

PostgreSQL

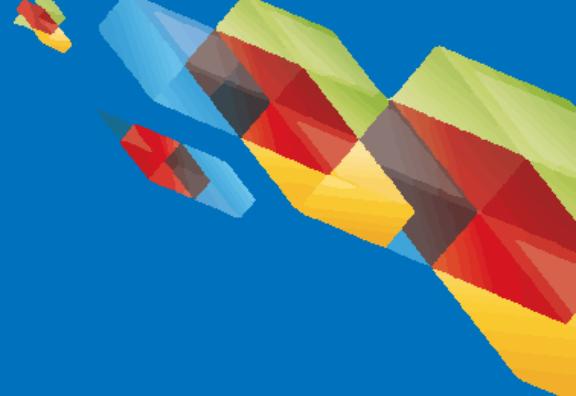
The world's most advanced
open source database.

PREVIRED
05/2016

EXE**E**

EXE**E**

www.exe.cl

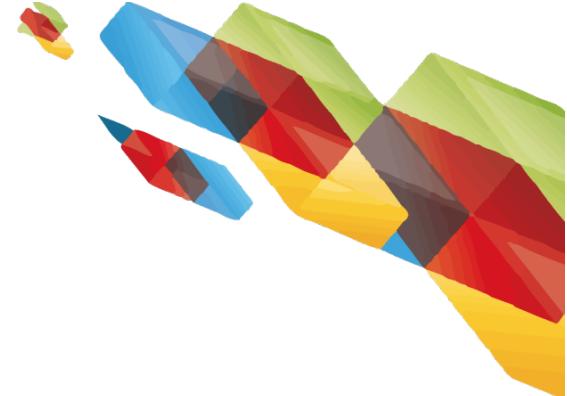


Tipos de datos





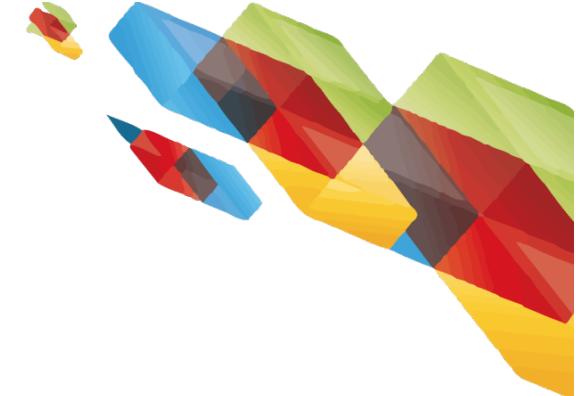
Introducción



- PostgreSQL soporta todos los tipos estándar de bases de datos
 - Numéricos
 - Strings
 - Fecha (Incluyendo tiempo)
 - Booleans
- Pero agrega soporte para
 - Arreglos
 - Fecha y hora con la zona de tiempo asociada
 - Intervalos de tiempos
 - Rangos
 - JSON
 - XML
- Y por supuesto crear tipos personalizados



Introducción



- Pero además de los tipos existen las funciones
 - Una función se entiende como algo de la forma $f(x)$.
- Y los operadores
 - Existen operadores unitarios (Solo un argumento)
 - O binarios (dos argumentos), como por ejemplo +, -, *, /.
 - Los operadores pueden ser alias de funciones
 - En algunos casos el operador, "opera" de manera distinta de acuerdo al tipo. Por ejemplo el operador +, significa sumar para tipos numéricos, pero en el caso de rangos, significa unión.



Numéricos



- Los tipos numéricos consisten en enteros de 2, 4 y 8 bytes, números de punto flotante de 4 y 8 bytes, y números de precisión fija

Nombre	Tamaño de almacenamiento	Descripción	Rango
smallint	2 bytes	Entero pequeño	-32768 a +32767
integer	4 bytes	Opción más usada para enteros	-2147483648 a +2147483647
bigint	8 bytes	Enteros grandes	-9223372036854775808 a +9223372036854775807
decimal	variable	Precisión exacta definida por el usuario	Hasta 131072 dígitos antes de los decimales, hasta 16383 dígitos en la parte decimal.
numeric	variable	Precisión exacta definida por el usuario	Hasta 131072 dígitos antes de los decimales, hasta 16383 dígitos en la parte decimal.
real	4 bytes	Precisión variable, inexactos	Precisión de 6 dígitos decimales
double precision	8 bytes	Precisión variable, inexactos	Precisión de 15 dígitos decimales
smallserial	2 bytes	Entero pequeño pero con autoincremento	1 a 32767
serial	4 bytes	Entero pero con autoincremento	1 a 2147483647
bigserial	8 bytes	Entero grande pero con autoincremento	1 a 9223372036854775807



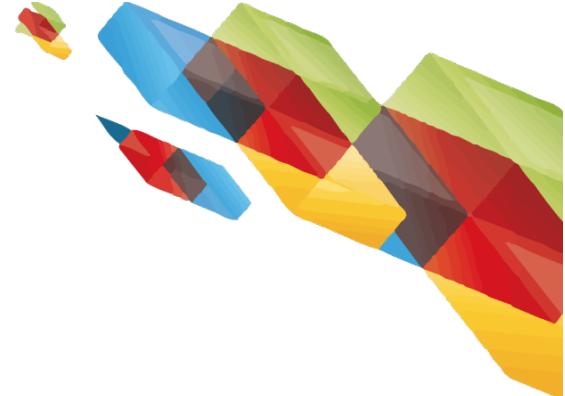
Numéricos



- **SMALLINT, INTEGER y BIGINT**
- Valores fuera de rango arrojan error
- Integer es la opción más utilizada.
 - Mejor balance entre rango, almacenamiento y performance
- Integer es equivalente a int
- También se puede utilizar int2, int4 e int8 (Están por compatibilidad)



Numéricos



- **NUMERIC(precisión, escala)**
 - Precisión es el total de dígitos (Enteros y decimales)
 - Escala es el total de dígitos decimales
 - Ejemplo 2116.33 es un número con precisión 6 y escala 2
- Las operaciones matemáticas son más lentas en comparación con los tipos enteros o los decimales de precisión variable
- Típicamente se usan para almacenar dinero.
- La precisión máxima es 1000
- Si no se indica la escala por omisión es 0.
- Cada grupo de 4 decimales es almacenado en 2 bytes, más tres bytes adicionales.
- **NaN**
 - Not a Number
 - Toda operación sobre un valor NaN devuelve NaN



Numéricos



- **REAL y DOUBLE**

- Precisión variable
 - Son implementaciones de IEEE Standard 754 for Binary Floating-Point Arithmetic (single and double precision)

- **REAL va de 1E-37 a 1E+37 con una precisión de al menos seis dígitos**

- **DOUBLE va de 1E-307 a 1E307 con una precisión de 15 dígitos**

- **Valores especiales**

- Infinity y -Infinity
 - NaN

```
Update tabla set columna='Infinity'
```

- **Consideraciones**

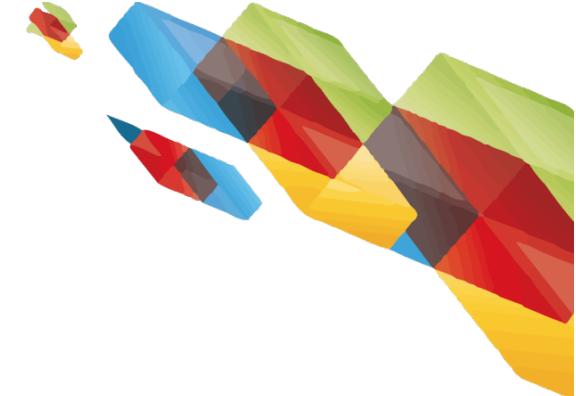
- Para efectos de ordenar NaN es mayor o igual que todo valor que no sea a NaN

- **Para efectos de compatibilidad se acepta float(p), donde p es la precisión**

- 1 a 24 es REAL
 - 25 a 53 DOUBLE
 - Si no se indica p, es DOUBLE



Numéricos



- **SMALLSERIAL, SERIAL y BIGSERIAL**
- Se autoincrementan
- Generan secuencias
- Al momento de agregar una columna SERIAL:

```
CREATE TABLE tabla( columna SERIAL );
```

– Es equivalente a:

```
CREATE SEQUENCE tabla_columna_seq;
```

```
CREATE TABLE tabla( columna integer NOT NULL DEFAULT
nextval('tabla_columna_seq'));
```

```
ALTER SEQUENCE tabla_columna_seq OWNED BY tabla.columna;
```

- La restricción de llave primaria no se agrega automáticamente
- Notar el uso de OWNED para que al momento de eliminar la columna se elimine la secuencia.
- Siempre existe la posibilidad de que hayan saltos de valores
 - Por ejemplo si al insertar se hace un rollback, ese valor es “saltado”
- Para que se asigne el valor de la secuencia la columna no debe ser incluida en el INSERT o bien se le debe asignar el valor DEFAULT
- Se pueden usar los alias serial2, serial4 y serial8



Sobre las secuencias

- Las secuencias pueden ser utilizadas en más de una columna
- Se pueden modificar utilizando ALTER SEQUENCE
- Y se puede establecer los valores de límites (Mínimo y máximo), así como también su incremento
- Es factible tener un incremento negativo (Secuencia va de mayor a menor)
- Para obtener el siguiente valor de una secuencia se utiliza nextval(nombre_secuencia)
- Un dato si la tabla es renombrada, la secuencia queda con el nombre original y no se cambia su nombre
- Existe la posibilidad de generar series, utilizando la función generate_series
- Toma tres parámetros, valor inicial, máximo e incremento

```
SELECT x FROM generate_series(1,51,13) As x;  
x  
----  
1  
14  
27  
40
```



Sobre las secuencias



- Las secuencias pueden ser utilizadas en más de una columna
- Se pueden modificar utilizando ALTER SEQUENCE
- Y se puede establecer los valores de límites (Mínimo y máximo), así como también su incremento
- Es factible tener un incremento negativo (Secuencia va de mayor a menor)
- Para obtener el siguiente valor de una secuencia se utiliza next_val(nombre_secuencia)
- Un dato si la tabla es renombrada, la secuencia queda con el nombre original y no se cambia su nombre
- Existe la posibilidad de generar series, utilizando la función generate_series
- Toma tres parámetros, valor inicial, máximo e incremento

```
SELECT x FROM generate_series(1,51,13) As x;  
x  
----  
1  
14  
27  
40
```



Moneda

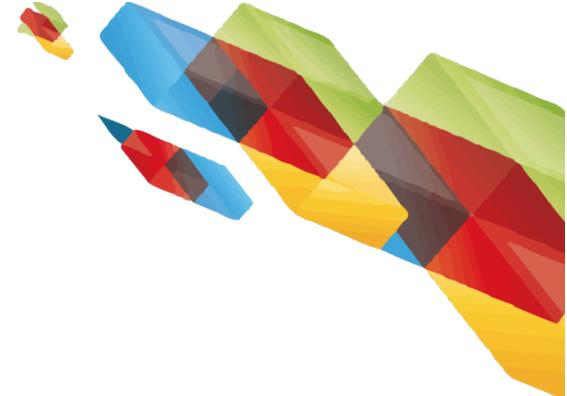


- Money
- Tamaño 8 Bytes, con un rango -92233720368547758.08 a +92233720368547758.07
- El formato es de moneda y depende de la configuración local
 - Se debe tener en cuenta al momento de realizar operaciones de backup y restore
- Conversiones

```
SELECT '12.34'::float8::numeric::money;  
SELECT '52093.89'::money::numeric::float8;
```



Carácteres

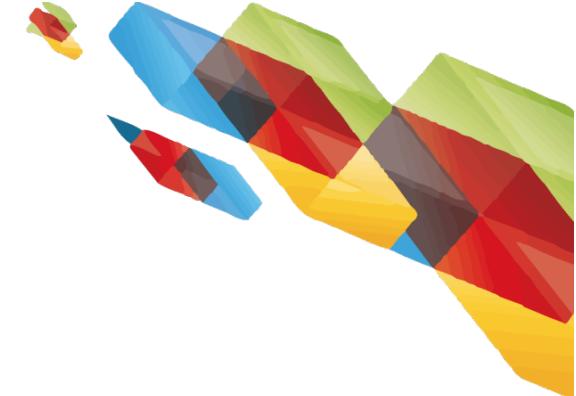


- Los tipos de caractéres almacenan “n” caracteres.
- Es importante entender que se almacenan caracteres y no bytes
- Almacenar un string de largo superior al que tiene el campo produce un error, a menos que todos los caracteres que sobran sean espacios.
- varchar es el alias para character varying y character para char

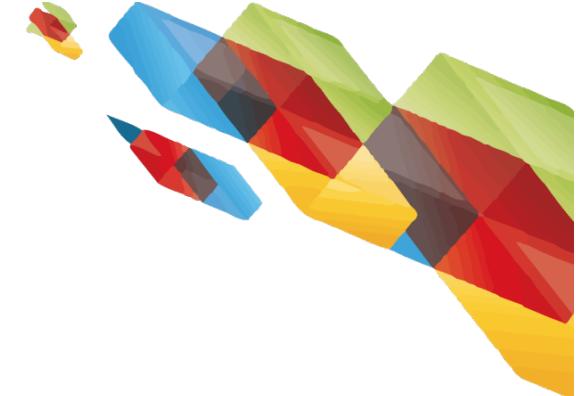
Nombre	Descripción
character varying(n), varchar(n)	Longitud variable con límite
character(n), char(n)	Flargo fijo, relleno con blancos
text	Variable sin límite



Carácteres



- Los strings de hasta **126 bytes**, ocupan **1 byte** adicional a su tamaño
- Strings más grandes ocupan **4 bytes** adicionales
- Strings muy largos son almacenados de manera diferenciada para no hacer lento el acceso a otras columnas
- El tamaño máximo de un String es de **1GB**
- No hay diferencias de performance entre los tres tipos.
 - Eso no sucede en otras bases de datos
 - En PostgreSQL usualmente el character(n) es el más lento de los tres.



Carácteres

```
CREATE TABLE test1 (a character(4));
INSERT INTO test1 VALUES ('ok');
SELECT a, char_length(a) FROM test1;
a    | char_length
-----+-----
ok   |      2
CREATE TABLE test2 (b varchar(5));
INSERT INTO test2 VALUES ('ok');
INSERT INTO test2 VALUES ('good      ');
INSERT INTO test2 VALUES ('too long');
ERROR: value too long for type character varying(5)
INSERT INTO test2 VALUES ('too long'::varchar(5)); -- truncación explícita
SELECT b, char_length(b) FROM test2;
b    | char_length
-----+-----
ok   |      2
good  |      5
too l |      5
```



Carácteres



- Tipos especiales

- char

- 1 Byte

- name

- 64 bytes

- Uso interno para nombres de objetos

- Escape

```
SELECT E'Esta es la primera parte \n y esta es la segunda';
```



Carácteres

- Algunas funciones

```
SELECT lpad('ab', 4, '0') As ab_lpad,
       rpad('ab', 4, '0') As ab_rpad,
       lpad('abcde', 4, '0') As ab_lpad_trunc;
ab_lpad | ab_rpad | ab_lpad_trunc
-----+-----+-----
00ab | ab00 | abcd
```

```
SELECT
    a As a_before, trim(a) As a_trim, rtrim(a) As a_rt,
    i As i_before, ltrim(i, '0') As i_lt_0,
    rtrim(i, '0') As i_rt_0, trim(i, '0') As i_t_0
  FROM ( SELECT repeat(' ', 4) || i || repeat(' ', 4) As a, '0' || i
    As i FROM generate_series(0, 200, 50) As i) As x;
a_before | a_trim | a_rt | i_before | i_lt_0 | i_rt_0 | i_t_0
-----+-----+-----+-----+-----+-----+-----+
0      | 0      | 0      | 00      |      |      |
50     | 50     | 50     | 050     | 50    | 05    | 5
100    | 100    | 100    | 0100    | 100   | 01    | 1
150    | 150    | 150    | 0150    | 150   | 015   | 15
200    | 200    | 200    | 0200    | 200   | 02    | 2
```



Carácteres



- Algunas funciones

```
SELECT split_part('abc.123.z45', '.', 2) As x;  
x  
-----  
123
```

- Unnest (Recordar para arreglos)

```
SELECT unnest(string_to_array('abc.123.z45', '.')) As x;  
x  
-----  
abc  
123  
z45
```



Carácteres



- **Expresiones regulares**

```
SELECT regexp_replace(  
    '6197306254',  
    '([0-9]{3})([0-9]{3})([0-9]{4})',  
    E'\\1 \\2-\\3'  
) AS x;  
  
-----  
(619) 730-6254
```

```
SELECT unnest(regexp_matches( 'Celular (619)852-5083. Casa 619-730-  
6254. Bésame mucho. ',  
    E'[(][0,1][0-9]{3}[ )- .]{0,1}[0-9]{3}[ - .]{0,1}[0-9]{4}' , 'g')  
) AS x;  
  
-----  
(619)852-5083  
619-730-6254
```



Binarios



- **BYTEA**
- Permite almacenar Strings binarios
- Un String binario es una secuencia de octetos.
 - Permite cualquier valor
 - En los Strings de caracteres solo se permiten secuencias de caracteres, donde cada carácter depende del formato (Ej. UTF-8)
 - Se pueden pensar como raw bytes
- Es equivalente a los campos BLOB (BINARY LARGE OBJECT).
- Para asignar valores se puede utilizar el formato HEX
 - 2 dígitos hexadecimales por byte
 - Los más significativos van primero
 - La secuencia es precedida por \x

```
SELECT E'\xDEADBEEF';
```
- Otro formato es el Escape
 - Representa el String como una secuencia de caracteres ASCII
 - La secuencia es precedida por \\

```
SELECT E'\\176\\176'::bytea;
```



Fecha y Hora



Nombre	Tamaño	Descripción	Mínimo	Máximo	Resolución
timestamp [(p)] [without time zone]	8 bytes	Fecha y hora sin zona horaria	4713 AC	294276 DC	1 microsegundo / 14 dígitos
timestamp [(p)] with time zone	8 bytes	Fecha y hora con zona horaria	4713 AC	294276 DC	1 microsegundo / 14 dígitos
date	4 bytes	Solo fecha	4713 AC	5874897 DC	1 día
time [(p)] [without time zone]	8 bytes	Solo hora	00:00:00	24:00:00	1 microsegundo / 14 dígitos
time [(p)] with time zone	12 bytes	Horas del día con zona horaria	00:00:00+1459	24:00:00-1459	1 microsegundo / 14 dígitos
interval [fields] [(p)]	16 bytes	Intervalos de tiempo	-178000000 years	178000000 años	1 microsegundo / 14 dígitos



Fecha y hora

- timestamp es equivalente a timestamp without time zone
- timestampz es un alias para timestamp with time zone
- P indica la precisión de la fracción de segundos (0 a 6 con almacenamiento entero y 0 a 10 con almacenamiento de punto flotante)
- Para el intervalo estas opciones están disponibles
 - YEAR
 - MONTH
 - DAY
 - HOUR
 - MINUTE
 - SECOND
 - YEAR TO MONTH
 - DAY TO HOUR
 - DAY TO MINUTE
 - DAY TO SECOND
 - HOUR TO MINUTE
 - HOUR TO SECOND
 - MINUTE TO SECOND



Fecha y hora



Ejemplo	Descripción
1999-01-08	ISO 8601; 8 de enero (Formato recomendado)
January 8, 1999	Modo no ambiguo
1/8/1999	Enero 8 en MDY ; Agosto 1 en DMY
1/18/1999	Enero 18 en MDY mode; rechazado en otros modos
01/02/03	January 2, 2003 in MDY mode; February 1, 2003 in DMY mode; February 3, 2001 in YMD mode
1999-Jan-08	Enero 8 en cualquier modo
Jan-08-1999	Enero 8 en cualquier modo
08-Jan-1999	Enero 8 en cualquier modo
99-Jan-08	Enero 8 en YMD mode, sino error
08-Jan-99	Enero 8 a excepción de YMD que arroja error
Jan-08-99	Enero 8 a excepción de YMD que arroja error
19990108	ISO 8601; Enero 8 de 1999 en cualquier modo
990108	ISO 8601; Enero 8 de 1999 en cualquier modo
1999.008	Año y día del año
J2451187	Fecha en formato de Calendario Juliano
January 8, 99 BC	Año 99 antes de Cristo



Fecha y hora



Ejemplo	Descripción
04:05:06.789	ISO 8601
04:05:06	ISO 8601
04:05	ISO 8601
040506	ISO 8601
04:05 AM	Igual que 04:05 AM no afecta al valor
04:05 PM	Igual que 16:05; la hora debe ser <= 12
04:05:06.789-8	ISO 8601
04:05:06-08:00	ISO 8601
04:05-08:00	ISO 8601
040506-08	ISO 8601
04:05:06 PST	Zona horaria especificada por abreviación
2003-04-12 04:05:06 America/New_York	Zona horaria con nombre completo



Fecha y hora

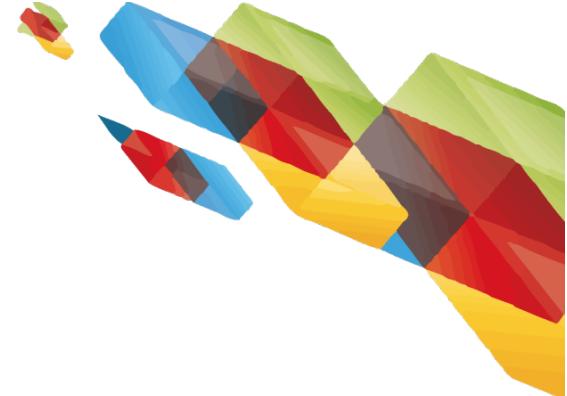


- Textos

Ejemplo	Descripción
PST	Abreviación (Pacific Standard Time)
America/New_York	Nombre de la zona horaria
PST8PDT	POSIX-style para la zona horaria
-8:00	ISO-8601 offset para PST
-800	ISO-8601 offset para PST
-8	ISO-8601 offset para PST
zulu	Abreviación militar para UTC
z	Abreviación de zulu



Fecha y hora



- **Timestamps**

- 1999-01-08 04:05:06
- 1999-01-08 04:05:06 -8:00
- January 8 04:05:06 1999 PST
- TIMESTAMP '2004-10-19 10:23:54'
- TIMESTAMP '2004-10-19 10:23:54+02' --no funciona en PostgreSQL
- TIMESTAMP WITH TIME ZONE '2004-10-19 10:23:54+02' –esto si



Fecha y hora



- Valores especiales

Valor	Tipos válidos	Descripción
epoch	date, timestamp	1970-01-01 00:00:00+00 (Unix system time zero)
infinity	date, timestamp	Mayor a cualquier timestamp
-infinity	date, timestamp	Menor a cualquier timestamp
now	date, time, timestamp	Fecha de inicio de la transacción
today	date, timestamp	Hoy a la media noche
tomorrow	date, timestamp	Mañana a la medianoche
yesterday	date, timestamp	Ayer a la medianoche
allballs	time	00:00:00.00 UTC



Fecha y hora



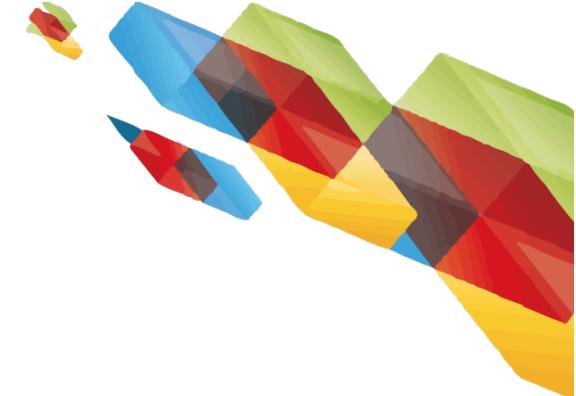
Estilo	Descripción	Ejemplo
ISO	ISO 8601, Estándar SQL	1997-12-17 07:37:16-08
SQL	Estilo tradicional	12/17/1997 07:37:16.00 PST
Postgres	Estilo original	Wed Dec 17 07:37:16 1997 PST
German	Estilo regional	17.12.1997 07:37:16.00 PST



Fecha y hora



Valor de datestyle	Orden	Ejemplo
SQL, DMY	Día/Mes/Año	17/12/1997 15:37:16.00 CET
SQL, MDY	Mes/Día/Año	12/17/1997 07:37:16.00 PST
Postgres, DMY	Día/Mes/Año	Wed 17 Dec 07:37:16 1997 PST



Fecha y hora

- **INTERVALOS**

- **cantidad unidad [cantidad unidad...] [dirección]**

- Cantidad es un número
- Unidad es microsecond, millisecond, second, minute, hour, day, week, month, year, decade, century, millennium o sus abreviaturas
- Dirección puede ser ago o no ir

- **Cantidades de días, horas, minutos y segundos no requieren de marcado explícito**

- ‘1 12:59:10’ es lo mismo que
- ‘1 day 12 hours 59 min 10 sec’

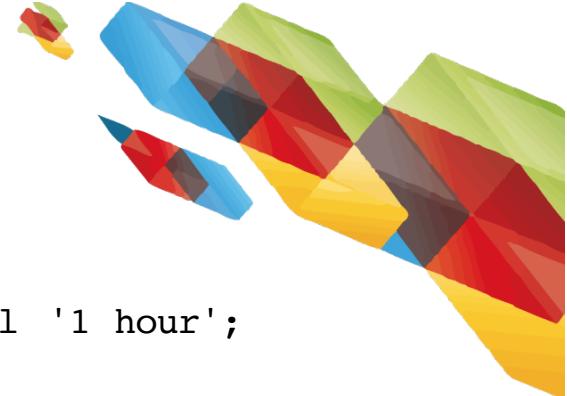
- **Combinaciones de años y meses se colocan con guión**

- ‘200-10’ es lo mismo que
- ‘200 years 10 months’

Abreviación	Significado
Y	Años
M	Meses (En la parte de fecha)
W	Semanas
D	Días
H	Horas
M	Minutos (en la parte tiempo)
S	Segundos



Fecha y hora



•SELECT

```
SELECT '2012-02-10 11:00 PM'::timestamp + interval '1 hour';
2012-02-11 00:00:00
```

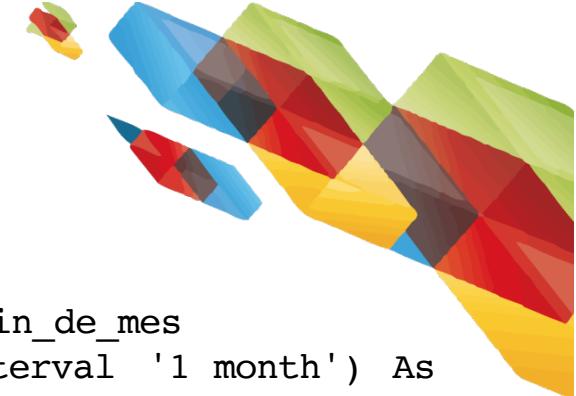
```
SELECT '23 hours 20 minutes'::interval + '1 hour'::interval;
24:20:00
```

```
SELECT '2012-02-10 11:00 PM'::timestamptz - interval '1 hour';
2012-02-10 22:00:00-05
```

```
SELECT ('2012-10-25 10:00 AM'::timestamp, '2012-10-25 2:00
PM'::timestamp) OVERLAPS
('2012-10-25 11:00 AM'::timestamp, '2012-10-26 2:00 PM'::timestamp)
AS x,
('2012-10-25'::date, '2012-10-26'::date) OVERLAPS
('2012-10-26'::date, '2012-10-27'::date)
AS y;
x | y
---+---
t | f
```



Fecha y hora



•SELECT

```
SELECT (primer_dia - interval '1 day')::date As fin_de_mes
FROM generate_series('2/1/2012', '6/30/2012', interval '1 month') As
primer_dia;
eom
-----
2012-01-31
2012-02-29
2012-03-31
2012-04-30
2012-05-31
```



Fecha y hora



•SELECT

```
SELECT intervalos, date_part('hour',intervalos) As hora,
       to_char(intervalos, 'HH12:MI AM') As hora_formato
  FROM
 generate_series( '2012-03-11 12:30 AM', '2012-03-11 3:00 AM',
 interval '15 minutes'
 ) As intervalos;
```

intervalos	hora	hora_formato
2012-03-11 00:30:00-05	0	12:30 AM
2012-03-11 00:45:00-05	0	12:45 AM
2012-03-11 01:00:00-05	1	01:00 AM
2012-03-11 01:15:00-05	1	01:15 AM
2012-03-11 01:30:00-05	1	01:30 AM
2012-03-11 01:45:00-05	1	01:45 AM
2012-03-11 03:00:00-04	3	03:00 AM



Boolean



- BOOLEAN
- 1 byte
- Verdadero o Falso
- Verdadero
 - TRUE
 - 't'
 - 'true'
 - 'y'
 - 'yes'
 - 'on'
 - '1'
- Falso
 - FALSE
 - 'f'
 - 'false'
 - 'n'
 - 'no'
 - 'off'
 - '0'

```
CREATE TABLE test1 (a boolean, b text);
INSERT INTO test1 VALUES (TRUE, 'sic
est');
INSERT INTO test1 VALUES (FALSE, 'non
est');
SELECT * FROM test1;
a | b
---+-----
t | sic est
f | non est
SELECT * FROM test1 WHERE a;
a | b
---+-----
t | sic est
```



Tipos enumerados

- **ENUM**

```
CREATE TYPE animo AS ENUM ('triste', 'ok', 'feliz');
CREATE TABLE persona ( nombretext, estado_animo animo );
    INSERT INTO persona VALUES ('EDUARDO', 'feliz');
SELECT * FROM persona WHERE estado_animo ='feliz';
nombre| estado_animo
-----+-----
Eduardo | feliz
```

- El orden está dado por el del listado de valores. En este caso 'triste' < 'ok' < 'feliz'

- Los valores son case sensitive

- Un enum ocupa 4 bytes en disco

- El string asociado a un enum es de máximo 63 bytes

- Cada tipo enumerado es distinto a otro
 - No se pueden comparar directamente aunque los valores sean los mismos
 - Si se puede utilizar el operador de cast y compararlos como texto
 - persona.estado_animo::text==dia.tipo_recuerdo::text



Arreglos



- ARREGLOS**

```
CREATE TABLE pagos_empleado (
    nombre text,
    pago_mensual integer[],
    agenda text[][]);
```

- Los [] determinan que el tipo de columna es un arreglo. En el caso de [][] es una matriz o arreglo de dos dimensiones

- Es factible definir el tamaño exacto del arreglo

```
CREATE TABLE ajedrez( partida text, cuadro integer[8][8] );
```

- El tamaño del arreglo es solo para efectos de documentación.

- Es factible usar la notación estándar SQL

- Pago_mensual integer ARRAY[12],

- Los valores se representan a lo JSON

- '{ val1 delim val2 delim ... }'

- '{{1,2,3},{4,5,6},{7,8,9}}'

- '{"1","2","3"}, {"4","5","6"}, {"7","8","9"}'

- Si el arreglo es multidimensional al momento de insertar se debe ingresar los valores de cada dimensión



Arreglos



- **ARREGLOS**

```
INSERT INTO sal_emp      VALUES ('Bill',
ARRAY[10000, 10000, 10000, 10000],
ARRAY[['meeting', 'lunch'], ['training', 'presentation']]);
INSERT INTO sal_emp      VALUES ('Carol',
ARRAY[20000, 25000, 25000, 25000],
ARRAY[['breakfast', 'consulting'], ['meeting', 'lunch']]);
```

- **Los arreglos parten del índice 1**

```
SELECT cuadro[1][1] from ajedrez;
```

- **Los arreglos parten del índice 1**

- Es factible obtener partes del arreglo

```
SELECT cuadro[2:4][5:8] from ajedrez;
```

- **Si cualquier dimensión se obtiene un parte del arreglo todo es tratado como partes**

```
SELECT cuadro[2][1:1] from ajedrez; --es lo mismo que
```

```
SELECT cuadro[1:2][1:1] from ajedrez;
```

- **Para obtener las dimensiones**

```
Select array_dims(cuadro) from ajedrez where partida="capablanca vs kasparov"
```

```
array_dims
```

```
-----
```

```
[1:8][1:8]
```



Arreglos

•ARREGLOS

```
insert into ajedrez values ('p1','{{1,2,3},{1,2,3}}')
select array_upper(cuadro,1) from ajedrez; --2
select array_upper(cuadro,2) from ajedrez; --3
select array_lower(cuadro,1) from ajedrez; --1
Select cardinality(cuadro) frim ajedrez; --6
```

•Update

```
Update ajedrez set cuadro='{{1,2,3},{1,2,3}}';
Update ajedrez set cuadro=ARRAY[[1,2,3], [1,2,3]];
Update ajedrez set cuadro[1][1]=8;
Update ajedrez set cuadro[2:2][2:3] = '{{4,5}}'
```

•Concatenación

```
SELECT ARRAY[1,2] || ARRAY[3,4];
```

```
?column?
```

```
-----
```

```
{1,2,3,4}
```

```
(1 row)
```

```
SELECT ARRAY[5,6] || ARRAY[[1,2],[3,4]];
```

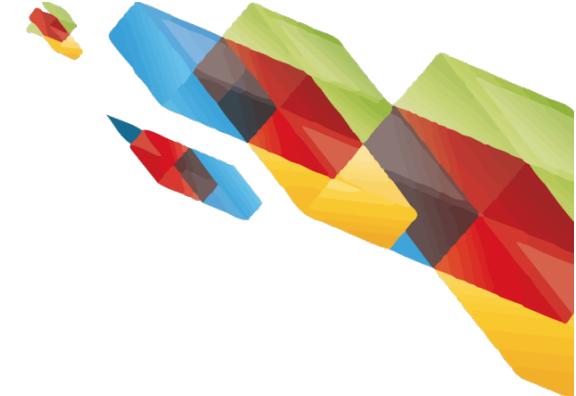
```
?column?-----
```

```
{{5,6},{1,2},{3,4}}
```

```
(1 row)
```



Arreglos



- Es factible agregar elementos a un arreglo

```
SELECT array_dims(1 || '[0:1]={2,3}'::int[]);
```

```
array_dims
```

```
-----
```

```
[0:2]
```

```
(1 row)
```

```
SELECT array_dims(ARRAY[1,2] || 3);
```

```
array_dims
```

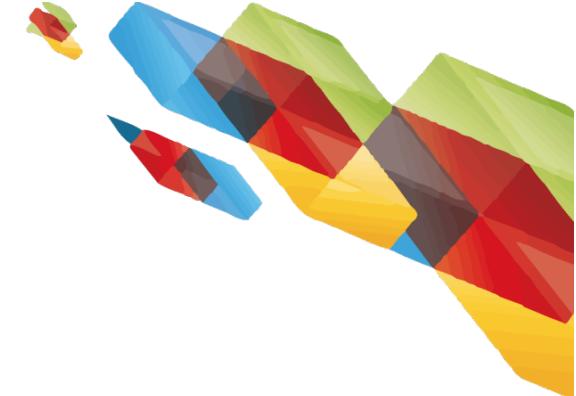
```
-----
```

```
[1:3]
```

```
(1 row)
```



Arreglos



- Es factible agregar elementos a un arreglo

```
SELECT array_dims(ARRAY[1,2] || ARRAY[3,4,5]);
```

```
array_dims
```

```
-----
```

```
[1:5]
```

```
(1 row)
```

```
SELECT array_dims(ARRAY[[1,2],[3,4]] || ARRAY[[5,6],[7,8],[9,0]]);
```

```
array_dims
```

```
-----
```

```
[1:5][1:2]
```

```
(1 row)
```

```
SELECT array_dims(ARRAY[1,2] || ARRAY[[3,4],[5,6]]);
```

```
array_dims
```

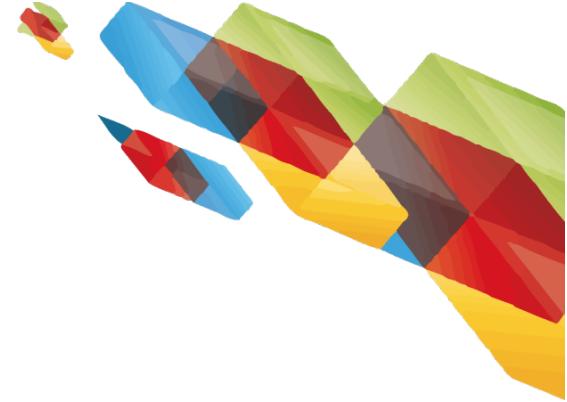
```
-----
```

```
[1:3][1:2]
```

```
(1 row)
```



Arreglos

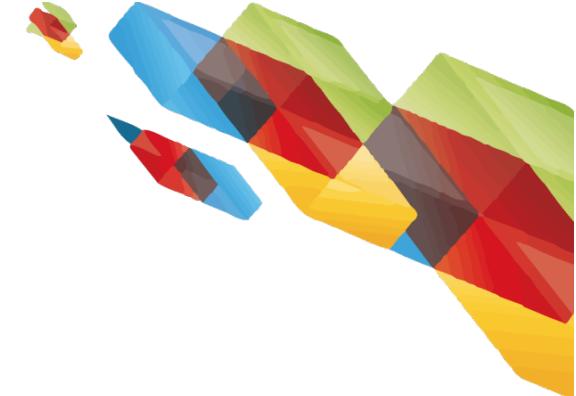


- **Algunas funciones**

```
SELECT array_prepend(1,  ARRAY[2,3]);
array_prepend
-----
{1,2,3}
(1 row)
SELECT array_append(ARRAY[1,2],  3);
array_append
-----
{1,2,3}
(1 row)
SELECT array_cat(ARRAY[1,2],  ARRAY[3,4]);
array_cat
-----
{1,2,3,4}
(1 row)
SELECT array_cat(ARRAY[[1,2],[3,4]],  ARRAY[5,6]);
array_cat
-----
{{1,2},{3,4},{5,6}}
(1 row)
SELECT array_cat(ARRAY[5,6],  ARRAY[[1,2],[3,4]]);
array_cat
-----
{{5,6},{1,2},{3,4}}
```



Arreglos



- Algunas funciones

```
SELECT ARRAY[1, 2] || '{3, 4}';
```

```
?column?
```

```
-----
```

```
{1,2,3,4}
```

```
SELECT ARRAY[1, 2] || '7';
```

```
ERROR: malformed array literal: "7"
```

```
SELECT ARRAY[1, 2] || NULL;
```

```
?column?
```

```
-----
```

```
{1,2}
```

```
(1 row)
```

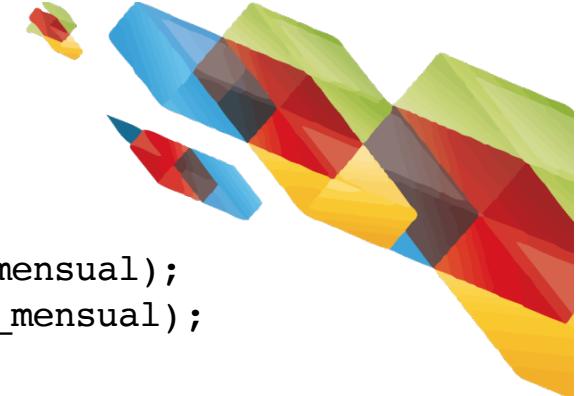
```
SELECT array_append(ARRAY[1, 2], NULL);
```

```
-----
```

```
{1,2,NULL}
```



Arreglos



•SELECT

```
SELECT * FROM empleado WHERE 1500000= ANY ( pago_mensual );
SELECT * FROM empleado WHERE 1500000 = ALL ( pago_mensual );
```

•generate_subscripts

```
SELECT * FROM (
    SELECT pago_mensual, generate_subscripts(pago_mensual, 1)
    AS p FROM empleado ) AS foo
WHERE pago_mensual[ p ] = 10000;
```

•array_position

```
SELECT
array_position(ARRAY[lun','mar','mie','jue','vie','sab','dom'], 'lun');
```

```
array_positions
```

```
-----
```

```
1
```

```
SELECT array_positions(ARRAY[ 1, 4, 3, 1, 3, 4, 2, 1 ], 1);
```

```
array_positions
```

```
-----
```

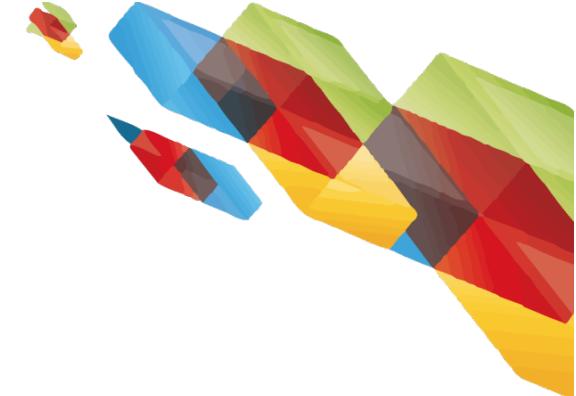
```
{1,4,8}
```

•&&

```
SELECT * FROM empleado WHERE pago_mensual && ARRAY[1500000];
```



Rangos



- **int4range** – Rango de integer
- **int8range** – Rango de bigint
- **numrange** – Rango de numeric
- **tsrange** – Rango de imestamp without time zone
- **tstzrange** – Rango de timestamp with time zone
- **daterange** – Rango de date

```
CREATE TABLE reserva(pieza int, durante tsrange);
INSERT INTO reserva VALUES (1108, '[2010-01-01 14:30, 2010-01-01
15:30)');
```

- – Contiene

```
SELECT int4range(10, 20) @> 3;
```

- – ¿Se sobreponen?

```
SELECT numrange(11.1, 22.2) && numrange(20.0, 30.0);
```

- – Extraer el límite superior

```
SELECT upper(int8range(15, 25));
```

- – Obtener la intersección

```
SELECT int4range(10, 20) * int4range(15, 25);
```

- – ¿Esta el rango vacío?

```
SELECT isempty(numrange(1, 5));
```



Rangos



- Select

- Incluir al 3, pero no al 7

```
SELECT '[3,7)::int4range;
```

- No incluir al 3 y al 7

```
SELECT '(3,7)::int4range;
```

- Incluye solo el 4

```
SELECT '[4,4)::int4range
```

- No incluye a nada, y es equivalente a empty

```
SELECT '[4,4)::int4range;
```

- Forma completa: límite inferior, límite superior, y texto que indica si se deben incluir los límites.

```
SELECT numrange(1.0, 14.0, '[]');
```

- Si se omite el tercer argumento, se asume '[]'.

```
SELECT numrange(1.0, 14.0);
```

- El uso de NULL permite crear un rango sin límite inferior o superior.

```
SELECT numrange(NULL, 2.2);
```



Object Id



- Son usados internamente por PostgreSQL
- No se pueden agregar a tablas creadas por el usuarios, a menos que al momento de crear la tabla se use la opción WITH OIDS.
- Es un entero de 4 bytes sin signo. No es apto para bases de datos grandes o para tablas grandes.
- No se debe usar como llave primaria



JSON



- Los tipos de datos JSON permiten almacenar datos JSON (JavaScript Object Notation), como dicta la especificación RFC 7159.
- Si bien es cierto pueden ser almacenados como texto (text), los tipos JSON validan que la data cumpla con las reglas de un documento JSON

```
Select '5'::json;
SELECT '[1, 2, "foo", null]'::json;
SELECT '{"bar": "baz", "balance": 7.77, "active": false}'::json
SELECT '{"foo": [true, "bar"], "tags": {"a": 1, "b": null}}'::json;
```

- Los tipos son json y jsonb

- Principal diferencia es la eficiencia
- Json almacena una copia exacta, preservando espacios en blanco que no son significativos
- Jsonb transforma el texto a un formato binario
- Más lento al insertar pero más eficiente para la recuperación y en espacio.
- En general se usa jsonb



JSON



- Los tipos de datos JSON permiten almacenar datos JSON (JavaScript Object Notation), como dicta la especificación RFC 7159.
- Si bien es cierto pueden ser almacenados como texto (text), los tipos JSON validan que la data cumpla con las reglas de un documento JSON

```
Select '5'::json;
SELECT '[1, 2, "foo", null]'::json;
SELECT '{"bar": "baz", "balance": 7.77, "active": false}'::json
SELECT '{"foo": [true, "bar"], "tags": {"a": 1, "b": null}}'::json;
```

- Los tipos son json y jsonb

– Principal diferencia es la eficiencia. Json almacena una copia exacta, preservando espacios en blanco que no son significativos. Jsonb transforma el texto a un formato binario. Más lento al insertar pero más eficiente para la recuperación y en espacio. En general se usa jsonb

```
SELECT '{"bar": "baz", "balance": 7.77, "active":false}'::json;
```

Json

```
{ "bar": "baz", "balance": 7.77, "active":false}
```

(1 row)

```
SELECT '{"bar": "baz", "balance": 7.77, "active":false}'::jsonb;
```

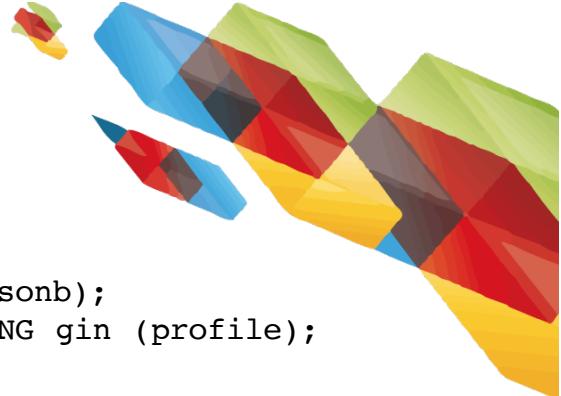
Jsonb

```
{ "bar": "baz", "active": false, "balance": 7.77}
```

(1 row)



JSON



•Inserción

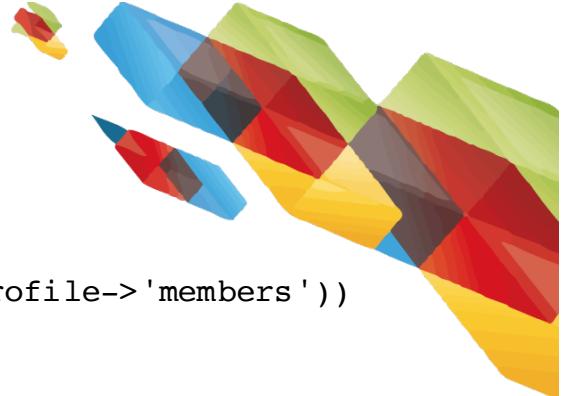
```
CREATE TABLE families_j (id serial PRIMARY KEY, profile jsonb);
CREATE INDEX idx_families_jb_profile_gin ON families_b USING gin (profile);
INSERT INTO families_j (profile) VALUES (
  '{"name": "Gomez", "members": [
    {"member": {"relation": "padre", "name": "Alex"}},
    {"member": {"relation": "madre", "name": "Sonia"}},
    {"member": {"relation": "hijo", "name": "Brandon"}},
    {"member": {"relation": "hija", "name": "Azaleah"}}
  ]}');
```

•Select

```
SELECT json_extract_path_text(profile, 'name') As family, json_ex
tract_path_text( json_array_elements( json_extract_path(profile, 'mem
bers') ), 'member', 'name' ) As member
FROM families_j;
family | member
-----+-----
Gomez | Alex
Gomez | Sonia
Gomez | Brandon
Gomez | Azalea
```



JSON



- > es `json_extract_path_text` para campos simples y #>> para arreglos

```
SELECT profile->>'name' As family, json_array_elements((profile->'members'))  
#>>  
'{member,name}'::text[] As member  
FROM families_j;
```

- Para obtener un elemento en específico

```
SELECT id, json_array_length(profile->'members') As numero,  
profile->'members'->0#>>'{member,name}'::text[] As primero  
FROM families_j;  
id | numero | primero  
---+-----+-----  
1 | 4 | Alex
```

- Convertir filas a json

```
SELECT row_to_json(f) As x  
FROM (SELECT id, profile->>'name' As name FROM families_j) As f;  
x  
-----  
{"id":1,"name":"Gomez"}
```



JSON



- **Operadores**

- igualdad
- @> contiene
- <@ contenido
- ? Existe la llave
- ?| cualquiera de un arreglo de llaves existe
- ?& todas las llaves de un arreglo existen

```
SELECT profile->>'name' As family
FROM families_b
WHERE profile @> '{"members": [{"member": {"name": "Alex"} }]}';
family
-----
Gomez
```



JSON



- Más ejemplos

```
SELECT '"foo"'::jsonb @> '"foo"'::jsonb;
SELECT '[1, 2, 3]'::jsonb @> '[1, 3]'::jsonb;
SELECT '[1, 2, 3]'::jsonb @> '[3, 1]'::jsonb;
SELECT '[1, 2, 3]'::jsonb @> '[1, 2, 2]'::jsonb;
SELECT '{"product": "PostgreSQL", "version": 9.4, "jsonb": true}'::jsonb
@> '{"version": 9.4}'::jsonb;
SELECT '[1, 2, [1, 3]]'::jsonb @> '[1, 3]'::jsonb;
SELECT '[1, 2, [1, 3]]'::jsonb @> '[[1, 3]]'::jsonb;
SELECT '{"foo": {"bar": "baz"}}'::jsonb @> '{"bar": "baz"}'::jsonb;
SELECT '{"foo": {"bar": "baz"}}'::jsonb @> '{"foo": {}}'::jsonb;
SELECT '[ "foo", "bar", "baz" ]'::jsonb ? 'bar';
SELECT '{"foo": "bar"}'::jsonb ? 'foo';
SELECT '{"foo": "bar"}'::jsonb ? 'bar';
SELECT '{"foo": {"bar": "baz"}}'::jsonb ? 'bar';
SELECT '"foo"'::jsonb ? 'foo';
```



XML



- Al igual que JSON al crear un tipo XML se valida que el texto sea un XML válido

- No se valida contra un DTD o un XSD

```
CREATE TABLE families (id serial PRIMARY KEY, profile xml);
INSERT INTO families(profile)
VALUES (
    '<family name="Gomez">
<member><relation>padre</relation><name>Alex</name></member>
<member><relation>madre</relation><name>Sonia</name></member>
<member><relation>hijo</relation><name>Brandon</name></member>
<member><relation>hija</relation><name>Azaleah</name></member>
</family>');

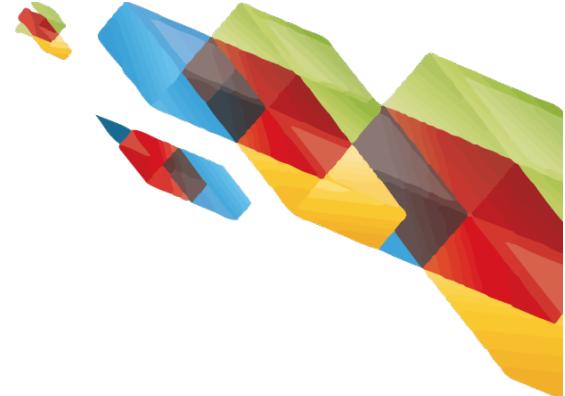
```

- Para validar se pueden agregar restricciones utilizando xpath

```
ALTER TABLE families ADD CONSTRAINT chk_has_relation
CHECK (xpath_exists('/family/member/relation', profile));
INSERT INTO families (profile) VALUES ('<family
name="HsuObe"></family>');
ERROR: new row for relation "families" violates check constraint
"chk_has_relation".
```



XML



- Select con xpath

- No se valida contra un DTD o un XSD

```
SELECT family,
(xpath('/member/relation/text()', f))[1]::text As relation,
(xpath('/member/name/text()', f))[1]::text As mem_name
FROM (SELECT (xpath('/family/@name', profile))[1]::text As family,
unnest(xpath('/family/member', profile)
) As f FROM families) x;
family | relation | mem_name
-----+-----+-----
Gomez | padre | Alex
Gomez | madre | Sonia
Gomez | hijo | Brandon
Gomez | hija | Azaleah
```



XML



- Para convertir texto se puede utilizar la función `xmlparse`:

- `XMLPARSE (DOCUMENT '<?xml`

- `version="1.0"?><book><title>Manual</title><chapter>...</chapter></book>')`

- `XMLPARSE (CONTENT 'abc<foo>bar</foo><bar>foo</bar>')`

- También se puede usar

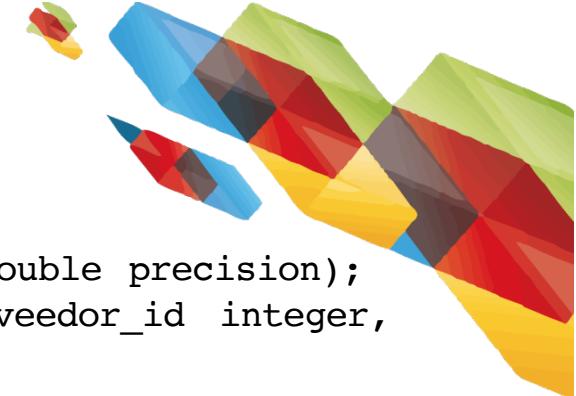
- `xml '<foo>bar</foo>' '<foo>bar</foo>'::xml`

- La variable `xmloption` controla si se admiten DOCUMENT o CONTENT

- `SET xmloption TO { DOCUMENT | CONTENT };`



Tipos compuestos



- Un tipo compuesto es la misma idea que un registro o record

```
CREATE TYPE complejo AS (r double precision, i double precision);
CREATE TYPE item_inventario AS (nombre text, proveedor_id integer,
precio numeric);
```

- Similar a un create table, pero sin restricciones (Constraints)

```
CREATE TABLE stock ( item item_inventario, cantidad integer);
Insert into stock VALUES(ROW('Bicicleta',12,250000.00), 50);
```

- Cada Tabla es un tipo compuesto

```
CREATE TABLE item_inventario( nombre text, proveedor_id
integer REFERENCES proveedores, precio numeric CHECK (price >
0));
```

- El tipo compuesto representa el tipo de la fila
- Las restricciones se ignoran

- La opción ROW es opcional si se ingresa más de un campo

- Para acceder al campo de un tipo compuesto la notación es columna.campo

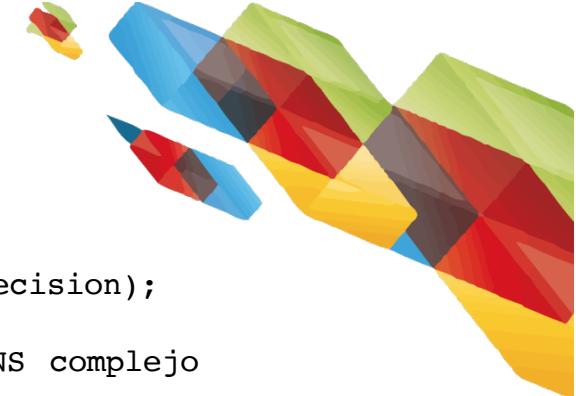
```
SELECT (item).nombre from stock where (item).precio > 500000;
```

- El uso de parentesis evita que se interprete como una tabla.
- Los mismo para update

```
UPDATE numeros SET numero_complejo= (numero_complejo).r + 1
```



Tipos compuestos



- Otro ejemplo

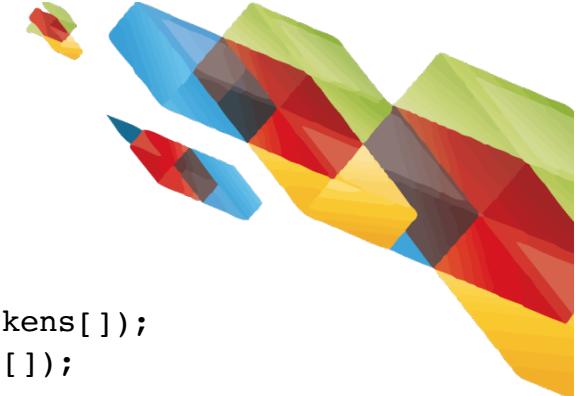
```
CREATE TYPE complejo AS (r double precision, i double precision);

CREATE OR REPLACE FUNCTION add(complejo, complejo) RETURNS complejo
AS
$$
SELECT ( (COALESCE((\$1).r,0) + COALESCE((\$2).r,0)),
(COALESCE((\$1).i,0) + COALESCE((\$2).i,0)) )::complejo;
$$
language sql;

CREATE OPERATOR +
PROCEDURE = add,
LEFTARG = complejo,
RIGHTARG = complejo,
COMMUTATOR = +;
SELECT (1,2)::complex_number + (3,-10)::complex_number;
```



Tipos compuestos



- Otro ejemplo

```
CREATE TABLE chickens (id integer PRIMARY KEY);
CREATE TABLE ducks (id integer PRIMARY KEY, chickens chickens[]);
CREATE TABLE turkeys (id integer PRIMARY KEY, ducks ducks[]);

INSERT INTO ducks VALUES (1, ARRAY[ROW(1)::chickens, ROW(1)::chickens]);

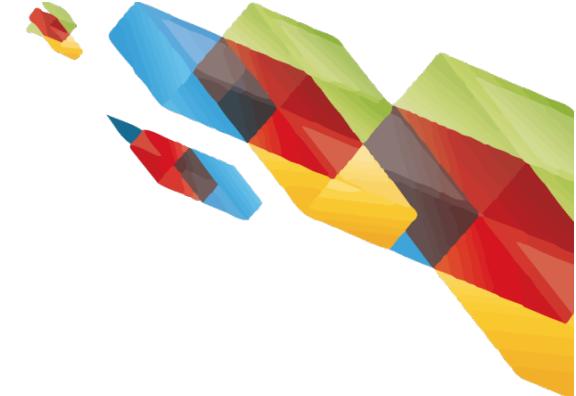
INSERT INTO turkeys VALUES (1, array(SELECT d FROM ducks d));

SELECT * FROM turkeys;
output
-----
id | ducks
---+
1  | {"(1, \"{{(1),(1)}}")}

UPDATE turkeys SET ducks[1].chickens[2] = ROW(3)::chickens
WHERE id = 1 RETURNING *;
output
-----
id | ducks
---+
1  | {"(1, \"{{(1),(3)}}")}
```



Tipos no revisados



- Geométricos
- De direcciones de redes
- Bit
- Text search
- UUID