



PostgreSQL

The world's most advanced
open source database.

PREVIEWED

05/2016

EDGE

EDGE

www.exe.cl

SQL a la postgres





VISTAS



- Vistas

```
CREATE OR REPLACE VIEW census.vw_facts_2011 AS  
SELECT fact_type_id, val, yr, tract_id FROM census.facts WHERE yr =  
2011;
```

- A partir de la versión 9.3, se pueden alterar utilizando INSERT, UPDATE, o DELETE. Updates y Deletes estan sujetas a la cláusula where de la vista.

```
DELETE FROM census.vw_facts_2011 WHERE val = 0;
```

- Esta instrucción no actualiza ningún registros

```
UPDATE census.vw_facts_2011 SET val = 1 WHERE val = 0 AND yr = 2012;
```

- Si es factible insertar o actualizar registros que no cumplan la condición del WHERE

```
UPDATE census.vw_facts_2011 SET yr = 2012 WHERE yr = 2011;
```



VISTAS



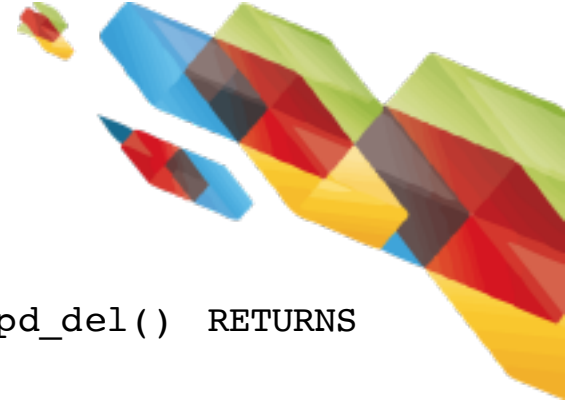
- **WITH CHECK OPTION** Permite chequear que no se hagan actualizaciones, inserciones fuera del alcance de la vista

```
CREATE OR REPLACE VIEW census.vw_facts_2011 AS
SELECT fact_type_id, val, yr, tract_id
FROM census.facts WHERE yr = 2011 WITH CHECK OPTION;
UPDATE census.vw_facts_2011 SET yr = 2012 WHERE val > 2942;
```

```
ERROR: new row violates WITH CHECK OPTION for view "vw_facts_2011"
DETAIL: Failing row contains (1, 25001010500, 2012, 2985.000, 100.00).
```



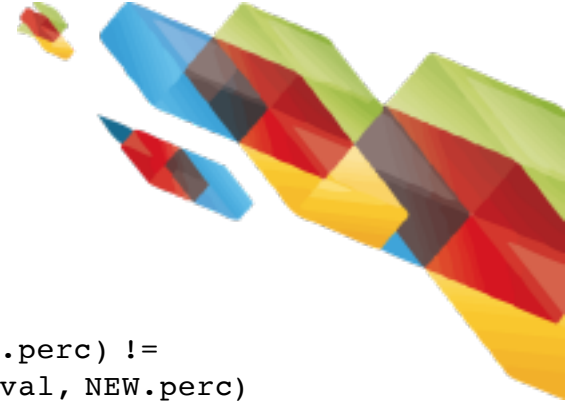
TRIGGER



```
CREATE OR REPLACE FUNCTION census.trig_vw_facts_ins_upd_del() RETURNS
trigger AS
$$
BEGIN
IF (TG_OP = 'DELETE') THEN
    DELETE FROM census.facts AS f
    WHERE
        f.tract_id = OLD.tract_id AND f.yr = OLD.yr AND
        f.fact_type_id = OLD.fact_type_id;
    RETURN OLD;
END IF;
IF (TG_OP = 'INSERT') THEN
    INSERT INTO census.facts(tract_id, yr, fact_type_id, val, perc)
    SELECT NEW.tract_id, NEW.yr, NEW.fact_type_id, NEW.val, NEW.perc;
    RETURN NEW;
END IF;
```



TRIGGER



```
IF (TG_OP = 'UPDATE') THEN
    IF ROW(OLD.fact_type_id, OLD.tract_id, OLD.yr, OLD.val, OLD.perc) !=
        ROW(NEW.fact_type_id, NEW.tract_id, NEW.yr, NEW.val, NEW.perc)
    THEN
        UPDATE census.facts AS f
        SET tract_id = NEW.tract_id,
            yr = NEW.yr,
            fact_type_id = NEW.fact_type_id,
            val = NEW.val,
            perc = NEW.perc
        WHERE
            f.tract_id = OLD.tract_id AND
            f.yr = OLD.yr AND
            f.fact_type_id = OLD.fact_type_id;
        RETURN NEW;
    ELSE
        RETURN NULL;
    END IF;
END IF;
END;
$$
LANGUAGE plpgsql VOLATILE;
```



TRIGGER



```
CREATE TRIGGER census.trig_01_vw_facts_ins_upd_del  
INSTEAD OF INSERT OR UPDATE OR DELETE ON census.vw_facts  
FOR EACH ROW EXECUTE PROCEDURE census.trig_vw_facts_ins_upd_del();
```



Vistas materializadas



- Las vistas materializadas realizan un caché de los datos..
- El caché se crea al momento de crear la vista
- Se puede refrescar el caché con el comando **REFRESH MATERIALIZED VIEW**.
- Las vistas materializadas están disponibles a partir de la versión 9.3

```
CREATE MATERIALIZED VIEW census.vw_facts_2011_materialized AS
SELECT fact_type_id, val, yr, tract_id FROM census.facts WHERE yr =
2011;
CREATE UNIQUE INDEX ix
ON census.vw_facts_2011_materialized (tract_id, fact_type_id, yr);
REFRESH MATERIALIZED VIEW census.vw_facts_2011_materialized;
```

- Para evitar locks a partir de la versión 9.4 es posible utilizar:

```
REFRESH MATERIALIZED VIEW CONCURRENTLY
census.vw_facts_2011_materialized;
```




DISTINCT ON

- DISTINCT**

- Retorna solo aquellos valores distintos.

- ```
SELECT DISTINCT columna1, columna2 FROM tabla;
```

- DISTINCT ON**

- Mantiene solo la primera fila de todas las filas donde la expresión es igual.

- El Order By debe partir con la expresión contenida en el DISTINCT ON

```
SELECT DISTINCT ON (left(tract_id, 5))
left(tract_id, 5) As county, tract_id, tract_name
FROM census.lu_tracts ORDER BY county, tract_id;
county | tract_id | tract_name
```

```
-----+-----+-----
25001 | 25001010100 | Census Tract 101, Barnstable County, Massachusetts
25003 | 25003900100 | Census Tract 9001, Berkshire County, Massachusetts
25005 | 25005600100 | Census Tract 6001, Bristol County, Massachusetts
25007 | 25007200100 | Census Tract 2001, Dukes County, Massachusetts
25009 | 25009201100 | Census Tract 2011, Essex County, Massachusetts
```



# Limit y offset

```
SELECT DISTINCT ON (left(tract_id, 5))
left(tract_id, 5) As county, tract_id, tract_name
FROM census.lu_tracts
ORDER BY county, tract_id LIMIT 3 OFFSET 2;
county | tract_id | tract_name
```

```
-----+-----+-----
25005 | 25005600100 | Census Tract 6001, Bristol County, Massachusetts
25007 | 25007200100 | Census Tract 2001, Dukes County, Massachusetts
25009 | 25009201100 | Census Tract 2011, Essex County, Massachusetts
```



# Inserción y selección múltiple

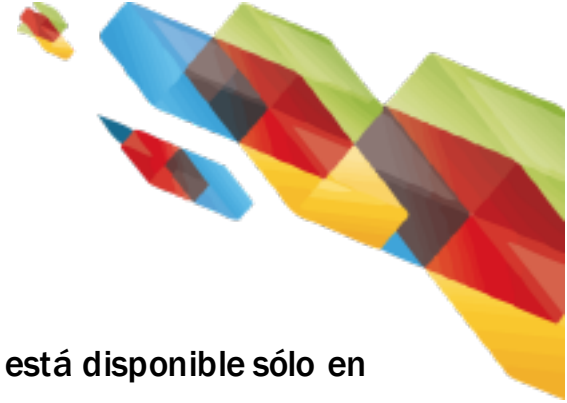


```
INSERT INTO logs_2011 (user_name, description, log_ts)
VALUES
('robe', 'logged in', '2011-01-10 10:15 AM EST'),
('lhsu', 'logged out', '2011-01-11 10:20 AM EST');
```

```
SELECT *
FROM (
VALUES
('robe', 'logged in', '2011-01-10 10:15 AM EST'::timestampz),
('lhsu', 'logged out', '2011-01-11 10:20 AM EST'::timestampz)
) AS l (user_name, description, log_ts);
```



# ILIKE



- PostgreSQL es case-sensitive.
- Para buscar de manera case-insensitive existe el operador ILIKE (~) que está disponible sólo en PostgreSQL:

```
SELECT tract_name FROM census.lu_tracts WHERE tract_name ILIKE '%duke%';
```



# CAST



- **CAST('2011-1-11' AS date)** casts el texto **2011-1-1** a fecha
- PostgreSQL puede hacer lo mismo ocupando una sintáxis más corta  
`'2011-1-1'::date.`
- También es factible convertir en varios pasos, ejemplo:  
`campoXML::text::integer.`



# Restricciones en tablas de herencia

- A veces se desea solo consultar, actualizar o borrar registros en una tabla hija, sin afectar a los registros de la tabla padre.
- Esto se puede hacer con el siguiente comando:

```
DELETE FROM ONLY logs_2011 WHERE log_ts < '2011-03-01' RETURNING *
```

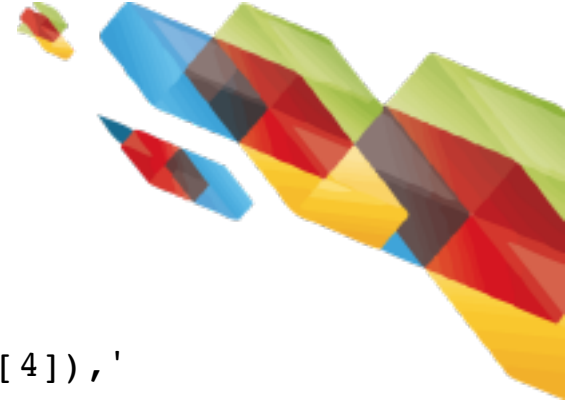


# Select con funciones

```
CREATE TABLE interval_periods (i_type interval);
INSERT INTO interval_periods (i_type)
VALUES ('5 months'), ('132 days'), ('4862 hours');
```

```
SELECT i_type,
generate_series('2012-01-01'::date, '2012-12-31'::date, i_type) As dt
FROM interval_periods;
```

| i_type     | dt                     |
|------------|------------------------|
| 5 months   | 2012-01-01 00:00:00-05 |
| 5 months   | 2012-06-01 00:00:00-04 |
| 5 months   | 2012-11-01 00:00:00-04 |
| 132 days   | 2012-01-01 00:00:00-05 |
| 132 days   | 2012-05-12 00:00:00-04 |
| 132 days   | 2012-09-21 00:00:00-04 |
| 4862 hours | 2012-01-01 00:00:00-05 |
| 4862 hours | 2012-07-21 15:00:00-04 |



# Update con retorno

```
UPDATE census.lu_fact_types AS f
SET short_name = replace(replace(lower(f.fact_subcats[4]), '
','_'),':','')
WHERE f.fact_subcats[3] = 'Hispanic or Latino:' AND f.fact_subcats[4] >
''
RETURNING fact_type_id, short_name;
```





# Delete using

- Cuando se desea borrar datos utilizando datos de otra tablar es factible utilizando la cláusula USING en conjunto con WHERE:

```
DELETE FROM census.facts
USING census.lu_fact_types As ft
WHERE facts.fact_type_id = ft.fact_type_id AND ft.short_name = 's01';
```



# Select de tablas

```
SELECT x FROM census.lu_fact_types As x LIMIT 2;
```

```
SELECT array_to_json(array_agg(f)) As cat
FROM (
 SELECT MAX(fact_type_id) As max_type, category
 FROM census.lu_fact_types
 GROUP BY category
) As f;
```



# DO



- El comando DO command permite utilizar al vuelo un procedimiento SQL

```
set search_path=census;
DROP TABLE IF EXISTS lu_fact_types;
CREATE TABLE lu_fact_types (
 fact_type_id serial,
 category varchar(100),
 fact_subcats varchar(255)[],
 short_name varchar(50),
 CONSTRAINT pk_lu_fact_types PRIMARY KEY (fact_type_id)
);
```



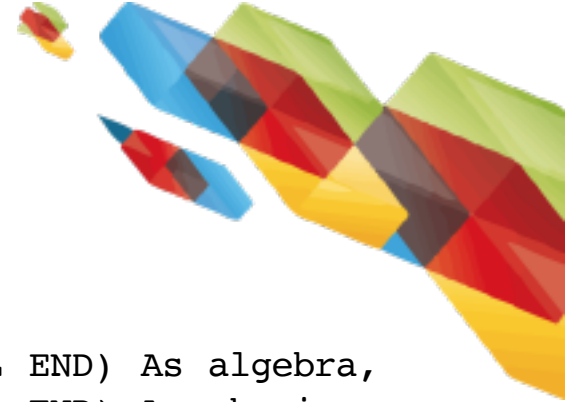
# DO



```
DO language plpgsql
$$
DECLARE var_sql text;
BEGIN
var_sql := string_agg(
'INSERT INTO lu_fact_types(category, fact_subcats, short_name)
SELECT
''Housing'',
array_agg(s' || lpad(i::text,2,'0') || ') As fact_subcats,
' || quote_literal('s' || lpad(i::text,2,'0')) || ' As short_name
FROM staging.factfinder_import
WHERE s' || lpad(I::text,2,'0') || ' ~ ''^[a-zA-Z]+''' ', ';'
)
FROM generate_series(1,51) As I;
EXECUTE var_sql;
END
$$;
```



# CASE WHEN



```
SELECT student,
AVG(CASE WHEN subject = 'algebra' THEN score ELSE NULL END) As algebra,
AVG(CASE WHEN subject = 'physics' THEN score ELSE NULL END) As physics
FROM test_scores
GROUP BY student;
```



# FILTER



```
SELECT student,
AVG(score) FILTER (WHERE subject = 'algebra') As algebra,
AVG(score) FILTER (WHERE subject = 'physics') As physics
FROM test_scores
GROUP BY student;
```



# Windows



- Las funciones WINDOW son una característica de ANSI SQL incorporadas a PostgreSQL desde la versión 8.4
- Una ventana puede ver y usar datos más allá de la fila actual. De ahí el nombre windows o ventana
- Una ventana define que filas considerar además de la fila actual.
- Las ventanas permite agregar información a cada fila
- Existen funciones asociadas a Window como row\_number y rank que son útiles para ordenar y procesar los registros.



# Windows



```
SELECT tract_id, val, AVG(val) OVER () as val_avg
FROM census.facts
WHERE fact_type_id = 86;
```

| tract_id    | val      | val_avg               |
|-------------|----------|-----------------------|
| 25001010100 | 2942.000 | 4430.0602165087956698 |
| 25001010206 | 2750.000 | 4430.0602165087956698 |
| 25001010208 | 2003.000 | 4430.0602165087956698 |
| 25001010304 | 2421.000 | 4430.0602165087956698 |

- La cláusula **OVER** establece los límites de la ventana. En este ejemplo, ya que no hay restricciones entre los paréntesis, la ventana cubre todas las filas del **WHERE**. En este caso el promedio es sobre todas las filas con **fact\_type\_id = 86**.
- También se ha modificado la función tradicional **AVG** para transformarla en una función agregada de la ventana.
- Para cada fila, PostgreSQL despacha todas las filas para el cálculo del **AVG** agregado y entrega ese valor para cada fila.
- Se pueden utilizar todas las funciones SQL de agregación como funciones **window**, pero adicionalmente se pueden usar **ROW**, **RANK**, **LEAD**





# PARTITION BY

- Se puede utilizar una función window sobre filas que contengan un valor específico, en vez de toda la tabla
- Para esto existe PARTITION BY, que le indica a PostgreSQL solo ejecutar los aggregate sobre las filas indicadas
- El código del condado son las primeras 5 letras de la columna tract\_id.

```
SELECT tract_id, val, AVG(val) OVER (PARTITION BY left(tract_id,5)) As
val_avg_county
```

```
FROM census.facts WHERE fact_type_id = 2 ORDER BY tract_id;
```

```
tract_id | val | val_avg_county
```

```
-----+-----+-----
```

```
25001010100 | 1765.000 | 1709.9107142857142857
```

```
25001010206 | 1366.000 | 1709.9107142857142857
```

```
25001010208 | 984.000 | 1709.9107142857142857
```

```
:
```

```
25003900100 | 1920.000 | 1438.2307692307692308
```

```
25003900200 | 1968.000 | 1438.2307692307692308
```



# ORDER BY



- Las funciones Window también permiten un **ORDER BY** en la cláusula **OVER** clause
- Es decir ordena las filas de la ventana utilizando el **ORDER**
- Ejemplo, numeración:

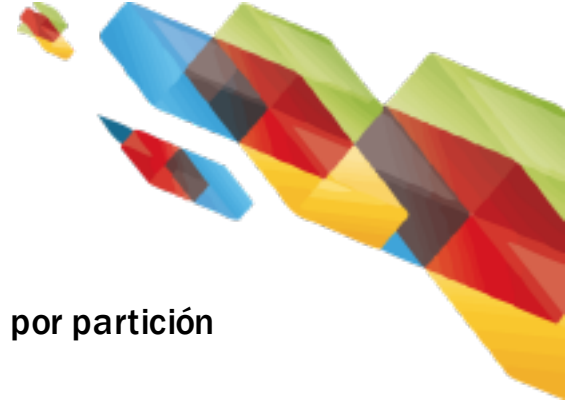
```
SELECT ROW_NUMBER() OVER (ORDER BY tract_name) As rnum, tract_name
FROM census.lu_tracts
ORDER BY rnum LIMIT 4;
```

```
rnum | tract_name
```

```
-----+-----
1 | Census Tract 1, Suffolk County, Massachusetts
2 | Census Tract 1001, Suffolk County, Massachusetts
3 | Census Tract 1002, Suffolk County, Massachusetts
4 | Census Tract 1003, Suffolk County, Massachusetts
```



# ORDER BY



- Se puede combinar **ORDER BY** con **PARTITION BY**, con lo cual el orden es por partición

```
SELECT tract_id, val,
SUM(val) OVER (PARTITION BY left(tract_id,5) ORDER BY val) As
sum_county_ordered
FROM census.facts
WHERE fact_type_id = 2
ORDER BY left(tract_id,5), val;
tract_id | val | sum_county_ordered
```

```
-----+-----+-----
25001014100 | 226.000 | 226.000
25001011700 | 971.000 | 1197.000
25001010208 | 984.000 | 2181.000
:
25003933200 | 564.000 | 564.000
25003934200 | 593.000 | 1157.000
25003931300 | 606.000 | 1763.000
```



# WINDOWS con nombre

```
SELECT * FROM (
SELECT
ROW_NUMBER() OVER(wt) As rnum,
substring(tract_id,1, 5) As county_code,
tract_id,
LAG(tract_id,2) OVER wt As tract_2_before,
LEAD(tract_id) OVER wt As tract_after
FROM census.lu_tracts
WINDOW wt AS (PARTITION BY substring(tract_id,1, 5) ORDER BY tract_id)
) As x
WHERE rnum BETWEEN 2 and 3 AND county_code IN ('25007','25025')
ORDER BY county_code, rnum;
```



# Common Table Expressions

- Esencialmente , las common table expressions (CTEs) permiten definir un query que puede ser reutilizado en un query más grande.
- Existen desde la versión 8.4 y en la versión 9.1 tuvieron la capacidad de escritura y de actuar como tablas temporales, es decir, son eliminadas una vez ejecutadas
- Existen tres maneras de utilizar CTEs:
  - **Basic CTE**
    - Uso normal de CTE, para hacer el código SQL más legible o para forzar al planner a materializar y calcular costos intermedios para una mejor performance.
  - **Writable CTE**
    - Es una extensión del CTE básico con UPDATE, INSERT, y DELETE. Además puede retornar las filas modificadas.
  - **Recursive CTE**
    - Esto permite utilizar un CTE de manera recursiva



# CTE Básico



```
WITH cte AS (
 SELECT
 tract_id, substring(tract_id,1, 5) As county_code,
 COUNT(*) OVER(PARTITION BY substring(tract_id,1, 5)) As cnt_tracts
 FROM census.lu_tracts
)
SELECT MAX(tract_id) As last_tract, county_code, cnt_tracts
FROM cte
WHERE cnt_tracts > 100
GROUP BY county_code, cnt_tracts;
```



# CTE Múltiple



```
WITH
cte1 AS (
SELECT
tract_id,
substring(tract_id,1, 5) As county_code,
COUNT(*) OVER (PARTITION BY substring(tract_id,1,5)) As cnt_tracts
FROM census.lu_tracts
),
cte2 AS (
SELECT
MAX(tract_id) As last_tract,
county_code,
cnt_tracts
FROM cte1
WHERE cnt_tracts < 8 GROUP BY county_code, cnt_tracts
)
SELECT c.last_tract, f.fact_type_id, f.val
FROM census.facts As f INNER JOIN cte2 c ON f.tract_id = c.last_tract;
```



# CTE Writable

```
WITH t AS (
DELETE FROM ONLY logs_2011 WHERE log_ts < '2011-03-01' RETURNING *
)
INSERT INTO logs_2011_01_02 SELECT * FROM t;
```





# CTE Recursive

- La documentación oficial dice lo siguiente: “El modificador opcional **RECURSIVE** cambia un CTE para realizar cosas que no son posibles con SQL estándar”
- Cambia el CTE para recursivamente combinar los resultados.
- Los resultados son combinados utilizando **UNION** o **UNION ALL**.
- **Sintáxis:**

```
WITH RECURSIVE nombre AS (
 -- valor inicial
 SELECT cols FROM tabla UNION ALL -- o UNION
 -- elemento recursivo: se apunta a sí mismo
 SELECT cols FROM nombre
 -- condición de terminación
 WHERE ...)
SELECT * FROM nombre;
```

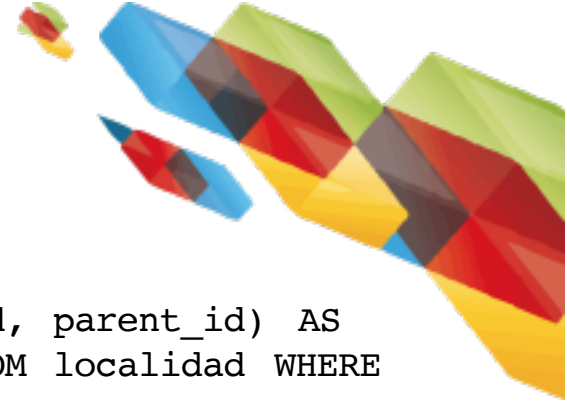


# CTE Recursive



```
CREATE TABLE localidad (id SERIAL PRIMARY KEY,nombreLocalidad
VARCHAR,parent_id INTEGER REFERENCES localidad(id));
```

```
INSERT INTO localidad (nombreLocalidad, parent_id) VALUES
('RM', NULL),
('Santiago', 1),
('Las Condes', 2),
('Avenida Kennedy', 3),
('Alto Las Condes', 3),
('Estadio Palestino', 3),
('Providencia', 2),
('Drugstore', 7),
('Paseo las palmas', 7);
```



# CTE Recursive

```
WITH RECURSIVE path(nombreLocalidad, path, parent, id, parent_id) AS
(SELECT nombreLocalidad, '/', NULL, id, parent_id FROM localidad WHERE
id = 1
UNION
SELECT localidad.nombreLocalidad, parentpath.path || CASE
parentpath.path WHEN '/' THEN " ELSE '/' END ||
localidad.nombreLocalidad,parentpath.path, localidad.id,
localidad.parent_idFROM localidad, path as parentpath
WHERE localidad.parent_id = parentpath.id)
SELECT * FROM path;
```



# Lateral Joins



- **LATERAL** es un comando ANSI SQL disponible desde la versión 9.3.
- **LATERAL** permite compartir datos de columnas entre tablas en cláusula from
- Veamos un ejemplo con error:

```
SELECT *
FROM
 census.facts L
 INNER JOIN
 (SELECT *
 FROM census.lu_fact_types
 WHERE category =
 CASE WHEN L.yr = 2011 THEN 'Housing' ELSE category END
) R
ON L.fact_type_id = R.fact_type_id;
```



# Lateral Joins

- Utilizando LATERAL es factible realizar ese query:

```
SELECT * FROM census.facts L INNER JOIN LATERAL
(SELECT * FROM census.lu_fact_types
WHERE category = CASE WHEN L.yr = 2011 THEN 'Housing' ELSE category END)
R
ON L.fact_type_id = R.fact_type_id;
```



# Lateral Join

- Lateral funciona de una sola manera :El lado derecho puede sacar datos del lado izquierdo pero no viceversa
- Otro ejemplo:

```
CREATE TABLE interval_periods(i_type interval);
INSERT INTO interval_periods (i_type)
VALUES ('5 months'), ('132 days'), ('4862 hours');
```

```
SELECT i_type, dt
FROM
interval_periods CROSS JOIN LATERAL
generate_series('2012-01-01'::date, '2012-12-31'::date, i_type) AS dt
WHERE NOT (dt = '2012-01-01' AND i_type = '132 days'::interval);
```



# Lateral



- Otro ejemplo, uso de **LATERAL** para limitar las filas de una tabla vinculada con JOIN

```
SELECT u.user_name, l.description, l.log_ts
FROM
super_users AS u CROSS JOIN LATERAL (
SELECT description, log_ts
FROM logs
WHERE
log_ts > CURRENT_TIMESTAMP - interval '100 days' AND
logs.user_name = u.user_name
ORDER BY log_ts DESC LIMIT 5
) AS l;
```