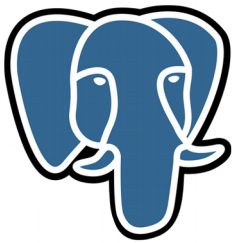
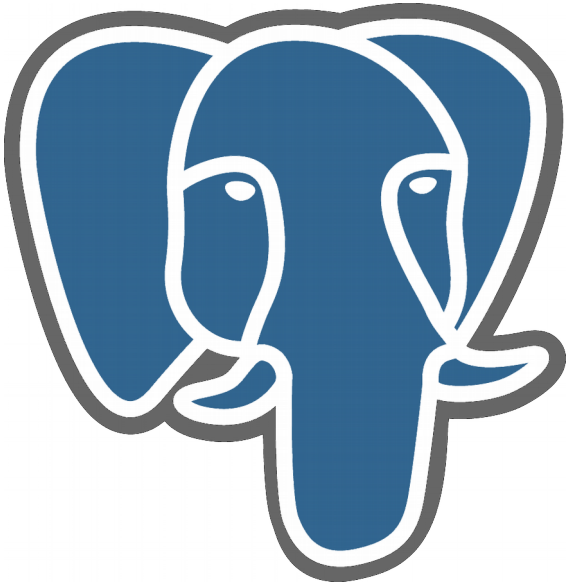


TodoPostgreSQL

Particionamiento de tablas

INSTRUCTOR:

José Segovia <info@todopostgresql.com>



Conceptos sobre particionado de tablas

- El particionado de tablas es una forma de **escalabilidad** dentro de la propia base de datos (*escalado vertical*).

- Se basa en dividir una tabla grande en otras tablas más pequeñas, de forma que **subconjuntos disjuntos de datos**, originalmente en la misma tabla, vayan a **tablas separadas**, y las operaciones habituales resulten más eficientes al ejecutarse sobre tablas más pequeñas.

Conceptos sobre particionado de tablas

- ¿Cuándo particionar? Depende de la aplicación y el rendimiento, pero típicamente con millones de registros.

- El particionado puede hacerse a nivel de aplicación, pero también **a nivel de base de datos** con PostgreSQL :)

Conceptos sobre particionado de tablas

- Un caso habitual para particionado es dividir una tabla grande con datos en series temporales (históricos) en tablas más pequeñas, donde las pequeñas tienen datos referentes a períodos concretos (por ejemplo, años).

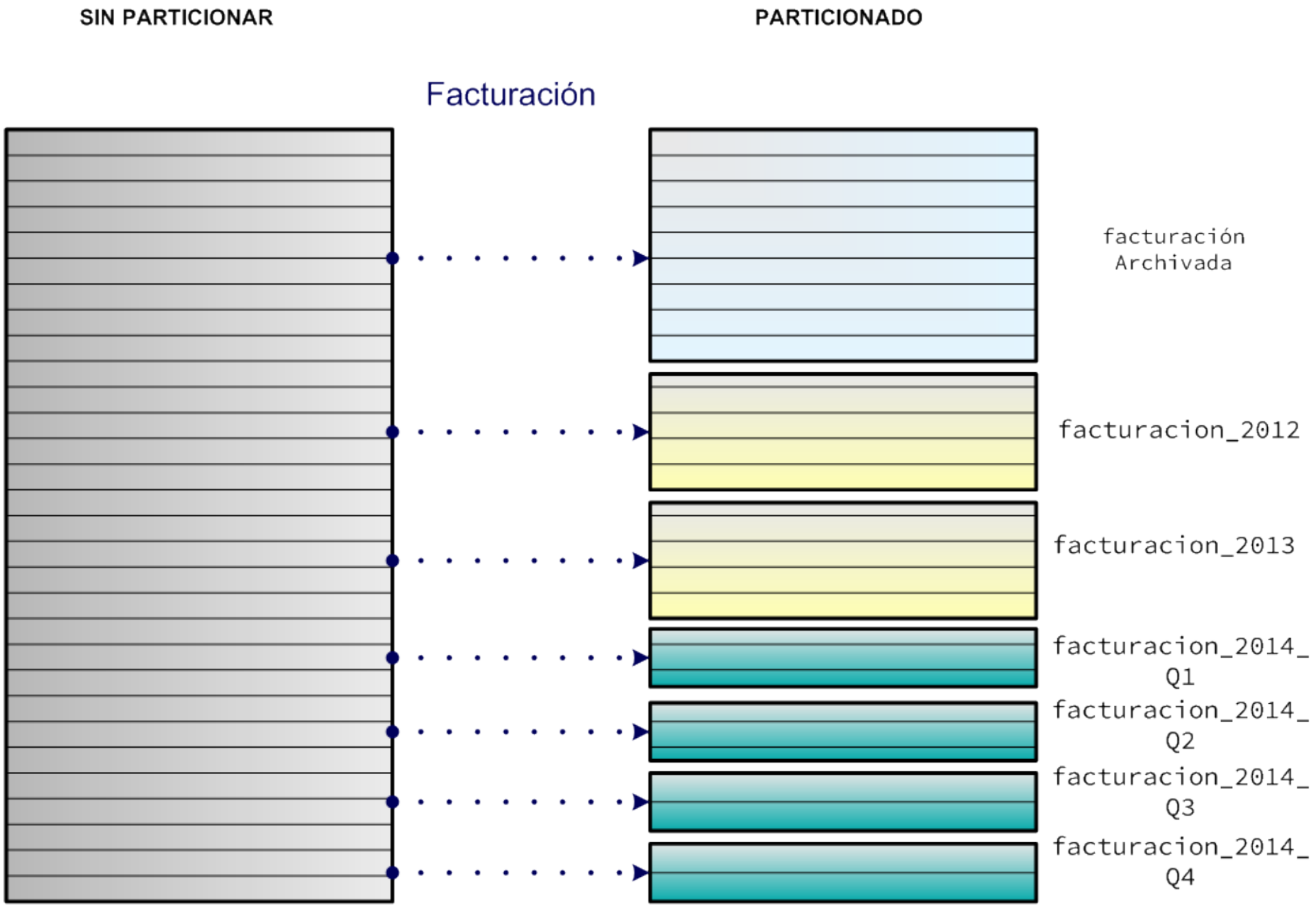
- Así, cada partición puede contener datos de sólo un año. Consultas habituales sobre datos de un año completo, referenciarán sólo a una de las tablas particionadas.

Conceptos sobre particionado de tablas

- El particionado **no es necesariamente uniforme**.

Si se particionan datos de facturación en años, nada impide que el año en curso esté también particionado por trimestres, o que datos históricos de más de 5 años estén todos en una única tabla.

Particionado



Ventajas del particionado de tablas

- Las operaciones que realizan *seq scans* (no usan índice) son lineales con el número de filas. Reducirlo aumenta el rendimiento.

- Aunque se usen índices, éstos son más grandes y también son más lentos de recorrer.

Ventajas del particionado de tablas

- Facilita la localización de datos (y por tanto las consultas).

- Permite gestionar la *"temperatura"* de los datos: datos más frecuentemente usados pueden estar en particiones más pequeñas para que su operativa sea más rápida.

Ventajas del particionado de tablas

- Permite borrar datos muy eficientemente (DROP TABLE frente a DELETE, que además genera presión MVCC).

- Usando tablespaces, se pueden asignar diferentes particiones a diferentes tablespaces, que a su vez estén sobre medios físicos de almacenamiento de diferentes características de rendimiento y tamaño.

Ventajas del particionado de tablas

- **Datos calientes.** Facturación año actual. SSDs.
- **Datos templados**. Facturación 5 años anteriores. Discos SAS.
- **Datos fríos**. Histórico de facturación. Discos SATA.

Ventajas del particionado de tablas

- El particionado + tablespaces también puede usarse para crecer una tabla sobre discos ya llenos no ampliables.

- El rendimiento de los índices puede mejorar drásticamente si en una partición el índice (o parte relevante) cabe en memoria, frente a la tabla completa donde la parte relevante pudiera no caber en memoria y requerir mucho más I/O para contemplar también partes no relevantes.

Ventajas del particionado de tablas

- Operaciones de mantenimiento como REINDEX, VACUUM, etc, pueden tener parámetros de configuración diferentes.

- Técnicamente, se pueden asignar permisos diferentes (como denegar la actualización o borrado) a cada partición.
  La facturación de años que no sean el actual ni el anterior no se pueden borrar ni modificar).

Criterios de particionado

El criterio de partición puede ser de varios tipos posibles:

1. **En serie** (por rangos). De un rango absoluto, se determinan subrangos, cada uno para una partición (ej. series temporales, como facturación).

2. **En lista** (tipos enumerados). El dato que determina la partición tiene un conjunto limitado de valores, y se elige un valor para cada partición (ej. tipos de productos).

Criterios de particionado

El criterio de partición puede ser de varios tipos posibles:

3. Por **función** de partición. Una función determina un algoritmo para decidir la partición resultante.

4. **Mixto**. Una combinación de los anteriores.

PostgreSQL como un OORDBMS: herencia de tablas

- PostgreSQL es una base de datos relacional (RDBMS).

Sin embargo, desde casi sus inicios, se añadieron algunas **funcionalidades de orientación a objetos** (OO) que claramente distinguieron a PostgreSQL de la competencia.

PostgreSQL como un OORDBMS: herencia de tablas

- La herencia de tablas permite:
  - Crear tablas que heredan la definición de columnas (incluyendo NOT NULL) y las restricciones CHECK de las mismas.
  - Que los SELECT sobre la tabla padre de la jerarquía incluyan, por defecto, todos los resultados de todas las tablas hijas (se puede usar SELECT ONLY para consultar sólo de la padre).

Herencia de tablas en PostgreSQL

- Creación de una tabla heredada:

```
CREATE TABLE tabla_hija (columnaHija tipoDatos…)
INHERITS (tablaPadre1, tablaPadre2…).
```

En la herencia múltiple, se produce error sólo si hay columnas de igual nombre pero diferentes tipos.

Herencia de tablas en PostgreSQL

- Usar "\d+ ..." para listar tablas hijas de una tabla padre.

- Las restricciones de tabla (incluyendo FKs), índices, etc, **no se heredan**. Pueden volver a definirse en las tablas hijas.

Particionado de tablas en PostgreSQL

- Se basa en el uso de la herencia de tablas.

- Tanto la tabla padre como cada una de las particiones contienen las mismas columnas y restricciones de datos (esto es, las tablas hijas no añaden columnas).

- La tabla padre es sólo la "definición": **no** debe tener datos.

Particionado de tablas en PostgreSQL

- Cada tabla hija implementa una restricción de datos adicional, de forma que los datos de la tabla hija correspondan con la definición de la partición (bien sea por rango, lista, función, etc). Dichas restricciones **no** deben solaparse (PostgreSQL no lo impide).

Particionado de tablas en PostgreSQL

- Opcionalmente (suele ser recomendable), **crear índices para cada partición resultante**.

- ¡**No** insertar nada en la tabla padre! (se puede evitar con restricciones como RULEs o permisos GRANT).

Ejemplo de particionado

```
CREATE TABLE facturacion (numero_factura varchar,
fecha_factura date, importe numeric);

CREATE TABLE facturacion_2016 (CHECK (fecha_factura
BETWEEN '2016-1-1' AND '2016-12-31'))
INHERITS (facturacion)
```

Ejemplo de particionado

```
CREATE TABLE facturacion_2015 (CHECK (fecha_factura
BETWEEN '2015-1-1 AND '2015-12-31'))
INHERITS (facturacion);

CREATE TABLE facturacion_historico (CHECK
(fecha_factura < '2015-1-1'))
INHERITS (facturacion);
```

Constraint exclusion

Configurar **constraint_exclusion = partition**. Este parámetro determina si el planificador de queries analiza las constraints de las tablas.

- En off, no lo hace nunca.
- En on, lo hace siempre (lo que puede tener un overhead importante incluso en queries simples).
- En partition analiza las restricciones de tablas en una jerarquía de herencia y realiza la query sólo en las tablas que aplique.

Constraint exclusion

```
SELECT ... FROM facturacion WHERE fecha_factura = '2016-6-20';
```

constraint_exclusion = (partition|on): consulta sólo la tabla facturacion_2016.

constraint_exclusion = off: consulta todas las tablas hijas y la padre.

Transparencia en INSERT

- Gracias a las características de OO de PostgreSQL y al parámetro **constraint_exclusion** se logra **transparencia en SELECT, UPDATE y DELETE**: con operar sobre la tabla padre, basta para obtener los resultados correctos y de forma eficiente.

Transparencia en INSERT

- Pero los **INSERT se realizan siempre sobre la tabla explícita** que se mencione en el comando (luego **no es transparente**: debe ser la aplicación quien lo haga). Alternativamente, puede dotarse de **transparencia**:
  - **Usando RULEs** sobre la tabla padre, que reescriben la query para insertar en la hija correspondiente.
  - **Usando TRIGGERs** sobre la tabla padre, que analizan los datos y deciden en qué tabla realizar la operación.

Transparencia en INSERT: RULEs

```
CREATE RULE facturacion_insert_facturacion_2016_rule AS ON INSERT
TO facturacion WHERE fecha_factura BETWEEN '2016-1-1' AND '2016-
12-31' DO INSTEAD  INSERT INTO facturacion_2016 VALUES (NEW.*);
```

```
CREATE RULE facturacion_insert_facturacion_2015_rule AS ON INSERT
TO facturacion WHERE fecha_factura BETWEEN '2015-1-1' AND '2015-
12-31' DO INSTEAD  INSERT INTO facturacion_2015 VALUES (NEW.*);
```

```
CREATE RULE facturacion_insert_facturacion_historico_rule AS ON
INSERT TO facturacion WHERE fecha_factura < '2015-1-1' DO INSTEAD
 INSERT INTO facturacion_historico VALUES (NEW.*);
```

Transparencia en INSERT: TRIGGERs

```
CREATE FUNCTION facturacion_insert() RETURNS TRIGGER AS $$
    BEGIN
        IF NEW.fecha_factura BETWEEN '2016-1-1' AND '2016-12-31' THEN
            INSERT INTO facturacion_2016 VALUES (NEW.*);
            ELSIF NEW.fecha_factura BETWEEN '2015-1-1' AND '2015-12-31' THEN
                INSERT INTO facturacion_2015 VALUES (NEW.*);
            ELSIF NEW.fecha_factura < '2015-1-1' THEN
                INSERT INTO facturacion_historico VALUES (NEW.*);
        END IF;
        RETURN NULL;
    END;
$$ LANGUAGE plpgsql;
CREATE TRIGGER facturacion_insert BEFORE INSERT ON facturacion
FOR EACH ROW EXECUTE PROCEDURE facturacion_insert();
```

Sistematización del proceso de particionado

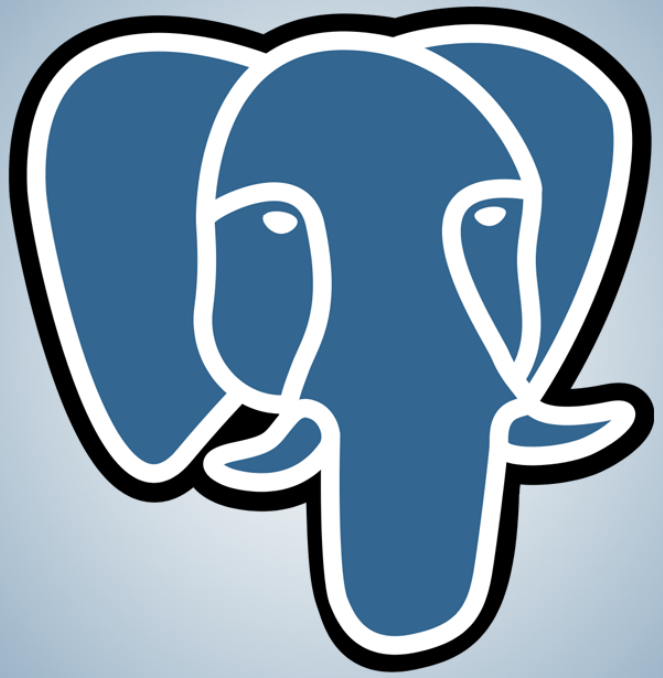
- Como se ha podido ver, el procedimiento de creación de tablas **es muy sistematizable** (escritura de los nombres de tablas, constraints asociadas, etc).

- También en la generación de los índices de las tablas, constraints que no se heredan (PKs, FKs, etc). Incluso la creación de RULEs es sistematizable.

Sistematización del proceso de particionado

- Se recomienda escribir funciones plpgsql que, de forma automática, generen las tablas, índices y rules (si aplica).

- Que se ejecuten por un DBA (no por otro trigger). Es muy peligroso (concurrencia) generar tablas como triggers de INSERTs, por ejemplo.



todopostgresql.com