

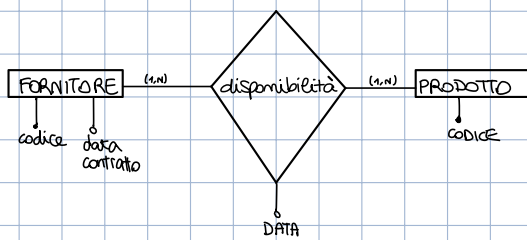
Funzioni definite dall'utente e trigger

Nicola Vitacolonna

Corso di Basi di Dati
Università degli Studi di Udine

26 novembre 2015





Non deve esserci un prodotto disponibile, da un fornitore, prima della sua data contratto

create or replace function verifica_data_contratto (f int, d date) => funzione che verifica e ritorna true/false -> controlla in Disponibilità, poi devo anche controllare in Fornitore
returns boolean

language plpgsql as

\$\$

begin -- se non devo selezionare nulla uso perform

perform * from Disponibilità

where fornitore = f and data < d

return not found; -- se ho trovato qualcosa resto

end;

\$\$

e poi la funzione la uso in una clausola con check: alter table Fornitore

add constraint controllo_data_contratto

check (verifica_data_contratto (codice, data_contratto));

Funzioni definite dall'utente

- Lo standard SQL definisce un insieme di **funzioni** per la manipolazione dei dati
 - **current_time** restituisce l'istante corrente
 - || concatena due stringhe
 - Funzioni aggregate
 - Etc...
- L'utente può inoltre definire le proprie funzioni (**user-defined function, UDF**)
- Le UDF sono memorizzate nel DBMS server, come gli altri oggetti della base di dati
- Le funzioni possono essere scritte in SQL o in un altro linguaggio
- La sintassi usata dai vari DBMS non è uniforme
- **Nota:** gli esempi seguenti usano la sintassi di PostgreSQL

Le definisco così poi nella mia applicazione la richiamo (senza usare nessun comando SQL nell'app)
sono definite all'interno del DB e possono essere richiamate da fuori

Funzioni nel linguaggio PL/pgSQL

```
create or replace function <nome> (<parametri>)  
returns <tipo>  
language plpgsql as $$  
    declare  
        <dichiarazioni di variabili>  
    begin  
        <istruzioni>  
    end;  
$$;
```

- Il corpo della funzione (tra \$\$) è tecnicamente una stringa
- Il risultato di un'interrogazione è assegnato a una variabile con **select... into**
- La funzione restituisce un valore mediante un'istruzione **return**
- Si possono definire funzioni con lo stesso nome e lo stesso valore di ritorno, purché con parametri diversi (**overloading**)

UDF: esempio

```
create or replace function dip_count(  
  nome_dip nomi_dipartimento → Dominio  
) returns integer language plpgsql as  
$$  
  declare  
    d_count integer;  
  begin  
    select count(*) into d_count  
      from Dipartimento D, Impiegato I  
     where I.dip = D.dnumero  
          and D.dnome = nome_dip;  
  
    return d_count;  
  end;  
$$;  
  
-- Esempio d'invocazione:  
select dnome, dip_count(dnome) from Dipartimento;
```

Parametri di input e parametri di output

Un'altra versione della funzione precedente:

```
create or replace function dip_count (  
    in nome_dip varchar(20), -- Parametro di input  
    out d_count integer -- Parametro di output  
)  
language plpgsql as $$  
begin  
    select count(*) into d_count  
    from Dipartimento D, Impiegato I  
    where I.dip = D.dnumero and D.dnome = nome_dip;  
end;  
$$;
```

- In presenza di parametri **out**, la clausola **return** è ridondante
- Una funzione può avere più parametri **out** (l'oggetto restituito in tal caso è di tipo **record**)
- **in** è opzionale (è il default)

Parametri out multipli

```
create or replace function somma_prodotto (x int, y int,  
    out somma int, out prod int) language plpgsql as  
$$  
    begin  
        somma := x + y;  
        prod := x * y;  
    end;  
$$;
```

```
select somma_prodotto(3, 4);
```

```
   somma_prodotto  
-----  
   (7,12)
```

```
select * from somma_prodotto(3, 4);
```

```
   somma | prod  
-----+-----  
       7 |    12
```


Funzione equivalente senza parametri out

```
create type somma_prod as (somma int, prod int);

create or replace function somma_prodotto (
    x int, y int
)
returns somma_prod
language plpgsql as
$$
    declare
        risultato record;
    begin
        risultato := (x + y, x * y);
        return risultato;
    end;
$$;
```


Funzioni come viste parametrizzate: definizione

- Il valore di ritorno può essere una tabella
- Generalizzazione del meccanismo delle viste

```
create or replace function donne_dip(nome_dip varchar(20))
returns table ( 
  cognome nomi_persona,
  iniziale iniziali_persona,
  nome nomi_persona
)
language plpgsql as
$$
begin
  return query
  select I.cognome, I.iniziale, I.nome
  from Dipartimento D, Impiegato I
  where I.dip = D.dnumero
  and D.dnome = nome_dip
  and I.sesso = 'F';
end;
$$;
```

Funzioni come viste parametrizzate: invocazione

La funzione precedente può essere usata come segue:

```
select * from donne_dip('Dipartimento 1');
```

cognome	iniziale	nome
Stroman	A	Alivia

```
select nome from donne_dip('Dipartimento 3')  
where iniziale = 'A';
```

nome
Alysha
Antonette

Funzioni per la verifica di vincoli

- Una funzione può essere invocata all'interno di una clausola **constraint... check**
- Una tale funzione può essere usata per implementare vincoli complessi che possono essere violati solo in fase di inserimento o aggiornamento (ma non da cancellazioni)
- Ciascuna clausola **constraint... check** è verificata subito dopo l'inserimento o l'aggiornamento di ciascun record ed è soddisfatta quando l'espressione che viene valutata è **true** o **unknown**
- **Nota:** fino alla versione 9.5, in PostgreSQL i vincoli espressi mediante **constraint... check** non sono differibili
- I trigger (vedi oltre) forniscono un meccanismo alternativo, più flessibile e in certi casi più efficiente per la verifica dei vincoli d'integrità

Funzioni per la verifica di vincoli: esempio

“Un impiegato e il suo supervisore devono afferire allo stesso dipartimento”

```
create or replace function → funzione controlla supervisore
controlla_supervisore(imp cf_persona)
returns boolean language plpgsql as
$$
  declare
    ok boolean;
  begin
    raise notice 'Controllo supervisore di %', imp;
    select I.dip = S.dip into ok → controllo e salvo l'esito (TRUE/FALSE) in ok
      from Impiegato I join Impiegato S on I.cf = imp
     where I.supervisore = S.cf;
    return ok;
  end;
$$;

alter table Impiegato add constraint supervisione
  check (controlla_supervisore(cf));
```

Estensioni procedurali di SQL

- Dichiarazioni di variabili (**declare**)
- Assegnamenti (e.g., `n := n + 1`)
- Costrutti condizionali (**if-then-else**, **case**)
- Costrutti iterativi (**while**, **for**, **loop**)
- Gestione delle eccezioni (**raise exception**)
- Fare riferimento ai §40.1–40.6 del manuale di PostgreSQL
- È possibile scrivere funzioni e procedure in un linguaggio esterno (C, Java, PHP, Python, Ruby, R, Scheme, ...)
- Per alcuni linguaggi (e.g., Java), è possibile eseguire le funzioni in un ambiente con accesso alla memoria ristretto (**sandbox**)

UDF: vantaggi

- Semplificazione delle applicazioni attraverso la condivisione tra applicazioni diverse di codice di interesse generale
- Semantica uniforme di alcune operazioni sulla base di dati
- Controllo centralizzato di vincoli d'integrità non esprimibili in SQL
- Riduzione del traffico di rete: il client deve solo inviare una chiamata di procedura remota invece di una sequenza di istruzioni SQL
- Sicurezza dei dati: si può consentire l'accesso ai dati soltanto tramite le procedure e le funzioni definite, senza permettere agli utenti di accedere direttamente alle tabelle (analogie col meccanismo delle viste)

UDF: svantaggi

- Riduzione della portabilità (ogni DBMS ha la propria sintassi)
- L'uso esteso di UDF non è considerato una buona pratica da una parte della comunità dell'ingegneria del software (discutibile)
- Alcune metodologie di progettazione del software non supportano lo sviluppo di tali procedure
 - Specialmente nel contesto delle metodologie “agili”, che prevedono un'evoluzione rapida del software
 - Attività di **debugging** e **testing** sono possibili per le UDF
- Maggiori competenze richieste: DBA (database administrator) e programmatori sono figure professionali con competenze diverse
- Spesso DBA e sviluppatori software sono in team separati, con conseguenti problemi di gestione e comunicazione tra gruppi di lavoro