Recherche zu PostgreSQL Features

Mengenoperatoren

Mengenoperatoren in SQL ermöglichen das Kombinieren von Resultaten mehrerer Abfragen zu einem einzigen Resultat.
Wichtig hierbei ist, dass alle Abfragen dieselbe Anzahl Spalten mit den gleichen Datentypen in derselben Reihenfolge erzeugen.

UNION

Bei UNION werden alle Duplikate aus dem Resultat gelöscht. Sprich, alle Datensätze, welche für alle selektierten Spalten die gleichen Werte aufweisen, werden zu einem einzelnen Datensatz zusammengefasst.

```
select emp_id as person_id
from employee
union
select cust_id as person_id
from customer;
```

UNION ALL

Der Mengenoperator UNION ALL funktioniert genau gleich wie UNION, mit dem Unterschied, dass alle doppelten Datensätze auch im Resultat aufgeführt werden.

```
select emp_id as person_id
from employee
union all
select cust_id as person_id
from customer;
```

INTERSECT

Mit INTERSECT wird die Schnittmenge zweier Resultate ausgegeben. Es werden also nur diese Datensätze aufgeführt, welche in den Resultaten beider Abfragen vorkommen.

```
select emp_id as person_id
from employee
intersect
select cust_id as person_id
from customer;
```

EXCEPT

Mit EXCEPT habe ich die Möglichkeit die Menge an Datensätzen der zweiten Abfrage von der ersten Abfrage «abzuziehen». In anderen Worten, zeige mir alle Datensätze des Resultats der Abfrage 1, ohne mir die Datensätze des Resultats von Abfrage 2 zu zeigen.

```
select emp_id as person_id
from employee
except
select cust_id as person_id
from customer;
```

Partial Indexes

Ein Partial Index dient wie ein herkömmlicher Index zur Performance-Verbesserung einer SQL-Abfrage. Die Indexierung erfolgt allerdings nur auf einem Teilset der gesamten Datenmenge. Dadurch sinkt auch der Speicherbedarf des Indexes.

Die Anwendung von Partial Indexes macht Sinn für seltene Typen von Datensätzen. Da häufige Datensätze ohnehin schnell gefunden werden, macht eine Indexierung nicht immer Sinn.

Views

Eine View ist eine Art «On-Demand» Tabelle, welche bei jeder Referenzierung eine vordefinierte Abfrage ausführt.

Der Unterschied zu einer herkömmlichen Tabelle ist, dass sie keinem eigenen, physischen Speicher unterliegt, sondern bestehende Datensätze temporär zu einer neuen Tabelle aggregiert.

Sie kann also ganz normal wie jede andere Tabelle in einer SELECT-Abfrage verwendet werden. UPDATE-, DELETE- und INSERT-Abfragen funktionieren nicht standardmässig in PostgreSQL, sondern müssen explizit erlaubt werden.

Beispiel-View auf der Bank-Datenbank:

```
create or replace view customer_balance as
select c.cust_id, coalesce(b.name, i.fname || ' ' || i.lname), a.avail_balance
from customer c
left join individual i on c.cust_id = i.cust_id
left join business b on c.cust_id = b.cust_id
join account a on c.cust_id = a.cust_id;
```

Automatisch fortlaufende Primärschlüssel

Um in PostgreSQL fortlaufende Primärschlüssel auf einer Tabelle zu generieren gibt es zwei weit verbreitete Ansätze. Beide legen intern eine Sequenz auf der Tabelle an und entfernen diese wieder mit dem Löschen der Spalte/Tabelle.

Generated Always As Identity (moderner Ansatz, Teil des SQL Standards):

```
create table customer (
   customer_id int generated always as identity primary key,
   name text
);
```

Serial (früherer Ansatz, PostgreSQL spezifisch):

```
create table customer (
   customer_id int serial primary key,
   name text
);
```

Triggers

Ein Trigger wird definiert, um eine Trigger-Funktion auszuführen, wenn das entsprechende Event auftritt. Trigger können auf INSERT-, UPDATE- und DELETE-Events reagieren. Wichtig ist, dass die Trigger-Funktion bereits existiert, wenn der Trigger erstellt wird. Neben pgSQL kann die Trigger-Funktion auch in Programmiersprachen wie Python oder C geschrieben werden.

Für die Bank-Datenbank würde es Sinn machen, einen Trigger zu erstellen, um die verfügbare Guthaben nach einer Transaktion zu aktualisieren.

Erstellen der Trigger-Funktion:

```
create or replace function update_account_after_transaction()
returns trigger as $$
begin
   update account
   set
      avail_balance = avail_balance + new.amount,
      last_activity_date = current_date
   where account_id = new.account_id;

   return new;
end;
$$ language plpgsql;
```

Erstellen des Triggers:

```
create or replace trigger trg_update_account_after_transaction
after insert on transaction
for each row
execute function update_account_after_transaction();
```