>1</td>
</tr>
<tr>
<td>car</td>
<td>Screw</td>
<td>20</td>
</tr>
<tr>
<td>car</td>
<td>window</td>
<td>4</td>
</tr>
<tr>
<td>Car</td>
<td>Screw</td>
<td>14</td>
</tr>
</tbody>
</table>
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 ,
LATERAL generate_series (1,max_num);

Results:

<table>
<thead>
<tr>
<th>Max_num</th>
<th>Generate_series</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>...</td>
<td>....</td>
</tr>
</tbody>
</table>
LATERAL – SUBQUERY EXAMPLE

This DOES NOT work:

```sql
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
```
“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:

```sql
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:

```sql
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:

<table>
<thead>
<tr>
<th>Customer_name</th>
<th>Country</th>
<th>Supplier_name</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stephen Frost</td>
<td>USA</td>
<td>Herpetoculture, LLC</td>
<td>USA</td>
</tr>
<tr>
<td>Stella Nisenbaum</td>
<td>USA</td>
<td>Herpetoculture, LLC</td>
<td>USA</td>
</tr>
<tr>
<td>Stephen Frost</td>
<td>USA</td>
<td>Goose Island Beer, Co</td>
<td>USA</td>
</tr>
<tr>
<td>Stella Nisenbaum</td>
<td>USA</td>
<td>Goose Island Beer, Co</td>
<td>USA</td>
</tr>
</tbody>
</table>
LATERAL – SUBQUERY EXAMPLE

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

```sql
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.
```
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_pgsq1_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