Vyer, Prepared Statements, Triggers

Vyer

- En vy är en virtuell tabell, som ej behöver existera fysiskt, en ”namngiven fråga”
- En vy är inte snabbare än SELECT-satsen som definierar vyn
- Det är möjligt att ställa alla typer av förfrågningar till en vy
- Av logiska skäl är det inte möjligt att stoppa in, ta bort och uppdatera alla typer av rader i en vy
- Materialiserade vyer lagras fysiskt

Vyer

Vyer kan användas för att

- spara vanligt förkommande förfrågningar
- skräddarsy olika vyer till olika användare efter deras behov
- begränsa rättigheter för användare

Vyer

Skapa vy

```
CREATE OR REPLACE VIEW ResearchStaff AS
  SELECT e.*
  FROM Employee e, Department d
  WHERE e.dno = d.dno AND d.name LIKE
'Research';
```

Fråga vyn

```
SELECT * FROM ResearchStaff
  WHERE salary > 40000;
```

”Materialiserade vyer”

```
CREATE TABLE Emp_salary_sum (
    sal_sum INT
);

INSERT INTO Emp_salary_sum(sal_sum
     SELECT sum(salary) FROM Employee;
```

Fråga:

```
SELECT * FROM Emp_salary_sum;
```

Prepared statements

- En fördefinierad fråga
- När ett prepared statement exekveras för första gången i sessionen, så parsas och planläggs den – kan alltså exekvera snabbare än motsvarande statement
- Prepared statement kan ta parametrar som substitueras vid exekvering.
- Prepared statements är serverobjekt (sessionen tar slut så försvinner objektet)
- Tas bort med en DEALLOCATE-sats.

Prepared statements

```
PREPARE PS_Emp_on_dno_sex (INT, CHAR) AS
  SELECT e.*, d.name
  FROM Employee e, Department d
  WHERE e.dno=d.dno AND d.dno = $1 AND sex =
$2;
```

Exekvering:

```
EXECUTE PS_Emp_on_dno_sex (11, 'M');
```

Avallokering:

```
DEALLOCATE PS_Emp_on_dno_sex ;
```

Aktiva databaser: Triggers

- Vid inläggning/uppdatering/borttagning av data - ytterligare uppdrag kan behöva utföras för att garantera att databasen är logiskt sammanhängande (consistent)
- Händelse (som triggar åtgärd) kan vara: INSERT/UPPDATE/DELETE
- Villkor
- Åtgärd

Triggers

Exempel på situationer

- Vid händelsen ta bort en (aktiv) kund ur en tabell Customer, lägg till motsvarande information i en tabell Old_Customer
- Uppdatera data i en materialiserad vy (härlett data som vi av någon anledning inte vill beräkna vid varje fråga); t.ex. total försäljning

Triggers

- Triggern kan definieras att exekveras före/efter operationerna INSERT, UPDATE, eller DELETE
- En gång per modifierad rad eller en gång per SQL-sats

PostgreSQL

- Triggern måste kopplas till en funktion skriven i ett procedurellt språk, t.ex. PLPGSQL

Triggers i PostgreSQL

- Syntax, PostgreSQL:

```
CREATE TRIGGER <namn> { BEFORE | AFTER } {
  event [ OR ... ] }
  ON <tabellnamn>
  [ FOR [ EACH ] { ROW | STATEMENT } ]
  EXECUTE PROCEDURE <funktionsnamn> (
  <argument> )
```

- NEW och OLD refererar till raden som triggade, före resp. efter uppdateringen
- BEFORE-trigger: RETURN NULL => uppdateringen som triggade skippas

Triggers i PostgreSQL

Förutsättning:
Employee (eNo, name, … , dNo)
Department (dNo, name, tot_salary)

- Department.tot_salary ska uppdateras vid INSERT, UPDATE och DELETE på Employee

Triggers i PostgreSQL - INSERT

```
CREATE OR REPLACE FUNCTION
update_tot_salary_on_insert() RETURNS TRIGGER AS $$
BEGIN
      UPDATE Department
      SET tot_salary = tot_salary + NEW.salary
      WHERE Department.dno = NEW.dno;
      RETURN NEW;
END;
$$ LANGUAGE PLPGSQL;

CREATE TRIGGER update_tot_salary_on_insert_trigger
AFTER INSERT ON Employee
FOR EACH ROW EXECUTE PROCEDURE
update_tot_salary_on_insert();
```

Triggers i PostgreSQL - UPDATE

```
CREATE OR REPLACE FUNCTION
update_tot_salary_on_update() RETURNS TRIGGER AS $$
BEGIN
      UPDATE Department
      SET tot_salary = tot_salary - OLD.salary +
                                    NEW.salary
      WHERE Department.dno = NEW.dno;
      RETURN NEW;
END;
$$ LANGUAGE PLPGSQL;

CREATE TRIGGER update_tot_salary_on_update_trigger
AFTER UPDATE OF salary ON Employee
FOR EACH ROW EXECUTE PROCEDURE
update_tot_salary_on_update();
```

Triggers i PostgreSQL – DELETE

```
CREATE OR REPLACE FUNCTION
update_tot_salary_on_delete() RETURNS TRIGGER AS $$
BEGIN
      UPDATE Department
      SET tot_salary = tot_salary - OLD.salary
      WHERE Department.dno = OLD.dno;
      RETURN NULL;
END;
$$ LANGUAGE PLPGSQL;

CREATE TRIGGER update_tot_salary_on_delete_trigger
AFTER DELETE ON Employee
FOR EACH ROW EXECUTE PROCEDURE
update_tot_salary_on_delete();
```

Triggers i PostgreSQL

```
CREATE TABLE emp (
       empname text,
       salary integer,
       last_date timestamp,
       last_user text
);
. . .

CREATE FUNCTION emp_stamp() RETURNS trigger AS $emp_stamp$
BEGIN
  . . .
END;
$emp_stamp$ LANGUAGE PLPQSQL;

CREATE TRIGGER emp_stamp BEFORE INSERT OR UPDATE ON emp
       FOR EACH ROW EXECUTE PROCEDURE emp_stamp();
```

Triggers i PostgreSQL

```
CREATE FUNCTION emp_stamp() RETURNS trigger AS $emp_stamp$
BEGIN
  -- Check that empname and salary are given
  IF NEW.empname IS NULL
    THEN RAISE EXCEPTION 'empname cannot be null';
  END IF;
  IF NEW.salary IS NULL
    THEN RAISE EXCEPTION '% cannot have null salary', NEW.empname;
  END IF;
  -- Who works for us when she must pay for it?
  IF NEW.salary < 0
    THEN RAISE EXCEPTION '% cannot have a negative salary',
                                                    NEW.empname;
  END IF;
  -- Remember who changed the payroll when
  NEW.last_date := current_timestamp;
  NEW.last_user := current_user;
  RETURN NEW;
END;
$emp_stamp$ LANGUAGE PLPQSQL;
```