



CONCEPTOS DE BASES DE DATOS II

Tipos de Datos - Índices

Tipos de Datos

Después de la fase de diseño de una base de datos, comienza la fase de implementación de la misma, es decir la creación de las tablas, índices, etc.

Al crear las tablas se debe tomar decisiones con respecto a la estructura de las mismas, especificando el tamaño y el tipo de dato para cada campo.

Una mala elección del tipo o tamaño de los campos puede derivar en errores, si el tamaño es insuficiente, de valores fuera de rango en caso de los numéricos o cadenas demasiado largas en caso de los datos de texto, por lo contrario, se puede desperdiciar mucho recurso si el tamaño es excesivo.

Tipos de Datos

Los tipos de datos que se puede asignar a un campo, se pueden agrupar en tres grandes grupos:

- ✓ **Tipos numéricos:** sin punto decimal y con punto decimal
- ✓ **Tipos de Cadena:** longitud fija o variable
- ✓ **Tipos de Fecha:** para guardar fecha con hora y sin hora

Tipos de Datos

La elección de los tipos de datos para las columnas de una tabla en una base de datos es crucial, ya que afecta el rendimiento, almacenamiento e integridad de los datos. Algunos criterios a considerar al seleccionar los tipos de datos para las columnas son:

Naturaleza de los Datos: Elegir tipos de datos que reflejen con precisión la naturaleza de los datos que se almacenarán. Por ejemplo, use INTEGER para números enteros, VARCHAR para cadenas de texto variables.

Longitud y Tamaño: Ajustar la longitud de los campos de texto para adaptarse al contenido real.

Utilizar tipos de datos que se ajusten al rango de valores esperados. Por ejemplo, si sabe que una columna solo contendrá valores positivos, considere usar UNSIGNED para enteros.

Tipos de Datos

Requerimientos de Precisión y Escala: Utilizar tipos de datos que ofrezcan la precisión y la escala adecuadas. Por ejemplo, DECIMAL por FLOAT si necesita precisión decimal exacta.

Indexación y Búsqueda: Considerar la posibilidad de indexar las columnas que se utilizarán con frecuencia en cláusulas WHERE o JOIN.

Restricciones de Integridad: Aplicar restricciones de integridad según sea necesario. Por ejemplo, usar Pk, Fk o constraints para garantizar la coherencia y la calidad de los datos.

Rendimiento: Evaluar el rendimiento de las consultas y operaciones que se realizarán en la base de datos. Algunos tipos de datos pueden ser más eficientes en términos de almacenamiento y velocidad de consulta que otros.

Tipos de Datos

Compatibilidad del Sistema: Utilizar tipos de datos estándar que sean compatibles con las tecnologías que planea utilizar.

Uso de Memoria y Almacenamiento: Algunos tipos de datos consumen más recursos que otros. Ajustar los tipos de datos según los requisitos de su aplicación y hardware.

Tendencias de Evolución de Datos: Anticipar posibles cambios en los datos a lo largo del tiempo. Elegir tipos de datos que permitan la evolución de la base de datos sin cambiar drásticamente la estructura existente.

Al considerar estos criterios y adaptarlos a las necesidades específicas de su aplicación, podrá diseñar tablas de base de datos que sean eficientes, escalables y que mantengan la integridad y consistencia de los datos almacenados.

Tipos de Datos

Los tipos de datos numéricos por lo general son más rápidos al momento de realizar consultas a la base de dato. Son estáticos, por lo que ocupan más espacios en la memoria y en el disco duro.

Supongamos que tenemos que definir el tipo y tamaño del campo cantidad de la tabla detalle_ventas (base de datos biblioteca) y debo elegir entre:

SMALLINT (-32768 a 32767) **2 bytes**

INTEGER (-2147483648 a 2147483647) **4 bytes**

BIGINT (-9.223.372.036.854.775.808 a 9.223.372.036.854.775.807) **8 bytes**

Tipos de Datos

Otra situación es la selección de tipo para campos que almacenaran valores monetarios, supongamos que tenemos que almacenar el precio de los productos (\$452,32.-) o el valor total de una venta (\$15.325,25.-), o simplemente el sueldo de un empleado (\$35.215,65.-)

NUMERIC(P,E) (PRECISIÓN ESPECIFICADA POR EL USUARIO) **VARIABLE**

REAL (AL MENOS 6 DÍGITOS DECIMALES DE PRECISIÓN) **4 BYTES**

DOUBLE PRECISION (AL MENOS 15 DÍGITOS DECIMALES DE PRECISIÓN) **8 BYTES**

Tipos de Datos Numéricos

Sin Punto Decimal			
Tipo	Descripción	Rango	Almacenamiento
SMALLINT	número entero con signo	(-32768 a 32767)	2 bytes
INTEGER	número entero con signo	(-2147483648 a 2147483647)	4 bytes
BIGINT	número entero con signo	(-9.223.372.036.854.775.808 a 9.223.372.036.854.775.807)	8 bytes
SERIAL	número entero auto-incrementable	1 a 2147483647	4 bytes
BIGSERIAL	número entero auto-incrementable	1 a 9223372036854775807	8 bytes

Tipos de Datos Numéricos

Con Punto Decimal			
Tipo	Descripción	Rango	Almacenamiento
NUMERIC	Precisión especificada por el usuario, exacto	numérico exacto de precisión seleccionable	variable
REAL	número en coma flotante de precisión simple	6 dígitos decimales de precisión	4 bytes
DOUBLE PRECISION	número en coma flotante de precisión doble	15 dígitos decimales de precisión	8 bytes

Tipos de Datos Numéricos

Tipo	Descripción	Rango	Almacenamiento
BOOL	tipo estándar de boolean, puede tener tres estados "true" , "false" o "unknow" (que es NULL)	TRUE, 't', 'true', 'y', 'yes', 'on', '1' FALSE, 'f', 'false', 'n', 'no', 'off', '0'	1 byte
MONEY	importe monetario	-92233720368547758.08 a +92233720368547758.07	8 bytes

Tipos de Datos

Los datos de tipo texto por lo general son más lentos al momento de realizar consultas a la base de datos. Son dinámicos (no todos) y eso hace que ocupen menos espacios en memoria y en disco duro.

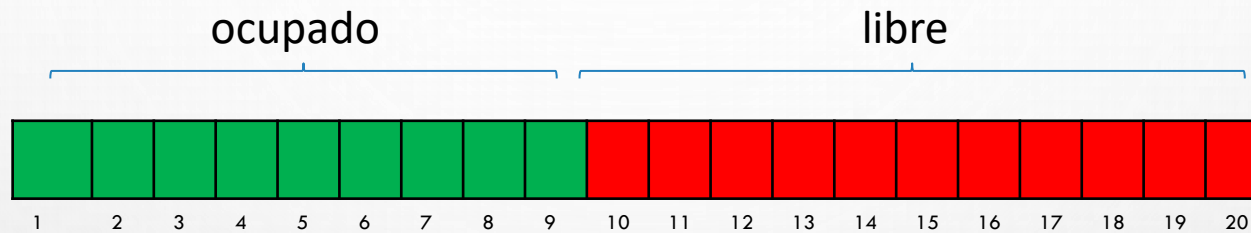
Supongamos que queremos almacenar la descripción de un producto. Este tipo de datos generalmente tienen longitudes muy diferentes y no podemos conocerlas de antemano.

CHARACTER(n) Reserva n lugares para almacenar la cadena

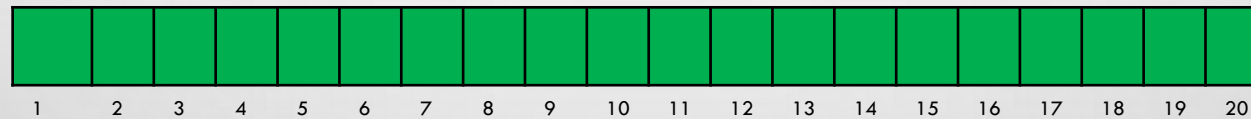
CHARACTER VARYING(n) Utiliza los espacios necesarios para almacenar una cadena menor o igual que n

Tipos de Datos

“desc prod” → 9 caracteres



“descripción producto” → 20 caracteres



si el campo de longitud variable, no se desperdicia la porción que se muerta libre, solo se ocupa lo que necesita, el resto lo libera.

Tipos de Datos

Supongamos ahora que necesitamos guardar el numero de documento de una persona (DNI) y el numero telefónico, en sus distintas variantes (fijo o celular).

DNI: 99999999 ó 99.999.999

Tel: 4444444 ó 54903814444444

Cel: 15555555 ó 3815555555

integer, numeric(p, e), float, Text, char(n), varchar(n)

No se aconseja dejar un campo texto como PK.

Tipos de datos de Cadena

Tipo	Descripción	Almacenamiento
CHAR(n), CHARACTER(n)	Reserva n espacios para almacenar la cadena	n +1 bytes
VARCHAR(n) CHARACTER VARYING(n)	Utiliza los espacios necesarios para almacenar una cadena menor o igual que n	Longitud+1 bytes
TEXT	Almacena cadenas de cualquier magnitud	Longitud variable

Tipos de datos de Json y Jsonb

Característica	JSON	JSONB
Almacenamiento	Texto plano (se almacena como una cadena de texto).	Binario optimizado (más eficiente).
Indexación	No admite índices GIN o GiST.	Admite índices GIN y GiST, lo que mejora búsquedas.
Acceso a datos	Más lento, ya que se debe parsear en cada consulta.	Más rápido, ya que está almacenado en un formato binario preprocesado.
Orden de claves	Mantiene el orden original de las claves.	No mantiene el orden original (las reorganiza).
Espacio ocupado	Más ligero en almacenamiento.	Puede ocupar más espacio debido a la optimización binaria.

Tipos de datos de Json y Jsonb

Uso principal de JSON/JSONB:

- Almacenamiento flexible de datos semiestructurados.
- Aplicaciones con esquemas dinámicos.
- APIs y servicios web que manejan datos en formato JSON.
- Optimización de búsquedas con índices GIN en JSONB.

Ejemplo:

```
create table productos (id serial primary key,  
                        nombre text,  
                        datos json);
```

```
insert into productos (nombre, datos)
```

```
values ('Laptop', '{"marca": "Dell", "modelo": "XPS", "precio": 1500}');
```

```
select datos->>'marca' from productos; -- Extrae el valor de la clave "marca"
```

Tipos de datos de Json y Jsonb

Uso de JSONB con índices GIN:

```
create table empleados (id serial primary key,  
                        info jsonb);
```

```
insert into empleados (info) values ('{"nombre": "Ana", "edad": 30, "cargo": "Ingeniera"} '),  
                                       ('{"nombre": "Juan", "edad": 25, "cargo": "Analista"} ');
```

```
create index idx_empleados_jsonb on empleados using gin (info);
```

```
select * from empleados where info @> '{"cargo": "Ingeniera"}'; -- Filtra empleados con cargo  
"Ingeniera"
```

Tipos de datos Fecha

Tipo	Descripción	Formato	Almacenamiento
DATE	Fecha	'YYYY-MM-DD'	4 bytes
TIMESTAMP()	Fecha y Horas	'YYYY-MM-DD HH:MM:SS'	8 bytes
TIME (HORA)	Horas del día sin o con zona horaria	'HH:MM:SS' 'HH:MM:SS+12'	8 bytes 12 bytes
INTERVAL[(p)]	Intervalo de hora	segundos, minutos, horas, días, semanas, meses, años, décadas, siglos, milenios	12 bytes

Tipos de datos Geométricos

Tipo	Descripción	Ejemplo	Almacenamiento
POINT	Un punto en un plano 2D ((x, y)).	(3.5, 4.2)	16 bytes
LINE	Una línea infinita definida por la ecuación $Ax + By + C = 0$.	{1, -1, 0}	32 bytes
LSEG	Un segmento de línea definido por dos puntos.	[(1,2), (3,4)]	32 bytes
BOX	Un rectángulo definido por dos puntos (esquina superior derecha e inferior izquierda).	((1,2), (3,4))	32 bytes
CIRCLE	Un círculo definido por un centro y un radio.	<(1,2), 3>	24 bytes
PATH	Una serie de puntos conectados (puede ser abierto o cerrado).	[(1,2), (3,4), (5,6)]	Variable
POLYGON	Un polígono cerrado con múltiples puntos.	((1,2), (3,4), (5,6), (1,2))	Variable

Tipos de datos Geométricos

Uso principal de datos geométricos:

- Sistemas de información geográfica (GIS).
- Aplicaciones de diseño y modelado.
- Representación de mapas y coordenadas.
- Juegos y simulaciones físicas.

Ejemplo:

```
create table lugares (id serial primary key,  
                      nombre text,  
                      ubicacion point);
```

```
insert into lugares (nombre, ubicacion) values ('Café Central', '(10.5, 20.3)');
```

```
select * from lugares where ubicacion <@ circle(point(10,20), 5); -- Lugares dentro de un radio de 5 unidades
```

Tipos de datos de Direcciones de Red

Tipo	Descripción	Almacenamiento
CIDR	Redes IPv4 ó IPv6	12 ó 24 bytes
INET	Hosts y redes IPv4 ó IPv6	12 ó 24 bytes
MACADDR	Dirección MAC	6 bytes

Datos Definidos por el Usuario – Create Type

PostgreSQL tiene un rico conjunto de tipos de datos nativos disponibles para los usuarios. Además, los usuarios pueden agregar nuevos tipos usando el comando **CREATE TYPE**.

CREATE TYPE registra un nuevo tipo de datos para su uso en la base de datos actual.

El usuario que define un tipo es su propietario. El nombre del **tipo de dato** debe ser distinto al de cualquier tipo, dominio o tabla existente en el mismo esquema.

Ejemplos de tipos de datos

Creamos tres tablas con los mismos campos, pero cada una tiene diferentes tipos de datos

```
create table persona_vchar(id_tex    character varying(8),  
                           nombre    character varying(100),  
                           fecha_nac character varying(100));
```

```
create table persona(id_int    integer,  
                    nombre    character varying(100),  
                    fecha_nac date);
```

```
create table persona_char(id_char    character (100),  
                          nombre    character (100),  
                          fecha_nac character (100));
```


Ejemplos de tipos de datos

Insertar 1,000,000 de registros aleatorios




```
insert into persona (id_int, nombre, fecha_nac)
select i, 'Persona_' || i, date '1950-01-01' + (random() * 25550)::int
from generate_series(1, 1000000) as i;
```

```
insert into persona_vchar (id_tex, nombre, fecha_nac)
select lpad(i::text, 8, '0'), 'Persona_' || i, (date '1950-01-01' + (random() * 25550)::int)::text
from generate_series(1, 1000000) as i;
```




```
insert into persona_char (id_char, nombre, fecha_nac)
select lpad(i::text, 100, '0'), 'Persona_' || i, (date '1950-01-01' + (random() * 25550)::int)::text
from generate_series(1, 1000000) as i;
```

Ejemplos de tipos de datos

```
select pg_table_size('persona') as tamPersona, pg_table_size('persona_char') as tamPersona_char,  
pg_table_size('persona_vchar') as tamPersona_vchar
```

	tampersona 	tampersona_char 	tampersona_vchar 
1	52215808	341442560	68304896

```
select pg_size_pretty(pg_table_size('persona')) as tamPersona,  
pg_size_pretty(pg_table_size('persona_char')) as tamPersona_char,  
pg_size_pretty(pg_table_size('persona_vchar')) as tamPersona_vchar
```

	tampersona 	tampersona_char 	tampersona_vchar 
1	50 MB	326 MB	65 MB

Ejemplos de tipos de datos

La función **pg_table_size** devuelve el tamaño de una tabla en bytes. Se utiliza para averiguar cuánto espacio ocupa una tabla en la base de datos.

La función **pg_size_pretty** devuelve una cadena de caracteres que representa un tamaño de bytes en un formato legible para humanos.

Por ejemplo, si el tamaño de la tabla es 1024 bytes, **pg_size_pretty** lo representaría como "1 KB". Esto se debe a que convierte el tamaño en una unidad más grande (por ejemplo, KB, MB, GB, etc.) para hacerlo más fácil de leer. En este caso, como 1024 bytes es igual a 1 kilobyte, la función devolverá la cadena "1 KB". Si el tamaño es 1073741824 bytes, la función "1 GB".

Ejemplos de tipos de datos

EXPLAIN ANALYZE es una instrucción en SQL, que brinda información sobre la ejecución de una consulta. Es una herramienta poderosa para analizar y optimizar el rendimiento de las consultas en una base de datos relacional.

Cuando ejecutas una consulta, el motor crea un plan de ejecución con los pasos que se llevarán a cabo para recuperar los datos solicitados. Este plan de ejecución es una especie de "hoja de ruta" para la base de datos, que indica cómo acceder y combinar los datos necesarios.

La instrucción **EXPLAIN ANALYZE** permite obtener el plan de ejecución detallado de una consulta, junto con información sobre el tiempo que lleva ejecutar cada paso. Al agregar **ANALYZE** a la instrucción **EXPLAIN**, la base de datos ejecutará la consulta y proporcionará estadísticas de ejecución detalladas.

Ejemplos de tipos de datos

La salida de EXPLAIN ANALYZE incluye información sobre el plan y las estadísticas de la ejecución de la consulta. La salida puede variar según el motor, pero estos son algunos elementos comunes:

Plan de ejecución:

- **Nodos del plan:** Cada paso de la consulta es un nodo. Pueden ser del tipo Seq Scan (escaneo secuencial) o **Index Scan** (escaneo de índice).
- **Costos y Estimaciones:** Se estima el costo de cada nodo. Esto incluye el costo de CPU, E/S (entrada/salida) y el costo total.

Estas estimaciones ayudan al motor de base de datos a determinar el mejor plan de ejecución posible.

Ejemplos de tipos de datos

Estadísticas de ejecución:

- **Tiempo total:** es el tiempo que llevó ejecutar la consulta.
- **Tiempo de ejecución de cada nodo:** Indica cuánto tiempo se dedicó a cada paso individual del plan de ejecución.

Información sobre filas y tamaño:

- **Número de filas:** Proporciona estimaciones y resultados reales del número de filas que se procesaron en cada nodo.
- **Tamaño de la salida:** Muestra el tamaño de los conjuntos de resultados en bytes.

Ejemplos de tipos de datos

Información sobre índices y restricciones:

- **Índices utilizados:** Indica si se utilizaron índices, en ese caso se utilizaron y cuáles.
- **Restricciones aplicadas:** Muestra las restricciones que se aplicaron durante la ejecución.

Un ejemplo de salida de EXPLAIN ANALYZE

```
explain analyze select * from persona
```

QUERY PLAN

```
Seq Scan on persona (cost=0.00..819.00 rows=50000 width=19) (actual time=0.861..16.107 rows=50000 loops=1)
Planning Time: 8.890 ms
Execution Time: 17.688 ms
```

Ejemplos de tipos de datos

Análisis de cada línea en el resultado de la consulta

La Línea principal del plan de ejecución:

Seq Scan on persona: se realiza un escaneo secuencial de la tabla.

(cost=0.00..819.00): Costos estimados de la ejecución de este nodo. La estimación se hace entre (0.00) el costo más bajo y (819.00) el costo más alto.

rows=50000: Se espera que el nodo devuelva alrededor de 50,000 filas según las estimaciones del plan de ejecución.

width=19: Representa el ancho estimado de cada fila en bytes. Se estima que cada fila de la tabla "persona" tiene un ancho de 19 bytes.

Ejemplos de tipos de datos

(actual time=0.861..16.107): Indica el tiempo real que llevó ejecutar este nodo. La consulta tardó entre 0.861 y 16.107 ms (milisegundos) para completarse.

rows=50000 loops=1: Indica el número real de filas devueltas por el nodo y el número real de iteraciones (loops) realizadas. Se devolvieron 50,000 filas y se realizó una sola iteración.

Tiempos de planificación y ejecución:

Planning Time: Es el tiempo que demoró el sistema de planificación de consultas en elaborar el plan de ejecución. En este caso, fueron 8.890 milisegundos.

Ejemplos de tipos de datos

Execution Time: Es el tiempo total tardó en ejecutar la consulta, desde la planificación hasta la finalización. En este caso, fueron 17.688 milisegundos.

Esto proporciona información sobre el rendimiento de la consulta, los tiempos estimados y reales, así como el costo asociado con el método de acceso a los datos.

Puedes utilizar esta información para entender cómo se ejecuta la consulta y para identificar una posibles optimización. Por ejemplo, si el tiempo de ejecución es alto, se podría considerar la adición de índices o ajustes en la consulta para mejorar el rendimiento.

En resumen, EXPLAIN ANALYZE es una herramienta valiosa para mejorar el rendimiento de las consultas SQL

Ejemplos de tipos de datos

explain analyze select * from persona

	QUERY PLAN text	
1	Seq Scan on persona (cost=0.00..16370.00 rows=1000000 width=22) (actual time=0.054..79.052 rows=1000000 loops=...	
2	Planning Time: 0.616 ms	
3	Execution Time: 101.149 ms	

explain analyze select * from persona_vchar

	QUERY PLAN text	
1	Seq Scan on persona_vchar (cost=0.00..18333.00 rows=1000000 width=34) (actual time=0.058..72.248 rows=1000000 loops=...	
2	Planning Time: 0.609 ms	
3	Execution Time: 91.570 ms	

explain analyze select * from persona_char

	QUERY PLAN text	
1	Seq Scan on persona_char (cost=0.00..51666.97 rows=999997 width=303) (actual time=0.047..172.791 rows=1000000 loops=...	
2	Planning Time: 0.644 ms	
3	Execution Time: 192.016 ms	

Ejemplos de tipos de datos

Creemos una tabla con los mismos campos, pero con tipos de datos diferentes

```
create table cliente ( id serial primary key,  
                       nombre text,  
                       nombre_varchar character varying (255)  
                       edad integer,  
                       edad_texto text,  
                       fecha_nacimiento date,  
                       fecha_texto text);
```

-- Insertar 1,000,000 de registros aleatorios

```
insert into cliente (nombre, nombre_varchar, edad, edad_texto, fecha_nacimiento, fecha_texto)  
select 'Cliente_' || i, 'Cliente_' || i, (random() * 100)::int,  
((random() * 100)::int)::text,  
date '1950-01-01' + (random() * 25550)::int,  
(date '1950-01-01' + (RANDOM() * 25550)::int)::text  
from generate_series(1, 1000000) as i;
```

Ejemplos de tipos de datos

explain analyze select * from cliente where nombre = 'Cliente_500000';

	QUERY PLAN text
1	Gather (cost=1000.00..17467.43 rows=1 width=53) (actual time=43.057..96.317 rows=1 loops=1)
2	Workers Planned: 2
3	Workers Launched: 2
4	-> Parallel Seq Scan on cliente (cost=0.00..16467.33 rows=1 width=53) (actual time=21.363..29.978 rows=0 loop...
5	Filter: (nombre = 'Cliente_500000')::text)
6	Rows Removed by Filter: 333333
7	Planning Time: 1.536 ms
8	Execution Time: 96.341 ms

explain analyze select * from cliente where nombre_varchar = 'Cliente_500000';

	QUERY PLAN text
1	Gather (cost=1000.00..17467.43 rows=1 width=53) (actual time=45.854..180.465 rows=1 loops=1)
2	Workers Planned: 2
3	Workers Launched: 2
4	-> Parallel Seq Scan on cliente (cost=0.00..16467.33 rows=1 width=53) (actual time=26.496..39.197 rows=0 loop...
5	Filter: ((nombre_varchar)::text = 'Cliente_500000')::text)
6	Rows Removed by Filter: 333333
7	Planning Time: 0.056 ms
8	Execution Time: 180.482 ms

Ejemplos de tipos de datos

explain analyze select * from cliente where nombre like 'Cliente_500000';

	QUERY PLAN text
1	Gather (cost=1000.00..17477.33 rows=100 width=53) (actual time=56.336..195.503 rows=1 loops=1)
2	Workers Planned: 2
3	Workers Launched: 2
4	-> Parallel Seq Scan on cliente (cost=0.00..16467.33 rows=42 width=53) (actual time=18.655..32.569 rows=0 loop...
5	Filter: (nombre ~~ 'Cliente_500000'::text)
6	Rows Removed by Filter: 333333
7	Planning Time: 0.073 ms
8	Execution Time: 195.534 ms

explain analyze select * from cliente where nombre_varchar like 'Cliente_500000';

	QUERY PLAN text
1	Gather (cost=1000.00..17477.33 rows=100 width=53) (actual time=76.236..98.703 rows=1 loops=1)
2	Workers Planned: 2
3	Workers Launched: 2
4	-> Parallel Seq Scan on cliente (cost=0.00..16467.33 rows=42 width=53) (actual time=31.463..37.655 rows=0 loop...
5	Filter: ((nombre_varchar)::text ~~ 'Cliente_500000'::text)
6	Rows Removed by Filter: 333333
7	Planning Time: 0.076 ms
8	Execution Time: 98.719 ms

Ejemplos de tipos de datos

explain analyze select * from cliente where edad < 30;

	QUERY PLAN text	🔒
1	Seq Scan on cliente (cost=0.00..23759.00 rows=289632 width=53) (actual time=0.050..72.009 rows=295057 loops=...	
2	Filter: (edad < 30)	
3	Rows Removed by Filter: 704943	
4	Planning Time: 0.062 ms	
5	Execution Time: 77.661 ms	

explain analyze select * from cliente where edad_texto < '30';

	QUERY PLAN text	🔒
1	Seq Scan on cliente (cost=0.00..23759.00 rows=237542 width=53) (actual time=0.047..259.648 rows=240337 loops=...	
2	Filter: (edad_texto < '30'::text)	
3	Rows Removed by Filter: 759663	
4	Planning Time: 0.093 ms	
5	Execution Time: 264.208 ms	

Ejemplos de tipos de datos

explain analyze select * from cliente where fecha_nacimiento = '1990-01-01';

	QUERY PLAN text	
1	Gather (cost=1000.00..17471.23 rows=39 width=53) (actual time=0.979..178.131 rows=40 loops=1)	
2	Workers Planned: 2	
3	Workers Launched: 2	
4	-> Parallel Seq Scan on cliente (cost=0.00..16467.33 rows=16 width=53) (actual time=1.794..37.404 rows=13 loop...	
5	Filter: (fecha_nacimiento = '1990-01-01'::date)	
6	Rows Removed by Filter: 333320	
7	Planning Time: 0.111 ms	
8	Execution Time: 178.171 ms	

explain analyze select * from cliente where fecha_texto = '1990-01-01';

	QUERY PLAN text	
1	Gather (cost=1000.00..17471.23 rows=39 width=53) (actual time=0.972..181.201 rows=40 loops=1)	
2	Workers Planned: 2	
3	Workers Launched: 2	
4	-> Parallel Seq Scan on cliente (cost=0.00..16467.33 rows=16 width=53) (actual time=0.217..32.294 rows=13 loop...	
5	Filter: (fecha_texto = '1990-01-01'::text)	
6	Rows Removed by Filter: 333320	
7	Planning Time: 0.065 ms	
8	Execution Time: 181.237 ms	

Ejemplos de tipos de datos

explain analyze select * from cliente where fecha_nacimiento between '1990-01-01' and '2000-01-01 ';

	QUERY PLAN text
1	Seq Scan on cliente (cost=0.00..26259.00 rows=141300 width=53) (actual time=0.047..81.691 rows=142687 loops=...
2	Filter: ((fecha_nacimiento >= '1990-01-01'::date) AND (fecha_nacimiento <= '2000-01-01'::date))
3	Rows Removed by Filter: 857313
4	Planning Time: 0.111 ms
5	Execution Time: 84.472 ms

explain analyze select * from cliente where fecha_texto between '1990-01-01' and '2000-01-01 ';

	QUERY PLAN text
1	Seq Scan on cliente (cost=0.00..26259.00 rows=139843 width=53) (actual time=0.043..403.873 rows=142687 loops=...
2	Filter: ((fecha_texto >= '1990-01-01'::text) AND (fecha_texto <= '2000-01-01'::text))
3	Rows Removed by Filter: 857313
4	Planning Time: 0.146 ms
5	Execution Time: 406.786 ms

Index (Índices)

La creación de índices en las bases de datos es fundamental, sirven para agilizar las consultas. Si bien en una tabla con pocas filas es casi inapreciable, cuando trabajo con millones de registros, el uso de índices es necesario.



Pensemos en el índice de un libro, donde un capítulo está asociado al número de página donde comienza el mismo. Para acceder a un capítulo no recorremos cada hoja hasta localizar el comienzo del capítulo. La búsqueda se realiza en el índice, se identifica la pagina donde comienza y se accede.

Por qué usar índices

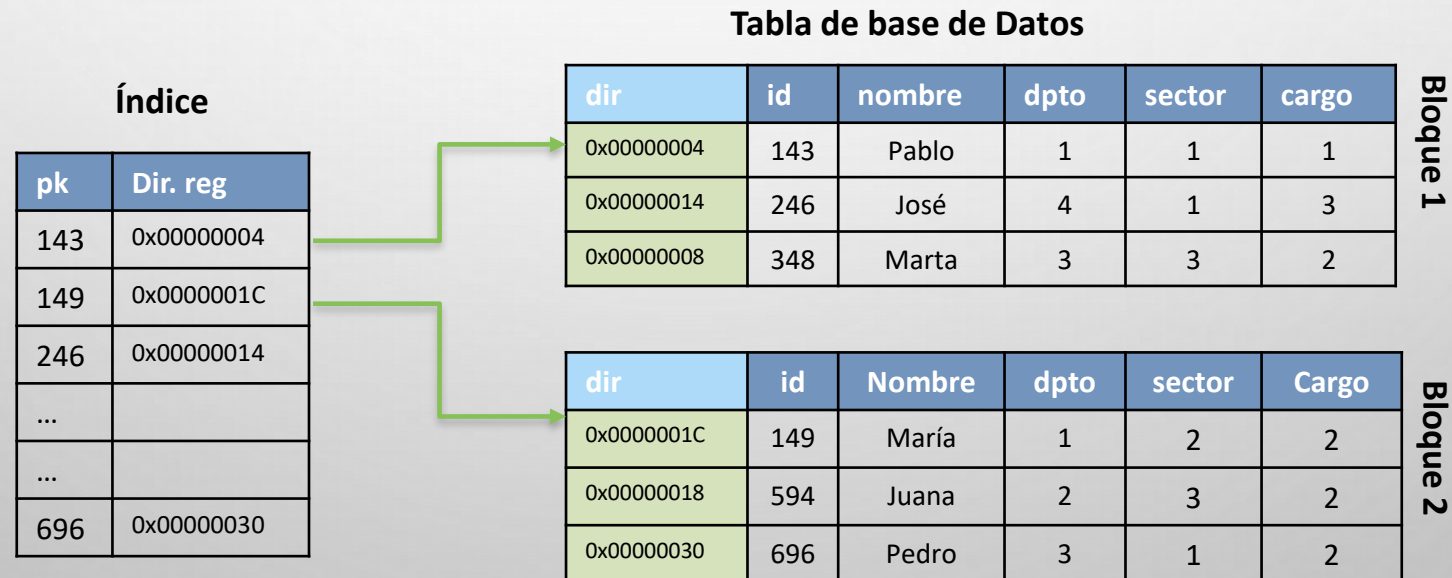
Sobrecarga de CPU. El proceso de leer cada uno de los registros en una tabla puede convertirse en un problema a medida que va aumentando la cantidad de registros en la misma

Concurrencia. Mientras se leen los datos de una tabla, la misma está bloquea para escritura, pero habilitada para lectura, en el mejor de los casos. Cuando se está actualizando o eliminando filas de una tabla, ésta se bloquea por completo.

Sobrecarga de disco. En una tabla muy grande, un escaneo completo consume una gran cantidad de entrada/salida en el disco. Esto puede hacer significativamente lento el servidor de bases de datos, debido al trafico de datos.

Index (Índices)

Un índice es una estructura de datos definida sobre una o varias columnas de una tabla, que permite localizar de forma rápida las filas de la misma. El índice almacena el dato buscado y un **puntero** a los registros de la tabla



Tipos de Índices

Índices Primarios: Se define sobre una de clave primaria PK. Se crea automáticamente cuando defines una clave primaria (PK) en una tabla.

```
create table cliente ( id serial primary key,  
                        nombre text,  
                        nombre_varchar character varying (255)  
                        edad integer,  
                        edad_texto text,  
                        fecha_nacimiento date,  
                        fecha_texto text);
```

Índices Secundarios: Se define sobre un campo que no es clave primaria del archivo (puede tener valores repetidos)

```
create index idx_clientes_edad on cliente(edad);  
create index idx_clientes_edad_texto on clientes(edad_texto);  
create index idx_clientes_nombre on clientes(nombre);  
create index idx_clientes_fecha_nacimiento on clientes(fecha_nacimiento);
```

Tipos de Índices

Índices Compuestos: Se definen con la combinación de dos o más campos, donde el orden lo da el primer campo y ante iguales ocurrencias para el mismo, se utiliza para ordenar el segundo campo. Y así sucesivamente en el caso de ser varios los campos.

```
create table automovil (id          int primary key,  
                        marca       varchar(255),  
                        modelo      varchar(255),  
                        color       varchar(255));
```

```
create index idx_marca on automovil (marca, modelo);
```

```
create index idx_modelo on automovil (modelo, marca);
```

¿Sabiendo que una marca de autos tiene muchos modelos, cual índice es mas efectivo (marca, modelo) ó (modelo, marca)?

Index (Índices)

Algunos de los tipos de índices más comunes en PostgreSQL:

Índices B-Tree: Eficientes con valores ordenados, como cadenas de texto y números. Mejoran el rendimiento de operaciones de igualdad y rangos.

```
create index nombre_indice on nombre_tabla using btree (nombre_columna);
```

Índices Hash: Son adecuados para búsquedas de igualdad. No son tan eficientes para rangos.

```
create index nombre_indice on nombre_tabla using hash (nombre_columna);
```

Índices GIN (Generalized Inverted Index): Son útiles conjuntos de datos complejos como arrays o listas.

Index (Índices)

Índices GiST (Generalized Search Tree): Similar a GIN, útil para realizar búsquedas espaciales y búsquedas de texto avanzadas.

```
create index nombre_indice on nombre_tabla using gist (nombre_columna);
```

Índices SP-GiST (Space-Partitioned Generalized Search Tree): Son útiles cuando se trabaja con tipos de datos geométricos.

```
create index nombre_indice on nombre_tabla using spgist (nombre_columna);
```

Índices BRIN (Block Range INdexes): Eficientes para grandes conjuntos de datos ordenados en bloques, como registros temporales o cronológicos. Almacenan información sobre el rango de bloques, reduciendo el tamaño del índice.

```
create index nombre_indice on nombre_tabla using brin (nombre_columna);
```


Crear Índices (Create Index)

```
create [ unique ] index [ if not exists ]  
name on table_name [ using method ] [ asc | desc ] [ null { first | last } ] [ where predicate ]
```

Podemos crear un índice cuando se crean las tablas

```
create table nombreTabla(campo1 tipoDato, campo2 tipoDato,..  
index [nombreIndice] (campo1 [,campo2...]));
```

Podemos crear el índice en una tabla ya existente

```
alter table nombreTabla add index [nombreIndice] (campo1 [,campo2...]);
```

Crear Índices (Create Index)

create [**unique**] **index** [**if not exists**]

name **on** table_name [**using** method] [**asc** | **desc**] [**null** { **first** | **last** }] [**where** predicate]

unique: no permite insertar datos duplicados, si hay valores duplicados en la tabla cuando se crea el índice da error al momento de crearlo.

if not exists: Si existe un índice con el mismo nombre en la tabla, no lo crea, para evitar errores.

name: El nombre del índice que se creará

table_name: El nombre de la tabla para ser indexada.

using method: método a utilizar por el índice. Las opciones son btree (árbolB), hash, GIST, spgist , gin y Brin . El método por defecto es btree.

column_name: El nombre de una columna de la tabla.

Crear Índices (Create Index)

asc : Especifica ordenamiento ascendente (valor por defecto).

desc Especifica ordenamiento descendente.

[nulls { **first** | last }]: Los valores nulos se ordenan antes de los no nulos. Valor por defecto es desc.

[nulls { first | **last** }]: Especifica que la clase después de los nulos no son nulos. Valor por defecto es desc.

[**where predicate**]: La expresión de restricción para un índice parcial.

Podemos crear un índice cuando se crean las tablas

```
create table nombreTabla(campo1 tipoDato, campo2 tipoDato,.. index [nombreIndice] (campo1 [,campo2...]));
```

Podemos crear el índice en una tabla ya existente

```
alter table nombreTabla add index [nombreIndice] (campo1 [,campo2...]);
```

Crear Índices (Create Index)

Crear un índice en una tabla existente.

```
create index nombreIndice on nombreTabla(campo1 [,campo2...]);
```

Ambas sentencias piden el nombre del índice, sin embargo, con la sentencia `create index` el nombre es obligatorio.

```
create table usuarios(id int, nombre varchar(50), apellidos varchar(70));
```

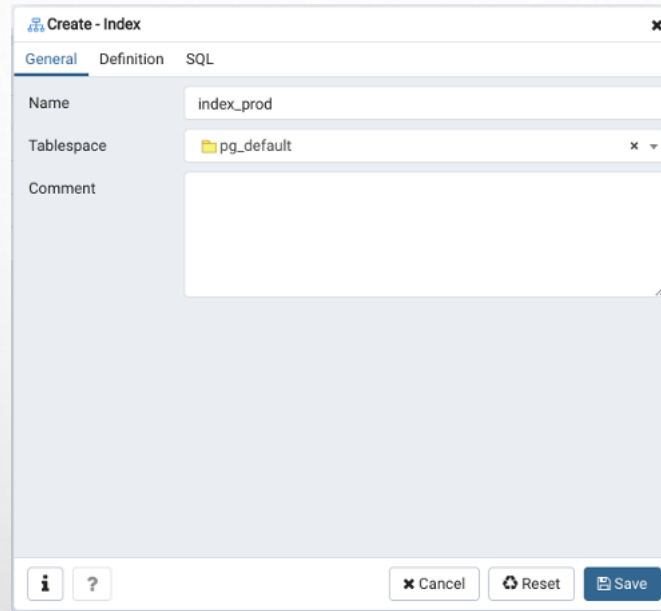
Se puede crear un índice en la columna apellidos con una sentencia `ALTER TABLE`:

```
alter table usuarios add index idx_apellidos (apellidos);
```

O bien, con una sentencia `CREATE INDEX`:

```
create index idx_nombre on usuarios(nombre);
```

Crear Índices (Create Index)



The screenshot shows a 'Create - Index' dialog box with the following fields and options:

- Name:** index_prod
- Tablespace:** pg_default
- Comment:** (empty text area)
- Buttons:** Cancel, Reset, Save

- ✓ Name: nombre del índice
- ✓ Tablespace: espacio de tablas en el que residirá el índice
- ✓ Comment: notas sobre el índice

Crear Índices (Create Index)

Create - Index

General **Definition** SQL

Access Method: btree

Fill factor:

Unique? ☐ No

Clustered? ☐ No

Concurrent build? ☐ No

Constraint: 1

Columns

Column	Operator class	Sort order	NULLs	Collation
c_name	varchar_ops	ASC	LAST	pg_catalog.\"aa_DJ\"

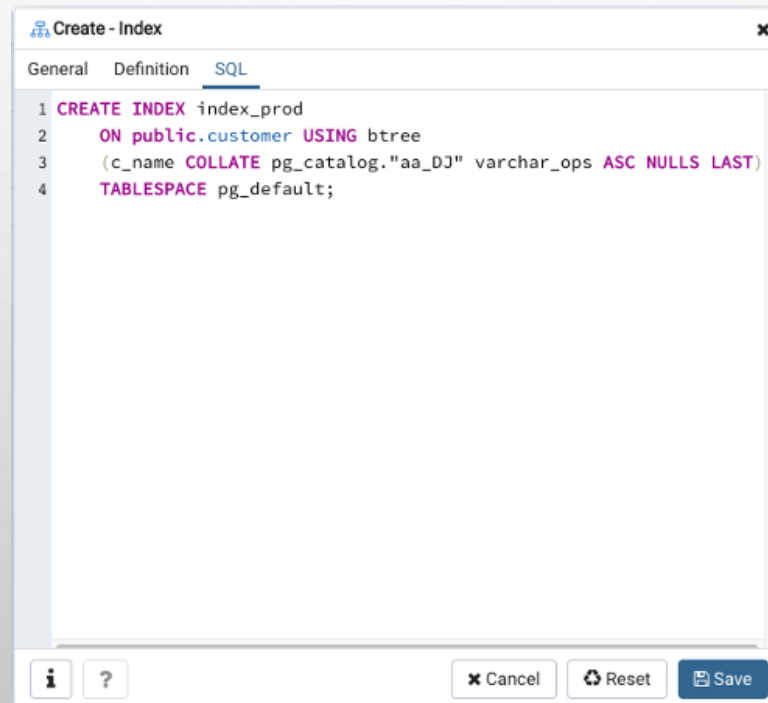
Include columns: Select the column(s)

Cancel Reset Save

- ✓ Access Method: btree, hash, GiST, GIN, GiST, BRIN.
- ✓ Fill factor: especifica qué tanto llenar cada página del índice
- ✓ Unique: No permite valores duplicados. El valor por defecto es No .
- ✓ Clustered: Si ordena la tabla físicamente por el índice
- ✓ Concurrent build: construir el índice sin tomar ningún bloqueo de tabla.

Crear Índices (Create Index)

- ✓ Constraint: imita las entradas en el índice a aquellas filas que satisfacen la restricción.
- ✓ Columns: columna (s) de la tabla en las que se creará el índice.
- ✓ ASC/ DESC : para especificar un orden (asc el predeterminado).



Tipos de índices disponibles en PostgreSQL

En PostgreSQL existen varios tipos de índices: **B-tree, Hash, GiST, SP-GiST, GIN, BRIN**. Cada uno utiliza un algoritmo diferente que se adapta mejor a cada tipos de consultas.

ÍNDICES B-TREE: El planificador de consultas de PostgreSQL utilizará un índice B-Tree siempre que una columna indexada esté involucrada en una comparación, usando alguno de estos operadores:

<, <=, =, >=, >, BETWEEN, IN, IS/IS NOT NULL, También LIKE y ~ si el patrón de búsqueda es una constante anclada al comiendo de la cadena.

ÍNDICES HASH: Almacenan un código hash de 32 bits a partir del valor de la columna indexada, solo pueden manejar comparaciones de igualdad simple.

ÍNDICES GiST: Los índices GiST no son un tipo de índice al uso sino una infraestructura dentro de la cual se pueden implementar muchas estrategias de indexación diferentes según la clase de operador. PostgreSQL incluye clases de operadores GiST para varios tipos de datos geométricos bidimensionales, que admiten consultas indexadas utilizando los operadores:

<<, &<, &>, >>, <<|, &<|, |&>, |>>, @>, <@, -=, &&

Los índices GiST son capaces de optimizar, por ejemplo, las consultas que buscan los lugares más cercanos a un punto geográfico dado.

Tipos de índices disponibles en PostgreSQL

ÍNDICES SP-GiST: ofrecen una infraestructura que admite varios tipos de búsquedas. Los operadores SP-GiST que incluye la versión estándar de PostgreSQL son: <<, >>, ~=, <@, <<|, |>>

ÍNDICES GIN: Los índices GIN se conocen como “**índices invertidos**” y son apropiados para datos que contienen múltiples componentes, como por ejemplo las matrices. Un índice invertido contiene una entrada para cada valor de componente.

Al igual que los dos anteriores, los índices GIN puede admitir muchas estrategias de indexación diferentes. Los operadores con los que se pueden usar un índice GIN son: <@, @>, = &&

ÍNDICES BRIN (Block Range Indexes): Para los tipos de datos que tienen un orden de lineal, los datos indexados corresponde a los valores mínimo y máximo en la columna para cada rango de bloque. El índice almacena un par de valores por conjunto de n filas. El tamaño del índice es mucho menor y aún más eficiente.

Los operadores con los que puede utilizar un índice BRIN son:

<, <=, =, >=, >

Índices Parciales

Un **índice parcial** es un índice construido sobre un subconjunto de una tabla; el subconjunto se define por una expresión condicional (llamado el predicado del índice parcial). El índice contiene entradas para sólo aquellas filas de la tabla que satisfacen el predicado.

Esto reduce el tamaño del índice, lo que acelerará las consultas que utilizar el índice. También se acelerará muchas operaciones de actualización de la tabla debido a que el índice no necesitan ser actualizados en todos los casos.

Ejemplos de tipos de datos

explain analyze select * from cliente where edad = 30;

	QUERY PLAN text
1	Gather (cost=1000.00..18447.33 rows=9800 width=53) (actual time=0.426..79.136 rows=10144 loops=1)
2	Workers Planned: 2
3	Workers Launched: 2
4	-> Parallel Seq Scan on cliente (cost=0.00..16467.33 rows=4083 width=53) (actual time=0.019..24.543 rows=3381 loop...)
5	Filter: (edad = 30)
6	Rows Removed by Filter: 329952
7	Planning Time: 0.646 ms
8	Execution Time: 79.445 ms

create index idx_cliente_edad on cliente(edad);
explain analyze select * from cliente where edad = 30;

	QUERY PLAN text
1	Bitmap Heap Scan on cliente (cost=112.38..11595.50 rows=9800 width=53) (actual time=2.050..6.457 rows=10144 loops=1)
2	Recheck Cond: (edad = 30)
3	Heap Blocks: exact=6697
4	-> Bitmap Index Scan on idx_cliente_edad (cost=0.00..109.92 rows=9800 width=0) (actual time=1.222..1.223 rows=10144 loop...)
5	Index Cond: (edad = 30)
6	Planning Time: 1.130 ms
7	Execution Time: 6.867 ms

Ejemplos de consultas con y sin índices

explain analyze select * from cliente where nombre = 'CLIENTE_500000';

	QUERY PLAN text	
1	Gather (cost=1000.00..17467.43 rows=1 width=53) (actual time=68.915..187.366 rows=1 loops=1)	
2	Workers Planned: 2	
3	Workers Launched: 2	
4	-> Parallel Seq Scan on cliente (cost=0.00..16467.33 rows=1 width=53) (actual time=35.366..49.228 rows=0 loop...	
5	Filter: (nombre = 'Cliente_500000')::text)	
6	Rows Removed by Filter: 333333	
7	Planning Time: 1.908 ms	
8	Execution Time: 187.408 ms	

create index idx_cliente_nombre on cliente(nombre); -- Creamos un índice por el campo nombre
explain analyze select * from cliente where nombre = 'CLIENTE_500000';

	QUERY PLAN text	
1	Index Scan using idx_cliente_nombre on cliente (cost=0.42..8.44 rows=1 width=53) (actual time=0.118..0.120 rows=1 loops=...	
2	Index Cond: (nombre = 'Cliente_500000')::text)	
3	Planning Time: 2.925 ms	
4	Execution Time: 0.145 ms	

Ejemplos de consultas con y sin índices

explain analyze select * from cliente where nombre_varchar = 'CLIENTE_500000';

	QUERY PLAN text	
1	Gather (cost=1000.00..17467.43 rows=1 width=53) (actual time=133.477..181.779 rows=1 loops=1)	
2	Workers Planned: 2	
3	Workers Launched: 2	
4	-> Parallel Seq Scan on cliente (cost=0.00..16467.33 rows=1 width=53) (actual time=47.016..62.934 rows=0 loop...	
5	Filter: ((nombre_varchar)::text = 'Cliente_500000')::text)	
6	Rows Removed by Filter: 333333	
7	Planning Time: 0.155 ms	
8	Execution Time: 181.817 ms	

create index idx_cliente_nombre_varchar on cliente(nombre_varchar);

explain analyze select * from cliente where nombre_varchar = 'CLIENTE_500000';

	QUERY PLAN text	
1	Index Scan using idx_cliente_nombre_varchar on cliente (cost=0.42..8.44 rows=1 width=53) (actual time=0.078..0.079 rows=1 loops=...	
2	Index Cond: ((nombre_varchar)::text = 'Cliente_500000')::text)	
3	Planning Time: 2.120 ms	
4	Execution Time: 0.100 ms	

Ejemplos de consultas con y sin índices

explain analyze select * from cliente where fecha_nacimiento between '1990-01-01' and '2000-01-01';

	QUERY PLAN text
1	Seq Scan on cliente (cost=0.00..26259.00 rows=142146 width=53) (actual time=0.080..115.381 rows=142508 loops=...
2	Filter: ((fecha_nacimiento >= '1990-01-01'::date) AND (fecha_nacimiento <= '2000-01-01'::date))
3	Rows Removed by Filter: 857492
4	Planning Time: 0.182 ms
5	Execution Time: 121.104 ms

create index idx_cliente_nombre_fecha_nacimiento on cliente(fecha_nacimiento);

explain analyze select * from cliente where fecha_nacimiento between '1990-01-01' and '2000-01-01';

	QUERY PLAN text
1	Bitmap Heap Scan on cliente (cost=1965.42..15356.61 rows=142146 width=53) (actual time=15.678..66.406 rows=142508 loops=1)
2	Recheck Cond: ((fecha_nacimiento >= '1990-01-01'::date) AND (fecha_nacimiento <= '2000-01-01'::date))
3	Heap Blocks: exact=11259
4	-> Bitmap Index Scan on idx_cliente_fecha_nacimiento (cost=0.00..1929.88 rows=142146 width=0) (actual time=13.012..13.012 rows=142508 loop...
5	Index Cond: ((fecha_nacimiento >= '1990-01-01'::date) AND (fecha_nacimiento <= '2000-01-01'::date))
6	Planning Time: 2.712 ms
7	Execution Time: 75.613 ms

Ejemplos de consultas con y sin índices

La instrucción "**EXPLAIN**" muestra el plan de ejecución previsto para una consulta, es decir, cómo el motor de la base de datos planea ejecutar la consulta y acceder a los datos.

Sin embargo, la instrucción "EXPLAIN" no proporciona información sobre el tiempo real de ejecución o los recursos utilizados. Es por eso que se utiliza la instrucción "ANALYZE" junto con "EXPLAIN" para ejecutar la consulta y medir su rendimiento.

La instrucción "**ANALYZE**" ejecuta la consulta y recopila información detallada sobre el tiempo y los recursos utilizados, como el tiempo total de ejecución, el número de filas escaneadas, el uso de CPU y la cantidad de memoria utilizada.

Ejemplos de consultas con y sin índices

EXPLAIN ANALYZE se utiliza para obtener información detallada sobre cómo se está ejecutando una consulta. Esta sentencia es útil para la optimización del rendimiento de las consultas.

Al utilizar EXPLAIN ANALYZE antes de una consulta, PostgreSQL ejecutará la consulta de manera simulada y mostrará información detallada sobre cómo se ejecutó la consulta, incluyendo información como:

- El plan de ejecución utilizado para la consulta
- El tiempo que tardó la consulta en ejecutarse
- La cantidad de filas que se escanearon para ejecutar la consulta
- El costo estimado y real de ejecutar la consulta

Ejemplos de consultas con y sin índices

Esta información puede ayudar a los desarrolladores a identificar cuellos de botella en las consultas, como consultas que escanean demasiadas filas o consultas que utilizan índices ineficientes. Con esta información, los desarrolladores pueden ajustar la consulta para mejorar su rendimiento.

Es importante tener en cuenta que "EXPLAIN ANALYZE" es una herramienta muy poderosa, pero también puede ser compleja de interpretar para aquellos que no están familiarizados con la planificación de consultas. Por lo tanto, es recomendable que los desarrolladores se familiaricen con los conceptos detrás de la planificación de consultas antes de utilizar esta sentencia en PostgreSQL.

Ejemplos de consultas con y sin índices

Bitmap Heap Scan y **Bitmap Index Scan** son dos técnicas de escaneo de datos utilizadas por PostgreSQL para recuperar filas de una tabla que cumplen con ciertas condiciones.

Bitmap Heap Scan: examina los datos almacenados en el heap (montón) de una tabla. Primero, se realiza un Bitmap Index Scan para identificar las páginas de heap que contienen las filas que cumplen con la condición de búsqueda. A continuación, PostgreSQL examina esas páginas de heap y recupera las filas correspondientes. Esta técnica es útil cuando la tabla es pequeña.

Bitmap Index Scan: examina los índices de una tabla en lugar de los datos almacenados en el heap. Primero, PostgreSQL utiliza el índice para identificar las páginas de heap que contienen las filas que cumplen con la condición de búsqueda. Luego, se utiliza un Bitmap para marcar las filas correspondientes. Por último, PostgreSQL utiliza ese Bitmap para buscar las filas reales en el heap. Esta técnica es útil cuando la tabla es grande .

Ejemplos de consultas con y sin índices

Planning time (tiempo de planificación) se refiere al tiempo que tarda en generar un plan de ejecución para la consulta. Este tiempo incluye la optimización de la consulta, la selección del mejor plan de ejecución y la asignación de recursos necesarios para la consulta. En general, cuanto más compleja es la consulta, mayor será el tiempo de planificación.

Execution time (tiempo de ejecución) se refiere al tiempo que tarda en ejecutar la consulta después de haber sido planificada. Este tiempo incluye la búsqueda de datos, la manipulación de datos y cualquier otro proceso que sea necesario para completar la consulta.

En resumen, ambos tiempos son importantes para evaluar el rendimiento de una consulta y para identificar cuellos de botella y oportunidades de mejora en la base de datos

Desventajas de los Índices

La siguiente consulta sirve para saber cuantos índices tiene una tabla.

```
select count(*) as cant_indices from pg_indexes  
where tablename = 'cliente';
```

	cant_indices bigint
1	4

```
select tablename, indexname, indexdef from pg_indexes  
where tablename = 'cliente';
```

	tablename name	indexname name	indexdef text
1	cliente	cliente_pkey	CREATE UNIQUE INDEX cliente_pkey ON public.cliente USING btree (id)
2	cliente	idx_cliente_nombre	CREATE INDEX idx_cliente_nombre ON public.cliente USING btree (nombre)
3	cliente	idx_cliente_nombre_varchar	CREATE INDEX idx_cliente_nombre_varchar ON public.cliente USING btree (nombre_varchar)
4	cliente	idx_cliente_fecha_nacimiento	CREATE INDEX idx_cliente_fecha_nacimiento ON public.cliente USING btree (fecha_nacimiento)

Esta consulta muestra la tabla, el nombre del índice y la definición de cada uno

Desventajas de los Índices

Podemos saber el tamaño de cada índice con la siguiente consulta




```
select indexrelname as index_name,  
pg_size_pretty(pg_total_relation_size(indexrelid)) as total_index_size  
from pg_stat_all_indexes where relname = 'cliente';
```

	index_name name	total_index_size text
1	cliente_pkey	21 MB
2	idx_cliente_nombre	30 MB
3	idx_cliente_nombre_varchar	30 MB
4	idx_cliente_fecha_nacimiento	7136 kB

Desventajas de los Índices

Crear índices no es gratuito, además de realizar las operaciones de inserción, eliminación y las actualizaciones sobre la tabla, debe actualizar el índice. Por otra parte, el índice también ocupa espacio físico, en ocasiones incluso más espacio que los propios datos.

```
select pg_size_pretty(pg_total_relation_size('cliente')) as "tamaño tabla + indice",  
       pg_size_pretty(pg_table_size('cliente')) as "tamaño tabla",  
       pg_size_pretty(pg_indexes_size( 'cliente ')) as "tamaño indices"
```

tamaño tabla + indice 	tamaño tabla 	tamaño indices 
text	text	text
177 MB	88 MB	89 MB

El tamaño del índice no es nada despreciable, de hecho la suma de los tamaños de los índices es mas grande que el tamaño de la tabla

Ejemplos índices

```
alter table cliente add column sexo varchar(10);
update cliente set sexo = case when id % 2 = 0 then 'masculino' else 'femenino' end;
explain analyze select * from cliente where sexo like 'femenino'
```

	QUERY PLAN text
1	Gather (cost=1000.00..31011.05 rows=1 width=62) (actual time=0.852..264.528 rows=500000 loops=1)
2	Workers Planned: 2
3	Workers Launched: 2
4	-> Parallel Seq Scan on cliente (cost=0.00..30010.95 rows=1 width=62) (actual time=0.333..179.406 rows=166667 loop...)
5	Filter: ((sexo)::text ~~ 'femenino'::text)
6	Rows Removed by Filter: 166667
7	Planning Time: 2.457 ms
8	Execution Time: 287.376 ms

```
create index cliente_sexo on cliente (sexo);
explain analyze select * from cliente where sexo like 'femenino'
```

	QUERY PLAN text
1	Bitmap Heap Scan on cliente (cost=5606.53..36436.53 rows=500400 width=62) (actual time=14.602..408.476 rows=500000 loops=1)
2	Filter: ((sexo)::text ~~ 'femenino'::text)
3	Heap Blocks: exact=12289
4	-> Bitmap Index Scan on cliente_sexo (cost=0.00..5481.43 rows=500400 width=0) (actual time=13.340..13.341 rows=500000 loop...)
5	Index Cond: ((sexo)::text = 'femenino'::text)
6	Planning Time: 2.945 ms
7	Execution Time: 438.273 ms

Ejemplos índices

```
alter table cliente add column pagos decimal(10,2);
update cliente set pagos = round(random() * (1000000 - 10000) + 10000)::numeric, 2);
create index cliente_pago on cliente (pagos) where pagos > 700000;
explain analyze select * from cliente where pagos = 550000
```


	QUERY PLAN text	
1	Gather (cost=1000.00..31828.43 rows=1 width=70) (actual time=99.435..105.172 rows=0 loops=1)	
2	Workers Planned: 2	
3	Workers Launched: 2	
4	-> Parallel Seq Scan on cliente (cost=0.00..30828.33 rows=1 width=70) (actual time=71.545..71.546 rows=0 loop...	
5	Filter: (pagos = '550000'::numeric)	
6	Rows Removed by Filter: 333333	
7	Planning Time: 3.546 ms	
8	Execution Time: 105.203 ms	

```
explain analyze select * from cliente where pagos = 750000
```

	QUERY PLAN text	
1	Index Scan using cliente_pago on cliente (cost=0.42..8.44 rows=1 width=70) (actual time=0.089..0.089 rows=0 loops=...	
2	Index Cond: (pagos = '750000'::numeric)	
3	Planning Time: 0.137 ms	
4	Execution Time: 0.105 ms	

Ejemplos de clase

explain analyze select nombre, presentacion **from** medicamento
where id_medicamento **not in** (**select** id_medicamento **from** tratamiento);

	QUERY PLAN text	
1	Seq Scan on medicamento (cost=0.00..12596289.75 rows=975 width=33) (actual time=109.712..566.916 rows=3 loops=...	
2	Filter: (NOT (ANY (id_medicamento = (SubPlan 1).col1)))	
3	Rows Removed by Filter: 1947	
4	SubPlan 1	
5	-> Materialize (cost=0.00..11929.05 rows=396070 width=4) (actual time=0.002..0.178 rows=2593 loops=1950)	
6	-> Seq Scan on tratamiento (cost=0.00..8400.70 rows=396070 width=4) (actual time=0.037..40.793 rows=396070 ...	
7	Planning Time: 3.689 ms	
8	Execution Time: 569.440 ms	

Ejemplos de clase

explain analyze select m.nombre, m.presentacion **from** medicamento m
left join tratamiento t **on** m.id_medicamento = t.id_medicamento **where** t.id_medicamento **is null**;

	QUERY PLAN text	🔒
1	Nested Loop Anti Join (cost=0.42..985.27 rows=3 width=33) (actual time=0.017..7.567 rows=3 loops=1)	
2	-> Seq Scan on medicamento m (cost=0.00..40.50 rows=1950 width=37) (actual time=0.010..0.141 rows=1950 loops=1)	
3	-> Index Only Scan using "Ref1197" on tratamiento t (cost=0.42..4.74 rows=203 width=4) (actual time=0.004..0.004 rows=1 loops=1...	
4	Index Cond: (id_medicamento = m.id_medicamento)	
5	Heap Fetches: 0	
6	Planning Time: 2.580 ms	
7	Execution Time: 7.586 ms	

explain analyze select m.nombre, m.presentacion **from** medicamento m
where not exists (**select** 1 **from** tratamiento t **where** t.id_medicamento = m.id_medicamento);

	QUERY PLAN text	🔒
1	Nested Loop Anti Join (cost=0.42..985.27 rows=3 width=33) (actual time=0.017..4.055 rows=3 loops=1)	
2	-> Seq Scan on medicamento m (cost=0.00..40.50 rows=1950 width=37) (actual time=0.011..0.122 rows=1950 loops=1)	
3	-> Index Only Scan using "Ref1197" on tratamiento t (cost=0.42..4.74 rows=203 width=4) (actual time=0.002..0.002 rows=1 loops=1...	
4	Index Cond: (id_medicamento = m.id_medicamento)	
5	Heap Fetches: 0	
6	Planning Time: 0.214 ms	
7	Execution Time: 4.072 ms	

Ventajas de los Índices

Evita Sobrecarga de CPU, sobrecarga de disco y concurrencia.

Con los índices evitamos hacer lecturas secuenciales.

Los índices nos permiten una mayor rapidez en la ejecución de las consultas tipo SELECT...

Y por último será una ventaja para aquellos campos que no tengan datos duplicados, si no es un campo con valores binarios (Ej. Masculino/Femenino, Si/No, True/False).

En aquellas tablas que se realizan operaciones de escritura (Insert, Delete, Update), esto es porque los índices se actualizan cada vez que se modifica una columna.

En tablas demasiado pequeñas puesto que no necesitaremos ganar tiempo en las consultas.

Tampoco son muy aconsejables cuando pretendemos que la tabla sobre la que se aplica devuelva una **gran cantidad de datos** en cada consulta.

Hay que tener en cuenta que ocupan espacio y en ocasiones incluso más espacio que los propios datos.

Nota

Ahora tenemos un dilema, ¿Usamos índices o no?

Pues como todo, depende, si tenemos una consulta en la que tenemos claro el resultado de la clausula WHERE la respuesta es sí, por el contrario, si tenemos un gran número de registros duplicados y lo que necesitamos la gran mayoría de veces es una lectura secuencial la respuesta es no.

Los índices se actualizan cada vez que se modifica la columna o columnas que utiliza. Por ello no es aconsejable usar como índices columnas en las que serán frecuentes operaciones de escritura (INSERT, UPDATE, DELETE).

Tampoco tiene sentido crear índices sobre columnas cuando cualquier “SELECT” sobre ellos va a devolver una gran cantidad de resultados; por ejemplo, una columna booleana que admita los valores SI/NO.

Tampoco es aconsejable usar índices en tablas demasiado pequeñas, ya que en estos casos no hay ganancia de rapidez frente a una consulta normal.

Nota

Si una columna es muy poco usada en una cláusula WHERE, no tiene mucho sentido indexar dicha columna. De esta manera, probablemente sea más eficiente sufrir el escaneo completo de la tabla las ocasiones en que se use esta columna en una consulta, que estar actualizando el índice cada vez que cambien los datos de la tabla.

Ante la duda, no tenemos otra alternativa que probar. Siempre podemos ejecutar algunas pruebas sobre los datos de nuestras tablas con y sin índices para ver como obtenemos los resultados más rápidamente. Lo único a considerar es que las pruebas sean lo más realistas posibles.

Finalmente, los índices ocupan espacio. A veces, incluso mas que la tabla de datos.

Ejemplos de Clase

```
explain analyze select nombre, presentacion  
from medicamento  
where id_medicamento not in (select id_medicamento from tratamiento);
```

```
explain analyze select m.nombre, m.presentacion  
from medicamento m  
left join tratamiento t on m.id_medicamento = t.id_medicamento  
where t.id_medicamento is null;
```

```
explain analyze select m.nombre, m.presentacion  
from medicamento m  
where not exists (select 1 from tratamiento t where t.id_medicamento = m.id_medicamento);
```




Gracias