Relazione di Laboratorio
Basi di dati e sistemi informativi
SQL3 vs. PostgreSQL 7.1.2

Federico Pellegrin

11 dicembre 2001

1 User-defined types

Una delle idee di base per l'espansione a linguaggio ad oggetti del SQL è l'aggiunta della possibilità di creare nuovi tipi di dati, usabili nello stesso modo dei dati built-in. Per ogni nuovo tipo di dati è chiaramente necessaria la definizione della sua struttura e delle procedure, dette *routines*, che li manipolano.
La definizioni di nuovi tipi da standard SQL3 è molto intuitiva e forniamo qui l'esempio fornito nello standard preso in considerazione:

```
CREATE TYPE employee_t
 (PUBLIC
   name CHAR(20),
   b_address address_t,
   manager employee_t,
   hiredate DATE,
  PRIVATE
  base_salary DECIMAL(7,2),
   commission DECIMAL(7,2),
  PUBLIC
   FUNCTION working_years (p employee_t) RETURNS INTEGER
    <code to calculate number of working years>,
  PUBLIC
   FUNCTION working_years (p employee_t, y years) RETURNS employee_t
   <code to update number of working years>,
  PUBLIC
   FUNCTION salary (p, employee_t) RETURNS DECIMAL
   <code to calculate salary>
);
```

La definizione del tipo si basa su un gruppo di definizioni in SQL degli attributi del tipo e un gruppo di funzioni alle quali viene assegnato un nome, dei parametri ed il tipo di valore di ritorno oltre che al codice che esegue il calcolo vero e proprio.

PostgreSQL supporta la creazione di nuovi tipi, ma l'implementazione attuale si differenzia notevolmente dallo standard proposto. Ogni nuovo tipo di PostgreSQL viene definito come uno spazio di memoria di una certa dimensione (fissa o variabile) e due funzioni: una, detta *input function*, per la trasformazione dal formato stabilito ricevuto in ingresso dalla query SQL al formato interno e l'altra, detta *output function*, per convertire i dati dal formato interno al formato utente. Il formato vero e proprio del nuovo tipo dunque non è nè visibile all'esterno né definito chiaramente dalla definizione di tipo nel linguaggio SQL ma è bensi interamente manipolato da queste due funzioni. Come esempio useremo la definizione di un nuovo tipo di numero complesso, la quale definizione in PostgreSQL è:

```
CREATE TYPE complex (
  internallength = 16,
  input = complex_in,
  output = complex_out
);
```

Le due funzioni *complex_in* e *complex_out* avranno invece il compito di trasformare il dato dal formato della query (nell'esempio il numero complesso verrà rappresentato nella forma *(Re,Im)* dove *Re* rappresenta la parte reale, *Im* la parte immaginaria) al formato interno (per la scelta della lunghezza il formato interno sarà rappresentato da una struttura composta da due cifre di 8 byte di tipo double) e viceversa. Le due funzioni dovranno chiaramente essere definite prima del tipo. I parametri alle funzioni, ovvero i dati rappresentati in forma di query, dovranno essere di tipo *opaque* ovvero dei semplici *puntatori ad una stringa*. PostgreSQL quindi semplicemente delegherà alle funzioni sottostanti il compito di manipolare i dati.
Un'altra problematica dell'implementazione dei tipi in PostgreSQL è la necessità di definire le due operazioni in input ed output per i tipi usando esclusivamente funzioni in linguaggio C in un file di oggetto esterno. Infatti non sono definibili, almeno al momento, statement SQL recanti tipi opaque. La definizione di un nuovo tipo in PostgreSQL necessita quindi la conoscenza del linguaggio C e l'implementazione di tutte le funzioni sui tipi in tale linguaggio. Questo certo permette grande flessibilità da una parte ma grande difficoltà e complessità dall'altra.
Le routine in C scritte e compilate in file di oggetto dovranno dunque avere lo stesso nome della funzione definita dal statement per la creazione del tipo. La funzione di input dovrà avere come parametro un puntatore a stringa (*char **) e ritornerà un puntatore al nuovo tipo di dato definito. Viceversa per la funzione di output.

```
CREATE FUNCTION complex_in(opaque)
RETURNS complex
AS '/usr/local/src/postgresql-7.1.2/src/tutorial/complex.so'
LANGUAGE 'c';

CREATE FUNCTION complex_out(opaque)
RETURNS opaque
AS '/usr/local/src/postgresql-7.1.2/src/tutorial/complex.so'
LANGUAGE 'c';
```

E vediamo ora le funzioni C che implementano le operazioni di input e di output:

```
Complex * complex_in(char *str)
{
        double x,y;
        Complex *result;

/* Viene eseguito un check sulla struttura corretta della stringa
   passata */
        if (sscanf(str, " ( %lf , %lf )", &x, &y) != 2)
        {
           elog(ERROR, "complex_in: error in parsing \"%s\"", str);
           return NULL;
        }

/* Viene allocata la memoria per il tipo usando la funzione
           di allocazione palloc fornita dalla libreria di PostgreSQL */

        result = (Complex *) palloc(sizeof(Complex));
        result->x = x;
        result->y = y;

/* Viene restituito un puntatore al nuovo dato */

        return result;
}

char * complex_out(Complex * complex)
{
        char       *result;

        if (complex == NULL)
                return NULL;

        result = (char *) palloc(60);
        sprintf(result, "(%g,%g)", complex->x, complex->y);
        return result;
}
```

E chiaramente c'e' la necessità di definire anche la nuova struttura *complex* come:

```
typedef struct Complex
{
        double          x;
        double          y;
} Complex;
```

Definite le funzioni di ingresso ed uscita il nuovo tipo può essere usato per le solite query SQL, ad es:

```
CREATE TABLE prova ( C1 complex );
INSERT INTO prova VALUES ('(2,3)');
INSERT INTO prova VALUES ('(4,5)');
INSERT INTO prova VALUES ('a');
```

Dove l'ultima query genererà un errore dato che la stringa passata non è conforme alla specifica definita nella routine di ingresso dei dati.
Definite le funzioni di base per la definizione di un tipo si possono aggiungere chiaramente nuovi operatori o funzioni al nuovo tipo di dati. Ad esempio per poter implementare la procedura per la somma di numeri complessi si deve definire una nuova funzione in modo simile al precedente (dunque sempre in C) e poi aggiungere la linea di definizione SQL del tipo

```
CREATE FUNCTION complex_add(complex, complex)
   RETURNS complex
   AS '/usr/local/src/postgresql-7.1.2/src/tutorial/complex.so'
   LANGUAGE 'c';
```

Tutte le funzioni definite sul nuovo tipo possono essere definite come operatore su tale tipo nel linguaggio di interrogazione SQL. Ad esempio, definita la funzione di somma, vogliamo che l'operatore $+$ nelle nostre query dove siano presenti due tipi complex definiti ci ritorni l'effettiva somma dei due numeri. Questo può essere eseguito con la seguente query:

```
CREATE OPERATOR + (
   leftarg = complex,
   rightarg = complex,
   procedure = complex_add,
   commutator = +
);
```

Per concludere l'esempio di creazione del tipo andiamo ancora ad esaminare la definizione dell'operatore $=$ necessario per qualsiasi tipo per poter eseguire anche le query più semplici (si veda ad esempio una cancellazione da un tabella con una query del tipo *DELETE FROM tabella WHERE A1=(1,2);*). Andranno dunque definite prima la funzione e dopodichè l'operatore come segue:

```
CREATE FUNCTION complex_abs_eq(complex, complex)
   RETURNS bool
   AS '/usr/local/src/postgresql-7.1.2/src/tutorial/complex.so'
   LANGUAGE 'c';

CREATE OPERATOR = (
   leftarg = complex,
   rightarg = complex,
   procedure = complex_abs_eq
);
```

Dove la funzione *complex_abs_eq* nel file oggetto specificato è una semplice routine per la comparazione in modulo tra due numeri complessi strutturati come definito sopra che ritorna un valore booleano *true* quando questi due sono uguali.

1.1 Richieste di attributi

Data la modalità di definizione dei tipi come strutture del tutto nascoste accedibili solo tramite funzioni esterne è chiaro che la modalità di richieste definita dallo standard SQL3 in paragrafo 2.2, nella cosìddetta *dot notation* non è implementata ed ogni accesso a una sottoparte del nuovo tipo di dato deve essere necessariamente eseguita definendo delle nuove funzioni in C.

1.2 Incapsulazione

Discorso simile all'accesso si applica sulla possibilità da standard di definire attributi o funzioni pubblici o privati usando le keyword *PUBLIC* e *PRIVATE* davanti alla definizione dell'attributo o della funzione. Nel caso di PostgreSQL non esistono attributi visibili dall'esterno e sono dunque tutti definiti implicitamente privati. L'eventuale necessità di accedere ad una parte dell'oggetto (questo pseudo-attributo) viene comunque risolto tramite la definizione di una funzione. Le funzioni altresí sono tutte pubbliche, le private, cioè quelle atte a manipolare internamente l'oggetto, sono comunque invisibili all'esterno e sono presenti eventualmente solo nel file di oggetto contenente il codice di implementazione.

2 Row type e named row type

Lo standard SQL3 offre oltre agli user-defined types i *row type* ovvero una generalizzazione per definire un nuovo tipo come gruppo di righe usando la parola chiave *ROW* che possono così essere visti come dato unico, ad esempio nel passaggio di dati a funzioni. Ad es. in

```
CREATE TABLE employees
(name CHAR(40),
 address ROW(street CHAR(30), city CHAR(20)));
```

l'attributo recante l'indirizzo *address* è visto come attributo composto da due attributi *street* e *city*. Se si va a definire un vero e proprio tipo contenente un gruppo di righe si parla di *named row type* e viene definito nel seguente modo:

```
CREATE ROW TYPE address
(street CHAR(30),
 city CHAR(20));
```

e può essere in seguito usato come tipo primitivo. Ogni tipo contenuto nel row-type è visibile all'esterno con il proprio nome, come da esempio:

```
CREATE TABLE addresses OF address
  (PRIMARY KEY street);
```

L'esempio per quanto realisticamente inconsistente (imporrebbe un abitante per via) ci dimostra come si referenzi l'attributo street incluso nel row-type definito pocanzi.
PostgreSQL *non* implementa in alcun modo nè i row type nè i named row type. Non ci sono strutture simili per poter implementarli. L'unico modo per ricavare qualcosa di vagamente simile è utilizzare gli user-defined types come descritto nella sezione precedente includendo le necessarie funzioni per accedere ad ogni attributo interno del nuovo tipo.
Lo standard SQL3 inoltre specifica il tipo di dato *reference type* come tipo di dato, generato unicamente dal sistema, usato per referenziare direttamente dei dati di tipo row-type. E' chiaro che neanche questa aggiunta è supportata da PostgreSQL.

3 Altre aggiunte ai tipi

3.1 Distinct type

Nello standard SQL3 si può marcare due tipi come *distinti* all'atto della creazione usando la keyword *DISTINCT* sebbene questi siano equivalenti. L'esempio dallo standard utilizzato è molto chiaro:

```
CREATE DISTINCT TYPE us_dollar AS DECIMAL(9,2);
CREATE DISTINCT TYPE canadian_dollar AS DECIMAL(9,2);
```

Per qualsiasi query i due tipi risulteranno differenti e quindi ogni tentativo di utilizzare un'istanza di un tipo come istanza dell'altro tipo genererà un errore.
PostgreSQL *non* implementa questa possibilità. La motivazione supponibile a questa mancanza è il modo in cui vengono gestiti gli user definied type in PostgreSQL che per definizione creano solo ed esclusivamente tipi distinti. Infatti nella definizione viene mostrata solo la lunghezza del tipo di dato ma non viene detto nulla della sua struttura che è accessibile solo tramite le funzioni citate.

3.2 Collection type

Lo standard SQL3 sono previste anche strutture per la definizione di collection type quali insiemi (*set*), multiinsiemi (*multisets*) e liste di altri tipi di base o UDT definibili nelle tabelle. PostgreSQL *non* implementa questi tipi di definizione. PostgreSQL in compenso ci offre la possibilità di definire dei vettori (e vettori multidimensionali) di un altro tipo di dato e, sebbene in maniera differente, si potrebbe immaginare di implementare i tre definitori di collection type usando questa struttura con un set di funzioni appropriate. La definizione di vettori in PostgreSQL è simile a quella di molti linguaggi di programmazione ad alto livello:

```
...
projects        integer[],
...
```

E i dati in essi sono accessibili indicando l'indice dell'elemento, che in PostgreSQL partono da 1 e non hanno dimensioni prefissate al momento della definizione.

3.3 Large Objects

Lo standard SQL3 offre due nuovi tipi BLOB (Binary Large Object) e CLOB (Character Large Object) per supportare oggetti di dimensioni notevoli direttamente nel database (e non in file esterni). PostgreSQL supporta questi due tipi sotto l'unico nome di *oid* (Object identifier) senza fare differenza tra i due tipi con o senza caratteri binari. L'unica differenza è che da standard i due LOB possono avere un'impostazione preimpostata della loro grandezza come parametro mentre PostgreSQL non accetta dimensioni ma bensí gestisce automaticamente le necessarie allocazioni di spazio.

4 Specializzazione

Nello standard SQL3 è prevista la specializzazione di tipi e delle tabelle. Un tipo può essere quindi definito come sottotipo di un altro tipo (o altri tipi se l'ereditarietà è multipla) e ne eredita tutti gli attributi e comportamenti. Per come abbiamo visto essere strutturata la definizione di nuovi tipi in PostgreSQL è chiaro che questa possibilità al momento non è implementabile in maniera decente, se non complicando di gran lunga le funzioni di accesso al nuovo tipo definito.
Nello standard viene poi definita l'ereditarietà per le tabelle. Ogni tabella può essere definita come sottotabella di una o più tabelle e ne eredita di conseguenza ogni colonna ed ogni riga della sottotabella corrisponde chiaramente ad esattamente una riga di ogni supertabella dalla quale deriva. La definizione proposta nello standard SQL3 usa la keyword *UNDER*, esemplificando:

```
CREATE TABLE person
  (name CHAR(20) PRIMARY KEY,
   sex  CHAR(1),
   age  INTEGER)

CREATE TABLE employee UNDER person
  (salary FLOAT);
```

Definita la tabella *person* viene definita la tabella specializzata *employee* da *person* con l'aggiunta di un attributo.
PostgreSQL supporta la specializzazione di tabelle, sebbene la notazione usata è leggermente diversa e il suo funzionamento sia alquanto strano. Avendo definito come nell'esempio la tabella *person* se vogliamo costruire in PostgreSQL la corrispondente tabella *employee* è necessario eseguire:

```
CREATE TABLE employee
  (salary FLOAT) INHERITS (person);
```

Il funzionamento dell'inheritance sotto PostgreSQL corrisponde allo standard proposto SQL3 è più precisamente:

- l'inserzione di una riga in una sottotabella comporta l'inserzione della riga nella supertabella;
- l'aggiornamento di una riga nella supertabella comporta l'aggiornamento della stessa riga in tutte le sue sottotabelle;
- l'aggiornamento di una riga in una sottotabella comporta l'aggiornamento della parte di riga ereditata in tutte le sue supertabelle;
- la cancellazione di una riga da una tabella comporta la cancellazione delle righe che ereditano tale informazione in tutte le corrispondenti sottotabelle.

Le difficoltà si presentano quando analizziamo il mantenimento dei vincoli di chiave delle tabelle dove notiamo due problemi. Innanzitutto la sottotabella creata non

mantiene le chiavi definite nella sopratabella. Queste devono essere necessariamente forzate simulando l'aggiunta degli stessi attributi chiave anche nella sottotabella (il nameclashing verrà risolto tentando un merge dei due attributi, sempre possibile se i due attributi sono di tipo identico). Nell'esempio dovremmo forzare:

```
CREATE TABLE employee
  (name CHAR(20) PRIMARY KEY,
   salary FLOAT) INHERITS (person);
```

Il secondo e ben più serio problema riguarda la modalità nella quale vengono effettivamente immagazzinati i dati delle tabelle specializzate. In realtà PostgreSQL immagazzinerà ogni sottotabella in una struttura dati a se stante e solo in fase di esecuzione di query andrà ad eseguire la query su ogni sottotabella raggruppando i dati in output ma *perdendo tutti i vincoli di chiave*. Ad esempio dalla definizione soprastante possiamo vedere un caso critico:

```
fede=# INSERT INTO person (name,sex,age) VALUES ('NomeT','f',36);
INSERT 27431 1
fede=# INSERT INTO employee (name,sex,age,salary) VALUES ('NomeT','f',36,1);
INSERT 27432 1
fede=# SELECT * FROM person;
         name          | sex | age
-----------------------+-----+-----
 NomeT                 | f   |  36
 NomeT                 | f   |  36
(2 rows)
```

Mentre se andiamo a vedere solo ed esclusivamente i dati nella sopratabella utilizzando la keyword ONLY avremo:

```
fede=# SELECT * FROM ONLY person;
         name          | sex | age
-----------------------+-----+-----
 NomeT                 | f   |  36
(1 row)
```

E l'altra tupla presente nella sottotabella. E' chiaro che bisogna fare molta attenzione all'uso della specializzazione definita in questo modo, dato che e' evidente che si potrebbe incorrere in situazioni critiche come la perdita del vincolo di chiave ignorando queste specialità' di PostgreSQL. Alternativamente si potrebbe definire dei trigger per il mantenimento incrociato del vincolo di chiave, cosa non certo molto elegante e pratica. E' da notare che nella TODO-list della versione analizzata ci sono note proprio riguardo al problema del mantenimento del vincolo di chiave esposto.

5 Polimorfismo

Per quanto riguarda le definizioni di polimorfismo o *overloading* di funzioni PostgreSQL è in linea con le richieste dello standard SQL3. Possono essere dunque definite funzioni con lo stesso nome ma con parametri diversi (in tipo o numero) e al momento della chiamata verrà definita quale delle routine deve essere chiamata secondo modalità interne in certi casi probabilmente anche complesse. Esemplificando:

```
CREATE FUNCTION testpoly() RETURNS int4
  AS 'SELECT 1 AS RESULT'
  LANGUAGE 'sql';

CREATE FUNCTION testpoly(int4) RETURNS int4
  AS 'SELECT 2 AS RESULT'
  LANGUAGE 'sql';

CREATE FUNCTION testpoly(float8) RETURNS int4
  AS 'SELECT 3 AS RESULT'
  LANGUAGE 'sql';
```

Definiamo la funzione *testpoly* che può non avere parametri, avere un intero o avere un numero a virgola mobile e rispettivamente ritornare 1, 2 o 3. Provando ad eseguire delle chiamate a testpoly ricaviamo:

```
fede=# SELECT testpoly();
 testpoly
----------
        1
(1 row)

fede=# SELECT testpoly(1);
 testpoly
----------
        2
(1 row)

fede=# SELECT testpoly(1.1);
 testpoly
----------
        3
(1 row)
```

Dall'esempio viene messa anche in evidenza la scelta, nel secondo caso, della chiamata a *testpoly(int4)* sebbene anche l'uso della *testpoly(float8)* potrebbe essere stato ritenuto corretto.
L'uso del polimorfismo invece per la specializzazione delle operazioni in sottotipi non è contemplato in alcun modo da PostgreSQL semplicemente per l'assenza della possibilità di definire sottotipi nell'usuale modo.

6 Trigger

Sebbene non presenti nello standard *SQL-92* i trigger sono presenti come estensioni più o meno compatibili in molti DBMS. Lo standard SQL3 si prefigge di introdurli come parte integrante del modello. I trigger sono specifiche di attuazione di un'azione al verificarsi di una qualche operazione di modifica di una tabella.
PostgreSQL implementa i trigger da diverse versioni e la sintassi è pressochè uguale a quella definita dallo standard. L'unica piccola differenza la ritroviamo nel fatto che PostgreSQL contiene come effetto dell'esecuzione del trigger una chiamata ad una funzione prima definita e non del codice SQL vero e proprio. La differenza evidentemente è minimale e, pensandoci, forza probabilmente una maggior leggibilità delle definizioni dei trigger.
Infine PostgreSQL non supporta (dalla documentazione è evidente che molto probabilmente questa funzionalità verrà implementata in futuro) l'esecuzione di trigger unicamente per tutta la condizione che l'ha attivato (tipo trigger *FOR EACH STATEMENT*) ma soltanto l'esecuzione di trigger per ogni riga della tabella che lo attiva (tipo trigger *FOR EACH ROW*).

7 Estensione del linguaggio procedurale

Lo standard SQL3 prevede l'aggiunta di una serie di statement per arricchire il linguaggio SQL nella definizione di procedure e farlo tendere ad un linguaggio abbastanza completo dal punto di vista computazionale. PostgreSQL supporta tutte le estensioni richieste nel linguaggio esteso interno già presente nelle versioni precedenti (e presente anche in altri sistemi di DBMS) di nome *PL/SQL* (*SQL Procedural Language*). I statement necessari definiti dallo standard SQL3 sono:

- definizioni e assegnamenti di variabili (*:=*);
- costrutti condizionali di tipo *IF-THEN-ELSE* e *CASE*;
- costrutti di chiamate procedurali (*CALL*) e ritorno (*RETURN*);
- costrutti iterativi quali *LOOP*, *WHILE* e *REPEAT*;

I costrutti, benchè con definizioni sintattiche diverse in piccoli particolari (ad esempio il costrutto REPEAT di per se non è presente ma può essere banalmente ricavato con un WHILE ed un'uscita forzata con il comando EXIT presente nel linguaggio), sono tutti presenti ed in aggiunta il linguaggio PL/SQL offre anche un notevole insieme di altri statement e definizioni.
L'unico capitolo scarsamente affrontato da PL/SQL è la gestione delle eccezioni. In occorrenza di un'eccezione infatti l'esecuzione verrà semplicemente abortita e il controllo verrà passato al chiamante precedente. Una gestione delle eccezioni è consigliata nella specifica SQL3 trattata ma non ci sono molte indicazioni a riguardo alla loro implementazione o altro.