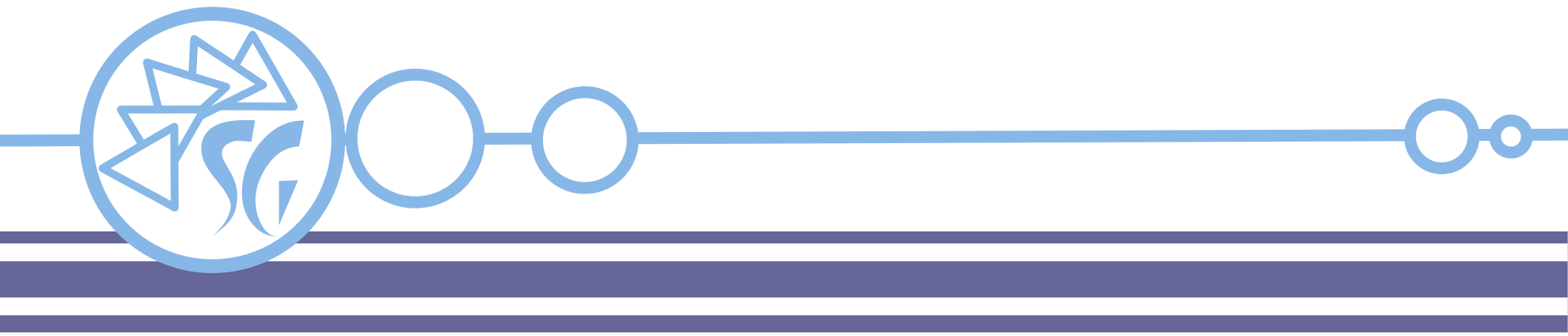

# PostgreSQL
# PL/pgSQL

**Ing. Simone Giustetti**

**www.giustetti.net**

# Estensioni Procedurali del Linguaggio SQL

PostgreSQL supporta svariati linguaggi procedurali per estendere le funzionalità di SQL:

- C / C++;
- PL/pgSQL;
- PL/Perl;
- PL/Python;
- PL/Tcl;
- ...

PostreSQL non interpreta direttamente i comandi, li passa invece ad un motore esterno.

# Installare un Linguaggio Procedurale

I linguaggi procedurali devono essere **“installati”** esplicitamente **in ogni database** che li userà.

Il comando **CREATE EXTENSION** installa uno dei linguaggi supportati:

```
CREATE EXTENSION <linguaggio>;

CREATE EXTENSION plpgsql;
```

# Installare un Linguaggio in Ogni DataBase

Creare un database battezzato **template1** ed installarvi il linguaggio desiderato.

Template1 funge da modello per popolare automaticamente ogni database creato successivamente.

I linguaggi procedurali installati in template1 sono duplicati in ogni database assieme agli altri oggetti ivi contenuti.

# Installare un Linguaggio Non Nativo

1) Copiare la libreria per l'interprete del linguaggio nella directory dei moduli.
2) Ripetere l'operazione se fossero installate più versioni di PostgreSQL.
3) Dichiarare la funzione di interfaccia:

```
CREATE FUNCTION <handler_function_name>()
RETURNS language_handler
AS '<percorso della libreria>'
LANGUAGE C;
```

# Installare un Linguaggio Non Nativo

4) Dichiarare una funzione di gestione “inline”.

```
CREATE FUNCTION <inline_function_name>( internal )
RETURNS void
AS '<percorso della libreria>'
LANGUAGE C STRICT;
```

5) Dichiarare una funzione di validazione:

```
CREATE FUNCTION <validator_function_name>( oid )
RETURNS void
AS '<percorso della libreria>'
LANGUAGE C STRICT;
```

# Installare un Linguaggio Non Nativo

6) Dichiarare il linguaggio procedurale.

```
CREATE [ TRUSTED ] LANGUAGE <linguaggio>
HANDLER <handler_function_name>
[ INLINE <inline_function_name> ]
[ VALIDATOR <validator_function_name> ];
```

L'istruzione **TRUSTED** deve essere impostata per ogni linguaggio non installato per scopi amministrativi.

# PL/pgSQL

Consente di raggruppare sequenze di istruzioni e salvarle sul server.

Riduce il numero di interazioni tra client e server con una conseguente riduzione della banda usata ed un guadagno prestazionale.

Sposta parte della logica dell’applicazione dentro al RDBMS complicandone la manutenzione e limitandone la scalabilità.

# Caratteristiche del Linguaggio PL/pgSQL

È un **linguaggio interpretato**, eseguito al volo dall'interprete, che deve essere presente sulla macchina.

È un **linguaggio imperativo**: Il programmatore deve scrivere codice dettagliato per tutte le operazioni eseguite dall'interprete.

È un **linguaggio specifico**: Non consente di sviluppare programmi / applicazioni generiche.

# Caratteristiche del Linguaggio PL/pgSQL

Supporta alcuni paradigmi propri della **programmazione ad oggetti**.

È un linguaggio a **tipizzazione debole**. La tipologia delle variabili deve essere definita durante la dichiarazione, ma il motore può eseguire conversioni implicite.

# Struttura del Linguaggio PL/pgSQL

Ogni segmento di codice è progettato per esistere nella forma di una funzione / blocco di codice.

Tutte le variabili devono essere **dichiarate prima del loro uso**.

Tutti i segmenti di codice accettano parametri durante la chiamata e possono rendere valori di ritorno alla conclusione.

# Sintassi del Linguaggio PL/pgSQL

Non distingue tra maiuscole e minuscole (**Case Insensitive**).

Lettere maiuscole e minuscole possono essere combinate nella dichiarazione delle variabili o delle funzioni.

Usa il doppio apostrofo come carattere di escape per l'apostrofo. Necessario perché le funzioni sono salvate in formato stringa nei database.

# Blocchi di Codice

Il codice è organizzato sotto forma di blocchi.

Un blocco di codice è definito attraverso l'istruzione **CREATE FUNCTION**.

La prima parte dell'istruzione **CREATE FUNCTION** è riservata per le dichiarazioni delle variabili.

Ogni dichiarazione deve essere terminata con un carattere ";" (Punto e virgola).

# Blocchi di Codice

Il corpo di un blocco di codice è delimitato dalle istruzioni **BEGIN** … **END**.

Il corpo di un blocco di codice deve essere posto dopo la sezione dichiarativa.

Ogni istruzione deve essere terminata con un carattere “;” (Punto e virgola).

È possibile inserire un blocco di codice all’interno di un altro blocco.

# Blocchi di Codice

Struttura di un segmento di codice PL/pgSQL:

```
CREATE FUNCTION <nome>( <argomento>, ... )
RETURNS <tipo> AS [ ' | $$ ]
    DECLARE
       <dichiarazione>;
       ...
    BEGIN
       <istruzione>;
       ...
    END; [ ' | $$ ]
LANGUAGE 'plsql';
```

# I Commenti

Esistono due tipologie di commenti in PL/pgSQL:

- I commenti di una singola riga

Ogni riga che incomincia con due meno.

```
Codice...  -- Commento
-- Commento
```

- I commenti inclusi in un blocco di testo

```
Codice...  /* Commento
Commento
Commento */  Codice...
```

# Le Variabili

Una variabile è una porzione di memoria in cui vengono salvate informazioni per la durata in esecuzione del codice.

Le variabili devono essere dichiarate prima dello uso.

```
DECLARE
   <variabile> <tipo> [ := <valore> ];
   quantity            integer := 80;
   2nd_quantity        integer DEFAULT 40;
   total               int4;  -- NULL
   name       NOT NULL varchar(20) DEFAULT '';
```

# Visibilità (Scope) delle Variabili

Ogni variabile è visibile / modificabile **nel blocco** di codice in cui è definita ed in tutti i relativi **sotto-blocchi**.

È possibile dichiarare variabili locali in un sotto-blocco. Le variabili sono distrutte quando il sotto-blocco finisce e **non sono disponibili per l'uso nel blocco padre**.

# Le Costanti

Una costante è una porzione di memoria in cui vengono salvate informazioni che non possono essere successivamente modificate.

La dichiarazione di una costante deve includere l'istruzione **CONSTANT**.

```
DECLARE
   quantity CONSTANT integer := 80;
```

# Gli Array

Un array è una variabile che contiene molti dati dello stesso tipo.

Per accedere ad un elemento di un array è necessario il ricorso alle parentesi quadre.

```
ar_01[ 3 ], ar_01[ 25 ], ...
```

Le componenti numeriche partono da **1**.

PostgreSQL supporta gli array multidimensionali.

# Inizializzare un Array

Per assegnare valori ad un array si può ricorrere alle parentesi graffe, oppure al costruttore **ARRAY[ ]**.

```
SELECT '{ 9, 10, 11, 12 }'::integer[];
SELECT '{{ 1, 2 }, { 3, 4 }}'::integer[];

ar_01 := ARRAY[ 1, 2, 3, 4, 5 ];
ar_02 := ARRAY[ ARRAY[1,2,3],
                ARRAY[4,5,6],
                ARRAY[7,8,9],
                ARRAY[10,11,12]];
```

# Cardinalità di un Array

La dimensione di un array è resa dalla funzione **cardinality()**.

```
cardinality( <array )
```

PostgreSQL **non controlla** i limiti di un array.

Quando si sforano i limiti di un array con lo operatore “[ ]” (Parentesi quadre) viene reso il valore **NULL**.

# Tipologie di Dato in SQL

Esistono 4 tipologie principali di dati:

- Testo
- Numeri
- Date / Orari
- Large Binary Objects

Ogni RDBMS implementa uno o più sotto-tipi

- Varia l'intervallo dei valori
- Varia la formattazione su supporto fisico
- Varia lo spazio occupato

# Tipologie di Dato Supportate

| SQL - DATO NUMERICO INTERO | | | |
|---|---|---|---|
| **Tipologia** | **Alias** | **Byte** | **Valori Ammessi** |
| **BIGINT** | int8 | 8 | [ -9223372036854775808; 9223372036854775807 ] |
| **BIGSERIAL** | serial8 | 8 | [ 1; 9223372036854775807 ] |
| **INTEGER** | int4 | 4 | [ -2147483648; 2147483647 ] |
| **SERIAL** | serial4 | 4 | [ 1; 2147483647 ] |
| **SMALLINT** | int2 | 2 | [ -32768; 32767 ] |
| **SMALLSERIAL** | serial2 | 2 | [ 1; 32767 ] |

| SQL - DATO LOGICO | | | |
|---|---|---|---|
| **Tipologia** | **Alias** | **Byte** | **Valori Ammessi** |
| **BOOLEAN** | bool | 1 | [ False; True ] |

# Tipologie di Dato Supportate

| SQL - DATO NUMERICO ESATTO | | | |
|---|---|---|---|
| **Tipologia** | **Alias** | **Byte** | **Valori Ammessi** |
| **NUMERIC** | **DECIMAL** | Variabile | 131072 cifre prima della virgola. 16383 cifre dopo la virgola |

| SQL - DATO NUMERICO A VIRGOLA MOBILE | | | |
|---|---|---|---|
| **Tipologia** | **Alias** | **Byte** | **Valori Ammessi** |
| **REAL** | **FLOAT(24)** | 4 | 6 decimali di precisione |
| **DOUBLE PRECISION** | **FLOAT(53)** | 8 | 15 decimali di precisione |

# Tipologie di Dato Supportate

| SQL - VALUTA | | | |
|---|---|---|---|
| **Tipologia** | **Alias** | **Byte** | **Valori Ammessi** |
| **MONEY** | | 8 | [ -9223372036854758.08; 9223372036854758.07 ]<br>Il numero di decimali dipende dalle impostazioni di sistema / dalla localizzazione |

| SQL - DATO NUMERICO BINARIO | | | |
|---|---|---|---|
| **Tipologia** | **Alias** | **Byte** | **Valori Ammessi** |
| **BIT** | | Variabile | Sequenza di [ 0; 1 ]. Lunghezza fissa. |
| **BIT VARYING** | | Variabile | Sequenza di [ 0; 1 ]. Lunghezza variabile. |

# Tipologie di Dato Supportate

| SQL - STRINGHE | | | |
|---|---|---|---|
| **Tipologia** | **Alias** | **Byte** | **Valori Ammessi** |
| **CHAR** | **BPCHAR CHARACTER** | Variabile | Testo di lunghezza fissa. |
| **TEXT** | | Infinita | Testo. |
| **VARCHAR** | **CHARACTER VARYING** | Variabile | Testo di lunghezza variabile. |

| SQL – STRINGHE BINARIE | | | |
|---|---|---|---|
| **Tipologia** | **Alias** | **Byte** | **Valori Ammessi** |
| **BYTEA** | | Lunghezza del campo + 1 o 4 byte | Sequenza di caratteri esadecimali di lunghezza variabile. |

# Tipologie di Dato Supportate

| SQL - DATE E ORARI | | | |
|---|---|---|---|
| **Tipologia** | **Alias** | **Byte** | **Valori Ammessi** |
| **DATE** | | 4 | [ 4713 BC; 5874897 AD ]<br>Livello di precisione = 1 giorno. |
| **TIME** | | 8 | [ 00:00:00; 24:00:00 ]<br>Livello di precisione = 1 microsecondo. |
| **TIME WITH TIME ZONE** | | 12 | [ 00:00:00 + 1559; 24:00:00 - 1559 ]<br>Livello di precisione = 1 microsecondo. |
| **TIMESTAMP** | | 8 | [ 4713 BC; 5874897 AD ]<br>Livello di precisione = 1 microsecondo. |
| **TIMESTAMP WITH TIME ZONE** | | 8 | [ 4713 BC; 5874897 AD ]<br>Livello di precisione = 1 microsecondo. |
| **INTERVAL** | | 16 | [ -178000000; 178000000 ] Anni<br>Livello di precisione = 1 microsecondo. |

# Tipologie di Dato Supportate

| SQL – DATO GEOMETRICO | | | |
|---|---|---|---|
| **Tipologia** | **Alias** | **Byte** | **Valori Ammessi** |
| **POINT** | | 16 | ( x, y ) |
| **LINE** | | 32 | { A, B , C } |
| **LSEG** | | 32 | (( x1, y1 ), ( x2, y2 )) |
| **BOX** | | 32 | (( x1, y1 ), ( x2, y2 )) |
| **PATH OPEN** | | 16 + 16n | (( x1, y1 ), … ) |
| **PATH CLOSED** | | 16 + 16n | (( x1, y1 ), … ) |
| **POLIGON** | | 40 + 16n | (( x1, y1 ), … ) |
| **CIRCLE** | | 24 | (( x1, y1 ), r ) |

# Tipologie di Dato Supportate

| SQL – INDIRIZZO DI RETE | | | |
|---|---|---|---|
| **Tipologia** | **Alias** | **Byte** | **Valori Ammessi** |
| **CIDR** | | 7 o 19 | Reti IPv4 o IPv6 |
| **INET** | | 7 o 19 | Host + Rete IPv4 o IPv6 |
| **MACADDR** | | 6 | Mac Address |
| **MACADDR8** | | 8 | Mac Address (Formato EUI-64) |

# Cambiare il Tipo di un Dato

La tipologia di una variabile può essere impostata / forzata utilizzando:

- La funzione **CAST()**;
- L'operatore **::**.

```
CAST( <espressione> AS <tipo dato> );
CAST( '999' AS int4 );

<espressione>::<tipo dato>
'2005-11-30'::date;
```

# Copiare il Tipo di un Dato

Per salvare un dato letto da una tabella in una variabile bisogna conoscere il tipo.

L'istruzione **%TYPE** legge la tipologia di un dato che può essere usata in una dichiarazione.

```
<variabile> <tabella>.<colonna>%TYPE;
```

```
DECLARE
   user_id user.user_id%TYPE;
    ...
```

# Copiare il “Tipo” di una Riga

Esistono variabili composite in cui è possibile salvare un’intera riga letta da una tabella.

L’istruzione **%ROWTYPE** legge la tipologia delle colonne di una tabella e la assegna ad una variabile.

```
<variabile> <tabella>%ROWTYPE;

DECLARE
   riga_user user%ROWTYPE;
   ...
```

# Accedere alle Componenti di una Variabile

Per accedere alle componenti di una variabile si ricorre al carattere “.” (Punto).

```
<variabile>.<colonna>
```

Le variabili composite possono essere usate come argomenti di una funzione. Il parametro corrispondente sarà una variabile di tipologia **%ROWTYPE** e le componenti potranno essere accedute usando il carattere “.” (Punto).

```
$n.<colonna>
```

# Le Variabili di Tipologia RECORD

La tipologia **RECORD** consente di salvare una riga resa da un'istruzione SQL in una variabile.

È simile a **%ROWTYPE**, ma non ha una struttura predefinita. É un "segnaposto": la struttura viene impostata solo durante la prima assegnazione.

```
DECLARE
   riga RECORD;
BEGIN
   INSERT INTO riga SELECT ...;
END;
```

# Le Variabili di Tipologia RECORD

Le componenti non esistono prima che sia effettuata l'assegnazione. **Ogni tentativo di accesso genera un errore**.

La struttura e le tipologie di dato sono dinamiche, variano ad ogni nuova assegnazione.

Le funzioni possono rendere un dato di tipologia **RECORD**. Struttura e tipologia sono determinati nel corpo della funzione.

# Assegnare Dati ad una Variabile

Per assegnare il risultato di una istruzione SQL ad una variabile si usa l'istruzione **INTO**.

Ogni comando SQL che renda una singola riga può essere usato per popolare una variabile.

```
DECLARE
   riga RECORD;
   ...
BEGIN
   SELECT ... INTO riga FROM ...;
   INSERT INTO riga SELECT ...;
END;
```

# Assegnare Dati ad una Variabile

```
DECLARE
   riga RECORD;
   ...
BEGIN
   SELECT * INTO riga FROM impiegato
      WHERE cognome = 'braschi';
   ...
   SELECT * INTO STRICT riga FROM impiegato
      WHERE cognome = 'giusti';
END;
```

Quando si specifica il vincolo **STRICT**, se la query rendesse un numero di righe **diverso da 1** verrebbe segnalato un errore.

# Assegnare Dati ad una Variabile

Quando **non** si specifica il vincolo **STRICT**, alla variabile verrebbe assegnata la prima riga resa e le successive verrebbero scartate.

“Prima riga” è un concetto aleatorio a meno di specificare una istruzione **ORDER BY**.

Quando una query non ritorna righe, la variabile viene popolata di **NULL**.

# Gestire Query Che Non Rendono Righe

La variabile di sistema **found** può essere interrogata per capire se l'ultima query eseguita abbia trovato dati o meno.

```
...
SELECT * INTO riga FROM impiegato
   WHERE cognome = 'braschi';
IF NOT found THEN
...
```

# Assegnazione Multipla

Molte variabili possono essere assegnate con una unica istruzione SQL.

L’elenco delle variabili usa la virgola (“,”) come separatore.

Il numero di variabili deve coincidere con quello delle colonne rese da una query.

```
SELECT nome, cognome INTO var_nome, var_cognome
   FROM impiegato WHERE cognome = 'braschi';
```

# Argomenti di Funzione

La dichiarazione di una funzione può includere la definizione di una lista di argomenti passati alla funzione stessa.

Nella dichiarazione è obbligatorio impostare la tipologia di ogni argomento. Opzionalmente può essere definito un alias.

Gli argomenti possono essere acceduti attraverso l'operatore **$** (Dollaro) e la rispettiva posizione nella dichiarazione.

# Argomenti di Funzione

```
CREATE FUNCTION somma( int, int ) RETURNS int AS '
    DECLARE
       add01 integer := $1;
       add02 ALIAS FOR $2;
    BEGIN
       RETURN add01 + add02;
    END;'
LANGUAGE 'plsql';
```

La funzione può essere chiamata attraverso una istruzione **SELECT**:

```
SELECT somma( 5, 19 );
```

# Argomenti di Output

Gli argomenti possono essere modificati nel corpo di una funzione. Gli argomenti modificabili sono identificati dall'istruzione **OUT**.

```
CREATE FUNCTION tasse( totale real, OUT tassa real )
AS '
   BEGIN
      tassa := totale * 0.22;
   END;'
LANGUAGE 'plsql';
```

Le istruzioni RETURNS e RETURN sono ridondanti e possono essere omesse.

# Overloading di Funzioni

PostgreSQL supporta la possibilità di assegnare il medesimo nome a più funzioni.

L’elenco degli argomenti delle funzioni **deve differire per numero, tipologia e/o posizione**.

Si tratta di una funzionalità controversa perché si lega male alle caratteristiche di un linguaggio tipizzato debole.

# Parametri Opzionali

È possibile definire un valore predefinito per i parametri di una funzione. Per assegnare il valore predefinito si usa l'istruzione **DEFAULT**.

L'elenco dei parametri opzionali deve essere posizionato dopo l'elenco di quelli standard.

```
CREATE FUNCTION <funzione>(
   <parametro> <tipo>, ...,
   <parametro> <tipo> DEFAULT <valore>, ...,
AS '
   ...
```

# Chiamata con Parametri Opzionali

I parametri per cui è definito un valore predefinito **possono essere omessi** dalla chiamata di funzione.

```
CREATE FUNCTION somma( int, int, int DEFAULT 0 ) ...

-- Usa il valore predefinito per il terzo parametro
SELECT somma( 5, 10 );
-- Usa un valore esplicito per il terzo parametro
SELECT somma( 5, 10, 31 );
```

# Parametri Nominali

I parametri di una funzione per cui sia definito un alias possono essere passati per nome anziché per posizione.

Si usa l'operatore di assegnazione **=>** (Freccia).

Non è necessario rispettare l'ordine della definizione, solo il numero totale.

```
CREATE FUNCTION somma( add01 int, add02 int ) ...

SELECT somma( add01 => 5, add02 => 10 );
SELECT somma( add02 => 10, add01 => 5 );
```

# Istruzione IF / THEN / ELSE

Un blocco di codice è eseguito solo quando è verificata una condizione.

```
IF <condizione> THEN
   <Blocco 1 di istruzioni>
   ...
ELSE
   <Blocco 2 di istruzioni>
   ...
END IF;
```

Il blocco **ELSE** è opzionale.

# Istruzione ELSIF

È possibile verificare molte condizioni in cascata annidando i cicli **IF**, oppure usare il costrutto **ELSIF** (Oppure il sinonimo **ELSEIF**).

```
IF <condizione> THEN
   <Blocco 1 di istruzioni>
   ...
ELSIF <condizione> THEN
   <Blocco 2 di istruzioni>
   ...
ELSE
   <Blocco conclusivo di istruzioni>
   ...
END IF;
```

# Istruzione CASE

Costrutto condizionale che permette di verificare tutti i valori resi da una espressione.

```
CASE <espressione>
   WHEN <valore 1> THEN
      <Blocco 1 di istruzioni>
      ...
   WHEN <valore  2>, <valore 3> THEN
      <Blocco 2 di istruzioni>
      ...
   ELSE
      <Blocco conclusivo di istruzioni>
      ...
END CASE;
```

# Istruzione Searched CASE

Garantisce l'esecuzione condizionale di codice, basata su espressioni booleane.

```
CASE
   WHEN <espressione 1> THEN
      <Blocco 1 di istruzioni>
      ...
   WHEN <espressione 2> THEN
      <Blocco 2 di istruzioni>
      ...
   ELSE
      <Blocco conclusivo di istruzioni>
      ...
END CASE;
```

# Cicli

È possibile far ripetere un blocco di codice più volte all'interno di una funzione ricorrendo alle istruzioni:

- **CONTINUE**;
- **EXIT**;
- **FOR**;
- **FOREACH**;
- **LOOP**;
- **WHILE**.

# Ciclo LOOP

Esegue un blocco di codice un numero infinito di volte fino a che non è terminato con **EXIT** o **RETURN**.

```
[ << <label> >> ]
LOOP
   <Blocco di istruzioni>
   ...
END LOOP [ <label> ];
```

L'identificatore opzionale <label> consente di identificare un ciclo quando ne esistono di molteplici annidati.

# Istruzione EXIT

Consente di uscire da un ciclo. Il controllo viene passato all'istruzione successiva ad **END LOOP**. Può essere usata con qualsiasi ciclo.

```
EXIT [ <label> ] [ WHEN <espressione boolenana> ];

LOOP
   <somme>
   ...
   IF count > 1 THEN
      EXIT;  -- Exit loop
   END IF;
END LOOP;
```

# Istruzione CONTINUE

Interrompe l'iterazione in corso e riprende il ciclo dall'iterazione successiva. Consente di saltare iterazioni.

```
CONTINUE [ <label> ] [ WHEN <espressione boolenana> ];

LOOP
   <somme>
   ...
      EXIT WHEN count > 100;
      CONTINUE  WHEN count < 50;
   ...
   <altre istruzioni>
END LOOP;
```

# Ciclo WHILE

Esegue un blocco di codice un numero molteplice di volte in base alla verifica di una condizione.

```
[ << <label> >> ]
WHILE <espressione boolenana> LOOP
   <Blocco di istruzioni>;
END LOOP [ <label> ];
```

È possibile che il blocco non sia mai eseguito

È possibile che il blocco sia eseguito all'infinito, se la condizione di stop non fosse mai verificata.

# Ciclo FOR

I cicli **FOR** usano un contatore incrementato ad ogni iterazione.

```
[ << <label> >> ]
FOR <variabile> IN <inizio> .. <fine> BY <incremento>
LOOP
   <Blocco di istruzioni>;
END LOOP [ <label> ];
```

Viene eseguito sempre un numero determinato di volte.

<variabile> è implicitamente di tipo intero.

# Ciclo FOR con Decremento

Specificando l'istruzione **REVERSE**, il contatore viene decrementato ad ogni iterazione.

```
FOR counter IN 1..10 BY 1 LOOP
   <Blocco di istruzioni>;
END LOOP;

FOR counter IN REVERSE 10..1 BY 2 LOOP
   <Blocco di istruzioni>;
   -- counter assume i valori 10, 8, 6, 4, 2
END LOOP;
```

Se il limite inferiore fosse maggiore di quello superiore il ciclo non viene mai eseguito.

# Leggere i Risultati di una Query

Un ciclo **FOR** può essere usato per leggere tutte le righe rese da una query SQL.

```
[ << <label> >> ]
FOR <variabili> IN <query> LOOP
   <Blocco di istruzioni>;
END LOOP [ <label> ];
```

<variabili> può indicare:

- Una variabile **RECORD**;
- Una variabile di tipo **%ROWTYPE**;
- Un elenco di variabili separate da virgola.

# Leggere i Risultati di una Query

Un altro modo per iterare tra le righe è la combinazione **FOR / EXECUTE**.

```
[ << <label> >> ]
FOR <variabili> IN EXECUTE <testo> LOOP
   <Blocco di istruzioni>;
END LOOP [ <label> ];
```

<testo> è una query di selezione espressa in formato stringa. La query può essere scritta dinamicamente nel corpo della funzione stessa. Le query dinamiche sono **meno prestazionali** di quelle statiche.

# Ciclo FOREACH

**FOREACH** funziona come un ciclo **FOR** per iterare attraverso le componenti di un array.

```
[ << <label> >> ]
FOREACH <variabile> [ SLICE <numero> ] IN ARRAY
<espressione> LOOP
   <Blocco di istruzioni>;
END LOOP [ <label> ];
```

<espressione>: istruzioni che rendono un array.
Quando **SLICE** = 0, viene letto l'intero array.
<Blocco di istruzioni> è eseguito per ogni componente dello array.

# I Cursori

Un cursore è una variabile che consente di gestire i risultati di una query una riga o un gruppo di righe per volta.

I cursori sono tutti del tipo speciale **refcursor**.

Esiste una sintassi specifica per dichiararli

```
<variabile> [[ NO ] SCROLL ] CURSOR [ <argomento>,
   ... ] FOR <query>;
```

# I Cursori

L’istruzione **SCROLL** consente ad un cursore di spostarsi indietro. Se non specificata, la direzione del cursore dipende dalla query.

La lista di argomenti è un elenco separato da “,” (Virgola) di coppie variabile / tipo che vengono assegnate ai parametri della query. La assegnazione è eseguita all’apertura del cursore.

# I Cursori

```
DECLARE
   curs01 refcursor;
   curs02 CURSOR FOR SELECT * FROM presenze;
   curs03 CURSOR (mese_in integer) FOR
      SELECT * FROM presenze WHERE mese = mese_in;
```

Tutte e tre le variabili sono di tipo **refcursor**. La prima, generica, può essere utilizzata per qualsiasi query. La query delle ultime due è inclusa nella definizione e non può variare.

# Aprire un Cursore

Ogni cursore deve essere aperto prima di poter estrarre righe dal database.

Esistono 3 modi per aprire un cursore.

```
OPEN <cursore> [[ NO ] SCROLL ] FOR <query>;

OPEN <cursore> [[ NO ] SCROLL ] FOR EXECUTE <query>
   [ USING <espressione> [, ... ]];
```

La query può essere passata al comando **EXECUTE** attraverso la funzione **format**.

# Aprire un Cursore

```
OPEN cur01 SCROLL FOR EXECUTE
   format( 'SELECT * FROM presenze' );
```

La funzione **format** consente di passare parametri alla stringa di testo rendendo la query dinamica.

```
OPEN <cursore> [( <argomento> := <valore> [, ... ] )];
```

I parametri possono essere passati per nome oppure per posizione.

```
OPEN cur02;
OPEN cur03( 42, 36 );
OPEN cur03( interno := 36, chiave := 42 );
```

# Usare un Cursore

Un cursore aperto può essere manipolato utilizzando appositi comandi:

- **FETCH**: Legge la prima riga disponibile di una query salvandola in una variabile;
- **UPDATE** / **DELETE**: Modifica / Cancella la riga puntata dal cursore;
- **MOVE**: Riposizione il cursore;
- **CLOSE**: Chiude un cursore;

# FETCH

Legge la riga “successiva” di una query e la salva in una variabile, un record oppure un elenco di variabili.

```
FETCH [ <direzione> { FROM | IN } ] <cursore>
   INTO <variabile>;
```

È possibile interrogare la variabile di sistema **found** per accertarsi che una riga sia stata letta.

Se non esistesse una riga successiva, rende una sequenza di **NULL**.

# FETCH

<direzione> può assumere un valore tra:

- FIRST;
- LAST;
- PRIOR;
- NEXT;
- ABSOLUTE <int>;
- RELATIVE <int>;
- BACKWARD <int>;
- FORWARD <int>.

# FETCH - Esempi

```
-- Standard fetch
FETCH curs01 INTO var_riga;
-- Retrieve 3 variables in one go
FETCH curs02 INTO var_01, var_02, var_03;
-- Retrieve last row into 2 variables
FETCH LAST FROM curs03 INTO var_01, var_02;
-- Move to 2nd previous row and retrive 1 value
FETCH RELATIVE -2 FROM curs04 INTO var_01;
```

# MOVE

Sposta il cursore in avanti oppure indietro. Funziona come **FETCH**, senza salvare quanto letto in una variabile.

```
MOVE [ <direzione> { FROM | IN } ] <cursore>;
```

<direzione> assume gli stessi valori usati per **FETCH**.

È possibile interrogare la variabile di sistema **found** per accertarsi che una riga sia stata letta.

# MOVE - Esempi

```
-- Standard move forward 1 row
MOVE curs01;
-- Move to last row
MOVE LAST FROM curs02;
-- Move to 2nd previous row
MOVE RELATIVE -2 FROM curs03;
-- Move forward 2 rows
MOVE FORWARD 2 FROM curs04;
```

# DELETE / UPDATE

Quando è posizionato su una riga, un cursore può essere usato per modificare o cancellare tale riga.

Non funziona con alcune query, ad esempio quelle contenti istruzioni **GROUP BY**.

```
UPDATE <tabella> SET <assegnazione>
   WHERE CURRENT OF <cursore>;
DELETE FROM <tabella> WHERE CURRENT OF <cursore>;

UPDATE anagrafica_personale SET nome = 'Simone'  WHERE
CURRENT OF cur01;
```

# CLOSE

Chiude un cursore.

Ha lo scopo di liberare risorse prima della fine di una transazione oppure di assegnare il cursore ad altra query.

Tutti i cursori sono chiusi automaticamente al termine di una transazione.

```
CLOSE <cursore>;
```

# Iterare per i Dati Attraverso un Cursore

Esiste una variante del ciclo **for** che consente di iterare le righe collegate ad un cursore.

```
[ << <label> >> ]
FOR <variabile> IN <cursore> [ <argomento> :=
   <valore>, ... ] LOOP
   <Blocco di istruzioni>;
END LOOP [ <label> ];
```

Il cursore deve essere associato ad una query durante la dichiarazione.

# Iterare per i Dati Attraverso un Cursore

Il ciclo **FOR** provvede automaticamente ad aprire e chiudere il cursore.

La variabile è automaticamente dichiarata di tipo **record** e può esistere solo all'interno del ciclo.

Ogni riga letta viene automaticamente passata alla variabile.

# Cursori come Argomenti di Funzione

Uno o più cursori possono essere passati come argomenti ad una funzione.

I cursori possono subire modifiche nel corpo di una funzione.

Una funzione può rendere un cursore come valore di ritorno.

Consente di passare molte righe tra funzioni distinte.

# Messaggi di Output

É utile produrre messaggistica per conoscere lo stato di una funzione ed eseguirne il debug.
L'istruzione **RAISE NOTICE** stampa messaggi a video.

```
RAISE NOTICE '<messaggio>';
RAISE NOTICE 'var = %', <variabile>;
```

# Gestione degli Errori

Ogni errore causa la terminazione immediata di una funzione e delle transazioni corrispondenti.

È possibile gestire gli errori per riprendere l'esecuzione del codice evitando che termini incondizionatamente.

Per abilitare le eccezioni in un blocco di codice si usa un sotto-blocco **EXCEPTION**.

Le eccezioni possono essere annidate.

# Dichiarazione delle Eccezioni

```
CREATE FUNCTION <nome>( <argomento>, ... )
RETURNS <tipo> AS [ ' | $$ ]
    DECLARE
       <dichiarazione>;
       ...
    BEGIN
       <istruzione>;
       ...
    EXCEPTION
       WHEN <condizione> [ OR <condizione> ] THEN
          <istruzioni di gestione>;
    ...
    END [ ' | $$ ];
LANGUAGE 'plsql';
```

# Condizioni

<condizione> deve corrispondere ad uno degli errori riconosciuti dal motore. Es:
division_by_zero

L'elenco completo delle eccezioni è reperibile nell'appendice A del manuale di PostgreSQL.

La condizione **OTHERS** gestisce ogni errore non in appendice.

# Gestione dei Dati Modificati

Il codice di gestione di una eccezione:

- Riconosce tutte le variabili del blocco di codice. Le variabili mantengono il valore che avevano prima dell'errore.
- Esegue automaticamente il rollback delle modifiche ai dati apportate da istruzioni incluse nel blocco di codice.
- Non tocca le modifiche ai dati apportate da istruzioni esterne al proprio blocco di codice.

# Impatto sulle Prestazioni

I blocchi di codice che contengono la definizione di eccezioni sono **molto più onerosi** dal punto di vista prestazionale di quelli che no.

Non usare le eccezioni quando non sussiste la necessità.

# Le Transazioni

Le transazioni legano molte operazioni in una atomica, che può essere confermata o revocata in toto.

Garantiscono la consistenza dei dati. In caso di errori scartano tutte le modifiche intermedie.

Per gestire una transazione bisogna includere tutti i sotto-comandi in un blocco **BEGIN** … **COMMIT** / **ROLLBACK**.

# Le Transazioni

```
BEGIN
   <istruzione>;
   ...
COMMIT;

BEGIN
   <istruzione>;
   ...
ROLLBACK;
```

Tutte le istruzioni SQL standard sono gestite attraverso transazioni implicite.

# Punti di Salvataggio

Per consentire una maggiore granularità, PostgreSQL consente di definire punti di salvataggio a cui eseguire un rollback.

```
SAVEPOINT <savepoint_id>;
```

Per tornare ad un salvataggio si usa l'istruzione **ROLLBACK TO** <savepoint_id>.

I savepoint definiti non sono rimossi dopo un rollback, per cui possono essere utilizzati anche più di una volta.

# Le Transazioni in PL/pgSQL

Le **funzioni** non supportano le transazioni. Non è possibile incapsulare una transazione nel corpo di una funzione.

Le **procedure** supportano le transazioni. Le transazioni possono essere incapsulate nel corpo di una procedura.

Le procedure sono state introdotte in PostgreSQL 11.

# Struttura di una Stored Procedure

```
CREATE PROCEDURE <nome>( <argomento>, ... )
AS [ ' | $$ ]
    DECLARE
       <dichiarazione>;
       ...
    BEGIN
       <istruzione>;
       ...
    END; [ ' | $$ ]
LANGUAGE 'plsql';
```

Le stored procedure **non ritornano** mai un valore.

L'istruzione **return** causa un errrore.

# Chiamare una Stored Procedure

Le stored procedure devono essere chiamate usando l'istruzione **call**.

```
CALL <procedura>( <argomento>, ... );

CALL trasferimento( 1, 2, 1000 );
```

# Informazioni & Licenze