Advanced Oracle PL/SQL - Tips

PL/SQL

PL/SQL offre una estensione procedurale all'SQL . E' un linguaggio procedurale strettamente legato a Oracle. Vantaggi: - sviluppo modularizzato programmi - integrazione con strumenti oracle (strumenti per costruizione applicazioni, report, ecc..) - portabilità (a livello di server, deve sempre girare su oracle) - gestione delle eccezioni.

Struttura blocchi PL\SQL:

anonomi, procedure, funzione I blocchi anonimi sono controllati (sintassi, referenze, ecc..) ogni volta, mentre i blocchi procedure e funzioni sono solamente eseguiti. In generale:

```
DECLARE
...
BEGIN
…
EXCEPTION
…
END;
```

Tipi di Variabili:

PL/SQL - Scalari (numeri, varchar, ..) - Composte (accoglie set di risultati: collections, array, matrici...) - Reference (puntatore) - Oggetti LOB (Large Object) Variabili non PL\SQL: - Bind variable. Se utilizzo uno strumento esterno per interagire con blocchi PL\SQL, queste variabili sono quelle dell'ambiente chiamante. (sintassi ":"nomeVariabile) Quindi il bind viene fatto solo al momento della eseguzione.

TIPS:

- Di default ogni variabile ha valore null: attenzione nei cicli per i contatori. - Nei booleani una variabile null rende nulle le condizioni a cui partecipa (e non false!).Le variabili boolean hanno 3 valori: TRUE, FALSE e NULL. - Le variabili di tipo INTERVAL lavorano sulle date con una conversione decisa esplicitamente,quindi posso esprimere intervalli temporali:

```
select sysdate+5 from dual --> restituisce la data tra 5 giorni. E' però una conversione implici

select sysdate+interval '01 04:25:15' day to second from dual --> restituisce la data con sommati
```

. - Per fare codice trasparente alla modifiche di tipo dati su tabelle, posso definire una variabile del tipo ricavato da una certa colonna:

```
v_ename employees.last_name%type
```

- Se si utilizzano due & la variabile dichiarata è mantenuta per la sessione, vale a dire si inizializza una variabile di ambiente per la sessione (si può usare per esempio se si utilizza più volte la stessa variabile e non la si può dichiarare). Per esempio:

```
select last_name, job_id, salary
from employee
 where job_id = (select job_id
          from employees
          where last_name = '&&last_name')
 and last_name <> '&last_name'
```

- E' bene evitare di dare ad una variabile lo stesso nome di una colonna. Quando c'è ambiguità tra colonna e var, Oracle intepreta l'etichetta come nome di colonna, mentre tra variabile e tabella, vince la variabile:

```
declare
```

```
 employee_id NUMBER
begin
 select employee_id
 into employee_id
 from employee;
```

-Una variabile può essere definita come dello stesso tipo di un'altra

```
balance NUMBER(7,2);
 min_balance balance%type:=1000;
```

-Blocchi nidificati: Comodo per la gestione delle eccezioni, che è unica per blocco. Si può così isolare internamente un blocco dove gestire direttamente l'eccezione, e continuare con l'esecuzione, come tanti try catch. Se nel blocco interno definisco una variabile con lo stesso nome di una già esistente, la nuova ha la priorità. Se si scrive outer. si può comunque riferire la variabile del blocco esterno.

Recupero e lettura dei dati

Cursore

Puntatore all'area di memoria riservata allocata dal server Oracle - Implicito: creato e gestito dal server per processare l'SQL Attributi: - sql%rowcount: numero di righe aggiornate dal cursore implicito nell'ultima istruzione - sql%found: se il cursore implicito ha trovato qualcosa - sql%notfound: se il cursore implicito non ha trovato qualcosa - Esplicito: dichiarato via codice

Costrutti

FOR

```
FOR <contatore implicito> in [REVERSE] in 1..10 LOOP
...
END LOOP;
```

Non c'è bisogno di dichiarare come variabile il contatore. Piu' comodo del LOOP, ma il contatore può avanzare solo di 1 per ogni ciclo.

IF

```
IF <cond>
THEN
...;
[
ELSIF <cond2>
THEN
...;
]
ELSE
...;
END IF;
```

Nota: se si usa un operatore > o < su di un valore nullo, si ottiene FALSE

CASE Si può incontrare sia come istruzione che come espressione

Istruzione

```
CASE <variabile>
[
WHEN <valore> THEN
<blocco di istruzioni>
]
END CASE;
```

Espressione

```
CASE <variabile>
[
WHEN <valore> THEN <risultato>;
]
[ ELSE <risultatoN> ]
END;
```

Tipi di dati composti

- RECORD - TABLE - NESTED TABLE - ARRAY - CURSORE ESPLICITO

RECORD Set di elementi tutti uguali, densi (no buchi vuoti) = varray. Deve essere dimensionata in fase di dichiarazione. I tipi contenuti nel record possono essere diversi e possono essere tipi composti a loro volta.

```
TYPE <type_name> IS RECORD (<nome_colonna1> <tipo_colonna1> [, <nome_colonnaN> <tipo_colonnaN>];
...
<nome_variabile> <type_name>;
...
SELECT <elenco di colonne> INTO <nome_variabile> FROM ... WHERE ... ;
dbms_output.put_line(<nome_variabile>.<nome_colonna1>);
```

Si può dichiarare una variabile composta come dello stesso tipo della riga di una tabella:

```
<nome_variabile> <nome_tabella>%rowtype;
```

TABLE Tabella (= matrice): index by table. Posso avere un elemento che fa da indice di tipo numerico (pls_integer, binary_integer) o alfanumerico (varchar2). Possono essere fino a 4 Giga: che però vanno a discapito della memoria. L'indice di tipo PLS_INTEGER è consigliato da Oracle, altrimenti si può usare VARCHAR2(N), però in quel caso non si potrà ciclare per indice (diventa tipo una hashmap).

```
TYPE <type_name> IS TABLE OF <nome_tabella>%rowtype INDEX BY PLS_INTEGER;
```

NESTED TABLE livello PL\SQL o livello storage. Non è possibile creare un indice custom. A livello di storage può servire per creare una tabella interna dentro a un campo di una altra tabella.

Valorizzazione della variabile table

I cicli di caricamento delle index by table possono essere molto onerosi in termini di memoria, è necessario pensare a metodi ottimizzati. Il punto delicato nelle table è il caricamento: per esempio non posso caricarle semplicemente con una select. Come fare? 1)sfruttare un ciclo 1.Svantaggi 1.l'indice degli elementi è quello della tabella da cui sono presi 2.ho una operazione sql (select) dentro un ciclo PL\SQL e quindi c'è un continuo switch fra i due moduli (SQL e PL/SQL) 3.Il for implica uno step sempre di 1 elemento 2)chiedo al motore sql di estrarre tutti i dati e metterli nella tabella e assegnare automaticamente un indice: costrutto bulk collect. (presente dalla versione 9) Questo costrutto è sintatticamente più semplice e ha prestazioni migliori.

La tabella può contenere anche elementi sparsi (index by table è indicizzata ma sparsa → indice on buchi), come leggerli? Non posso usare il ciclo for, ma dovrò sfruttare i metodi offerti e un ciclo while.

Se non si specifica l'idex by si ha comunque un tipo dato composto tabella: ho una nested table PL\SQL. Anche se non realtà non è innestata da nessuna parte. Si può popolare come fosse un array ma non ha una dimensione prestabilita (si può estendere con il metodo extend), ha anche questa un indice ma non è manipolabile.

Nel caso degli array il metodo extend esiste, ma sempre nel limite degli elementi settati in fase di dichiarazione.

```
FOR i IN 100..104 LOOP
```

```
 SELECT * into var_table(i)
 FROM employees
 WHERE employees_id=i;
END LOOP;
```

```
SELECT * BULK COLLECT INTO <var_table> ...
```

Ciclo sulla una variabile table

```
FOR i in var_table.first..var_table.last LOOP
...var_table(i);
END LOOP;
```

Per variabile table con indice varchar2:

```
indice varchar(<stesso valore dell'indice di var_table>);
...
indice:=<var_table>.first;
WHILE indice IS NOT NULL LOOP;
 ...<var_table>(indice);
 indice:= <var_table>.next(indice);
END LOOP;
```

NESTED TABLE

```
TYPE <nested_table> IS TABLE OF <rowtype>;
<var> <nested table>;
...
<var>:=<nested_table>(<elenco di valori di tipo rowtype...>);
FOR i IN <var>.first..<var>.last LOOP
 dbms_output.put_line(<var>(i));
END LOOP;
```

Nella nested table, l'indice first è sempre 1.

.extend(N); Aggiunge N posizioni vuote alla variabile. E' buona norma estendere prima di inserire dati nella variabile.

ARRAY

TYPE VARRAY() OF ; ; ... :=(); FOR i IN .first...count() LOOP ... END LOOP;

C'è un controllo sul numero di elementi (fisso) che si provano a mettere nell'array.

CURSORE ESPLICITO Un corsore esplicito viene dichiarato e gestito da programma, mantiene i dati recuperati in memoria ed è possibile scorrerli, dopo la lettura il corsore si può chiudere e la memoria viene liberata. Dato il costo cmq elevato, sul db esiste un parametro che limita il numero di cursori apribili.

5 fasi per l'utilizzo di un cursore: 1)declare 2)open: esecuzione effettiva della query e set del risultato nell'area di memoria del db 3)fetch: caricamento di una riga dentro una variabile 4)ciclo su ogni riga 5)chiusura

La select associata al cursore è SQL puro. In memoria ci finisce il risultato di questa select.

```
CURSOR <type> IS SELECT ... FROM ... WHERE ...;
<var> <type>%ROWTYPE;
...
OPEN <var>;
 LOOP
  FETCH <type> INTO <var>;
  EXIT WHEN <var>%notfound;
  dbms_output.put_line(<var>. ...);
```

```
 END LOOP;
CLOSE <var>;
```

Variabile, open, close, fetch implicite

```
FOR <var_implicita> IN <type> LOOP
dbms_output.put_line(<var_implicita>. ...);
END LOOP;
```

La versione con variabile e OPEN si usa quando si ha un cursore parametrico. E' possibile mettere la SELECT direttamente al posto di :

```
FOR <var_implicita> IN (SELECT ...) LOOP
 dbms_output.put_line(<var_implicita>. ...);
END LOOP;
```

In questo modo si perdono però gli attributi % del cursore.

Utilizzo di parametri nel cursore

```
CURSOR <type>(<nome_par> <tipo_par>) IS SELECT... WHERE <espressione che utilizza nome_par>;
...
FOR <var_implicita> IN <type>(<valore_parametro>) LOOP
...
END LOOP;
```

Se qualcuno modifica i dati dopo che li ho aperti col cursore, continuo a vedere quelli dell'apertura. Oppure li si potrebbe voler bloccare:

```
CURSOR <type> IS SELECT ... FOR UPDATE [OF <nome_colonna>] NOWAIT;
```

NOWAIT serve a dire di non attendere (e dare errore subito) se qualcun altro sta bloccando quei dati. Se la select è una JOIN, il lock avviene su tutti i record coinvolti nella query. Specificare OF fa bloccare solo la riga proprietaria di quella colonna.

Aggiornamento tramite cursore

```
FOR <var_implicita> IN <type>
LOOP
 ...
 UPDATE <table_originale>
 SET ...
 WHERE CURRENT OF <type>;
END LOOP;
```

Gestione Eccezioni

Gestione anomalie che si verificano durante l'eseguzione del programma. 3 modalità di gestione delle eccezioni: 1)Eccezioni predefinite a cui sono assegnati un numero e un nome logico di riferimento: possono essere catturati espressamente 2)Eccezioni non predefinite classificate solo in base al numero. Per catturarle posso definire delle variabili di tipo eccezione e associare il nome logico da me definito con l'eccezione vera a propria tramite il comando pragma exception_init. 3)Inoltre è possibile definire eccezioni custom che derivino dalla logica di business. Questi errori non sarebbero rilevati da oracle (per esempio aggiornamento su un record con id che non è presente nella tabella) ma sono situazioni che si ha cmq bisogno di segnalare: errori applicativi.

Quando viene rilevata una eccezione, il flusso di comandi è immediatamente interrotto e si viene rimandati, eventualmente, nel blocco exception che si occupa di gestire le eccezioni.

Con raise_application_error si possono lanciare eccezioni bloccanti su un rage di numeri a disposizione dello sviluppatore.

Note: se volessi fare un insert in una tabella di log dentro un blocco di gestione eccezioni e poi lanciare una

raise, il commit non sarebbe eseguito. La soluzione può essere fare una procedura apposta che gestisca autonomamente la commit usando pragma autonomous_transaction. Si ha una transazione secondaria dentro quella principale.

```
BEGIN
...
EXCEPTION
WHEN <label_eccezione>
 THEN dbms_output.put_line(sqlcode || sqlerrm);
 ...
END;
```

Le label di eccezioni sono standard, come ad esempio - NO_DATA_FOUND: unica eccezione non bloccante. Può essere lanciata solo da una SELECT, non da un UPDATE - OTHERS

Se si gestiscono molteplici eccezioni tra le quali c'è OTHERS, deve comunque andare in coda.

Una EXCEPTION generata ma non bloccata da un blocco interno, può comunque essere bloccata da uno più esterno. Il problema di queste eccezioni è che non si sa da dove arrivino. Si può risolvere con il comando RAISE_APPLICATION_ERROR, che permette di gestire eccezioni personalizzate.

```
EXCEPTION
WHEN <label_eccezione>
THEN RAISE_APPLICATION_ERROR(<codice_numerico>, <messaggio>);
```

Il codice numerico è inventato, basta che non sia uno di quelli riservati dal sistema. Il sistema lo tratta comunque come un errore Oracle, nel senso che un WHEN OTHERS esterno acchiapperebbe quello applicativo.

Dichiarazione di eccezione come variabile

```
DECLARE
<nome_var> EXCEPTION;
BEGIN
 UPDATE...
 IF SQL%NOT_FOUND THEN RAISE <nome_var>
 END IF;
 ...
EXCEPTION
WHEN <nome_var> THEN RAISE_APPLICATION_ERROR(...
END;
```

```
DECLARE
 <nome_var> EXCEPTION;
 PRAGMA EXCEPTION_INIT(<nome_var>,<codice_di_eccezione_di_sistema>);
BEGIN
...
EXCEPTION
WHEN <nome_var> THEN ...;
END;
```

Questo codice riconosce l'eccezione definita nel PRAGMA, per eccezioni di sistema che hanno il codice ma non l'etichetta, ad esempio per il vincolo di integrità referenziale, che non ha una propria etichetta.

Stored procedures

```
CREATE OR REPLACE PROCEDURE <nome_procedura> [( <nome_parametro> { IN || OUT } <tipo_parametro>
...
BEGIN
...
END <nome_procedura>;
```

I parametri delle procedure non hanno informazioni di precisione I comandi di CREATE, ALTER, DROP, RENAME, TRUNCATE, GRANT, REVOKE, COMMENT hanno l'autocommit

```
EXEC <nome_procedura> [ ( <nome_parametro1> => ] <valore_parametro1>[ , ...])
```

I blocchi anonimi stanno nella shared pool e poi, una volta eseguiti, vengono "buttati". Le procedure invece sono blocchi PL\SQL con un nome fisso e che sono incapsulati sul db e il cui codice è visibile tramite interrogazione di tabelle di sistema. Vantaggi: 1)organizzativo: tutto il codice è mantenuto in un unico posto 2)quando viene incapsulato il codice viene anche controllato e viene eseguito il piano di eseguzione: queste operazioni sono fatte una volta, alla creazione o compilazione della procedura e non ogni volta come accade per i blocchi anonimi. Anche gli errori di compilazione sono mantenuti in una tabella, detta user_errors. Ne risulta che le prestazione sono superiori. 3)Ho più flessibilitù nella gestione dei permessi: per esempio potrei non dare il grant di lettura in una tabella ma darli solo ad una store procedure che si occuperà di leggere i dati e che funge da interfaccia verso la tabella. 4)Maggiore chiarezza nella sintassi dei blocchi: un blocco diventa una procedura, richiamabile in modo più semplice.

Nota: perchè usare REPLACE per aggiornare una procedura invece che droppare e ricreare? Se ho dato dei grant la drop mi rimuove i grant e la create non li rimette, invece con la replace i grant sono mantenuti.

Tabelle di sistema che contengono informazioni sulle procedure:

1) user_source: contiene il codice

```
select text
   from user_source
   where name = 'NOME_PROC'
```

2) user_object: contiene l'indicazione dello stato di validità

```
select object_name, object_type, status
   from user_objects
   where object_name = 'NOME_PROC'
```

3) user_dependencies: nomi e tipi di quegli oggetti che dipendono dall'oggetto passato

```
select name, type, referenced_name
   from user_dependencies
   where referenced_name = 'NOME_OBJ'
```

Se uno degli oggetti da cui la store procedure dipende viene modificato (per esempio aggiungo un campo alla tabella) la procedura viene messa in stato invalido. Quando viene richiamata, il server tenta di ricompilarla e quindi la esegue. Il problema nasce con i db link: quando la procedura è remota rispetto alla tabella. In questo caso la procedura non si accorge della modifica alla tabella in modo automatico, quindi alla prima chiamata vedrà la differenza e sarà posta in stato invalido. Solo alla seconda eseguzione parte la ricompilazione automatica. I controlli fatti dalla procedura remota sono decisi da un parametro che dice se controllare il tempo di modifica degli oggetti coinvolti.

Il passaggio di parametri è per default posizionale, ma non è l'unico modo: con la sintassi "=>" si effettua un passaggio nominale di parametri. Altra cosa importante, se si hanno dei default non come ultimi parametri il metodo posizionale non può funzionare, ma la nominale può essere l'unico modo di richiamare la procedura.

create or replace procedure add_dept (p_name in varchar2 default 'new', p_id in departments.departement_id%type) is ~~variabili locali begin~~ blocco PL\SQL insert into departments(department_id, department_name) values (p_id, p_name); end add_dept;

se volessi chiamare passando solo p_id, non ci riuscirei con il posizionale

add_dept(113); → errore, 113 verrebbe associato a p_name

devo per forza usare il metodo nominale

add_dept(p_id => 113);

Functions

```
CREATE OR REPLACE FUNCTION <nome_funzione> [( <nome_parametro> [ IN || OUT] <tipo_parametro> [de
...
BEGIN
...
RETURN <valore_ritorno>;
...
END <nome_funzione>;
```

Anche le funzioni, come le procedure sono mantenute persistenti all'interno della base dati in più hanno sempre un valore di ritorno. Le funzioni che trattano dati semplici sono richiamabili da SQL. Se la funzione manipola i dati non è richiamabile da un contesto SQL. Non è consigliabile fare funzioni che manipolino i dati.

Usare funzioni nelle clausole di where si ha un vantaggio di prestazioni, perchè la query non viene ogni volta riesaminata e non viene fatto un nuovo piano di esecuzione. Esempio:

```
select * from tabella where id = n
```

per ogni n che chiedo viene fatto un nuovo piano di esecuzione e quindi per ogni valore richiesto la query viene aggiunta nello shared pool mentre se avessi f() return n verrebbe memorizzata una sola query nello shared pool e non verrebbe rifatto ogni volta il piano di esecuzione.

~~3 fasi di compilazione diverse e indipendenti: 3 cursori e 3 aree di memoria nello shared pool select last_name, salary from employees where employee_id = 100;~~

~~select last_name, salary from employees where employee_id = 101;~~

~~select last_name, salary from employees where employee_id = 124;~~

controlliamo nello shared pool select sql_text, executions from v$sql where sql_text like '%select last_name%'

~~se invece avessi una funzione che assume solo valori differenti, la query verrebbe reciclata: ne vedrei il codice solo una volta nello~~ shared pool (-->concetto di BIND VARIABLE
)

Le funzioni accettano anche parametri di output (!!!). Per esempio:

```
create or replace function demo (p_id in number, p_name out varchar2) return boolean is begin
select last_name into p_name from employees where employee_id = p_id; return (true); exception
where no_data_found then p_name := 'nessun dato'; return (false); end; /

declare appo varchar2(25); begin if demo(100,appo) then dbms_output.put_line(appo); else
dbms_output.put_line(appo); end if; end; /
```

Package

Racchiude procedure e funzioni, permette l'offuscamento e l'overloading. Se nelle specifiche si fa una dichiarazione di variabile, poi la si può usare anche al di fuori del package: si potrebbe pensare a un package con solo parte di spec che faccia da contenitore di variabili. La definizione è comune ma l'istanziazione va per utente. Stesso ragionamento per fare dei custom mapping di eccezioni: si mettono nella spec di un package le varie pragma exception in modo da avere una gestione centralizzata. Le variabili all'interno del package di default hanno persistenza per sessione, ma si può modifcare questo comportamento aggiungendo

```
pragma serially_reusable
```

nella package specification per renderlo persistente per la durata di una chiamata a un sottoprogramma.

Intestazione

```
CREATE OR REPLACE PACKAGE <nome_package>
IS
{[ TYPE ... ]}
{[ <nome_var> <tipo_var>:=<val_default>; }}
{[ PROCEDURE <nomeproc> (<variabili proc >); ]}
...
{[ FUNCTION <nomefunc> (<parametri func>)RETURN <tiporeturn>; ]}
```

```
...
END;
```

Body

```
CREATE OR REPLACE PACKAGE BODY <nome_package>
IS
{[ PROCEDURE <definizione proc completa> ]}
...
{[ FUNCTION <definizione func completa> ]}
...
[
BEGIN --blocco anonimo facoltativo eseguito prima della prima invocazione del package nella sess
...
END;
]
END;
```

Ha senso fare replace solo dei BODY, quando non è necessario aggiornare le specifiche, in questo modo non si forza la de-compilazione dei dipendenti dalle specifiche

Blocco di inizializzazione del package Posso mettere blocchi anonimi (i blocchi anonimi non devono terminare con end)all'interno del package body: questo viene eseguito una volta alla prima chiamata di un elemento del package e viene utilizzato per inizializzare variabili pubbliche e private all'interno del package (valori che poi avranno validità per la sessione).

Overloading Si possono definire diverse procedure o funzioni con lo stesso nome ma un diverso elenco di parametri. Il compilatore ammette più procedure con lo stesso elenco di tipi di parametri (in java non si potrebbe), ma in quel caso si possono chiamare solo nominando esplicitamente i parametri con l'operatore -> (ovviamente i nomi dei parametri devono essere diversi). Se ci può essere ambiguità nella chiamata a runtime, il sistema dà errore.

Forward declaration Per chiamare in un package body una procedura che è invece definita più in fondo, bisogna scrivere all'inizio del body la riga di intestazione di quella procedura

Trigger

Sono blocchi di codice associati a determinati eventi di tabelle Mai fare trigger che: - fanno già cose che può fare Oracle - duplicano altri trigger. Possono essere a livello di: - istruzione - riga. Possono essere attivati: - before - after - instead of

```
CREATE OR REPLACE TRIGGER <nome_trig>
{ BEFORE || AFTER || INSTEAD OF} {INSERT || UPDATE || DELETE || OR ...}
ON <tabella> [FOR EACH ROW]
BEGIN
...
END;
```

Disabilitare:

```
ALTER TRIGGER <nome> DISABLE
```

- Un Trigger può fare un commit indipendentemente dall'utente: un rollback dato da utente, non inibisce il commit del trigger - Un Trigger può avere informazioni sull'evento che lo ha generato - Si può condizionare l'azione del trigger con la clausola WHEN

```
...
WHEN (NEW.<colonna>=<valore>)
BEGIN
...
```

Altre istruzioni

CUBE

```
SELECT ...
GROUP BY CUBE(<colonna1>, [...<colonnaN>)
...
```

questa query fa tutti i raggruppamenti, parziali e non, possibili tra le colonne specificate, ad esempio:

```
GROUP BY CUBE (A,B) corrisponde a GROUP BY (A) UNION GROUP BY (B) UNION GROUP BY (A,B) UNION GR
```

INSERT in vista

Se una vista non fa join, la si può usare per inserire dati. Se una vista però è filtrata, può capitare che inserendo un dato in vista poi non lo si ritrovi.

Se nella vista alla fine c'è WITH CHECK OPTION, in fase di inserimenti dati si controlla che tali dati siano ammissibili per la vista.

Se una vista mostra poche colonne, e non ci sono abbastanza valori per fare un inserimento per bene, si può scrivere un trigger sulla vista che completi l'inserimento con i dati mancanti

```
CREATE OR REPLACE TRIGGER <nome>
INSTEAD OF INSERT ON <vista>
FOR EACH ROW
 BEGIN
 INSERT INTO <tabella>(<elenco colonne>)
 VALUES (<valori>, ... :new.<colonna di vista del quale abbiamo in input il valore> ...)
END;
```

SQL DINAMICO CON DDL

```
EXECUTE IMMEDIATE '<codice sql sotto forma di stringa>';
```

da chiamare dentro una procedura o funzione

Cursore variabile

Strong cursor: - ha un RETURN type - può associarsi solo a query compatibili - meno errori

Weak cursor: - si può associare ad ogni query - flessibile - non si può usare FOR UPDATE

```
VAR <nomevar> REFCURSOR
BEGIN
 OPEN :<nomevar> FOR SELECT ...;
END;
PRINT <nomevar>
```

questa stampa tutta la query

Subtype

```
SUBTYPE <nometipo> IS <tipo esistente>;
```

è un modo per rinominare i tipi e renderli più leggibili

Nested table storicizzata

```
CREATE TYPE <nometipo> AS OBJECT (<elenco colonne e tipi...>);

CREATE TYPE <nested> AS TABLE OF <nometipo>;

CREATE TABLE <nometab> (<elencocolonne e tipi>, <colonnaN> <nested>) NESTED TABLE <colonnaN> <nest
```

Pipelined

Una pipelined table function è in grado di ritornare i record al chiamante uno alla volta. La funzione, una volta invocata, comincia l'elaborazione e ritorna al chiamante il primo record poi continua con il secondo, lo ritorna e così via. E' chiaro che in questo modo sia le performance che l'utilizzo di memoria sono molto migliori.

```
FUNCTION <nomefun> ... RETURN <tipo> PIPELINED;
 BEGIN
 ...
 FOR...
  PIPE <oggetto di tipo <tipo>>;
 END LOOP;
 ...
 RETURN;
END;

....

SELECT * FROM TABLE(<nomefun>(<par...>);
```

Esempio: una funzione che ritorna un numero non precisato di numeri casuali:
SQL> create or replace type t_type as table of number; 2 /

Type created.

SQL> create function random_num return t_type PIPELINED 2 is 3 begin 4 loop 5 pipe row(dbms_random.value(1,1000)); 6 end loop; 7 return; 8 end; 9 /

Function created.

*SQL> select * 2 from table(random_num) 3 where rownum<11;*

COLUMN_VALUE —--- 978,297758 644,640576 130,253134 891,206621 39,4187734 461,34422 61,5183674 765,650893 824,892227 651,643861

Gestione eccezioni per blocchi con FORALL

```
TYPE <var type> IS TABLE OF <tabella>%ROWTYPE;
<var_table> <var_type>
BEGIN
 SELECT * BULK COLLECT INTO <var_table>;
 FORALL i IN <var_table>.FIRST..<var_table>.LAST
  SAVE EXCEPTIONS
 INSERT INTO <tabella> VALUES <var_table>(I);

FOR j IN ... SQL%BULK_EXCEPTIONS
```

Il forall è un for finto, perchè in realtà inserisce tutto in un'unica soluzione. Il SAVE EXCEPTIONS fa "conservare" le eccezioni per la fine del for.

Select da file

L'utente deve avere i diritti di lettura sulle directory di oracle. Scriviamo una query che legga da un file di testo. Dobbiamo creare una tabella apposita, tabella "per uso esterno" : organization external (dalla versione 9 di oracle). Le select fatte su questa tabella in realtà andranno a leggere dal file txt (che deve stare sul server).

Quindi si potranno estrarre i dati dal file tramite select e importarli per esempio su una altra tabella.

Tisp: - PL/SQL usa la logica short circuit: una catena di IF ELSIF THEN è meglio di tanti if. - Le conversioni implicite sono da evitare (esempio: assegnare un pls_integer a un number) - PLS_INTEGER è meglio di NUMBER per gli interi (utilizza codice macchina e coprocessore) - Dichiarare una variabile come NOT NULL è meno performante di fare controlli espliciti