PL/SQL

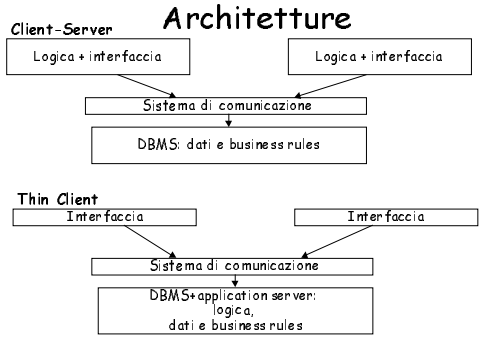
Un linguaggio per la realizzazione di applicazioni di basi di dati

Realizzazione di applicazioni di basi di dati: alternative

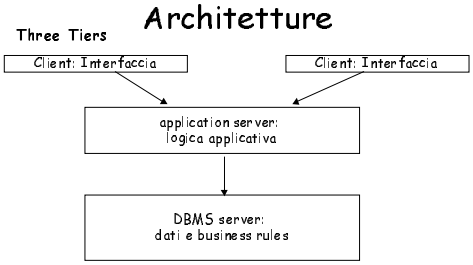
Realizzazione di applicazioni

- Quattro parti:
  - Gestione dati
  - Business rules
  - Logica applicativa
  - Interfaccia utente
- Molte possibili architetture
- Approccio tradizionale: uso di un linguaggio

Architetture



Architetture



Linguaggio: il problema

- Scrivere un'applicazione che visualizzi una schermata, raccolga dei dati, segnali eventuali inconsistenze con i dati nella BD oppure inserisca i nuovi dati
- Serve un linguaggio che possa:
  - Effettuare I/O
  - Effettuare interrogazioni
  - Controllare il flusso in un modo che dipende dal risultato dell'interrogazione
  - Effettuare aggiornamenti
- Tre soluzioni:
  - Linguaggio di programmazione + API
  - Linguaggio immerso
  - Linguaggio integrato

Linguaggio: il problema

- Come abbiamo visto, SQL permette in modo dichiarativo di accedere i dati memorizzati in una base di dati
- il suo potere espressivo è però limitato
- questo significa che non tutte le elaborazioni che possiamo volere applicare ai dati possono essere espresse in SQL
- Esempio: Supponiamo di avere una base di dati avente il seguente schema:

  Impiegati(#Imp,Nome,Stipendio,#Manager)
- supponiamo adesso di volere determinare tutti i manager diretti o indiretti dell'impiegato Rossi
- questa computazione non è esprimibile in SQL

Linguaggio

- Nasce quindi l'esigenza di estendere il potere espressivo di SQL
- **idea**: combinare SQL con linguaggi di programmazione
  - SQL: utilizzato per l'accesso ai dati
  - linguaggio di programmazione: utilizzato per manipolare i dati
- Tre possibili approcci:
  - **estensioni imperative di SQL**: si estende SQL con tipici costrutti dei linguaggi di programmazione
  - **interfacce verso il DBMS**: si utilizza un'interfaccia per utilizzare le funzionalità del DBMS da linguaggio di programmazione
  - **embedded SQL**: si utilizza SQL all'interno di un

Estensioni imperative di SQL

- Combinazione di statement SQL e costrutti imperativi
- possibilità di definire funzioni e procedure
- ogni procedura viene compilata e può essere direttamente eseguita o chiamata da programmi applicativi, utilizzando SQL embedded o interfacce per DBMS

Interfacce per DBMS

- Libreria di funzioni del DBMS che possono essere chiamate dai programmi applicativi
- Anche chiamate Call Level Interface (CLI)
- simile alle tipiche librerie C
- Funzioni tipiche:
  - Connessione
  - Invio statement per esecuzione
  - Analisi risultato
  - Disconnessione
- Esempio:
  - la CLI di Oracle si chiama OCI (Oracle Call Interface)
  - librerie standard: ODBC, JDBC

Embedded SQL

- Possibilità di utilizzare statement SQL in programmi scritti con tipici linguaggi di programmazione (C, Java)
- Gli statement SQL vengono processati da uno speciale precompilatore che invia lo statement al DBMS per l'esecuzione e recupera il risultato
- Necessità di gestire la comunicazione tra programma applicativo e DBMS

Flusso

- Qualunque sia l'approccio scelto, il programma deve eseguire i seguenti passi fondamentali:
  - connessione alla base di dati
  - dichiarazione variabili di comunicazione con il DBMS
  - esecuzione statement SQL
  - analisi risultato
  - disconnessione dalla base di dati

Flusso - Connessione

- Le procedure scritte con estensioni imperative di SQL possono venire eseguite direttamente dal DBMS,quindi non richiedono una connessione esplicita
- negli altri due casi è necessario specificare a quale base di dati ci si vuole connettere, con quale utente e quale password

Flusso - Variabili di comunicazione

- Il linguaggio di programmazione e gli statement SQL possono in genere comunicare utilizzando opportune variabili
- tali variabili possono essere utilizzate per definire gli statement SQL o per inserire valori ottenuti come risultato di interrogazioni SQL

Flusso - Esecuzione statement SQL

- Tutti e tre gli approcci devono dare la possibilità di eseguire statement SQL
  - interrogazioni: restituiscono in generale un insieme di tuple
  - insert, delete, update: restituiscono in genere il numero di tuple inserite, cancellate, aggiornate
  - statement DDL: restituiscono valore predefinito

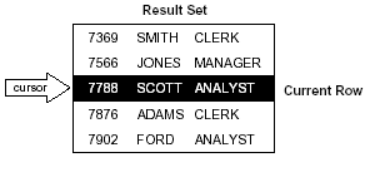
Flusso - Analisi del risultato

- Problema relativo all'esecuzione di interrogazioni SQL
- se l'interrogazione restituisce solo una tupla:
  - Il numero di informazioni da analizzare è noto a tempo di compilazione e pari al numero degli attributi della relazione
  - è possibile definire variabili di comunicazione da utilizzare per inserire i valori degli attributi della tupla restituita
- se l'interrogazione restituisce più tuple:
  - è necessario un meccanismo che permetta di muoversi sulle tuple del risultato, una ad una, e manipolarle
  - questo è possibile utilizzando un **cursore**

Flusso - Cursore

- Un cursore si può definire in astratto come un puntatore alle tuple ottenute come risultato di un'interrogazione SQL
- Operazioni sui cursori:
  - **dichiarazione**: associa un cursore ad un'interrogazione
  - **apertura**: esegue l'interrogazione associata al cursore e lo inizializza
  - **avanzamento**: sposta il cursore sulla tupla successiva del risultato
  - **chiusura**: disabilita il cursore

Flusso - Cursore



Flusso - Disconnessione

- Chiude la connessione tra il programma applicativo e la base di dati
- nel caso di procedure scritte con estensioni imperative di SQL, quando vengono eseguite direttamente dal DBMS, questa operazione non è necessaria

SQL statico e dinamico

- SQL statico:
  - Uso di statement SQL noti a tempo di compilazione
- SQL dinamico:
  - uso di statement SQL noti solo a tempo di esecuzione
  - tipico esempio:
    - clausola WHERE in uno statement SELECT cambia in relazione al fatto che una certa condizione sia vera o falsa
    - se siamo in maggio, determina gli impiegati con stipendio > 1000, altrimenti con stipendio > 2000;

PL/SQL: Introduzione

Estensioni imperative di SQL

- Ogni DBMS in genere supporta un'estensione di SQL che supporta concetti tipici dei linguaggi di programmazione
- Linguaggio integrato:
  - integrazione dei tipi di dato
  - integrazione dello scoping
  - integrazione del DML
- Tale linguaggio può essere usato per scrivere programmi che coinvolgano statement SQL
- Tali programmi possono essere:
  - scritti ed eseguiti direttamente da una shell
  - memorizzati e richiamati quando necessario (procedure, package)

Estensioni imperative di SQL

- Non esiste uno standard per le estensioni imperative di SQL
  - Oracle: PL/SQL
  - SQL Server: T-SQL
- ogni variante può differire dalle altre per la specifica sintassi
- ogni estensione deve permettere di specificare
  - dichiarazioni
  - assegnamenti
  - strutture di controllo
  - istruzioni per cursori
  - modularità
  - (transazioni)

PL/SQL: Un esempio

```
procedure prenotaIf(
   ilLogin    in prenota.login%TYPE,
   laData     in date,
   lOra       in prenota.ora%TYPE) is
unaPrenotazione prenota%ROWTYPE;
cursor c is select *
          from prenota
          where data = laData and ora = lOra;
begin
  open c;
  fetch c into unaPrenotazione;
  if c%NOTFOUND
  then insert into prenota
     values (codiceSeq.nextval, ilLogin, laData,
              lOra , ilTerm, laDataP , ilTermP );
  else ...;
  end if;
end prenotaIf;
```

PL/SQL: Un esempio

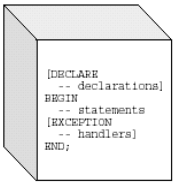
```
procedure creaElaborato(CodTemp   AllocazioniTemp.CodTemp%type,
                        Resp  number) is i  binary_integer;
 cursor callocazioni is select CodEl, Matricola,
 from Allocazioni
 where Allocazioni.CodEl = CreaElaborato.CodTemp;
   begin
    insert into Elaborati (CodEl, PassEl, CodCo)
         values (CodEl,PassEl, '1');
   open callocazioni;
   loop
   fetch callocazioni into IlCodTemp ,LaMatricola;
   exit when callocazioni%notfound;
   insert into Allocazioni (Matricola,CodEl,Responsabile)
   values(LaMatricola,CodEl,'N');
   end loop;
   close callocazioni;
end creaElaborato;
```

PL/SQL

- Linguaggio per manipolare basi di dati che integra DML (SQL) con il linguaggio ospite
- linguaggio a blocchi con una struttura del controllo completa che contiene l'SQL come sottolinguaggio
- permette:
  - di definire variabili di tipo scalare, record (annidato), insieme di scalari, insiemi di record piatti, cursore
  - di definire i tipi delle variabili a partire da quelli della base di dati
  - di eseguire interrogazioni SQL ed esplorarne il risultato
  - di modificare la base di dati
  - di definire procedure e moduli
  - di gestire il flusso del controllo, le transazioni, le eccezioni

PL/SQL

- É un linguaggio con struttura a blocchi
- ogni blocco è composto da tre parti:
  - parte dichiarativa
  - parte di esecuzione
  - parte di gestione delle eccezi



PL/SQL: Struttura

- Il blocco:
  - DECLARE <dichiarazioni>
  - BEGIN <comandi>
  - EXCEPTION <gestori>
  - END;
- La procedura:
  - PROCEDURE <parametri> IS
  - <dichiarazioni>
  - BEGIN <comandi>
  - EXCEPTION <gestori>
  - END;
- Il modulo: interfaccia e implementazione
  - CREATE PACKAGE <nome> AS … END <nome>;
  - CREATE PACKAGE BODY <nome> AS …
    END <nome>;

Dichiarazioni

- Le dichiarazioni devono essere precedute dalla parola chiave DECLARE
- è possibile dichiarare sia costanti che variabili ed usarle ovunque possa comparire un'espressione
- le variabili possono avere un qualunque tipo SQL
- è possibile assegnare un valore di default alle variabili al momento della dichiarazione
- è possibile specificare vincoli di NOT NULL

Dichiarazioni - Esempio

```
DECLARE
   stipendio NUMBER(4) NOT NULL;
   limite_di_credito CONSTANT NUMBER(7)
   := 3000000;
   data_nascita DATE;
   num_impiegati INTEGER := 0;
   manager BOOLEAN;
```

Dichiarazioni - Uso di attributi

- Possibilità di specificare che il tipo di una variabile coincide con il tipo di un'altra variabile o di un campo di una tabella
  - impiegato impiegati.nome%TYPE;
    - il tipo della variabile impiegato coincide con il tipo del campo nome della tabella impiegati
  - credit REAL(7,2);
    debit credit%TYPE;
    - il tipo della variabile debit coincide con il tipo della variabile credit
- Possibilità di specificare che una variabile rappresenta un record, corrispondente ad una tupla di una tabella
  - dept_rec dept%ROWTYPE;
    - la variabile dept_rec è dichiarata come un record, i cui elementi

Dichiarazioni

- <nome> <tipo>;
- <nome> CONSTANT <tipo> := <expr>;
- CURSOR <nome> IS <query>;
  - Operazioni: OPEN, FETCH, CLOSE
- Attributi di variabili e cursori:
  - Var%TYPE, dove var è una variabile o una colonna
  - Cur%ROWTYPE dove cur è un cursore o una tabella

Dichiarazioni

- Possono avere come inizializzatore un'espressione complessa, o non averlo (NULL):
  - nome tipo;
  - nome tipo [not null] := expr;
    (oppure: nome tipo [not null] DEFAULT expr;)
  - nome CONSTANT tipo [not null] := expr;
- Un tipo si può specificare: variabile%TYPE, colonna%TYPE (non riceve in questo caso il NOT NULL)
- tabella%ROWTYPE, cursore%ROWTYPE: tipo record
  - selezione e scrittura come var.field
  - assegnamento totale solo tra due record il cui tipo è tratto dalla stessa tabella o cursore, o usando SELECT * INTO reg FROM tab, se reg ha tipo tab%ROWTYPE

Istruzioni - Assegnamenti

- Assegnamenti di base
  - stipendio := 2000000;
  - manager := TRUE;
  - bonus := stipendio * 0.5;
- assegnamenti con valori estratti dal database
  - si esegue una select che restituisce un'unica tupla
  - si assegnano i campi della tupla ad opportune variabili
- operatori per ogni tipo supportato, come in SQL
- possibilità di effettuare assegnamenti tra variabili record basate sulla stessa tabella

Istruzioni - Esempio

```
DECLARE
emp_id emp.empno%TYPE;
emp_name emp.ename%TYPE;
wages NUMBER(7,2);
BEGIN
...
SELECT ename, sal + comm
INTO emp_name, wages FROM emp
WHERE empno = emp_id;
...
END;
```

Istruzioni: Flusso di controllo

- IF - THEN - (ELSIF THEN) - ELSE - END IF
- LOOP <comandi> END LOOP (EXIT WHEN <cond>)
- FOR var IN seq LOOP <comandi> END LOOP
- GOTO label ... <<label>> comandi

Istruzioni - Strutture di controllo

- Costrutti di scelta:
  - IF cond THEN seq of stat;
    [ELSIF cond THEN seq of stat;]*
    [ELSE seq of stat;]
    END IF;
  - come in C
    IF condition1 THEN sequence_of_statements1
    ELSIF condition2 THEN sequence_of_statements2
    ELSE sequence_of_statements3
    END IF;
  - i rami *then* sono valutati solo se la condizione è *true*; *false* e *null* non valutano il *then*, ma valutano l'*else*

Istruzioni - Esempio

```
BEGIN
...
IF sales > 50000 THEN
bonus := 1500;
ELSIF sales > 35000 THEN
bonus := 500;
ELSE
bonus := 100;
END IF;
INSERT INTO payroll VALUES (emp_id, bonus, ...);
END;
```

Istruzioni - Strutture di controllo

- cicli
  - **LOOP**: LOOP
    sequence_of_statements
    EXIT WHEN boolean_expression;
    END LOOP;
  - **WHILE LOOP**: WHILE condition LOOP
    sequence_of_statements
    END LOOP;
  - **FOR LOOP**:FOR counter IN [REVERSE]
    lower_bound..higher_bound
    LOOP
    sequence_of_statements
    END LOOP;

Istruzioni - Strutture di controllo

- [WHILE cond] LOOP seq of stat; END LOOP;
- Per uscire:
  - Se cond è false-null, non viene (più) eseguito
  - EXIT
  - EXIT WHEN cond: uguale a IF cond THEN EXIT END IF
- Loop etichettati:
  - <<ciclo>> LOOP seq of stat; END LOOP ciclo;
  - EXIT ciclo: può uscire da più cicli in un colpo
- FOR counter IN [REVERSE] lower..higher LOOP ....
  - se lower>higher il loop non è eseguito; si può uscire con EXIT
  - counter è dichiarato implicitamente, è costante, ha il loop come scope

Istruzioni - Esempio

```
WHILE total <= 25000 LOOP
   ...
   SELECT sal INTO salary FROM emp
     WHERE ...
   total := total + salary;
END LOOP;
```

Istruzioni - Strutture di controllo

- GOTO
  - è necessario associare label a determinati punti del programma
  - con l'istruzione GOTO è possibile modificare il flusso, portandolo ad eseguire l'istruzione associata alla label specificata

```
BEGIN
...
GOTO insert_row;
...
<<insert_row>>
INSERT INTO emp VALUES ...
END;
```

Cursori

- Dichiarazione:
  - CURSOR cursor_name [(parameter[, parameter]...)] IS select_statement;
- apertura:
  - OPEN cursor_name;
- avanzamento:
  - FETCH cursor_name INTO record_name
  - FETCH cursor_name INTO lista_variabili
  - il cursore viene associato ad una variabile booleana NOT FOUND che diventa vera quando non ci sono più tuple da analizzare
- chiusura:
  - CLOSE cursor_name

Cursori - Esempio

```
DECLARE
my_sal emp.sal%TYPE;
my_job emp.job%TYPE;
factor INTEGER := 2;
CURSOR c1 IS SELECT factor*sal FROM emp WHERE job = my_job;
BEGIN
...
OPEN c1;
LOOP
   FETCH c1 INTO my_sal;
   EXIT WHEN c1%NOTFOUND;
   ....
END LOOP;
CLOSE c1;
END;
```

definizione

apertura

avanzamento

chiusura

Cursori - Esempio: uso di parametri

- DECLARE
  emp_name emp.ename%TYPE;
  CURSOR c1 **(name VARCHAR2)** IS
    SELECT * FROM emp WHERE ename = name
- Al momento dell'apertura:
  OPEN c1**("John")**;

Eccezioni

- L'ultima sezione di un programma PL/SQL permette di gestire gli errori
- le eccezioni possono essere:
  - definite dal sistema
  - definite dall'utente

Eccezioni di sistema

Exception	Oracle Error	SQLCODE Value
ACCESS_INTO_NULL	ORA-06530	-6530
COLLECTION_IS_NULL	ORA-06531	-6531
CURSOR_ALREADY_OPEN	ORA-06511	-6511
DUP_VAL_ON_INDEX	ORA-00001	-1
INVALID_CURSOR	ORA-01001	-1001
INVALID_NUMBER	ORA-01722	-1722
LOGIN_DENIED	ORA-01017	-1017
NO_DATA_FOUND	ORA-01403	+100
NOT_LOGGED_ON	ORA-01012	-1012
PROGRAM_ERROR	ORA-06501	-6501
ROWTYPE_MISMATCH	ORA-06504	-6504
SELF_IS_NULL	ORA-30625	-30625
STORAGE_ERROR	ORA-06500	-6500
SUBSCRIPT_BEYOND_COUNT	ORA-06533	-6533
SUBSCRIPT_OUTSIDE_LIMIT	ORA-06532	-6532
SYS_INVALID_ROWID	ORA-01410	-1410
TIMEOUT_ON_RESOURCE	ORA-00051	-51
TOO_MANY_ROWS	ORA-01422	-1422
VALUE_ERROR	ORA-06502	-6502
ZERO_DIVIDE	ORA-01476	-1476

Eccezioni definite dall'utente

- Dichiarazione:
  DECLARE
    …
    my_exception EXCEPTION;
- sollevamento eccezione:
    RAISE my_exception;
- definizione eccezione:
  EXCEPTION
    WHEN my_exception
    … codice PL/SQL

Eccezioni – Esempio (eccezione di sistema)

```
DECLARE
   pe_ratio NUMBER(3,1);
BEGIN
    SELECT price / earnings INTO pe_ratio FROM stocks
    WHERE symbol = 'XYZ'; -- might cause division-by-zero
      error
    INSERT INTO stats (symbol, ratio) VALUES ('XYZ',
      pe_ratio);
    COMMIT;
EXCEPTION
    WHEN ZERO_DIVIDE THEN -- handles 'division by zero'
      error
    INSERT INTO stats (symbol, ratio) VALUES ('XYZ',
      NULL);
    COMMIT;
    ...
```

Eccezioni – Esempio (eccezione definita dall'utente)

```
DECLARE
   ...
  comm_missing EXCEPTION; -- declare exception
BEGIN
   ...
   IF commission IS NULL THEN
   RAISE comm_missing; -- raise exception
   END IF;
   bonus := (salary * 0.10) + (commission * 0.15);
   EXCEPTION
   WHEN comm_missing THEN ... -- process the exception
END;
```

Modularità

- Il codice PL/SQL può essere organizzato in unità programmative:
  - procedure, funzioni: concetto analogo a quello visto per C e Java
  - package: insieme di dichiarazioni di variabili, costanti, procedure e funzioni che possono essere memorizzate nel database e utilizzate dalle applicazioni

Tabelle e record

- Una tabella è un'array associativo dinamico, di tipo
  - TABLE OF <tipo> INDEX BY <tipo>
- I tipi record sono annidabili:
  - RECORD (nome tipo,...,nome tipo)

PL/SQL: Architettura

- Tre possibilità:
  - un programma PL/SQL può essere trasformato da uno strumento in un programma che invia comandi SQL al server
  - un programma PL/SQL può essere inviato dallo strumento al server che lo esegue
  - un programma PL/SQL può risiedere nel server che lo esegue

PL/SQL: Preliminari

Unità lessicali

- Simboli, identificatori, costanti, commenti
- Simboli: +,-,*,/,=,<,>,";",%,.,@,',",:,**,<>,!=,~=...
- Identificatori: lettere, numeri, $, _, # ($,_,#: una sola volta, non all'inizio), oppure "caratteri ascii"(fino a 30)
- Costanti:
  - numeriche: interi e reali (6, -14, +.5, 25., 2E5, -9.5e-3)
  - caratteri e stringhe: 'a', 'abc'
  - bool: TRUE, FALSE, NULL
- Commenti:

```
select *    -- commento su singola linea
from        /* commenti su più
               linee              */
```

Tipi: Tipi numerici

- BINARY_INTEGER
  - interi con segno, range -2147483647… 2147483647
  - sottotipi:
    - NATURAL, POSITIVE interi positivi
    - NATURALN, POSITIVEN interi positivi non nulli
    - SIGNTYPE solo -1, 0, 1
- PLS_INTEGER: stesso range, operazioni più efficienti (machine arithmetic invece di library arithmetic)
- NUMBER[(cifre:=38,posvirgola:=0)]
  - numeri interi o reali in in notazione virgola fissa
  - range di magnitudine 1E-130..10E125

Tipi: Tipi numerici

  - Reali virgola fissa NUMBER(cifre,posvirgola)
  - Interi: NUMBER(cifre)
  - sottotipi:
    - dec, decimal, numeric: reali in notazione a virgola fissa con massimo 38 cifre decimali
    - integer, int, smallint: interi cin massimo 38 cifre decimali
  - Ex: Number(8), Number(8,4)
- FLOAT[(cifre binarie:=126)]
  - reali in notazione virgola mobile (floating point)
  - sottotipi:
    - double precision: reali in notazione a virgola mobile max 126 cifre decimali
    - real: reali in notazione a virgola mobile max 63 cifre decimali

Tipi: Stringhe

- CHAR[(max length:=1)] dimensione fissa (max<256 per le colonne, 32767 per le variabili)
  - sottotipo: character
- LONG: dim variabile (<2MB per le colonne, 32760 per le variabili)
- RAW(max length): non interpretati (dati binari o stringhe di bit) (max colonna: 255; variabile: 32767)
- LONG RAW: long+raw
- ROWID: memorizza rowid in formato ffff. bbbbbbbb.rrrr (numero di file, di blocco, di riga)
- VARCHAR2(max length) dimensione variabile (max colonna: 2000, variabile: 32767)
  - sottotipi: string, varchar

Tipi: Date e booleani

- Boolean:
  - booleani, con possibili valori *TRUE, FALSE* e *NULL*
  - solo variabili, non nella basi di dati
- Date:
  - data + ora, minuto e secondo
  - diversi possibili formati
  - es. 'DD-MON-YY hh:mm:ss'
  - funzioni TO_DATE e TO_CHAR per conversioni a e da stringhe
  - SYSDATE per data e ora corrente

Subtypes

- Servono a dare un altro nome ad un tipo:
  - DECLARE SUBTYPE Accumulator IS NUMBER;
- Possono contenere un'indicazione di dimensione, ma solo se sono definiti dal tipo di una variabile:
  - DECLARE
    - temp NUMBER(4);
    - SUBTYPE InteroQuattro IS temp%TYPE; -- ok
    - SUBTYPE Intero IS NUMBER; --ok
    - SUBTYPE InteroTre IS NUMBER(3); --no

Conversioni

- TO_CHAR: data, numero a stringa
- TO_DATE: stringa, numero a data:
  - to_date('11-03-1988','dd-mm-yyyy')
- TO_NUMBER: stringa a numero
- stringa e raw-rowid: ROWTOHEX, HEXTORAW, CHARTOROWID, ROWIDTOCHAR
- Conversioni implicite: ogni conversione immaginabile; non usarle

Nomi

- Classi di nomi:
  - Semplici: emp
  - Qualificati: luigi.emp
  - Remoti: emp@rome
  - Qualificati e remoti: luigi.emp@rome
- Non è permessa ambiguità tra gli identificatori dichiarati nello stesso livello; quelli dichiarati internamente coprono quelli esterni, ma non i nomi di colonna
- Negli identificatori maiuscole e minuscole sono uguali (non case sensitive)

Assegnamenti

- ide := expr
- select { col1,..,coln | * }
  into { var1,..,varn | rcdvar}
  from ... where
  - Fallisce se la select non trova ennuple (NO_DATA_FOUND) o ne trova più di una (TOO_MANY_ROWS)
  - per ogni colonna, ci deve essere una variabile con tipo compatibile

Espressioni

- Operatori:
  - ** (esp.), NOT ; +, -; *, / ; +, -, || (concatenazione)
  - =, !=, <, >, <=, >=, IS NULL, LIKE, BETWEEN, IN
  - AND; OR
- stringa LIKE pattern:
  - _: un carattere; %: 0 o più caratteri; case sensitive
- a BETWEEN 10 AND 20
- a IN (3,5,8)
- IS NULL viene introdotto perché NULL = NULL dà NULL, non TRUE

Espressioni- Valori nulli

- un predicato semplice valutato su un attributo a valore nullo dà come risultato della valutazione NULL
- il valore di verità di un predicato complesso viene calcolato in base alle seguenti tabelle di verità

AND

	T	F	N
T	T	F	N
F	F	F	F
N	N	F	N

OR

	T	F	N
T	T	T	T
F	T	F	N
N	T	N	N

NOT

T	F
F	T
N	N

Espressioni- Valori nulli

- Una tupla per cui il valore di verità è NULL non viene restituita dalla query
- if a then b else c end if esegue c se a è NULL
- una stringa vuota vale NULL, quindi X = Y dà sempre NULL se Y è una stringa vuota
- nelle espressioni (ad es. aritmetiche) se un argomento è NULL allora il valore dell'intera espressione è NULL
- nelle funzioni di gruppo i valori nulli sono invece ignorati (tranne count(*))

Espressioni- Valori nulli

- alcuni comportamenti strani:
  - a:= NULL;
    b:= NULL;
    ...
    IF a = b THEN st
    ENDIF; --- lo statement non viene eseguito
  - SUM(Colonna1 + Colonna2)
    può dare risultato diverso da
    SUM(Colonna1) + SUM(Colonna2)
  - (a = b) OR (a!=b) non è sempre vero:
    se a e b sono NULL non dà TRUE, ma NULL

Funzioni built-in

- Numeriche: trigonometriche, logaritmiche, trunc-round-floor-ceil...
- Char: char, concat, lower, lpad, ltrim, replace, substr, translate, upper, ascii, instr, length
- Date: add_months, last_day, months_between, next_day, sysdate
- Conversione
- NVL(e1,e2): ritorna e1 se non è null, altrimenti e2
- user, uid
- Funzioni di gruppo:
  - accettano distinct o (default) all
  - ignorano i null, tranne count(*)
  - avg, count(*), max, min, stddev, sum, variance

PL/SQL: Modularità

Procedure e funzioni

- Possono essere create come oggetti di database con costrutti DDL
  - CREATE PROCEDURE
  - CREATE FUNCTION
- possono essere dichiarate nella sezione di dichiarazione di un programma PL/SQL
  - tutto uguale a prima
  - si toglie la parola chiave CREATE

Procedure

- Creazione procedura:
  [CREATE [OR REPLACE]]
  PROCEDURE procedure_name[(parameter[,parameter]...)]
  {IS|AS}
  [local declarations]
  BEGIN
  executable statements
  [EXCEPTION
  exception handlers]
  END [name];
- definizione parametri:
  parameter_name [IN | OUT | IN OUT] datatype
  [{:= | DEFAULT} expression]
- chiamata:
  procedure_name(p1,….,pn);

Procedure

- Dichiarazione:
  procedure nome [(parametri)];
- Definizione:
  procedure nome [(parametri)] is
  dichiarazioni ;
  begin comandi;
  exception gestori;
  end [nome];

Definizione di procedure

- In un corpo di modulo (globali)
- In coda in una qualunque sezione declare
- Dichiarazione forward: ha la sintassi di una dichiarazione
- Nella base di dati:
  - create procedure / function [(..)] is / as ....
- È ammesso in una certa misura l'overloading di funzioni i cui parametri hanno un tipo diverso

Parametri

- Sintassi:
  - parametri := parametro {, parametro}*
  - parametro
    ::= nome [in | out | in out] tipo [default expr]
- Tipo:
  - uno scalare (boolean, number, varchar2...) senza dimensioni
  - nome%type, nome%rowtype
- possono essere in (caso default), out, in out
- i parametri in possono avere un valore default (default expr oppure := expr)

Parametri

- Notazione posizionale:
  - dividi(10,5)
- Nominale:
  - dividi(dividendo => 10, divisore => 5)
  - dividi(divisore => 5, dividendo => 10)
- Mista:
  - dividi(10, divisore => 5): sì
  - dividi(dividendo => 10, 5): no
  - dividi(10, dividendo =>5): no

Parametri

- in: caso default; è una costante; una funzione dovrebbe avere solo parametri in
- out: possono solo subire assegnamenti nel corpo della procedura
- in out: sono come variabili
- non è definito se i parametri sono passati per valore o per riferimento; evitare quindi l'aliasing

Procedure - Esempio

```
PROCEDURE raise_salary (emp_id INTEGER, amount
   REAL) IS
current_salary REAL;
salary_missing EXCEPTION;
BEGIN
    SELECT sal INTO current_salary FROM emp
    WHERE empno = emp_id;
    IF current_salary IS NULL THEN
    RAISE salary_missing;
    ELSE
    UPDATE emp SET sal = sal + amount
    WHERE empno = emp_id;
    END IF;
```

Esempio (continua)

```
EXCEPTION
    WHEN NO_DATA_FOUND THEN
    INSERT INTO emp_audit VALUES (emp_id,
      'No such number');
    WHEN salary_missing THEN
    INSERT INTO emp_audit VALUES (emp_id,
      'Salary is null');
END raise_salary;
```

Funzioni

```
[CREATE [OR REPLACE]]
FUNCTION function_name[(parameter[,
   parameter]...)] RETURN datatype {IS|AS}
[local declarations]
BEGIN
executable statements
[EXCEPTION
exception handlers]
END [name];
```

Funzioni

- Dichiarazione:
  function nome [(parametri)] return tipo;
- Definizione:
  function nome [(parametri)] return tipo is
  dichiarazioni;
  begin comandi;
  exception gestori;
  end [nome];
- Il corpo contiene uno o più comandi return expr (meglio uno solo e meglio in coda)
- Se la funzione esce senza ritornare un valore vi è un errore a run-time
- Chiamata: nome [(attuali)]

Funzioni - Esempio

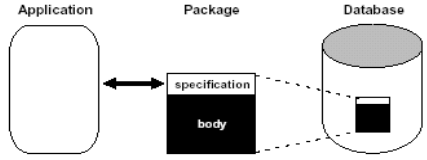
```
FUNCTION sal_ok (salary REAL, title VARCHAR2)
   RETURN BOOLEAN IS
min_sal REAL;
max_sal REAL;
BEGIN
    SELECT losal, hisal INTO min_sal, max_sal FROM
      sals
    WHERE job = title;
    RETURN (salary >= min_sal) AND (salary <=
      max_sal);
END sal_ok;
```

Package (modulo)

- Un package ha un'interfaccia ed un'implementazione; entrambe sono memorizzate in uno schema
- Vantaggi: modularità, information hiding, compilazione separata
- Interfaccia:
  - Definisce:
    - Tipi, variabili, costanti, eccezioni
    - Dichiara: cursori, procedure/funzioni
- Implementazione:
  - Definisce:
    - Cursori e procedure pubbliche
    - Tipi, variabili, costanti, eccezioni, cursori, procedure private
  - Non è indispensabile

Package (modulo)

- Un package è un oggetto contenuto nello schema di una base di dati che contiene definizioni di variabili, costanti, procedure, funzioni, eccezioni, ecc.
- L'interfaccia del package è accessibile alle applicazioni



Package - Sintassi

- create package nome as
  - definizioni pubbliche;
  - dichiarazioni pubbliche di cursori;
  - dichiarazioni di procedure pubbliche;

  end [nome];
- create package body nome as
  - definizioni private e definizioni dei cursori pubblici;
  - definizioni di procedure private e pubbliche;
  - [begin codice di inizializzazione rieseguito per ogni sessione]

  end [nome];
- Lo stato di un package è legato alla sessione (connessione/disconnessione); due sessioni concorrenti vedono due stati indipendenti

Dichiarazioni e definizioni di funzioni e procedure

- Procedure:
  - procedure nome[(parametri)];
  - procedure nome[(parametri)] is ...;
- Funzioni:
  - function nome[(parametri)] return tipo;
  - function nome[(parametri)] return tipo is ... ;
- Cursori:
  - cursor nome[(parametri)] return tipo;
  - cursor nome[(parametri)] return tipo is query;
  - Nei cursori la parte return tipo serve solo per la definizione di un cursore pubblico nel corpo di un modulo

Creazione interfaccia package

```
CREATE [OR REPLACE] PACKAGEpackage_name
{IS | AS}
...
[constant_declaration ...]
[exception_declaration ...]
[record_declaration ...]
[variable_declaration ...]
[cursor_spec ...]
[function_spec ...]
[procedure_spec ...]
[call_spec ...]
END [package_name];
```

Interfaccia package - Esempio

```
CREATE OR REPLACE PACKAGEemp_actions AS -- spec
TYPE EmpRecTyp IS RECORD (emp_id INT, salary REAL);
CURSOR desc_salary RETURN EmpRecTyp;
PROCEDURE hire_employee (
ename VARCHAR2,
job VARCHAR2,
mgr NUMBER,
sal NUMBER,
comm NUMBER,
deptno NUMBER);
PROCEDURE fire_employee (emp_id NUMBER);
END emp_actions;
```

Creazione body package

```
[CREATE [OR REPLACE] PACKAGE BODY package_name {IS |
   AS}
...
[constant_declaration ...]
[exception_declaration ...]
[record_declaration ...]
[variable_declaration ...]
[cursor_body ...]
[function_spec ...]
[procedure_spec ...]
[call_spec ...]
[BEGIN
sequence_of_statements]
END [package_name];]
```

Body package - Esempio

```
CREATE OR REPLACE PACKAGE BODY emp_actions AS -- body
CURSOR desc_salary RETURN EmpRecTyp IS
SELECT empno, sal FROM emp ORDER BY sal DESC;
PROCEDURE hire_employee (
ename VARCHAR2,
job VARCHAR2,
mgr NUMBER,
sal NUMBER,
comm NUMBER,
deptno NUMBER) IS
BEGIN
INSERT INTO emp VALUES (empno_seq.NEXTVAL, ename,
   job,
mgr, SYSDATE, sal, comm, deptno);
END hire_employee;
```

Body package - Esempio (continua)

```
PROCEDURE fire_employee (emp_id NUMBER) IS
BEGIN
DELETE FROM emp WHERE empno = emp_id;
END fire_employee;
END emp_actions;
```

Package predefiniti

- standard: contiene le funzioni standard (che quindi restano accessibili dopo un'eventuale redefinizione)
- dbms_standard
- dbms_sql: esecuzione dinamica di sql
- dbms_alert
- dbms_output: output verso sqlplus
- dbms_pipe
- utl_file: I/O su files del sistema operativo

Eccezioni

- Le eccezioni si dichiarano:
  - in un modulo (visibilità globale)
  - in un blocco (visibilità locale)
- sintassi
  - nomeecc exception;
- si sollevano con un raise nomeecc
- si propagano ai blocchi esterni ed alle funzioni chiamanti fino a che non trovano una sezione exception in grado di gestirle
- terminata l'esecuzione del gestore, il blocco termina
- due eccezioni diverse con lo stesso nome sono diverse
- un fallimento nella sezione declare o exception può essere solo gestito da un blocco più esterno

Condivisione di eccezioni

```
create or replace package eccp as
  f exception;
  g exception;
  procedure uu;
end eccp;

create or replace package pack2 as ...;

create or replace package body pack2 as
  procedure ss is
  begin
    raise eccp.f ;
  exception
    when eccp.g then stampaP.stampa('eccezione g');
  end ss;
end pack2;
```

Eccezioni interne

- Per gestire le eccezioni interne che hanno un numero, non un nome è possibile darvi un nome mediante la pragma (direttiva al compilatore) exception_init
- exception_init associa un nome di eccezione con un numero di errore Oracle

```
declare
    pochi_privilegi exception;
    pragma exception_init(pochi_privilegi, -1031);
```

Eccezioni interne

- La procedura raise_application_error del package DBMS_STANDARD permette di sollevare messaggi di errore user-defined nelle stored procedure e funzioni
- sintassi:

  raise_application_error(error_number, message, [TRUE|FALSE]);
  - error_number: numero negativo nel range [-2000,-20999]
  - message stringa
  - TRUE impila l'errore sullo stack degli errori generati, FALSE solleva solo l'errore corrente
- termina il sottoprogramma e restituisce il numero e il messaggio all'applicazione

Eccezioni interne - esempio

```
CREATE PROCEDURE raise_salary (emp_id NUMBER, amount
  NUMBER) AS
  curr_sal NUMBER;
BEGIN
SELECT sal INTO curr_sal FROM emp
  WHERE empno = emp_id;
IF curr_sal IS NULL THEN
  /* solleva messaggio di errore user defined */
  raise_application_error(-20101, 'Salary is missing');
ELSE
  UPDATE emp SET sal = curr_sal + amount
  WHERE empno = emp_id;
END IF;
END raise_salary;
```

Eccezioni interne - esempio

```
EXEC SQL EXECUTE
/* esegue unnlocco PL/SQL embedded usando le variabili del
   linguaggio ospite my_emp_id e my_amount */
DECLARE
   …
   null_salary EXCEPTION;
   PRAGMA EXCEPTION_INIT (null_salary, -20101);
   /* associa il nome null_salary al numero -20101 */
BEGIN
   …
   raise_salary(:my_emp_id, :my_amount);
EXCEPTION
   WHEN null_salary THEN INSERT INTO emp_audit
   VALUES (:my_emp_id, …);
END;
END-EXEC;
```

Gestori di eccezioni

- Sintassi:

```
exception
    when ecc1 or ... or ecc5then
        seq di comandi;
    when ecc6 then
        seq di comandi;
    when others then
        seq di comandi;
  end
```

- raise (solo nella sezione exception) risolleva l'eccezione corrente

Propagazione di eccezioni

```
BEGIN
  BEGIN
     IF X = 1 THEN RAISE A;
     ELSIF X = 2 THEN RAISE B;
     ELSE RAISE C;
     …
  EXCEPTION
    WHEN A THEN …
  END;

  EXCEPTION
    WHEN B THEN …
  END;
```

L'eccezione A viene gestita localmente, poi l'esecuzione riprende nel blocco immediatamente più esterno

Propagazione di eccezioni

```
BEGIN
  BEGIN
     IF X = 1 THEN RAISE A;
     ELSIF X = 2 THEN RAISE B;
     ELSE RAISE C;
     …
  EXCEPTION
    WHEN A THEN …
  END;

  EXCEPTION
    WHEN B THEN …
  END;
```

Si propaga l'eccezione B al primo blocco che racchiude l'istruzione RAISE e contiene un handler per B, poi l'esecuzione riprende nell'ambiente esterno

Propagazione di eccezioni

```
BEGIN
  BEGIN
     IF X = 1 THEN RAISE A;
     ELSIF X = 2 THEN RAISE B;
     ELSE RAISE C;
     …
  EXCEPTION
    WHEN A THEN …
  END;

  EXCEPTION
    WHEN B THEN …
  END;
```

Per C non viene trovato un handler, un'eccezione non gestita viene restituita all'ambiente esterno (*unhandled exception* error)

SQLCODE e SQLERRM

- In un gestore di eccezioni si possono usare le funzioni built-in SQLCODE e SQLERRM per determinare quale errore si è verificato e ottenere il messaggio di errore associato
- sqlcode: il numero dell'eccezione:
  - utente: +1
  - sql: negativo, o +100 (no data found)
- sqlerrm: il messaggio dell'eccezione:
  - utente: 'User-Defined Exception'
  - sql: un qualche messaggio
- sqlerrm(i): il messaggio associato ad i

SQLCODE e SQLERRM - esempio

```
DECLARE
  …
  err_msg VARCHAR2(100)
  BEGIN
  /* ritrova tutti i messaggi di errore di Oracle */
  FOR err_num IN 1..9999 LOOP
      err_msg := SQLERRM(-err_num);
      INSERT INTO errors VALUES (err_msg);
  END LOOP;
END;
```

Continuare dopo un'eccezione

```
DECLARE
  pe_ratio NUMBER (3,1);
BEGIN
  DELETE FROM stats WHERE symbol = 'XYZ';
  SELECT price/NVL(earnings,0) INTO pe_ratio
    FROM stocks WHERE symbol = 'XYZ';
  INSERT INTO stats (symbol,ratio)
    VALUES ('XYZ', pe_ratio);
EXCEPTION
  WHEN ZERO_DIVIDE THEN
  …
END;
```

- qualsiasi sia la gestione specificata nell'handler per ZERO_DIVIDE se la SELECT solleva ZERO_DIVIDE, l'INSERT non viene eseguita

Continuare dopo un'eccezione

```
DECLARE
  pe_ratio NUMBER (3,1);
BEGIN
  DELETE FROM stats WHERE symbol = 'XYZ';
  BEGIN                          --inizio sottoblocco
    SELECT price/NVL(earnings,0) INTO pe_ratio
          FROM stocks WHERE symbol = 'XYZ';
  EXCEPTION
    WHEN ZERO_DIVIDE THEN pe_ratio := 0;
  END;                           --fine sottoblocco
  INSERT INTO stats (symbol,ratio)
    VALUES ('XYZ', pe_ratio);
EXCEPTION
  WHEN OTHERS THEN …
END;
```

Rieseguire una transazione

```
suffisso:=0;
nome:=ilNome;
while suffisso < 100  loop
    begin
        savepoint riparti;
        ....
        insert into persone(nome, …)
        commit;
        exit;
    exception
        when dup_val_on_index then
            rollback to riparti;
            nome := ilNome || tochar(suffisso);
            suffisso := suffisso + 1;
    end;
end loop;
```

PL/SQL:Tipi strutturati

Tipi record

- Sintassi:
  - **TYPE mioTipoRecord IS RECORD (field[,field]*);**
  - **field ::= nome tipo [[NOT NULL] := expr]**
- i campi possono essere scalari o record (oracle 8: anche collezioni)
- sono tipi di prima classe
- i campi si aggiornano e si leggono con la sintassi rec.campo
- due record dello stesso tipo (cioè con lo stesso tipo campo per campo) si possono assegnare per intero

SELECT INTO

- Se una query ritorna una sola riga, si può metterne il risultato dentro un record o un insieme di campi:

```
type impiegato emp%ROWTYPE;
oppure: type impiegato is record(a number, b char(30));
unImp impiegato;
x number; y char(30);
...
select * into unImp
from emp where codice=100;
```

  - ma anche:
    - select codice, nome into x, y
    - select * into x, y
    - select codice, nome into unImp

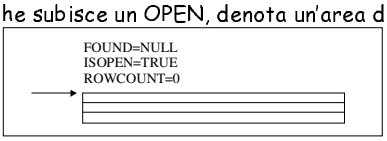
SELECT INTO

- **select** ... **into** ... fallisce se la query dà n righe con n <> 1
- per evitare problemi:

```
select count(*) into i
from ... ecc.;
if i=1 then select ...;
else ....;
end if;
```

Cursori

- Un cursore è associato ad una query
- dopo che subisce un OPEN, denota un'area di lavoro:



- ogni operazione **fetch c into var** legge una riga ed avanza il puntatore
- dopo l'ultima riga, l'effetto della fetch su **var** è indefinito (ma non fallisce)
- dopo la OPEN, FOUND = Null
- dopo le prime tre FETCH, FOUND=True
- dopo la quarta FETCH, FOUND=False

Cursori

- Un cursore è associato ad una query nella sezione di dichiarazioni
- un cusore può avere parametri:
  - **illogin varchar2;**
    cursor c is
    select ora, data
    from prenotazioni where login = illogin;
  - **cursor c(nome varchar2) is**
    select ora, data
    from prenotazioni where login = nome;

  in (1), **illogin** è valutata al momento della open del cursore

Cursori

- **Operazioni sui cursori**
  - **open**: esegue la query
  - **fetch c into <dest>**
    - dest: o una lista di variabili, o un record
  - estrazione di attributi: **c%FOUND, c%NOTFOUND, c%ISOPEN,c%ROWCOUNT**
  - **close c**: libera il risultato; si può riaprire
- Attributi dei cursori
  - FOUND/NOTFOUND: TRUE/FALSE se ultima FETCH ha restituito una tupla
  - ISOPEN: TRUE se il cursore è aperto
  - ROWCOUNT: numero di tuple ritrovate

Cursori nei FOR

- Se c è un cursore, allora:

```
for locale in c loop
  body (locale);
end loop;
```

equivale (più o meno) a:

```
declare
   locale c%rowtype;
begin
   open c;
   loop
     fetch c into locale;
     exit when c%notfound
     body(locale);
   end loop;
   close c;
end;
```

Cursori impliciti

- Se c è un cursore, allora:

```
for locale in c loop
  body(locale);
end loop;
```

equivale a:

```
for locale in ( query ) loop
  body(locale);
end loop;
```

Cursori FOR UPDATE

- E' possibile dichiarare un cursore FOR UPDATE
- talecursore può essere usato nella clausola CURRENT OF di un comando UPDATE o DELETE

```
DECLARE
CURSOR c1 IS SELECT empno, sal FROM emp
  WHERE job = 'salesman' AND comm > sal
  FOR UPDATE [OF sal];
…
BEGIN
OPEN c1;
LOOP
  FETCH c1 INTO …
  …
  UPDATE emp SET sal = new_sal WHERE CURRENT OF c1;
END LOOP;
```

Sequence

- La generazione di chiavi numeriche si può fare come segue:

```
select max(codice)+1 into nuovocodice from persone;
insert into persone
  values (nuovocodice, nome, cognome)
```

- questa tecnica può causare deadlock
- ORACLE mette a disposizione contatori persistenti, detti sequence:

```
create sequence codicePersone
increment by 1
start with 1
maxvalue 99999
cycle;
```

Sequence

- L'inserzione diventa:

```
insert into persone
values (codicePersone.nextval, nome, cognome)
```

- in seguito `s.currval` restituisce l'ultimo valore restituito da `s.nextval`
- per leggere `s.currval`

```
select s.currval into plsqlvar from dual
```

Binding di PL/SQL

- PL/SQL è compilato, per cui:
  - i nomi di tabelle e colonne devono essere specificati come costanti
  - può riferire solo tabelle e colonne già specificate
  - non può eseguire comandi del DDL
- se lo schema è cambiato al momento di eseguire una funzione, il sistema rieffettua il binding, che però può fallire se il nuovo schema è incompatibile con la procedura
- esiste un package (DBMS_SQL) per effettuare generazione e compilazione dinamica di PL/SQL

SQL in PL/SQL

- Solo il DML ed il controllo delle transazioni
- tutte le funzioni SQL, ma le funzioni aggregate solo nelle query
- pseudocolonne nelle query:
  - CURRVAL, NEXTVAL: usano una SEQUENCE, dentro una select o dentro una insert / set
  - ROWID: identifica una ennupla
  - ROWNUM: in una query ne assegno uno diverso (crescente e consecutivo) ad ogni ennupla trovata
- nella clausola where:
  - confronti, con eventualmente **some**(**any**) ed **all**
  - **between**, **exists**, **in**, **is null**
- tra due **select**: **intersect**, **minus**, **union**, **union all**

Cursori variabili

- Sono cursori su cui si possono fare assegnamenti, o puntatori assegnabili ad aree di lavoro
- Utili per:
  - fare aprire un cursore da un subroutine
  - comunicazione tra ambiente esterno e PL/SQL
  - avere un cursore che può essere legato a tabelle, query o anche tipi diversi

Definizione di cursori variabili

- Prima si dichiara il tipo poi la variabile

```
declare
    type curtipo is ref cursor
                        return prenota%rowtype;
    curVar curtipo ;
```

- la parte return è opzionale
- le variabili di cursore non possono essere variabili persistenti (variabili di package, colonne nel db)
- anche di una variabile di cursore si può estrarre il %rowtype
- operazioni:
  - open cur for query;
  - attributi, fetch into, close

Tipi tabella Index-By

- Array sparsi in memoria centrale:
  - **TYPE mioTipoTabella IS TABLE OF tipoElem [NOT NULL] INDEX BY BINARY_INTEGER;**
  - **miaTabella mioTipoTabella;**
- tipoElem: o scalare, o record di scalari (char, date, number), ottenuto usando %type, %rowtype, dichiarazione di scalare, dichiarazione di record
- una tabella può essere un parametro o il risultato di una funzione
- accesso alle righe: **miaTabella (expr);**
- tabelle di uguale tipo si possono assegnare per intero

Attributi di una tabella

- **EXISTS(i): bool**
- **PRIOR(i), NEXT(i), FIRST, LAST, COUNT: binary_integer**
- Esempio:

```
DECLARE
    i BINARY_INTEGER
BEGIN
    i := tab.FIRST;
    WHILE i IS NOT NULL
    LOOP ..;
    i := tab.NEXT(i);
    END LOOP;
```

- tabella.DELETE, tabella.DELETE(i), tabella.DELETE(i,j)

Uso di una tabella

- Con la clausola SELECT .. INTO tab(i) è possibile creare una riga di una tabella
- Con un loop si può caricare una tabella:

```
DECLARE
    TYPE MioTipoTabella IS TABLE OF emp%ROWTYPE
      INDEX BY ...
    miaTab MioTipoTabella;
    i BINARY_INTEGER := 0;
    CURSOR c IS SELECT * FROM emp;
BEGIN
    OPEN c;
    LOOP
        i:=i+1;
        FETCH c INTO miaTab(i);
        EXIT WHEN c%NOTFOUND;
    END LOOP;
```

Cicli sulle tabelle

- Si può usare un cursore implicito:

```
DECLARE
    TYPE TNomeTab IS TABLE OF emp.nome%TYPE...
    TYPE TSalTab IS TABLE OF emp.sal%TYPE INDEX...
    miaNomeTab TNomeTab ;
    miaSalTab TSalTab ;
    i BINARY_INTEGER := 0;
BEGIN
    FOR imp IN (SELECT nome, sal FROM emp)
    LOOP
        i:=i+1;
        miaNomeTab(i) := imp.nome;
        miaSalTab(i) := imp.sal;
    END LOOP;
END;
```

Tabelle come parametri

- Un parametro tabella non può avere default null, ma:

```
CREATE OR REPLACE PACKAGE pp AS
 TYPE MyTableT IS TABLE OF varchar(80)
              INDEX BY binary_integer;
  myEmptyTable MyTableT;
  PROCEDURE test(
          t MyTableT DEFAULT myEmptyTable
  );
END pp;
```

Collezioni in Oracle 8

- Tipi meno flessibili, ma che i cui valori possono essere memorizzati in una casella del DB
- **Tabelle annidate**: simili alle index-by, ma:
  - Alcune procedure in più (trim,extend)
  - Una nested table vuota è uguale a NULL
  - Una nested table va creata ed estesa in modo esplicito
- **Varray**: simili alle tabelle annidate, ma:
  - Hanno un maximum size
  - Non hanno buchi, ma solo un upper bound
  - Conservano ordine e subscript nel DB

Dichiarazione di Nested Table e Varray

```
TYPE CourseList IS TABLE OF VARCHAR2(10);
TYPE Project IS
      OBJECT(
            project_no NUMBER(2),
            title VARCHAR2(35),
            cost NUMBER(7,2));
TYPE ProjectList IS VARRAY(50) OF Project;
```

Inizializzazione di Nested Table e Varray

- Una Nested table o Varray vale **null** fino a che non viene inizializzata con il costruttore:

```
DECLARE my_courses CourseList;
BEGIN my_courses := CourseList('Econ 2010', 'Acct
  3401', 'Mgmt 3100', 'PoSc 3141', 'Mktg 3312',
  'Engl 2005');
```

- per modificare la dimensione, metodo extend
  - my_courses.extend
  - my_courses.extend(3): aggiunge tre elementi nulli
  - my_courses.extend(3,1): aggiunge tre nuovi elmenti, copie del primo
- **trim** annulla l'effetto di **extend**

Bulk binds

- Per minimizzare i passaggi di controllo tra l'esecutore di statement SQL e quello di statement procedurali
- chiamiamo bind l'assegnazione di valori a variabili PL/SQL all'interno di comandi SQL
- lo statement

  `forall var in e1..e2 sqlstatement`

  viene eseguito in modo molto più efficiente del loop:

  `for var in e1..e2 sqlstatement`
- in modo analogo esistono versioni bulk di select into e di fetch into

Bulk SELECT INTO

```
TYPE MyTable IS TABLE OF char(15)  -- tipo scalare
             index by binary_integer;
i binary_integer;
t myTable;
s myTable;
begin
  SELECT nome, cognome BULK COLLECT INTO t, s
  FROM persone WHERE ROWNUM <= 100;
  i := t.first;
  while (i is not null)
  loop
    …;
    i := t.next(i);
  end loop;
end;
```