UE (GK+EK) Datenbankseitige Programmierung

Verfasser: **Leonhard Stransky, 4AHIT**

Datum: **24.02.2024**

Projektbeschreibung

Moderne Datenbankmanagementsysteme wie PostgreSQL erlauben es, Programmcode direkt in der Datenbank laufen zu lassen. Dabei werden die Daten nicht wie sonst ueblich mittels SELECT-Queries zum Client kopiert, dort bearbeitet und ausgewertet (und ggf wieder mit INSERT/UPDATE/DELETE Statements zurueck zum Server geschickt) -- stattdessen bleiben die Daten am Server und die Verarbeitung geschieht im DBMS selbst. Dies kann in der Praxis enorme Geschwindigkeitsvorteile bringen, schon alleine dadurch, dass weniger Daten zwischen Client und Server hin- und hergeschickt werden muessen. In SQL ist der generelle Ueberbegriff hierfuer Stored Procedures.

Wir werden die Datenbank aus der letzten Laborangabe zur Verwaltung von Kontodaten verwenden bzw. erweitern:

```
CREATE TABLE accounts (
    client_id integer primary key,
    amount decimal,
    deactivated boolean
);
```

```
CREATE TABLE transfers (
    transfer_id integer primary key,
    from_client_id integer,
    to_client_id integer,
    date date,
    amount decimal
);
```

Programmcode kann auf verschiedene Art und Weise in die Datenbank eingebunden und damit auch aufgerufen werden, die gebraeuchlichsten Methoden sind Functions, Procedures und Trigger:

Functions

Eine einfache eigene Funktion kann in Postgres zum Beispiel mit

```
CREATE OR REPLACE FUNCTION meineFunktion(f1 INTEGER, f2 INTEGER)
        RETURNS INTEGER
        AS $$
        BEGIN
           RETURN f1 * f2;
```

```
        END;
        $$ LANGUAGE plpgsql;
```

definiert werden. Diese Funktion koennte man dann in einer SQL-Query mit SELECT meineFunktion(2,3); aufgerufen werden. Die verwendete Sprache ist hier plpgsql, eine prozedurale Erweiterung von SQL, die Variablen, Schleifen, If-Statementents, usw. umfasst. Zum Beispiel liefert die folgende Funktion eine Bilanz (Einzahlungen - Auszahlungen) fuer einen gegebenen Benutzer:

```
CREATE OR REPLACE FUNCTION bilanz(client_id INTEGER)
        RETURNS INTEGER
        AS $$
        DECLARE
           einzahlungen DECIMAL;
           auszahlungen DECIMAL;
        BEGIN
           SELECT SUM(amount) INTO einzahlungen FROM transfers WHERE to_client_id
= client_id;
           SELECT SUM(amount) INTO auszahlungen FROM transfers WHERE
from_client_id = client_id;
           RETURN einzahlungen - auszahlungen;
        END;
        $$ LANGUAGE plpgsql;
```

Beachte, dass Variablen extra deklariert werden muessen und das Ergebnis einer SQL Query hier mit INTO in einer Variablen abgelegt wird. Neben reinen SELECT-Statements sind natuerlich auch INSERT, UPDATE und DELETE-Queries moeglich.

Aufgabe 1

Erweitere die Funktion 'bilanz' wie folgt: Die Bank hat bis 2019 bei jeder Transaktion eine Steuer von 2% eingehoben - d.h. ein Transfer von 100.- soll in der Bilanz des Empfaengers nur mit 98.- gewertet werden. 2020 wurde diese Steuer auf 1% gesenkt. Erstelle eine Funktion bilanz_mit_steuer, die diese beiden Steuersaetze beruecksichtigt. (Hint: mit date_part('year',date) laesst sich das Jahr zu einem Datum ermitteln.) Rufe deine Funktion mit select bilanz() auf und teste anhand von passenden Daten, ob sie auch funktioniert.

```
CREATE OR REPLACE FUNCTION bilanz_mit_steuer(client_id INTEGER)
RETURNS DECIMAL
AS $$
DECLARE
    einzahlungen DECIMAL;
    auszahlungen DECIMAL;
BEGIN
    -- Einzahlungen mit Steuer
    SELECT SUM(CASE
                WHEN date_part('year', date) <= 2019 THEN amount * 0.98
                ELSE amount * 0.99
            END) INTO einzahlungen
    FROM transfers
```

```
    WHERE to_client_id = client_id;

    -- Auszahlungen ohne Steuer
    SELECT SUM(amount) INTO auszahlungen
    FROM transfers
    WHERE from_client_id = client_id;

    -- Berechnung der Bilanz
    RETURN einzahlungen - auszahlungen;
END;
$$ LANGUAGE plpgsql;
```

Procedures

Procedures sind Functions sehr aehnlich, mit einem wichtigen Unterschied: Wahrend Functions immer komplett in einer Transaktion ablaufen, hat man in einer Procedure die Kontrolle ueber die aktuelle Funktion. (D.h. man kann Kommandos wie COMMIT und ROLLBACK ausfuehren.) Sie werden daher haeufiger fuer Datenmanipulationen eingesetzt.

```
CREATE  OR  REPLACE  PROCEDURE transfer_direct(sender INTEGER, recipient INTEGER,
howmuch DECIMAL)
    AS $$
    BEGIN
        UPDATE accounts SET amount = amount - howmuch WHERE client_id = sender;
        UPDATE accounts SET amount = amount + howmuch WHERE client_id = recipient;
        COMMIT;
    END;
    $$ LANGUAGE plpgsql;
```

Eine Procedure wird im Gegensatz zu einer Function ausserhalb von einer Query mit dem Kommando CALL aufgerufen: CALL transfer_direct(1,2,100); ueberweist 100,- von Konto 1 an Konto 2

Aufgabe 2

Passe die Procedure transfer_direct() so an, dass die Ueberweisung nicht durchgefuehrt wird, falls das Konto nicht gedeckt ist, d.h., wenn am Konto des Senders weniger Geld vorhanden ist, als ueberwiesen werden soll. (Hint 1: mit zum Beispiel IF a < b THEN ... END IF; lassen sich Fallunterscheidungen durchfuehren.) (Hint 2: mit RAISE EXCEPTION 'meine Fehlermeldung' laesst sich die Procedure abbrechen.)

```
CREATE OR REPLACE PROCEDURE transfer_direct(sender INTEGER, recipient INTEGER,
howmuch DECIMAL)
AS $$
DECLARE
    sender_balance DECIMAL;
BEGIN
    -- Ermittlung des aktuellen Guthabens des Senders
    SELECT amount INTO sender_balance FROM accounts WHERE client_id = sender;
```

```
    -- Überprüfung, ob das Guthaben für die Überweisung ausreicht
    IF sender_balance < howmuch THEN
        -- Auslösen einer Exception, wenn das Guthaben nicht ausreicht
        RAISE EXCEPTION 'Nicht genügend Guthaben für die Überweisung. Verfügbar:
%, erforderlich: %', sender_balance, howmuch;
    ELSE
        -- Durchführung der Überweisung, wenn das Guthaben ausreicht
        UPDATE accounts SET amount = amount - howmuch WHERE client_id = sender;
        UPDATE accounts SET amount = amount + howmuch WHERE client_id = recipient;
    END IF;
    COMMIT;
END;
$$ LANGUAGE plpgsql;
```

Triggers

Als Trigger bezeichnet man eine spezielle Art von Funktionen, welche vom Datenbankmanagementsystem automatisch aufgerufen werden, wenn bestimmte Ereignisse eintreten. Solche Ereignisse sind zum Beispiel das Einfuegen, Aendern oder Loeschen von Datensaetzen. Trigger koennen dieses Ereigniss verhindern, die zu einzufuegenden Daten modifizieren, oder auch in andere Tabellen schreiben. Zuerst definiert man hierfuer eine Function, die den Returntyp trigger besitzt und richtet diese dann mit dem Kommando CREATER TRIGGER als Trigger ein. Dabei kann man angeben, ob der Trigger vor oder nach einer Operation aufgerufen werden soll. Mit dem Kommando DROP TRIGGER name ON tabelle; kann ein Trigger wieder entfernt werden.

Moechte man zum Beispiel verhindern, dass Konten geloescht werden koennen (zB aus rechltichen Gruenden), koennte man wie folgt vorgehen: Man definiert einen BEFORE DELETE Trigger auf die Accounts Tabelle. In diesem Trigger wird der zu loeschende Datesatz lediglich deaktiviert, anstatt geloescht. Dadurch, dass der Trigger NULL zurueckgibt, wird die Loesch-Aktion nicht durchgefuert:

```
CREATE OR REPLACE FUNCTION deactivate_user()
RETURNS trigger
AS $$
BEGIN
    update accounts set deactivated = TRUE where client_id = OLD.client_id;
    RETURN NULL;
END;
$$ LANGUAGE plpgsql;
```

```
-- Implement Trigger
CREATE TRIGGER no_del_accounts BEFORE DELETE ON accounts FOR EACH ROW EXECUTE
PROCEDURE deactivate_user();
```

Moechte man nun ein Konto mittels delete from accounts where client_id = 1 loeschen, so wird dieses stattdessen nur deaktiviert. Beachte dabei, wie unter OLD.client_id die id des zu loeschenden Datensatzes zugreifbar ist. Ebenso existiert bei INSERT und UPDATE Triggern der Datensatz NEW, welcher den neuen zu speichernden Datensatz beinhaltet.

Aufgabe 3

Erstelle einen Trigger, welcher bewirkt, dass bei dem Anlegen eines neuen Transfers der aktuelle Kontostand in der accounts-Tabelle automatisch angepasst wird. Erstelle dafuer eine Function update_accounts(), die die noetigen Anpassungen durchfuehrt und mache diese Funktion mittels CREATE TRIGGER new_transfer BEFORE INSERT ON transfers FOR EACH ROW EXECUTE PROCEDURE update_accounts(); zum Trigger. (Hint: Damit der Transfer auch gespeichert wird, muss dein Trigger RETURN NEW zurueckgeben);

```
CREATE OR REPLACE FUNCTION update_accounts()
RETURNS TRIGGER AS $$
BEGIN
    -- Erhöhung des Kontostands des Empfängers
    UPDATE accounts SET amount = amount + NEW.amount WHERE client_id =
NEW.to_client_id;

    -- Verringerung des Kontostands des Absenders
    UPDATE accounts SET amount = amount - NEW.amount WHERE client_id =
NEW.from_client_id;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

```
-- Implementing Trigger
CREATE TRIGGER new_transfer BEFORE INSERT ON transfers
FOR EACH ROW EXECUTE FUNCTION update_accounts();
```

Aufgabe 4

Passe den Trigger aus Aufgabe 3 so an, dass nur Transfers angenommen werden, die das Senderkonto nicht ueberziehen. D.h. ueberpruefe, ob das Konto nach der Ueberweisung unter 0.- fallen wuerde und verhindere in diesem Fall mit RETURN NULL das Einfuegen der Transaktion

```
CREATE OR REPLACE FUNCTION update_accounts_safe()
RETURNS TRIGGER AS $$
DECLARE
    sender_balance DECIMAL;
BEGIN
    -- Ermittlung des aktuellen Guthabens des Senders
    SELECT amount INTO sender_balance FROM accounts WHERE client_id =
NEW.from_client_id;

    -- Überprüfung, ob das Guthaben ausreicht
    IF sender_balance < NEW.amount THEN
        -- Wenn das Guthaben nicht ausreicht, wird die Transaktion nicht
durchgeführt
        RAISE EXCEPTION 'Nicht genügend Guthaben für die Überweisung. Verfügbar:
```

```
%, erforderlich: %', sender_balance, NEW.amount;
    ELSE
        -- Ansonsten Durchführung der Kontostandanpassung
        UPDATE accounts SET amount = amount - NEW.amount WHERE client_id =
NEW.from_client_id;
        UPDATE accounts SET amount = amount + NEW.amount WHERE client_id =
NEW.to_client_id;
    END IF;

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

```
-- Implementing Trigger
DROP TRIGGER IF EXISTS new_transfer ON transfers;

CREATE TRIGGER new_transfer BEFORE INSERT ON transfers
FOR EACH ROW EXECUTE FUNCTION update_accounts_safe();
```

Zusammenfassung

Functions:

Funktionen sind Routinen, die bestimmte Aufgaben in der Datenbank ausführen und ein Ergebnis zurückgeben. Sie können in der prozeduralen Sprache PL/pgSQL definiert werden und ermöglichen Operationen wie Berechnungen und Datenmanipulationen innerhalb der Datenbank.

```
-- Example Function (That adds two Numbers)
CREATE OR REPLACE FUNCTION add_numbers(a INTEGER, b INTEGER)
RETURNS INTEGER AS $$
BEGIN
    RETURN a + b;
END;
$$ LANGUAGE plpgsql;
```

```
-- Executing Function
SELECT add_numbers(5, 10) AS result;
```

Procedures:

Prozeduren ähneln Funktionen, führen jedoch Aufgaben aus, ohne zwingend einen Wert zurückzugeben. Sie bieten mehr Kontrolle über Transaktionen, da sie Befehle wie COMMIT und ROLLBACK innerhalb ihres Körpers enthalten können.

```
-- Example Procedure (That outputs a Message)
CREATE OR REPLACE PROCEDURE show_message()
LANGUAGE plpgsql
AS $$
BEGIN
    RAISE NOTICE 'This is an example Function';
END;
$$;
```

```
-- Executing Procedure:
CALL show_message();
```

Trigger:

Trigger sind spezielle Routinen, die automatisch in Reaktion auf bestimmte Datenbankereignisse ausgelöst werden, wie z.B. das Einfügen, Ändern oder Löschen von Datensätzen. Sie können verwendet werden, um Datenintegrität sicherzustellen, automatische Aktionen durchzuführen oder Änderungen in anderen Tabellen zu protokollieren.

```
-- Example Tables:
CREATE TABLE IF NOT EXISTS accounts (
    account_id SERIAL PRIMARY KEY,
    account_name TEXT NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE IF NOT EXISTS logs (
    log_id SERIAL PRIMARY KEY,
    message TEXT NOT NULL,
    logged_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

```
-- Example Trigger (That creates an Log-Entry)
CREATE OR REPLACE FUNCTION log_account_creation()
RETURNS TRIGGER AS $$
BEGIN
    INSERT INTO logs (message) VALUES ('Account created');
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

```
-- Example Implementation:
CREATE TRIGGER account_creation_trigger
```

```
AFTER INSERT ON accounts
FOR EACH ROW
EXECUTE FUNCTION log_account_creation();
```

Quellen:

[1] https://www.postgresqltutorial.com/postgresql-create-function/

[2] https://www.postgresql.org/docs/current/plpgsql.html

[3] https://www.postgresql.org/docs/12/trigger-definition.html