PREVIRED

05/2016

Tablas e índices

Introducción

- **PostgreSQL soporta creación de tablas estándar SQL**
- **Pero además soporta:**
  - –Temporary (Temporales)
  - –Unlogged (Sin Log)
  - –Inherited (Con herencia)
  - –Typed (A partir de tipos)
  - –Foreign (Foráneas). Las revisaremos ccomo FDW Foreign Data Wrappers.
- **Un Ejemplo:**

```
CREATE TABLE logs ( log_id serial PRIMARY KEY,
user_name varchar(50),
description text,
log_ts timestamp with time zone NOT NULL DEFAULT current_timestamp);

CREATE INDEX idx_logs_log_ts ON logs USING btree (log_ts);
```

CREATE TABLE

```
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } | UNLOGGED ]
TABLE
[ IF NOT EXISTS ] table_name ( [  { column_name data_type [ COLLATE
collation ] [ column_constraint [ ... ] ]    | table_constraint     |
LIKE source_table [ like_option ... ] }    [, ... ]] )[ INHERITS (
parent_table [, ... ] ) ][ WITH ( storage_parameter [= value] [, ... ] )
| WITH OIDS | WITHOUT OIDS ][ ON COMMIT { PRESERVE ROWS | DELETE ROWS |
DROP } ][ TABLESPACE tablespace_name ]
```

Temporales

- Las tablas temporales son creadas con el calificador TEMPORARY o TEMP
- Las tablas temporales son eliminadas (Dropped) automáticamente al final la sesión o al finalizar la transacción (ON COMMIT)
- Las tablas permanentes con el mismo nombre no son visibles durante la sesión. Aunque las permantes pueden ser referenciadas utilizando el esquema.
- Cualquier indice creado también es descargado al final de la sesión.
- El daemon autovacuum no tiene acceso a estas tblas por lo que operaciones de vacuum o de análisuis deben ser hechas vía comandos SQL en la misma sesión.
  - Por ejemplo, si la tabla temporal participa de queries complejos es necesario ejecutar ANALYZE antes realizarlos y después de haber poblado de datos la tabla.
- GLOBAL y LOCAL están obsoletos y no hacen diferencia.

Herencia

- **PostgreSQL soporta herencia de tablas**
  - –Al parecer es el único RDBMS que soporta herencia (AFAIK)
- **La tabla hija "hereda" todas las columnas de la tabla "padre"**
  - –La tabla hija es creada con sus columnas y con todas las columnas de la tabla padre
  - –La relación padre-hija es almacenada por PostgreSQL
  - –Cualquier cambio posterior en la tabla padre es reflejado en la tabla hija
  - –Al consultar por la tabla padre, las filas de la tabla hija se incluyen
  - –No todo se hereda. Lo que no se hereda
    - •Primary keys
    - •Unique Constraints
    - •Indexes
  - –Check constraints se heredan, pero se pueden agregar otras

```
CREATE TABLE logs_2011 (PRIMARY KEY(log_id)) INHERITS (logs);
CREATE INDEX idx_logs_2011_log_ts ON logs USING btree(log_ts);
ALTER TABLE logs_2011 ADD CONSTRAINT chk_y2011
CHECK (log_ts >= '2011-1-1'::timestamptz
AND log_ts < '2012-1-1'::timestamptz );
```

Tablas sin log

- Para datos que pueden ser reconstituidos en caso de falla del disco o que no es necesario restaurarlos se puede utilizar la opción UNLOGGED
- Estas tablas no son parte de los log write-ahead. Si accidentalmente se desenfucha el servidor al momento de encenderlo y reiniciar , todos los datos se perderan durante el proceso de rollback.
- pg_dump permite no respaldar los datos unlogged
- data.
- Ejemplo

```
CREATE UNLOGGED TABLE web_sessions ( session_id text PRIMARY KEY, add_ts
time
stamptz, upd_ts timestamptz, session_state xml);
```

- La gran ventaja de estas tablas es que son muy rápidas. 15 veces más rápidas en la práctica.
- Si el servidor falla, PostgreSQL realizará un truncate de todas las tablas unlogged. (Si, Truncate signigica que borra todas las filas)
- Las tablas Unlogged no soportan los índices GiST

Tablas a partir de tipos

- Cada vez que se crea un tabl, PostgreSQL automáticamente crear tipo de dato compuesto que corresponde a tabla.
- A partir de la versión 9.0 es posible utilizar un tipo compuesto como plantillla para la creación de tablas.

```
CREATE TYPE usuario AS (nombre varchar(50), pwd varchar(10));
```

- Luego es factible crear talas con filas que son instancias de este tipo utilizando la claúsula OF

```
CREATE TABLE super_usuarios OF usuario (CONSTRAINT pk_su PRIMARY KEY
(usuario));
```

- Cuando se crean tablas a partir de tipos no es factible modificar las columnas vía ALTER.
- Sin embargo, los cambios hechos al tipo se propagan automáticamente.
- Los cambios deben ser hechos con CASCADE

```
ALTER TYPE usuario ADD ATTRIBUTE telefono varchar(10) CASCADE;
```

Constraints

- Las restricciones en PostgreSQL constraints son las más avanzadas y quizás las más complejas que cualquier otra base de datos (AFAIX)
- Se pueden controlar todos los aspectos asociados a los datos, la manera de realizar cascada, condiciones, índices, etc.
- Para efectos de este curso revisaremos:
  - Foreign key
  - Unique
  - Check
  - Exclusion

Foreign keys

- PostgreSQL sigue las mismas reglas que todas las bases de datos para integridad referencial.
- Es factible aplicar reglas de cascada de actualización y borrado para evitar registros huérfanos.
- RESTRICT es el comportamiento por omisión

```
set search_path=censo, public;
ALTER TABLE hechos ADD CONSTRAINT fk_hechos FOREIGN KEY (tipo_hecho_id)
REFERENCES tipo_hecho(tipo_echo_id)
ON UPDATE CASCADE ON DELETE RESTRICT;
CREATE INDEX fki_facts_1 ON facts (fact_type_id);
```

Unique Constraints

- Cada tabla solo tiene una llave primaria.
- Para indicar que el valor de una columna es único se utilizan las unique constraints o los unique indexes.
- Al agregar una unique constraint automáticamente se crea un unique index asociado
  - Esto es similar a las llaves primarias
- Las unique key constraints pueden participar en la part REFERENCES de una foreign key y no pueden tener valores NULL.
- Un unique index sin una unique key constraint permite valores NULL

```
ALTER TABLE logs_2011 ADD CONSTRAINT uq UNIQUE (user_name,log_ts);
CREATE UNIQUE INDEX name ON table (column [, ...]);
```

- Para agregar restricciones de unicidad a un subconjunto de datos se pueden utilizar unique index parciales.

Check Constraints

- Las Check constraints son condiciones que deben ser satisfechas por un campo o por un grupo de campos de una fila.
- El planificador de queries toma ventajas de las check constraints y descarta aquellos que no cumplen las restricciones.

```
ALTER TABLE logs ADD CONSTRAINT chk CHECK (user_name =
lower(user_name));
```

- Otro ejemplo:

```
CREATE TABLE productos(numero integer,numero text,
precio numeric CONSTRAINT precio_positivo CHECK (precio> 0));
```

- asdasd

Exclusion Constraints

- Aparecieron en la versión 9.0.
- Estas restricciones permiten incorporar operadores adicionales para reforzar la unicidad y que no pueden ser satisfechas por una doncidicón de igualdad.
- Son muy útiles para resolver problemas de agendamiento.
- En PostgreSQL 9.2 aparecieron los tipos de datos de rango , que son muy usados para las exclusiones.
- Las exclusiones funcionan muy bien enconjunto con lo índices GiST indexes, pero también es factible usar índices compuestos B-Tree
  - Se debe instalar la extensión btree_gist extension.
- Ejemplode uso de exclusioens es el agendamiento de un recurso. Supongamos que existen una cierta cantidad de salas de reuniones y queremos prevenir que dos reuniones calcen en horario en la misma sala.
  - Se puede usar el operador && para chequear el traslape de horarios y la igualda para el número de sala

```
CREATE TABLE reuniones(id serial primary key, sala smallint, horario
tstzrange);
ALTER TABLE reuniones ADD CONSTRAINT ex_reuniones
EXCLUDE USING gist (sala WITH =, horario WITH &&);
```

- PostgreSQL automácticamente crea un índice asociado a la restricción

Indices

- **Tratar de cubrir de manera completa el tema de índices escapa al alcance del curso**
- **PostgreSQL posee 4 tipos de índices “Out of the box”**
- **Estos índices se pueden mezclar**
  - –Por ejemplo , una columna puede usar un B-Tree, otra columna utilizar GIST
  - –El Query Planner resuelve cual índice utilizar
  - –A esto se le denomina bitmap index scan strategy.

B-Tree

- Los índices B-Tree, son de propósito general y se encuentran en la mayoría de las bases de datos.
- Es factible obtener rendimientos razonables solo utilizando B-Trees
- Si no se indica el tipo del índice se utiliza B-Tree
- Es el único tipo de índice disponible para primary keys y unique keys.

Indices

GiST

- Generalized Search Tree (GiST), es un índice optimizado para búsquedas de texto (full-text search), espaciales, de datos científicos, no estructurados y jerárquicos.
- Aunque no se pueden utilizar en unique keys, se puede lograr el mismo efecto con una restricción de exclusión.
- GiST es un índice de tipo lossy (con pérdidas), en el sentido que el índice en sí mismo no almacena los valores de lo que está indexando. Por ejemplo, para un polígono puede guardar una caja, por lo tanto, se necesita un paso extra para recuperar los valores

Indices

GIN

- Generalized Inverted Index (GIN) es el índice preferencial para búsquedas de texto y jsonb en PostgreSQL.
- Muchas extensiones como hstore y pg_trgm la usan
- GIN es una variante de GiST, pero sin pérdidas. GIN hace una copia de los valores de las columnas que son parte del índice
- Si solo se requiere traer datos asociados a las columnas asociadas GIN es más rápido que GIST.
- Sin embargo, la copia extra requerida por GIN hace que sea más lento de actualizar que GIST.
- Como cada fila del índice tiene un límite de tamaño tampoco es factible utilizar GIN para objetos grandes como documentos hstore grandes o texto.
- Ejemplo si es un documento de 600 páginas que se almacena en una columna text, no es factible utilizar un índice GIN en esa columna.

Indices

SP-GiST

- Space-Partitioning Trees Generalized Search Tree (SP-GiST)
- Disponibles a patir de la versión 9.2
- Se utilizan en las mismas situaciones que GiST, pero pueden ser mucho más rápidos para ciertos tipos de datos y distribuciones de ellos.
- Los tipos geométricos nativos de PostgreSQL como punto (point) o caja (box) soportan SP-GiST
- Text también es soportado
- En la versión 9.3 se agregaron los tipos de rango.
- En el futuro PostGIS también lo utilizará.

Indices

hash

- Los índices hash, se utilizaron mucho antes de que aparecieran GiST y GIN.
- Sin embargo, hoy son más usados GiST y GIN por sobre hash, ya que tienen un mejor desempeño
- El log write-ahead no matiene un registro de los índices hash
  - por lo tanto no se pueden utilizar en replicación basada en WAL
- Hash tiene un estado actual de “legacy”
- Se recomiendo no utilizarlo

Indices

- B-Tree-GiST/B-Tree-GIN
- Más allá de lo que viene instalado en PostgreSQL existen los índices compuestos B-Tree-GiST r B-Tree-GIN
- Estan disponibles como extensiones.
- Estos híbridos soportan los operadores especializados de f GiST o GIN, pero también ofrecen la performance de indexabilidad del operador de igualdad tan buena como los índices B-Tree
- Se utilizan para índices compuestos por múltiples columnas.

Operadores

- ¿Por qué el Query Planner no utiliza mi índice?
- Los índices trabajan con ciertos tipos de datos y ciertos operadores de comparación
- Por ejemplo para rangos el operador de traslape (overlap &&), se utiliza mucho en rangos, pero tienen poca aplicabilidad en búsquedas de texto.
- Qué tipos de operadores soporta B-Tree

```
SELECT am.amname AS index_method, opc.opcname AS opclass_name,
opc.opcintype::regtype AS indexed_type, opc.opcdefault AS is_default
FROM pg_am am INNER JOIN pg_opclass opc ON opc.opcmethod = am.oid
WHERE am.amname = 'btree'
ORDER BY index_method, indexed_type, opclass_name;
index_method | opclass_name | indexed_type | is_default
-------------+--------------------+--------------------------+------
------
btree | bool_ops | boolean | t
:
btree | text_ops | text | t
btree | text_pattern_ops | text | f
btree | varchar_ops | text | f
btree | varchar_pattern_ops | text | f
```

Indices

- Cada índice podemos ver que actúa sobre un (1, uno) conjunto de operadores
- Con esto podemos evaluar que B-Tree con operadores de texto (text_ops o varchar_ops) no incluye el operador ~~ (LIKE).
- Ninguna búsqueda por LIKE utilizará ese índice
- Sin embargo, se puede crear el índice de tal forma que si sea utilizado.

```
CREATE INDEX idx1 ON census.lu_tracts USING btree (tract_name
text_pattern_ops);
```

- Si se requiere más de un conjunto de operadores se debe crear más de un índice
- Para agregar las operaciones comunes de texto (text_ops) se debe crear un índice adicional

```
CREATE INDEX idx2 ON census.lu_tracts USING btree (tract_name);
```

- Con esto tenemos dos índices sobre la misma columna (No existe un límite para la cantidad de índices para una columna)
- El Planner eligirá idx2 para queries que hagan búsquedas de igualdad y el índice idx1 para comparaciones utilizando like.

Indices y funciones

- PostgreSQL permite crear índices que usen funciones.
- Por ejemplo PostgreSQL es una base de datos case-sensitive. Por eso es factible construir un índice ocupando la función upper y así superar este obstáculo:

```
CREATE INDEX fidx ON featnames_short
USING btree (upper(fullname) varchar_pattern_ops);
```

- Esto permite que consultas como:

```
SELECT fullname FROM feat
names_short WHERE upper(fullname) LIKE 'S%';
```

- Utilicen el índice
- Es importante utilizar la misma función al momento de realizar la consulta de tal forma de asegurar el uso del índice.

Indices parciales

- Los índices parciales, llamados también índices con filtros o filtraados, son índices que solo abarcan filas que calzan con una condición WHERE.
- Por ejemplo si se tienen un tabla de un 1.000.000 de filas pero lo que interesa son 10.000, esta es una situación donde amerita el uso de índices parciales. Estos índices son muy rápidos porque se pueden operar completamente en RAM.
- Ejemplo queremos filtrar solo suscriptores activos.

```
CREATE TABLE suscriptores(
id serial PRIMARY KEY,
nombre varchar(50) NOT NULL, tipo varchar(50),
activo boolean);
```

- Y luego el índice parcial

```
CREATE UNIQUE INDEX uq_suscriptor ON suscriptores USING
btree(lower(nombre)) WHERE activo;
```

Indices parciales

- Las funciones que se utilicen en la condición WHERE deben ser inmutable. Esto significa que por ejemplo, no puede utilizar funciones de tiempo como CURRENT_DATE o datos de otras tablas
- Se debe tener cuidado que al momento de seleccionar datos con un las condiciones deben ser un subconjunto del índice
- Una manera de garantizar esto, es con el uso de vistas

```
CREATE OR REPLACE VIEW vw_suscriptores AS
SELECT id, lower(nombre) As nombre FROM suscriptores WHERE activo= true;

SELECT * FROM vw_suscriptores WHERE nombre= 'sandy';
```

Indices multicolumnas

•Durante esta clase hemos visto índices multicolumnas.

```
CREATE INDEX idx ON suscriptores USING btree (tipo, upper(nombre)
varchar_pattern_ops);
```

•El Planner de PostgreSQL planner utiliza una estrategia llamada bitmap index scan que automática intenta combinar índices al vuelo, a menudo índices de una sola columna para lograr el mismo resultado que un índice multicolumna.
•Quizás una buena estrategia en ciertos casos es crear índices monocolumna y dejar al planner que resuelva.

Formalidad

- **Para crear un índice:**

```
CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] [ [ IF NOT EXISTS ] name ] ON
table_name [ USING method ]    ( { column_name | ( expression ) } [
COLLATE collation ] [ opclass ] [ ASC | DESC ] [ NULLS { FIRST | LAST }
] [, ...] )    [ WITH ( storage_parameter = value [, ... ] ) ]    [
TABLESPACE tablespace_name ]    [ WHERE predicate ]
```

- **asdasdas**

Ejemplos

•Crear un índice B-tree en la columna title de la tabla films:

```
CREATE UNIQUE INDEX title_idx ON films (title);
```

•Crear un índice en la columna título para crar búsquedas eficientes case insensitive

```
CREATE INDEX ON films ((lower(title)));
```

•Un índice con un collation distinto al default:

```
CREATE INDEX title_idx_german ON films (title COLLATE "de_DE");
```

•Crear un índice con el ordenamiento de nulls distinto al default:

```
CREATE INDEX title_idx_nulls_low ON films (title NULLS FIRST);
```

•Crear un índice con un factor de llenado distinto al default:

```
CREATE UNIQUE INDEX title_idx ON films (title) WITH (fillfactor = 70);
```

•Crear un índice GIN con los fast updates deshabilitados:

```
CREATE INDEX gin_idx ON documents_table USING GIN (locations) WITH
(fastupdate = off);
```

Ejemplos

•**Crear un índice en un tablespace específico:**

```
CREATE INDEX code_idx ON films (code) TABLESPACE indexspace;
```

•**Crear un índice GiST utilizando box sobre un punto para poder realizar búsquedas eficientes utilizando el operador de traslape:**

```
CREATE INDEX pointloc    ON points USING gist (box(location,location));
SELECT * FROM points    WHERE box(location,location) &&
'(0,0),(1,1)'::box;
```

•**Crear un índice sin bloquear las escrituras de una tabla:**

```
CREATE INDEX CONCURRENTLY sales_quantity_index ON sales_table
(quantity);
```