```
CREATE TABLE empregado (
   nome_emp      text,
   salario       integer,
   ultima_data   timestamp,
   ultimo_usuario text
);

CREATE FUNCTION emp_gatilho() RETURNS trigger AS $emp_gatilho$
   BEGIN
      -- Verificar se foi fornecido o nome e o salário do empregado
      IF NEW.nome_emp IS NULL THEN
         RAISE EXCEPTION 'O nome do empregado não pode ser nulo';
      END IF;
      IF NEW.salario IS NULL THEN
         RAISE EXCEPTION '% não pode ter um salário nulo', NEW.nome_emp;
      END IF;

      -- Definir um salário válido?
      IF NEW.salario < 0 THEN
         RAISE EXCEPTION '% não pode ter um salário negativo', NEW.nome_emp;
      END IF;

      -- Registrar quem alterou a folha de pagamento e quando
      NEW.ultima_data := 'now';
      NEW.ultimo_usuario := current_user;
      RETURN NEW;
   END;
$emp_gatilho$ LANGUAGE plpgsql;

CREATE TRIGGER emp_gatilho BEFORE INSERT OR UPDATE ON empregado
   FOR EACH ROW EXECUTE PROCEDURE emp_gatilho();

INSERT INTO empregado (nome_emp, salario) VALUES ('Ilson',2000);
INSERT INTO empregado (nome_emp, salario) VALUES ('Douglas',2500);
INSERT INTO empregado (nome_emp, salario) VALUES ('Gaby',3500);

SELECT * FROM empregado;

------
```

Exemplo 2. Procedimento de gatilho PL/pgSQL para registrar inserção e atualização

O gatilho deste exemplo garante que quando é inserida ou atualizada uma linha na tabela, fica sempre registrado nesta linha o usuário que efetuou a inserção ou a atualização, e quando isto ocorreu. Porém, diferentemente do gatilho anterior, a criação e a atualização da linha são registradas em colunas diferentes. Além disso, o gatilho verifica se é fornecido o nome do empregado, e se o valor do salário é um número positivo.

```
CREATE TABLE empregado (
    nome_emp      text,
    salario       integer,
    usu_cria      text,       -- Usuário que criou a linha
    data_cria     timestamp,  -- Data da criação da linha
    usu_atu       text,       -- Usuário que fez a atualização
    data_atu      timestamp   -- Data da atualização
);

CREATE FUNCTION emp_gatilho() RETURNS trigger AS $emp_gatilho$
    BEGIN
        -- Verificar se foi fornecido o nome do empregado
        IF NEW.nome_emp IS NULL THEN
            RAISE EXCEPTION 'O nome do empregado não pode ser nulo';
        END IF;
        IF NEW.salario IS NULL THEN
            RAISE EXCEPTION '% não pode ter um salário nulo', NEW.nome_emp;
        END IF;

        -- Quem paga para trabalhar?
        IF NEW.salario < 0 THEN
            RAISE EXCEPTION '% não pode ter um salário negativo', NEW.nome_emp;
        END IF;

        -- Registrar quem criou a linha e quando
        IF (TG_OP = 'INSERT') THEN
            NEW.data_cria := current_timestamp;
            NEW.usu_cria  := current_user;
        -- Registrar quem alterou a linha e quando
        ELSIF (TG_OP = 'UPDATE') THEN
            NEW.data_atu := current_timestamp;
            NEW.usu_atu  := current_user;
        END IF;
        RETURN NEW;
    END;
$emp_gatilho$ LANGUAGE plpgsql;

CREATE TRIGGER emp_gatilho BEFORE INSERT OR UPDATE ON empregado
```

```
   FOR EACH ROW EXECUTE PROCEDURE emp_gatilho();

INSERT INTO empregado (nome_emp, salario) VALUES ('Ilson',2000);
INSERT INTO empregado (nome_emp, salario) VALUES ('Douglas',2500);
INSERT INTO empregado (nome_emp, salario) VALUES ('Gaby',3500);
UPDATE empregado SET salario = 4500 WHERE nome_emp = 'Gaby';

SELECT * FROM empregado;


------
```

Exemplo 35-3. Procedimento de gatilho PL/pgSQL para auditoria

Este gatilho garante que todas as inserções, atualizações e exclusões de linha na tabela emp são registradas na tabela emp_audit, para permitir auditar as operações efetuadas na tabela emp. O nome de usuário e a hora corrente são gravadas na linha, junto com o tipo de operação que foi realizada.

```
CREATE TABLE empregado (
   nome_emp    text NOT NULL,
   salario     integer
);

CREATE TABLE emp_audit(
   operacao    char(1)   NOT NULL,
   usuario     text      NOT NULL,
   data        timestamp NOT NULL,
   nome_emp    text      NOT NULL,
   salario     integer
);

CREATE OR REPLACE FUNCTION processa_emp_audit() RETURNS TRIGGER AS
$emp_audit$
   BEGIN
      --
      -- Cria uma linha na tabela emp_audit para refletir a operação
      -- realizada na tabela emp. Utiliza a variável especial TG_OP
      -- para descobrir a operação sendo realizada.
      --
      IF (TG_OP = 'DELETE') THEN
         INSERT INTO emp_audit SELECT 'E', user, now(), OLD.*;
         RETURN OLD;
      ELSIF (TG_OP = 'UPDATE') THEN
         INSERT INTO emp_audit SELECT 'U', user, now(), NEW.*;
         RETURN NEW;
      ELSIF (TG_OP = 'INSERT') THEN
         INSERT INTO emp_audit SELECT 'I', user, now(), NEW.*;
         RETURN NEW;
      END IF;
      RETURN NULL; -- o resultado é ignorado uma vez que este é um gatilho AFTER
   END;
$emp_audit$ language plpgsql;

CREATE TRIGGER emp_audit
AFTER INSERT OR UPDATE OR DELETE ON empregado
   FOR EACH ROW EXECUTE PROCEDURE processa_emp_audit();

INSERT INTO empregado (nome_emp, salario) VALUES ('Khalil',2001);
```

```
INSERT INTO empregado (nome_emp, salario) VALUES ('Jhonathan',2500);
INSERT INTO empregado (nome_emp, salario) VALUES ('Rodrigo',250);
UPDATE empregado SET salario = 2500 WHERE nome_emp = 'Rodrigo';
DELETE FROM empregado WHERE nome_emp = 'Khalil';

SELECT * FROM empregado;

SELECT * FROM emp_audit;
```

Exemplo 35-4. Procedimento de gatilho PL/pgSQL para auditoria no nível de coluna

Este gatilho registra todas as atualizações realizadas nas colunas nome_emp e salario da tabela empregado na tabela emp_audit (isto é, as colunas são auditadas). O nome de usuário e a hora corrente são registrados junto com a chave da linha (id) e a informação atualizada. Não é permitido atualizar a chave da linha. Este exemplo difere do anterior pela auditoria ser no nível de coluna, e não no nível de linha.

```
CREATE TABLE empregado (
   id         serial  PRIMARY KEY,
   nome_emp    text    NOT NULL,
   salario     integer
);

CREATE TABLE emp_audit(
   usuario        text      NOT NULL,
   data           timestamp NOT NULL,
   id             integer   NOT NULL,
   coluna         text      NOT NULL,
   valor_antigo    text      NOT NULL,
   valor_novo      text      NOT NULL
);

CREATE OR REPLACE FUNCTION processa_emp_audit() RETURNS TRIGGER AS
$emp_audit$
   BEGIN
      --
      -- Não permitir atualizar a chave primária
      --
      IF (NEW.id <> OLD.id) THEN
         RAISE EXCEPTION 'Não é permitido atualizar o campo ID';
      END IF;
      --
      -- Inserir linhas na tabela emp_audit para refletir as alterações
      -- realizada na tabela emp.
      --
      IF (NEW.nome_emp <> OLD.nome_emp) THEN
        INSERT INTO emp_audit SELECT current_user, current_timestamp,
                NEW.id, 'nome_emp', OLD.nome_emp, NEW.nome_emp;
      END IF;
      IF (NEW.salario <> OLD.salario) THEN
        INSERT INTO emp_audit SELECT current_user, current_timestamp,
                NEW.id, 'salario', OLD.salario, NEW.salario;
      END IF;
      RETURN NULL; -- o resultado é ignorado uma vez que este é um gatilho AFTER
   END;
$emp_audit$ language plpgsql;
```

```
CREATE TRIGGER emp_audit
AFTER UPDATE ON empregado
   FOR EACH ROW EXECUTE PROCEDURE processa_emp_audit();

INSERT INTO empregado (nome_emp, salario) VALUES ('Eros',1000);
INSERT INTO empregado (nome_emp, salario) VALUES ('Marcelo',1500);
INSERT INTO empregado (nome_emp, salario) VALUES ('Maria',2500);
UPDATE empregado SET salario = 2500 WHERE id = 2;
UPDATE empregado SET nome_emp = 'Maria Cecília' WHERE id = 3;
UPDATE empregado SET id=100 WHERE id=1;


SELECT * FROM empregado;


SELECT * FROM emp_audit;


------
```

Uma das utilizações de gatilho é para manter uma tabela contendo o sumário de outra tabela. O sumário produzido pode ser utilizado no lugar da tabela original em diversas consultas — geralmente com um tempo de execução bem menor. Esta técnica é muito utilizada em Armazém de Dados (Data Warehousing), onde as tabelas dos dados medidos ou observados (chamadas de tabelas fato) podem ser muito grandes. O Exemplo 5 mostra um procedimento de gatilho em PL/pgSQL para manter uma tabela de sumário de uma tabela fato em um armazém de dados.

Exemplo 5. Procedimento de gatilho PL/pgSQL para manter uma tabela sumário

O esquema que está detalhado a seguir é parcialmente baseado no exemplo Grocery Store do livro The Data Warehouse Toolkit de Ralph Kimball.

```
--
-- Main tables - time dimension and sales fact.
--
CREATE TABLE time_dimension (
   time_key               integer NOT NULL,
   day_of_week            integer NOT NULL,
   day_of_month           integer NOT NULL,
   month                  integer NOT NULL,
   quarter                integer NOT NULL,
   year                   integer NOT NULL
);
CREATE UNIQUE INDEX time_dimension_key ON time_dimension(time_key);

CREATE TABLE sales_fact (
   time_key               integer NOT NULL,
   product_key            integer NOT NULL,
   store_key              integer NOT NULL,
   amount_sold            numeric(12,2) NOT NULL,
   units_sold             integer NOT NULL,
   amount_cost            numeric(12,2) NOT NULL
);
CREATE INDEX sales_fact_time ON sales_fact(time_key);

--
-- Summary table - sales by time.
--
CREATE TABLE sales_summary_bytime (
   time_key               integer NOT NULL,
   amount_sold            numeric(15,2) NOT NULL,
   units_sold             numeric(12) NOT NULL,
   amount_cost            numeric(15,2) NOT NULL
);
CREATE UNIQUE INDEX sales_summary_bytime_key ON
sales_summary_bytime(time_key);
```

```
--
-- Function and trigger to amend summarized column(s) on UPDATE, INSERT, DELETE.
--
CREATE OR REPLACE FUNCTION maint_sales_summary_bytime() RETURNS TRIGGER AS
$maint_sales_summary_bytime$
  DECLARE
    delta_time_key        integer;
    delta_amount_sold     numeric(15,2);
    delta_units_sold      numeric(12);
    delta_amount_cost     numeric(15,2);
  BEGIN

    -- Work out the increment/decrement amount(s).
    IF (TG_OP = 'DELETE') THEN

      delta_time_key = OLD.time_key;
      delta_amount_sold = -1 * OLD.amount_sold;
      delta_units_sold = -1 * OLD.units_sold;
      delta_amount_cost = -1 * OLD.amount_cost;

    ELSIF (TG_OP = 'UPDATE') THEN

      -- forbid updates that change the time_key -
      -- (probably not too onerous, as DELETE + INSERT is how most
      -- changes will be made).
      IF ( OLD.time_key != NEW.time_key) THEN
        RAISE EXCEPTION 'Update of time_key : % -> % not allowed', OLD.time_key,
NEW.time_key;
      END IF;

      delta_time_key = OLD.time_key;
      delta_amount_sold = NEW.amount_sold - OLD.amount_sold;
      delta_units_sold = NEW.units_sold - OLD.units_sold;
      delta_amount_cost = NEW.amount_cost - OLD.amount_cost;

    ELSIF (TG_OP = 'INSERT') THEN

      delta_time_key = NEW.time_key;
      delta_amount_sold = NEW.amount_sold;
      delta_units_sold = NEW.units_sold;
      delta_amount_cost = NEW.amount_cost;

    END IF;


    -- Update the summary row with the new values.
```

```
    UPDATE sales_summary_bytime
       SET amount_sold = amount_sold + delta_amount_sold,
          units_sold = units_sold + delta_units_sold,
          amount_cost = amount_cost + delta_amount_cost
       WHERE time_key = delta_time_key;


    -- There might have been no row with this time_key (e.g new data!).
    IF (NOT FOUND) THEN
       BEGIN
          INSERT INTO sales_summary_bytime (
                time_key,
                amount_sold,
                units_sold,
                amount_cost)
             VALUES (
                delta_time_key,
                delta_amount_sold,
                delta_units_sold,
                delta_amount_cost
               );
       EXCEPTION
          --
          -- Catch race condition when two transactions are adding data
          -- for a new time_key.
          --
          WHEN UNIQUE_VIOLATION THEN
             UPDATE sales_summary_bytime
                SET amount_sold = amount_sold + delta_amount_sold,
                   units_sold = units_sold + delta_units_sold,
                   amount_cost = amount_cost + delta_amount_cost
                WHERE time_key = delta_time_key;

       END;
    END IF;
    RETURN NULL;

  END;
$maint_sales_summary_bytime$ LANGUAGE plpgsql;

CREATE TRIGGER maint_sales_summary_bytime
AFTER INSERT OR UPDATE OR DELETE ON sales_fact
  FOR EACH ROW EXECUTE PROCEDURE maint_sales_summary_bytime();

-------
```

REFERÊNCIA
http://pgdocptbr.sourceforge.net/pg80/plpgsql-trigger.html