Capitolo 1

Che cos'è un server PostgreSQL

Se pensate che un server PostgreSQL sia soltanto un sistema di archiviazione e l'unico metodo di comunicazione è l'esecuzione di query SQL, vi state sbagliando. Questa è solo una piccola parte delle caratteristiche del database.

Un server PostgreSQL è un framework potente che può essere utilizzato per ogni tipo di elaborazione dei dati e altre attività di tipo server. Si tratta di una piattaforma che permette di mescolare in modo semplice funzioni e librerie di molti linguaggi popolari. Considerate una sequenza di operazioni multilinguaggio complicata come la seguente.

1. Chiamare una funzione per il parsing di una stringa in Perl.
2. Convertire la stringa in XSLT ed elaborarla utilizzando JavaScript.
3. Richiedere un marchio di sicurezza da un servizio esterno di time-stamping come `www.guardtime.com`, utilizzando il relativo SDK per C.
4. Scrivere una funzione Python per firmare in modo digitale il risultato.

Tutto questo può essere realizzato con una serie di semplici chiamate a funzione, utilizzando molti dei linguaggi di programmazione server a disposizione. Lo sviluppatore che deve portare a termine queste attività può farlo con una singola funzione PostgreSQL senza preoccuparsi di come passare i dati tra i linguaggi e le librerie:

```
SELECT convert_to_xslt_and_sign(raw_data_string);
```

In questo capitolo

- **Perché programmare sul server?**
- **Il codice di esempio del libro**
- **Migliorare le funzioni semplici**
- **Gestire i dati collegati con i trigger**
- **Auditing delle modifiche**
- **Pulizia dei dati**
- **Ordinamenti personalizzati**
- **Best practice della programmazione**
- **Cache**
- **Ricapitoliamo: perché programmare nel server?**
- **Riepilogo**

In questo libro si parlerà dei diversi aspetti della programmazione PostgreSQL. Questo possiede tutte le caratteristiche disponibili nei maggiori sistemi di database, come i trigger, azioni automatizzate che sono attivate ogni volta che i dati cambiano. Quello che spicca è l'abilità di eludere il comportamento di default con operatori di base. Ecco alcuni esempi di personalizzazione.

- Scrivere *funzioni definite dall'utente* (UDF, *user-defined functions*) in C per calcoli complessi.
- Aggiungere vincoli per assicurarsi che i dati nel server restino entro i limiti stabiliti.
- Creare trigger in molti linguaggi per applicare modifiche collegate alle altre tabelle, registrare i log delle attività o impedire che l'azione sia eseguita se non sono rispettati alcuni criteri.
- Definire nuovi tipi di dati e operatori nel database.
- Utilizzare i tipi geografici definiti nel pacchetto PostGIS.
- Aggiungere metodi di accesso personalizzati sia per i tipi esistenti, sia per i nuovi, realizzando query più efficienti.

Che cosa potete realizzare con tutte questo? Non ci sono limiti alle possibilità:

- potete scrivere funzioni per estrarre i dati e ottenere solo la parte di interesse da una struttura, per esempio XML o JSON, senza dover trasferire un documento intero, magari anche pesante, all'applicazione client;
- potete processare eventi asincroni, come l'invio di una e-mail senza rallentare l'applicazione principale. Potete creare una coda per le modifiche alle informazioni dell'utente, popolata da un trigger. Un processo separato si preoccuperà di utilizzare i dati non appena riceve la notifica dall'applicazione.

Il resto del capitolo descrive una serie di attività comuni per la gestione dei dati, mostrando come possono essere completate in modo elegante ed efficiente attraverso la programmazione server.
Gli esempi in questo capitolo sono stati testati e funzionanti, ma riportano pochi commenti. Sono stati inseriti solo per mostrarvi cosa si può realizzare con la programmazione. Le tecniche saranno spiegate nel dettaglio nei prossimi capitoli.

Perché programmare sul server?

Gli sviluppatori programmano il codice utilizzando linguaggi diversi che possono essere eseguiti quasi ovunque. Quando si scrive un'applicazione, alcune persone seguono la filosofia secondo cui la maggior parte della logica dovrebbe essere gestita dal client; potete notarlo nella grande disponibilità di applicazioni JavaScript all'interno dei browser. Altri preferiscono mantenerla a metà strada, con un server applicativo che gestisce le regole operative. Sono tutti sistemi validi per realizzare un'applicazione, perciò perché dovreste programmare nel server database?
Partiamo da un esempio semplice. Molte applicazioni prevedono una lista di clienti che hanno un bilancio del proprio conto. Utilizziamo questo schema di esempio e questi dati:

```
CREATE TABLE accounts(owner text, balance numeric);
INSERT INTO accounts VALUES ('Bob',100);
INSERT INTO accounts VALUES ('Mary',200);
```

Scaricare il codice di esempio

Potete scaricare i file del codice di esempio del libro dal sito di Apogeo all'indirizzo `http://www.apogeonline.com/libri/9788850331482/scheda`.

Quando si utilizza un database, il modo più comune per interagire è utilizzare query SQL. Se volete spostare 14 euro dal conto di Bob a quello di Mary, potete farlo con questo semplice SQL:

```
UPDATE accounts SET balance = balance - 14.00 WHERE owner = 'Bob';
UPDATE accounts SET balance = balance + 14.00 WHERE owner = 'Mary';
```

Dovete però controllare che Bob abbia abbastanza soldi (o credito) sul proprio conto. È importante che se qualcosa andasse storto non avvenga alcuna transazione. In un'applicazione, il codice precedente diventa così:

```
BEGIN;
SELECT balance FROM accounts WHERE owner = 'Bob' FOR UPDATE;
-- controllate nell'applicazione che il saldo attuale sia superiore a 14
UPDATE accounts SET balance = balance - 14.00 WHERE owner = 'Bob';
UPDATE accounts SET balance = balance + 14.00 WHERE owner = 'Mary';
COMMIT;
```

Mary ha davvero un conto? Se così non fosse, l'ultima istruzione `UPDATE` non aggiornerà alcuna riga. Se uno qualsiasi dei controlli dovesse fallire, eseguite un `ROLLBACK` invece di `COMMIT`. Dopo che avete fatto tutto questo in tutti i client che trasferiscono denaro, arriverà di sicuro una nuova richiesta. Per esempio, l'ammontare del trasferimento minimo è `5.00`. Dovrete rivedere di nuovo il vostro codice in tutti i client.
Che cosa potete fare per rendere queste operazioni più gestibili, sicure ed efficienti? La programmazione e l'esecuzione del codice nel server database possono esservi d'aiuto. Potete spostare i calcoli, i controlli e la manipolazione dei dati in una funzione UDF sul server. Questo non vi assicura che dovrete gestire solo una copia della logica operativa, ma rende tutto più veloce perché evita un continuo passaggio dal client al server. Se richiesto, potete anche fare in modo che solo le informazioni necessarie escano dal database. Per esempio, non c'è motivo che le applicazioni client conoscano il saldo del conto di Bob. Di solito, hanno bisogno di sapere solo se c'è abbastanza denaro per completare la transazione o se questa è avvenuta correttamente.

Utilizzare PL/pgSQL per controlli di integrità

PostgreSQL include il proprio linguaggio di programmazione, chiamato PL/pgSQL che si propone di integrare in modo semplice i comandi SQL. PL sta per *programming*

language, ed è uno dei molti linguaggi a disposizione per scrivere codice server. pgSQL è un'abbreviazione di PostgreSQL.
In modo diverso da SQL, PL/pgSQL comprende elementi procedurali, come la possibilità di utilizzare costrutti `if/then/else` e iterazioni. Potete lanciare in modo semplice istruzioni SQL, oppure scorrere i risultati di una query.
I controlli di integrità necessari all'applicazione possono essere realizzati con una funzione PL/pgSQL che accetta tre argomenti: nome di chi paga, di chi riceve e ammontare del pagamento. Questo esempio restituisce anche lo stato del pagamento:

```
CREATE OR REPLACE FUNCTION transfer(
             i_payer text,
             i_recipient text,
             i_amount numeric(15,2))
RETURNS text
AS
$$
DECLARE
  payer_bal numeric; BEGIN
  SELECT balance INTO payer_bal
    FROM accounts
  WHERE owner = i_payer FOR UPDATE; IF NOT FOUND THEN
    RETURN 'Payer account not found';
  END IF;
  IF payer_bal < i_amount THEN
    RETURN 'Not enough funds';
  END IF;

  UPDATE accounts
       SET balance = balance + i_amount
    WHERE owner = i_recipient;
  IF NOT FOUND THEN
    RETURN 'Recipient does not exist'; END IF;

  UPDATE accounts
       SET balance = balance - i_amount
    WHERE owner = i_payer; RETURN 'OK';
END;
$$ LANGUAGE plpgsql;
```

Ecco un paio di esempi di utilizzo di questa funzione (assumendo che non avete eseguito l'istruzione `UPDATE` precedente):

```
postgres=# SELECT * FROM accounts;
owner  | balance
-------+---------
   Bob | 100
  Mary | 200
(2 rows)
```

```
postgres=# SELECT * FROM transfer('Bob','Mary',14.00);
transfer
---------- OK
(1 row)

 postgres=# SELECT * FROM accounts;
 owner  | balance
 -------+---------
   Mary | 214.00
    Bob |  86.00
 (2 rows)
```

La vostra applicazione dovrebbe controllare il codice restituito e decidere come gestire gli errori. Potete estendere la funzione per fare qualche controllo in più invece che rifiutare ogni valore non previsto, come l'ammontare minimo da trasferire, e assicurarsi che sia rispettato. Gli errori che possono essere restituiti sono tre:

```
postgres=# SELECT * FROM transfer('Fred','Mary',14.00);
        transfer
-------------------------
 Payer account not found
(1 row)

postgres=# SELECT * FROM transfer('Bob','Fred',14.00);
     transfer
-------------------------
 Recipient does not exist
(1 row)

postgres=# SELECT * FROM transfer('Bob','Mary',500.00);
     transfer
-----------------
 Not enough funds
(1 row)
```

Affinché i controlli funzionino sempre, dovrete gestire tutti i trasferimenti attraverso questa funzione invece che farlo manualmente con istruzioni SQL.

Il codice di esempio del libro

L'output di esempio appena visto è stato creato con lo strumento PostgreSQL `psql`, di solito utilizzato su un sistema Linux. La maggior parte del codice funzionerà allo stesso modo se utilizzate una GUI `pgAdmin3` per accedere al server. In una riga come questa

```
postgres=# SELECT 1;
```

la parte `postgres=#` è il prompt mostrato dal comando `psql`.

Gli esempi di questo libro sono stati testati con PostgreSQL 9.2. È probabile che funzionino sulla versione 8.3 e successive, dato che non ci sono state modifiche importanti nelle ultime release di PostgreSQL. La sintassi è diventata più rigida nel corso del tempo, per ridurre i possibili errori nella programmazione server e, considerando la natura di questi cambiamenti, il codice delle versioni più recenti funzionerà su quelle più vecchie, sempre che non si faccia uso delle caratteristiche più nuove. Al contrario, il codice più vecchio è probabile che abbia problemi a causa delle ultime restrizioni introdotte.

Passare alla visualizzazione espansa

Quando si utilizza lo strumento `psql` per eseguire una query, PostgreSQL di solito mostra il risultato con colonne allineate verticalmente:

```
$ psql -c "SELECT 1 AS test"
 test
------
    1
(1 row)

$ psql
psql (9.2.1)
Type "help" for help.
postgres=# SELECT 1 AS test;
 test
------
    1
(1 row)
```

Potete capire che l'output è corretto perché termina con il numero di righe. Questo tipo di visualizzazione non si adatta al meglio in un libro come questo. È più semplice utilizzare quella che il programma chiama *visualizzazione espansa*, che porta ogni colonna su una riga separata. Potete attivare questa modalità sia con l'opzione `-x`, sia passando il modificatore `\x` al programma `psql`. Ecco un esempio dell'utilizzo di entrambi:

```
$ psql -x -c "SELECT 1 AS test"
-[ RECORD 1 ]
test | 1

$ psql
psql (9.2.1)
Type "help" for help.

postgres=# \x
Expanded display is on. postgres=# SELECT 1 AS test;
-[ RECORD 1 ]
test | 1
```

Notate che la visualizzazione espansa non mostra il totale di righe in fondo e che numera ciascuna riga di output. Per risparmiare spazio, non tutti gli esempi utilizzeranno la visualizzazione espansa. Potete capire quale tipo state visualizzando da differenze come questa, a seconda che visualizziate delle righe o dei `RECORD`. Quando il risultato della query è troppo lungo rispetto alla larghezza della pagina del libro sarà preferita la modalità espansa.

Migliorare le funzioni semplici

Programmazione server può voler dire cose differenti. Non si tratta solo di scrivere funzioni server. Ci sono molte altre cose da fare su un server che possono essere considerate programmazione.

Confrontare i dati con gli operatori

Per le operazioni più complesse, potete definire tipi e operatori personalizzati e fare il cast da un tipo all'altro, con la possibilità, per esempio, di confrontare mele e arance. Come vedrete nel prossimo esempio, potete definire il tipo `fruit_qty` che si riferisce a frutta con quantità e insegnare a PostgreSQL il confronto, dichiarando che un'arancia vale 1,5 mele e fare quindi la conversione:

```
postgres=# CREATE TYPE FRUIT_QTY as (name text, qty int);

postgres=# SELECT '("APPLE", 3)'::FRUIT_QTY;
 fruit_quantity
----------------
 (APPLE,3)
(1 row)

CREATE FUNCTION fruit_qty_larger_than(left_fruit FRUIT_QTY,
                                      right_fruit FRUIT_QTY)
RETURNS BOOL AS $$
BEGIN
    IF (left_fruit.name = 'APPLE' AND right_fruit.name = 'ORANGE')
    THEN
        RETURN left_fruit.qty > (1.5 * right_fruit.qty);
    END IF;
    IF (left_fruit.name = 'ORANGE' AND right_fruit.name = 'APPLE' )
    THEN
        RETURN (1.5 * left_fruit.qty) > right_fruit.qty;
    END IF;
    RETURN        left_fruit.qty > right_fruit.qty;
END;
$$
LANGUAGE plpgsql;
```

```
postgres=# SELECT fruit_qty_larger_than('("APPLE", 3)'::FRUIT_
QTY,'("ORANGE", 2)'::FRUIT_QTY);
 fruit_qty_larger_than
-----------------------
 f
(1 row)

postgres=# SELECT fruit_qty_larger_than('("APPLE", 4)'::FRUIT_
QTY,'("ORANGE", 2)'::FRUIT_QTY);
 fruit_qty_larger_than
----------------------- t
(1 row)

CREATE OPERATOR > (
    leftarg = FRUIT_QTY,
    rightarg = FRUIT_QTY,
    procedure = fruit_qty_larger_than,
commutator = >
);

 postgres=# SELECT '("ORANGE", 2)'::FRUIT_QTY > '("APPLE", 2)'::FRUIT_
QTY;
 ?column?
----------
 t
(1 row)

postgres=# SELECT '("ORANGE", 2)'::FRUIT_QTY > '("APPLE", 3)'::FRUIT_
QTY;
 ?column?
----------
 f
(1 row)
```

Gestire i dati collegati con i trigger

È possibile anche impostare attività automatiche (*trigger*) in modo che alcune operazioni sul database facciano capitare altre cose. Per esempio, potete impostare un processo in modo che sia possibile fare un'offerta soltanto per gli oggetti disponibili.
Partiamo con la creazione di una tabella per la disponibilità della frutta:

```
CREATE TABLE fruits_in_stock (
    name text PRIMARY KEY,
    in_stock integer NOT NULL,
    reserved integer NOT NULL DEFAULT 0,
```

```
    CHECK (in_stock between 0 and 1000 ),
    CHECK (reserved <= in_stock)
);
```

Il vincolo `CHECK` controlla che siano rispettate alcune regole base: non potete avere più di `1000` frutti disponibili (è possibile che vadano a male), la disponibilità non può essere negativa e non potete vendere più di quanto possedete:

```
CREATE TABLE fruit_offer (
    offer_id serial PRIMARY KEY, recipient_name text,
    offer_date timestamp default current_timestamp,
    fruit_name text REFERENCES fruits_in_stock,
    offered_amount integer
);
```

La tabella `offer` ha un ID per l'offerta (in modo da distinguerle in seguito), il nome del compratore, la data, il nome del frutto e la quantità.
Per automatizzare la gestione delle prenotazioni, avete bisogno innanzitutto di una funzione `TRIGGER`, che implementa la logica:

```
CREATE OR REPLACE FUNCTION reserve_stock_on_offer () RETURNS trigger
AS $$
    BEGIN
        IF TG_OP = 'INSERT' THEN
            UPDATE fruits_in_stock
        SET reserved = reserved + NEW.offered_amount
      WHERE name = NEW.fruit_name;
    ELSIF TG_OP = 'UPDATE' THEN
        UPDATE fruits_in_stock
          SET reserved = reserved - OLD.offered_amount
                                  + NEW.offered_amount
      WHERE name = NEW.fruit_name;
    ELSIF TG_OP = 'DELETE' THEN
        UPDATE fruits_in_stock
          SET reserved = reserved - OLD.offered_amount
      WHERE name = OLD.fruit_name;
        END IF;
        RETURN NEW;
    END;

$$ LANGUAGE plpgsql;
```

Dovete indicare a PostgreSQL di chiamare questa funzione ogni qualvolta un'offerta cambia:

```
CREATE TRIGGER manage_reserve_stock_on_offer_change
AFTER INSERT OR UPDATE OR DELETE ON fruit_offer
    FOR EACH ROW EXECUTE PROCEDURE reserve_stock_on_offer();
```

Adesso siete pronti per fare un test. Per prima cosa, aggiungete qualche frutto al magazzino:

```
INSERT INTO fruits_in_stock(name,in_stock)
```

Dopodiché, controllate la disponibilità (l'output è mostrato in visualizzazione espansa):

```
postgres=# \x
Expanded display is on.
postgres=# SELECT * FROM fruits_in_stock;
-[ RECORD 1 ]----
name     | APPLE
in_stock | 500
reserved | 0
-[ RECORD 2 ]----
name     | ORANGE
in_stock | 500
reserved | 0
```

Fate un'offerta a nome di Bob per 100 mele:

```
postgres=# INSERT INTO fruit_offer(recipient_name,fruit_name,offered_
amount) VALUES('Bob','APPLE',100);
INSERT 0 1
postgres=# SELECT * FROM fruit_offer;
-[ RECORD 1 ]--+--------------------------
offer_id       | 1
recipient_name | Bob
offer_date     | 2013-01-25 15:21:15.281579
fruit_name     | APPLE
offered_amount | 100

postgres=# SELECT * FROM fruits_in_stock;
-[ RECORD 1 ]----
name     | ORANGE
in_stock | 500
reserved | 0
-[ RECORD 2 ]----
name     | APPLE
in_stock | 500
reserved | 100
```

Se ricontrollate la disponibilità, vedrete che in effetti sono state prenotate 100 mele:

```
postgres=# SELECT * FROM fruits_in_stock;
-[ RECORD 1 ]----
name     | ORANGE
in_stock | 500
reserved | 0
-[ RECORD 2 ]----
name     | APPLE
```

```
in_stock | 500
reserved | 100
```

Se modificate l'offerta, cambierà anche la prenotazione:

```
postgres=# UPDATE fruit_offer SET offered_amount = 115 WHERE offer_id
= 1;
UPDATE 1
postgres=# SELECT * FROM fruits_in_stock;
-[ RECORD 1 ]----
name     | ORANGE
in_stock | 500
reserved | 0
-[ RECORD 2 ]----
name     | APPLE
in_stock | 500
reserved | 115
```

Otterrete anche altri benefici. Per prima cosa, grazie al vincolo sulla tabella `stock`, non potrete vendere le mele già prenotate:

```
postgres=# UPDATE fruits_in_stock SET in_stock = 100 WHERE name =
'APPLE';
ERROR:   new row for relation "fruits_in_stock" violatesi check
constraint "fruits_in_stock_check"
DETAIL:  Failing row contains (APPLE, 100, 115).
```

Ancora più interessante, non potete prenotare più mele di quante disponibili, anche se i vincoli sono su un'altra tabella:

```
postgres=# UPDATE fruit_offer SET offered_amount = 1100 WHERE offer_id
= 1;
ERROR:  new row for relation "fruits_in_stock" violates check
constraint "fruits_in_stock_check"
DETAIL:  Failing row contains (APPLE, 500, 1100).
CONTEXT: SQL statement "UPDATE fruits_in_stock
       SET reserved = reserved - OLD.offered_amount
                                 + NEW.offered_amount
    WHERE name = NEW.fruit_name"
PL/pgSQL function reserve_stock_on_offer() line 8 at SQL statement
```

Se cancellate l'offerta, la prenotazione viene anch'essa cancellata:

```
postgres=# DELETE FROM fruit_offer WHERE offer_id = 1;
DELETE 1
postgres=# SELECT * FROM fruits_in_stock;
-[ RECORD 1 ]----
name     | ORANGE
in_stock | 500
reserved | 0
```

```
-[ RECORD 2 ]----
name     | APPLE
in_stock | 500
reserved | 0
```

In una situazione reale, è probabile che vogliate archiviare l'offerta prima di cancellarla.

Auditing delle modifiche

Se volete sapere chi ha fatto cosa ai dati e quando è stato fatto qualcosa, un metodo è quello di registrare ogni azione eseguita in una tabella. Ci sono almeno due sistemi, entrambi validi, di realizzare l'auditing:

- utilizzare i trigger specifici;
- permettere l'accesso alle tabelle solo attraverso le funzioni e fare l'auditing al loro interno.

Vedremo esempi semplici di entrambi gli approcci. Per prima cosa, create le tabelle:

```
CREATE TABLE salaries(
    emp_name text PRIMARY KEY,
    salary integer NOT NULL
);

CREATE TABLE salary_change_log(
    changed_by text DEFAULT CURRENT_USER,
    changed_at timestamp DEFAULT CURRENT_TIMESTAMP,
    salary_op text,
    emp_name text,
    old_salary integer,
    new_salary integer
);
REVOKE ALL ON salary_change_log FROM PUBLIC;
GRANT ALL ON salary_change_log TO managers;
```

Di solito non darete agli utenti la possibilità di modificare i log dell'auditing, perciò assegnate i diritti di accesso soltanto ai gestori del sistema. Se pensate di consentire agli utenti un accesso diretto alla tabella `salary`, dovreste inserire un trigger per l'auditing:

```
CREATE OR REPLACE FUNCTION log_salary_change () RETURNS trigger AS $$
    BEGIN
        IF TG_OP = 'INSERT' THEN
      INSERT INTO salary_change_log(salary_op,emp_name,new_salary)
     VALUES (TG_OP,NEW.emp_name,NEW.salary);
  ELSIF TG_OP = 'UPDATE' THEN        INSERT INTO salary_change_
log(salary_op,emp_name,old_salary,new_salary)
     VALUES (TG_OP,NEW.emp_name,OLD.salary,NEW.salary);
  ELSIF TG_OP = 'DELETE' THEN
      INSERT INTO salary_change_log(salary_op,emp_name,old_salary)
```

```
      VALUES (TG_OP,NEW.emp_name,OLD.salary);
        END IF;
        RETURN NEW;
    END;
$$ LANGUAGE plpgsql SECURITY DEFINER;

CREATE TRIGGER audit_salary_change
AFTER INSERT OR UPDATE OR DELETE ON salaries
    FOR EACH ROW EXECUTE PROCEDURE log_salary_change ();
```

Adesso, fate un test per la gestione degli stipendi:

```
postgres=# INSERT INTO salaries values('Bob',1000);
INSERT 0 1
postgres=# UPDATE salaries set salary = 1100 where emp_name = 'Bob';
UPDATE 1
postgres=# INSERT INTO salaries values('Mary',1000);
INSERT 0 1
postgres=# UPDATE salaries set salary = salary + 200;
UPDATE 2
postgres=# SELECT * FROM salaries;
-[ RECORD 1 ]--
emp_name | Bob
salary   | 1300
-[ RECORD 2 ]--
emp_name | Mary
salary   | 1200
```

Ognuna di queste modifiche è registrata nella tabella dei log per scopi di auditing:

```
postgres=# SELECT * FROM salary_change_log;
-[ RECORD 1 ]-------------------------
changed_by | frank
changed_at | 2012-01-25 15:44:43.311299
salary_op  | INSERT
emp_name   | Bob
old_salary |
new_salary | 1000
-[ RECORD 2 ]-------------------------
changed_by | frank
changed_at | 2012-01-25 15:44:43.313405
salary_op  | UPDATE
emp_name   | Bob
old_salary | 1000
new_salary | 1100
-[ RECORD 3 ]-------------------------
changed_by | frank
changed_at | 2012-01-25 15:44:43.314208
```

```
salary_op  | INSERT
emp_name   | Mary
old_salary |
new_salary | 1000
-[ RECORD 4 ]-------------------------
changed_by | frank
changed_at | 2012-01-25 15:44:43.314903
salary_op  | UPDATE
emp_name   | Bob
old_salary | 1100
new_salary | 1300
-[ RECORD 5 ]-------------------------
changed_by | frank
changed_at | 2012-01-25 15:44:43.314903
salary_op  | UPDATE
emp_name   | Mary
old_salary | 1000new_salary | 1200
```

D'altro canto, potreste non volere alcun accesso alla tabella `salary` ed eseguire l'istruzione seguente:

```
REVOKE ALL ON salaries FROM PUBLIC;
```

Darete accesso soltanto a due funzioni: la prima è per visualizzare gli stipendi e l'altra per modificarli, accessibile soltanto ai gestori.
Le funzioni avranno invece accesso completo alle tabelle sottostanti, perché sono chiarate come `SECURITY DEFINER`, cioè con gli stessi privilegi dell'utente che le ha create.
La funzione per controllare lo stipendio sarà simile a questa:

```
CREATE OR REPLACE FUNCTION get_salary(text)
RETURNS integer
AS $$
    -- la visualizzazione dello stipendio di altre persone sarà registrata
    INSERT INTO salary_change_log(salary_op,emp_name,new_salary)
    SELECT 'SELECT',emp_name,salary
      FROM salaries
    WHERE upper(emp_name) = upper($1)
      AND upper(emp_name) != upper(CURRENT_USER); - non registra
la visualizzazione del vostro stipendio
    -- restituisce lo stipendio richiesto
SELECT salary FROM salaries WHERE upper(emp_name) = upper($1);$$ LANGUAGE SQL SECURITY
DEFINER;
```

Avete implementato un approccio "morbido", in cui potete visualizzare lo stipendio di altre persone, ma dovrete farlo in modo responsabile – cioè solo quando ne avete bisogno – perché il vostro superiore potrà saperlo.
La funzione `set_salary()` elimina la necessità di controllare se l'utente esiste; in caso negativo, viene creato. Se impostate uno stipendio a `0`, la persona sarà rimossa dalla tabella.

In questo modo, l'interfaccia è semplificata e l'applicazione client di queste funzioni ha bisogno di meno informazioni e fa meno cose:

```
CREATE OR REPLACE FUNCTION set_salary(i_emp_name text, i_salary int)
RETURNS TEXT AS $$
DECLARE
    old_salary integer;
BEGIN
    SELECT salary INTO old_salary
      FROM salaries
     WHERE upper(emp_name) = upper(i_emp_name);
    IF NOT FOUND THEN
        INSERT INTO salaries VALUES(i_emp_name, i_salary);
  INSERT INTO salary_change_log(salary_op,emp_name,new_salary)
      VALUES ('INSERT',i_emp_name,i_salary);
        RETURN 'INSERTED USER ' || i_emp_name;
    ELSIF i_salary > 0 THEN
        UPDATE salaries
     SET salary = i_salary
   WHERE upper(emp_name) = upper(i_emp_name);
  INSERT INTO salary_change_log
              (salary_op,emp_name,old_salary,new_salary)
      VALUES ('UPDATE',i_emp_name,old_salary,i_salary);
        RETURN 'UPDATED USER ' || i_emp_name;
   ELSE -- stipendio impostato a 0
       DELETE FROM salaries WHERE upper(emp_name) = upper(i_emp_name);
  INSERT INTO salary_change_log(salary_op,emp_name,old_salary)
      VALUES ('DELETE',i_emp_name,old_salary);
        RETURN 'DELETED USER ' || i_emp_name;
    END IF;
END;
$$ LANGUAGE plpgsql SECURITY DEFINER;
```

Adesso disattivate il trigger `audit` (altrimenti il log delle modifiche sarà doppio) e fate il test della nuova funzionalità:

```
postgres=# DROP TRIGGER audit_salary_change ON salaries;
ROP TRIGGER
postgres=#
postgres=# SELECT set_salary('Fred',750);
-[ RECORD 1 ]------------------
set_salary | INSERTED USER Fred

postgres=# SELECT set_salary('frank',100);
-[ RECORD 1 ]-------------------
set_salary | INSERTED USER frank

postgres=# SELECT * FROM salaries ;
```

```
-[ RECORD 1 ]---
emp_name | Bob
salary   | 1300
-[ RECORD 2 ]---
emp_name | Mary
salary   | 1200
-[ RECORD 3 ]---
emp_name | Fred
salary   | 750
-[ RECORD 4 ]---
emp_name | frank
salary   | 100

postgres=# SELECT set_salary('mary',0);
-[ RECORD 1 ]-----------------
set_salary | DELETED USER mary

postgres=# SELECT * FROM salaries ;
-[ RECORD 1 ]---
emp_name | Bob
salary   | 1300
-[ RECORD 2 ]---
emp_name | Fred
salary   | 750
-[ RECORD 3 ]---
emp_name | frank
salary   | 100

postgres=# SELECT * FROM salary_change_log ;
...
-[ RECORD 6 ]-------------------------
changed_by | gsmith
changed_at | 2013-01-25 15:57:49.057592
salary_op  | INSERT
emp_name   | Fred
old_salary |
new_salary | 750
-[ RECORD 7 ]-------------------------
changed_by | gsmith
changed_at | 2013-01-25 15:57:49.062456
salary_op  | INSERT
emp_name   | frank
old_salary |
new_salary | 100
-[ RECORD 8 ]-------------------------
changed_by | gsmith
changed_at | 2013-01-25 15:57:49.064337
```

```
salary_op  | DELETE*
emp_name   | mary
old_salary | 1200
new_salary |
```

Pulizia dei dati

Avrete notato che i nomi degli impiegati non sono omogenei per quanto riguarda le maiuscole. Sarebbe meglio aumentare la coerenza aggiungendo un vincolo:

```
CHECK (emp_name = upper(emp_name))
```

Un'alternativa è verificare soltanto che il nome sia archiviato con la lettera maiuscola, e il modo più semplice è attraverso un `trigger`:

```
CREATE OR REPLACE FUNCTION uppercase_name ()
  RETURNS trigger AS $$
    BEGIN
        NEW.emp_name = upper(NEW.emp_name);
        RETURN NEW;
    END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER uppercase_emp_name
BEFORE INSERT OR UPDATE OR DELETE ON salaries
    FOR EACH ROW EXECUTE PROCEDURE uppercase_name ();
```

La prossima chiamata a `set_salary()` per un nuovo impiegato inserirà `emp_name` in modo corretto:

```
postgres=# SELECT set_salary('arnold',80);
-[ RECORD 1 ]-------------------
set_salary | INSERTED USER arnold
```

Quando la conversione avviene all'interno di un trigger, la funzione di risposta mostra ancora il nome originale, anche se nel database è già in maiuscolo:

```
postgres=# SELECT * FROM salaries ;
-[ RECORD 1 ]---
emp_name | Bob
salary   | 1300
-[ RECORD 2 ]---
emp_name | Fred
salary   | 750
-[ RECORD 3 ]---
emp_name | frank
salary   | 100
-[ RECORD 4 ]---
emp_name | ARNOLD
```

```
salary   | 80
```

Dopo aver sistemato la situazione esistente con il mix di maiuscole e minuscole, potete assicurarvi che tutti gli `emp_names` siano sempre in maiuscolo aggiungendo il vincolo:

```
postgres=# update salaries set emp_name = upper(emp_name) where not
emp_name = upper(emp_name);
UPDATE 3
postgres=# alter table salaries add constraint emp_name_must_be_
uppercasepostgres-# CHECK (emp_name = upper(emp_name));
ALTER TABLE
```

Se volete questo comportamento in più posizioni, potrebbe aver senso definire un nuovo tipo, magari `u_text`, che è sempre archiviato in maiuscolo. Approfondiremo questo approccio nel capitolo relativo alla definizione dei tipi personalizzati.

Ordinamenti personalizzati

L'ultimo esempio di questo capitolo riguarda l'utilizzo di funzioni per sistemi di ordinamento diversi.
Supponete di voler ordinare delle parole sulla base delle vocali e fare in modo che l'ultima sia quella più importante durante l'ordinamento. Sembra un'attività complicata, ma è semplice risolverla con le funzioni:

```
CREATE OR REPLACE FUNCTION reversed_vowels(word text)
    RETURNS text AS $$
  vowels = [c for c in word.lower() if c in 'aeiou']
  vowels.reverse()
  return ''.join(vowels)
$$ LANGUAGE plpythonu IMMUTABLE;

postgres=# select word,reversed_vowels(word) from words order by
reversed_vowels(word);
    word    | reversed_vowels
------------+-----------------
Abracadabra | aaaaa
Great       | ae
Barter      | ea
Revolver    | eoe
(4 rows)
```

La cosa migliore è che potete utilizzare questa nuova funzione nella definizione di un indice:

```
postgres=# CREATE INDEX reversed_vowels_index ON words (reversed_
vowels(word));
CREATE INDEX
```

Il sistema utilizzerà in automatico questo indice ogni volta che chiamerete la funzione `reversed_vowels(word)` in una clausola `WHERE` o `ORDER BY`.

Best practice della programmazione

Lo sviluppo di un'applicazione è complicato. Ci sono approcci che aiutano nella gestione e sono così popolari da meritarsi un acronimo per ricordarli. Nei prossimi paragrafi scoprirete alcuni di questi principi e vedrete come siano facili da seguire con la programmazione server.

KISS – Keep It Simple Stupid

Una delle tecniche principali per scrivere codice di successo è farlo in modo semplice, scrivendo codice che potrete capire con facilità anche fra tre anni e che altre persone non avranno difficoltà a comprendere. Non si può sempre realizzare, ma è consigliabile scrivere il codice nel modo più lineare possibile. In futuro potrete riscriverne alcune parti per ragioni come velocità, compattezza, precisione e così via. Prima di tutto scrivetelo in modo semplice, in modo che faccia esattamente ciò che desiderate. Lavorare in questo modo non sarà soltanto più veloce, ma avrete un termine di paragone quando proverete tecniche più avanzate per fare la stessa cosa.
E ricordate, il debug è più difficile della scrittura del codice: perciò, se scrivete in modo astruso, sarà ancora più complicato fare il debug.
Spesso è più facile creare una funzione che restituisce un valore piuttosto che una query complessa. È vero, con tutta probabilità l'esecuzione sarà più lenta perché l'ottimizzatore non potrà fare molto sul codice delle funzioni, ma la velocità sarà sufficiente per le vostre necessità. Se volete più rapidità, è sempre meglio riscrivere il codice pezzo per pezzo, trasformando parti delle funzioni in query più grandi, su cui l'ottimizzatore può fare un lavoro migliore per rendere accettabili le performance.
Ricordate che la maggior parte delle volte non avrete bisogno del codice più veloce in assoluto.
Per i vostri clienti e i capi, il codice migliore è quello che fa bene il proprio lavoro e lo finisce in tempo.

DRY – Don't Repeat Yourself

Con questo metodo dovrete provare a implementare ogni parte della logica operativa soltanto una volta, e collocare il codice nel posto giusto.
A volte può essere difficile, per esempio se volete fare qualche controllo nei form del browser ma volete che quelli conclusivi avvengano nel database. In linea generale, però, è molto valida.
La programmazione server in questo caso aiuta molto. Se il codice per la manipolazione dei dati si trova sul server database, tutti gli utenti possono accedere con facilità e non dovrete gestirlo con un programma C++ in Windows, due siti web PHP e un paio di

script Python che lavorano di notte. Se dovete fare delle operazioni su una tabella di clienti, lanciate soltanto questa istruzione:

```
SELECT * FROM   do_this_thing_to_customers(arg1, arg2, arg3);
```

Ecco fatto!
Se la logica sottesa alla funzione richiede una modifica, potrete cambiare la funzione senza interruzioni e operazioni complicate per lanciare query di aggiornamento su molti client. Una volta che la modificate sul server database, lo avrete fatto per tutti gli utenti.

YAGNI – You Ain't Gonna Need It

In altre parole, non fate più di quanto necessario. Se avete la sensazione che il vostro cliente non abbia ben chiaro quale sarà la dimensione finale del database o che cosa dovrà fare, è meglio resistere alla tentazione di progettare "tutto il mondo" nel database. È meglio un'implementazione minima che soddisfa le specifiche correnti, ma con un'idea di estensibilità. È molto facile finire in un vicolo cieco quando si implementa una grande specifica con molte parti immaginarie.
Se organizzate il vostro accesso al database attraverso le funzioni, spesso potete applicare modifiche consistenti alla logica senza toccare il front end dell'applicazione. La vostra app eseguirà `SELECT * FROM do_this_thing_to_ customers(arg1, arg2, arg3)` anche dopo aver riscritto la funzione cinque volte e cambiato per intero due volte la struttura della tabella.

SOA – Service-Oriented Architecture

Di solito, quando sentite parlare di SOA si tratta di qualcuno che prova a vendervi una soluzione SOAP complessa. L'essenza di SOA è l'organizzazione della vostra piattaforma come un set di servizi che i client e gli altri servizi chiamano per compiere attività atomiche ben definite, come le seguenti:

- controllare la password dell'utente e le credenziali;
- mostrare una lista dei siti web preferiti;
- vendere un nuovo collare rosso per cani con associazione al club dei cani dal collare rosso.

Questi servizi possono essere implementati come chiamate SOAP con le definizioni WSDL relative, server Java con contenitori servlet e infrastrutture di gestione complesse. Possono anche essere un set di funzioni PostgreSQL che accettano un set di argomenti e restituiscono un set di valori. Se i risultati restituiti sono complessi, possono essere in formato XML o JSON, ma spesso un semplice set di dati PostgreSQL è sufficiente. Nel Capitolo 9 imparerete a rendere questi servizi PostgreSQL basati su SOA altamente scalabili.

Estensibilità dei tipi

Alcune delle tecniche precedenti sono disponibili anche in altri database, ma l'estensibilità di PostgreSQL non si ferma qui. Potete scrivere funzioni personalizzate in un qualsiasi

linguaggio di script tra quelli più popolari. Potete anche definire tipi sviluppati in modo completo, non soltanto domini (cioè tipi standard con alcuni vincoli extra collegati).
Per esempio, la compagnia olandese MGRID ha sviluppato il set di dati *value with unit*, in modo da poter dividere 10 km per 0,2 ore e ottenere il risultato di 50 km/h. Ovviamente potete fare il cast del risultato in metri al secondo o ogni altra unità di misura, oltre a mostrarlo come frazione di *c*, la velocità della luce.
Per questo tipo di funzionalità avete bisogno sia dei tipi stessi, sia dell'overload degli operatori, in modo che la divisione di distanza per tempo dia come risultato una velocità. Avrete anche bisogno di cast personalizzati e funzioni di conversione automatiche o manuali tra tipi diversi.
MGRID ha sviluppato questa soluzione per l'utilizzo in applicazioni mediche, dove il costo di un errore può essere molto alto (la differenza tra 10 ml e 10 cc può essere vitale). Il ricorso a un sistema simile avrebbe potuto evitare anche altri disastri, laddove l'utilizzo di unità errate ha comportato risultati di calcolo errati. Se l'unità è sempre disponibile con la quantità, questi tipi di errori sono molto meno probabili. Potete anche aggiungere i vostri metodi personalizzati se avete delle competenze di programmazione e l'indice esistente non risolve i vostri problemi. PostgreSQL integra un set corposo di indici, e molti altri vengono sviluppati all'esterno.
L'ultimo metodo incluso in modo ufficiale in PostgreSQL è KNN (*K Nearest Neighbor*), un indice intelligente che può restituire K righe ordinate in base alla distanza dal target. Uno degli impieghi di KNN è nella ricerca testuale libera, con la creazione di un punteggio per i risultati delle ricerche in base alla corrispondenza con i termini. Prima di KNN, questo tipo di ricerche era effettuato con una query su tutte le righe che corrispondevano anche in minima parte, ordinate secondo la funzione distanza che restituiva le prime K come ultimo passo.
Utilizzando l'indicizzazione KNN, l'accesso all'indice può restituire le righe nell'ordine desiderato; sarà sufficiente un semplice `LIMIT K` per ottenere i primi K elementi trovati. L'indice KNN può essere impiegato anche per distanze reali, per esempio rispondendo alla query "mostrami le 10 pizzerie più vicine alla stazione centrale".
Come potete notare, gli indici sono separati dai dati. Prendete per esempio GIN (*General Inverted Index*): può essere utilizzato sia per ricerche testuali (insieme a derivazioni, sinonimi e altre elaborazioni), sia per l'indicizzazione degli elementi di un array di interi.

Cache

Un altro modo di utilizzare la programmazione lato server riguarda la memorizzazione nella cache dei valori, spesso difficile da realizzare. Il pattern di base è il seguente.

1. Controllate se il valore è nella cache.
2. In caso negativo, oppure se il valore è troppo vecchio, elaboratelo e inseritelo nella cache.
3. Restituite il valore.

Per esempio, il calcolo delle vendite di un'azienda è un ottimo dato da mettere nella cache. Un'impresa commerciale di grandi dimensioni può avere mille negozi, con milioni di transazioni individuali al giorno. Se la casa madre vuole conoscere l'andamento delle

vendite, è molto più efficiente calcolare in anticipo i dati e immagazzinarli piuttosto che sommare milioni di transazioni giornaliere.
Se il valore è semplice, per esempio l'informazione su un utente presa da una tabella singola basata sull'ID, non dovete fare nulla. Questo viene inserito nella cache interna di PostgreSQL e tutte le ricerche saranno rapide anche in una rete molto veloce, tanto che la maggior parte del tempo sarà speso nel traffico di rete, non nella ricerca in sé. In un caso come questo, le prestazioni sono simili a quelle di altri sistemi di cache in memoria (come mecached) ma senza nessun sovraccarico nella gestione.
Un altro caso di utilizzo tipico della cache è l'implementazione delle viste materializzate. Si tratta di viste calcolate solo quando necessario, prima di lanciare una query. Alcuni database SQL le trattano come oggetti separati, ma in PostgreSQL dovete fare tutto voi, utilizzando altre caratteristiche del database per automatizzare il processo.

Ricapitoliamo: perché programmare nel server?

I vantaggi principali di manipolare i dati lato server sono i seguenti.

Performance

Eseguire i calcoli vicino ai dati è spesso un successo per quanto riguarda le prestazioni, perché la latenza per recuperarli è minima. In una situazione tipica di calcolo intensivo, la maggior parte del tempo è dedicata al prelievo dei dati. Inoltre, rendere più rapido l'accesso ai dati è il modo migliore per velocizzare tutto. Il mio portatile impiega 2,2 ms per eseguire una query su una riga casuale in un database di 100.000 righe nel client, ma solo 0,12 ms recuperare i dati all'interno del database. Il processo è 20 volte più veloce e sempre sulla stessa macchina basata su Unix. La differenza aumenta se client e server sono connessi in rete.
Ecco un breve esempio reale: un mio amico è stato chiamato per aiutare una grande azienda (sono sicuro che tutti la conosciate, ma non posso dirvi quale) a far diventare il sistema di generazione delle e-mail più veloce. L'implementazione originale impiegava tutte le ultime tecnologie Java EE, recuperando prima i dati dal database per passarli ai vari servizi che si occupavano di serializzarli e deserializzarli molte volte, prima di trasformarli in XSLT e generare il testo dell'e-mail. Il risultato finale era la produzione di poche centinaia di mail per secondo, il che tradiva tutte le attese.
Quando riscrisse il processo utilizzando una funzione PL/Perl interna al database per formattare i dati e la query restituiva e-mail già pronte, immediatamente iniziarono a essere create decine di migliaia di e-mail per secondo e dovettero aggiungere un secondo server di posta per essere in grado di spedirle davvero.

Facilità di manutenzione

Se tutto il codice per la manipolazione dei dati è nel database (comprese le funzioni e le viste), il processo di aggiornamento diventa davvero semplice. Tutto quello che serve è lanciare uno script DDL per ridefinire le funzioni, in modo che tutti i client utilizzino

in automatico il nuovo codice senza interruzioni né attività di coordinazione complesse tra vari sistemi di front end e gruppi di lavoro.

Sistemi semplici per aumentare la sicurezza

Se l'accesso ai server con problemi di sicurezza passa attraverso le funzioni, è possibile concedere agli utenti del database l'accesso solo alle funzioni necessarie e nient'altro. Non potranno vedere le tabelle dei dati, né sapere che queste esistono. In questo modo, se la sicurezza del server fosse compromessa, tutto quello che può succedere è che siano chiamate di continuo le stesse funzioni. Non è nemmeno possibile rubare password, e-mail o altri dati sensibili utilizzando query del tipo `SELECT * FROM users;` per recuperare tutti i dati che ci sono nel database. E la cosa più importante è che la programmazione server è divertente!

Riepilogo

Programmare in un server database non è sempre la prima cosa che viene in mente a molti sviluppatori, ma la sua posizione all'interno dello stack applicativo porta vantaggi enormi. Le vostre applicazioni possono essere più veloci, sicure e gestibili se inserite la logica nel database. Con la programmazione lato server PostgreSQL, potete:

- rendere sicuri i vostri dati utilizzando le funzioni;
- fare l'auditing dell'accesso ai dati utilizzando i trigger;
- arricchire i vostri dati con tipi personalizzati;
- analizzare i dati con operatori personalizzati.

Questo è soltanto l'inizio di quello che potete fare con PostgreSQL. Continuando nella lettura del libro, imparerete molti altri modi per scrivere applicazioni potenti sfruttando questo ambiente di programmazione.