Collections

Zusammengesetzte Datentypen

- Bestehen aus mehreren Einzeldaten
- Es gibt 2 Arten:
  - PL/SQL Records
  - PL/SQL Collections
    - Assoziative Arrays (`INDEX BY` table)
    - Nested Table
    - `VARRAY`

PL/SQL Records oder Collections?

- PL/SQL Records speichern verschiedene Datentypen, aber nur einmal.
- PL/SQL Collections speichern viele Werte vom gleichen Typ.

PL/SQL Record:

TRUE	23-DEC-98	ATLANTA	

PL/SQL Collection:

1	SMITH
2	JONES
3	BENNETT
4	KRAMER

VARCHAR2

PLS_INTEGER

Unterschiede der PL/SQL Collection Typen

Collection Type	Elements	Type	Sparse	Created	Type Attribute
Associative array (or index-by table)	Unbounded	String or integer	Either	Only in PL/SQL block	No
Nested table	Unbounded	Integer	Starts dense, can become sparse	Either in PL/SQL block or at schema level	Yes
Variable-size array (varray)	Bounded	Integer	Always dense	Either in PL/SQL block or at schema level	Yes

Associative Arrays (`INDEX BY` Tables)

Ein assoziatives Array besteht aus zwei Spalten (im Tabellensinn):

- Primary key: Integer oder Zeichenkette
- Werte: Skalare Typen oder Records

Key	Values
1	JONES
2	HARDEY
3	MADURO
4	KRAMER

Nutzung von Assoziativen Arrays

```
...
DECLARE
  TYPE ename_table_type IS TABLE OF
    employees.last_name%TYPE
    INDEX BY PLS_INTEGER;
  TYPE hiredate_table_type IS TABLE OF DATE
    INDEX BY PLS_INTEGER;
  ename_table       ename_table_type;
  hiredate_table    hiredate_table_type;
BEGIN
  ename_table(1)    := 'CAMERON';
  hiredate_table(8) := SYSDATE + 7;
    IF ename_table.EXISTS(1) THEN
    INSERT INTO ...
    ...
END;
/
...
```

Einheitliche Methodik für die Collection-Typen

Function or Procedure	Description
EXISTS	Returns TRUE if the nth element in a collection exists; otherwise, EXISTS(N) returns FALSE.
COUNT	Returns the number of elements that a collection contains.
LIMIT	For nested tables that have no maximum size, LIMIT returns NULL; for varrays, LIMIT returns the maximum number of elements that a varray can contain.
FIRST and LAST	Returns the first and last (smallest and largest) index numbers in a collection, respectively.
PRIOR and NEXT	PRIOR(n) returns the index number that precedes index n in a collection; NEXT(n) returns the index number that follows index n.
EXTEND	Appends one null element. EXTEND(n) appends n elements; EXTEND(n, i) appends n copies of the ith element.
TRIM	Removes one element from the end; TRIM(n) removes n elements from the end of a collection
DELETE	Removes all elements from a nested or associative array table. DELETE(n) removes the nth element ; DELETE(m, n) removes a range. **Note:** Does not work on varrays.

INDEX BY Table of Records - 1

Kann Teile einer Tabelle (viele Records) im Speicher halten

```
DECLARE
  TYPE dept_table_type IS TABLE OF
       departments%ROWTYPE INDEX BY PLS_INTEGER;
  dept_table dept_table_type;
  -- Each element of dept_table is a record
Begin
  SELECT * INTO dept_table(1) FROM departments
    WHERE department_id = 10;
  DBMS_OUTPUT.PUT_LINE(dept_table(1).department_id || '
' ||
   dept_table(1).department_name || ' ' ||
   dept_table(1).manager_id);
END;/
```

INDEX BY Table of Records - 2

```
DECLARE
   TYPE emp_table_type IS TABLE OF
      employees%ROWTYPE INDEX BY PLS_INTEGER;
   my_emp_table  emp_table_type;
   max_count         NUMBER(3):= 104;
BEGIN
  FOR i IN 100..max_count
  LOOP
   SELECT * INTO my_emp_table(i) FROM employees
   WHERE employee_id = i;
  END LOOP;
  FOR i IN my_emp_table.FIRST..my_emp_table.LAST
  LOOP
     DBMS_OUTPUT.PUT_LINE(my_emp_table(i).last_name);
  END LOOP;
END;
/
```

Assoziatives Array mit Zeichenkettenindex:

```
TYPE type_name IS TABLE OF  element_type
INDEX BY VARCHAR2(size)
```

```
CREATE OR REPLACE PROCEDURE report_credit
  (p_last_name    customers.cust_last_name%TYPE,
   p_credit_limit customers.credit_limit%TYPE)
IS
  TYPE  typ_name IS TABLE OF customers%ROWTYPE
    INDEX BY customers.cust_email%TYPE;
  v_by_cust_email   typ_name;
  i VARCHAR2(30);

  PROCEDURE load_arrays IS
  BEGIN
    FOR rec IN  (SELECT * FROM customers
                     WHERE cust_email IS NOT NULL)
      LOOP
        -- Load up the array in single pass to database table.
         v_by_cust_email (rec.cust_email) := rec;
      END LOOP;
  END;
```

Traversieren des Array

```
...
BEGIN
  load_arrays;
  i:= v_by_cust_email.FIRST;
  dbms_output.put_line ('For credit amount of: ' || p_credit_limit);
  WHILE i IS NOT NULL LOOP
    IF v_by_cust_email(i).cust_last_name = p_last_name
    AND v_by_cust_email(i).credit_limit > p_credit_limit
      THEN dbms_output.put_line ( 'Customer '||
        v_by_cust_email(i).cust_last_name || ': ' ||
        v_by_cust_email(i).cust_email || ' has credit limit of: ' ||
        v_by_cust_email(i).credit_limit);
    END IF;
    i := v_by_cust_email.NEXT(i);
  END LOOP;
END report_credit;
/
```

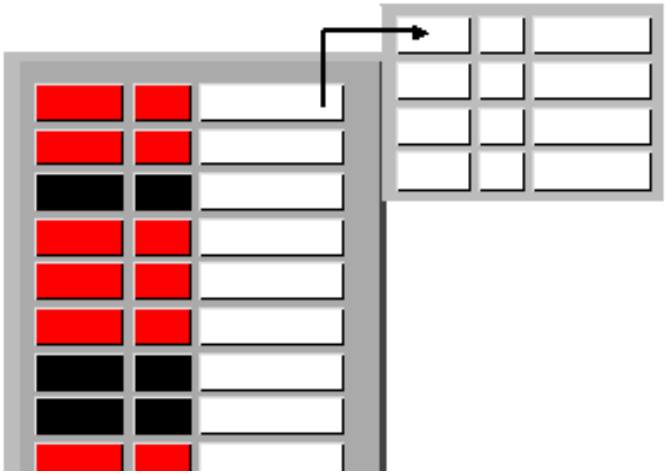
```
EXECUTE report_credit('Walken', 1200)

For credit amount of: 1200
Customer Walken: Emmet.Walken@LIMPKIN.COM has credit limit of: 3600
Customer Walken: Prem.Walken@BRANT.COM has credit limit of: 3700
```

Nested Tables

Eigenschaften:

- Unbounded
- Sowohl in SQL (Datenbank-Typ) als auch PL/SQL nutzbar
  - Als Datenbank-Typ: Eine Tabelle innerhalb einer Tabelle
- Feld-artiger Zugriff auf die einzelnen Zeilen

Nested table:



Erzeugen eines Nested Table

Als Datenbanktyp:

```
CREATE [OR REPLACE] TYPE type_name AS TABLE OF
Element_datatype [NOT NULL];
```

In PL/SQL:

```
TYPE type_name IS TABLE OF element_datatype
[NOT NULL];
```

Erzeugen eines Nested Table

Als Datenbanktyp:

```
CREATE [OR REPLACE] TYPE type_name AS TABLE OF
Element_datatype [NOT NULL];
```

In PL/SQL:

```
TYPE type_name IS TABLE OF element_datatype
[NOT NULL];
```

Nested Table: Beispiel - 1

```
CREATE TYPE typ_item AS OBJECT  --create object
 (prodid  NUMBER(5),
  price   NUMBER(7,2) )
/
CREATE TYPE typ_item_nst -- define nested table type
  AS TABLE OF typ_item
/
```

```
CREATE TABLE pOrder (  -- create database table
     ordid    NUMBER(5),
     supplier NUMBER(5),
     requester       NUMBER(4),
     ordered  DATE,
     items    typ_item_nst)
     NESTED TABLE items STORE AS item_stor_tab
/
```

Nested Table: Beispiel - 2

Hinzufügen von Daten:

```
INSERT INTO pOrder
  VALUES (500, 50, 5000, sysdate, typ_item_nst(
     typ_item(55, 555),
     typ_item(56, 566),
     typ_item(57, 577)));

INSERT INTO pOrder
  VALUES (800, 80, 8000, sysdate,
    typ_item_nst (typ_item (88, 888)));
```

`pOrder` Nested Table

ORDID	SUPPLIER	REQUESTER	ORDERED	ITEMS
500	50	5000	30-OCT-07	
800	80	8000	31-OCT-07	

PRODID	PRICE
55	555
56	566
57	577

PRODID	PRICE
88	888

Nested Table: Beispiel - 3

Abfrage:

```
SELECT * FROM porder;

     ORDID   SUPPLIER  REQUESTER ORDERED
---------- ---------- ---------- ---------
ITEMS(PRODID, PRICE)
--------------------------------------------------------------------
       500         50       5000 31-OCT-07
TYP_ITEM_NST(TYP_ITEM(55, 555), TYP_ITEM(56, 566), TYP_ITEM(57, 577))
       800         80       8000 31-OCT-07
TYP_ITEM_NST(TYP_ITEM(88, 888))
```

Abfrage mit `TABLE` Expression:

```
SELECT p2.ordid, p1.*
FROM porder p2, TABLE(p2.items) p1;

     ORDID     PRODID      PRICE
---------- ---------- ----------
       800         88        888
       500         57        577
       500         55        555
       500         56        566
```

Nested Table: Beispiel in PL/SQL - 1

```
CREATE OR REPLACE PROCEDURE add_order_items
(p_ordid NUMBER, p_new_items typ_item_nst)
IS
  v_num_items      NUMBER;
  v_with_discount  typ_item_nst;
BEGIN
  v_num_items := p_new_items.COUNT;
  v_with_discount := p_new_items; -- Vorsicht Kopie
  IF v_num_items > 2 THEN
  --ordering more than 2 items gives a 5% discount
    FOR i IN 1..v_num_items LOOP
      v_with_discount(i) :=
      typ_item(p_new_items(i).prodid,
               p_new_items(i).price*.95);
    END LOOP;
  END IF;
  UPDATE pOrder
    SET  items = v_with_discount
    WHERE ordid = p_ordid;
END;
```

Nested Table: Beispiel in PL/SQL - 2

```
-- caller pgm:
DECLARE
  v_form_items  typ_item_nst:= typ_item_nst();
BEGIN
  -- let's say the form holds 4 items
  v_form_items.EXTEND(4);
  v_form_items(1) := typ_item(1804, 65);
  v_form_items(2) := typ_item(3172, 42);
  v_form_items(3) := typ_item(3337, 800);
  v_form_items(4) := typ_item(2144, 14);
  add_order_items(800, v_form_items);
END;
```

Erzeugen von Varrays

Als Datenbanktyp:

```
CREATE [OR REPLACE] TYPE type_name AS VARRAY
(max_elements) OF element_datatype [NOT NULL];
```

In PL/SQL:

```
TYPE type_name IS VARRAY  (max_elements) OF
element_datatype [NOT NULL];
```

Varray: Beispiel - 1

```
CREATE TYPE typ_Project AS OBJECT(  --create object
   project_no NUMBER(4),
   title      VARCHAR2(35),
   cost       NUMBER(12,2))
/
CREATE TYPE typ_ProjectList AS VARRAY (50) OF typ_Project
         -- define VARRAY type
/
```

```
CREATE TABLE department (  -- create database table
   dept_id  NUMBER(2),
   name     VARCHAR2(25),
   budget   NUMBER(12,2),
   projects typ_ProjectList)  -- declare varray as column
/
```

Varray: Beispiel - 2

Hinzufügen von Daten:

```
INSERT INTO department
  VALUES (10, 'Executive Administration', 30000000,
     typ_ProjectList(
     typ_Project(1001, 'Travel Monitor',  400000),
     typ_Project(1002, 'Open World',    10000000)));

INSERT INTO department
  VALUES (20, 'Information Technology', 5000000,
     typ_ProjectList(
     typ_Project(2001, 'DB11gR2', 900000)));
```

Varray: Beispiel - 3

Abfrage:

```
SELECT * FROM department;

   DEPT_ID NAME                           BUDGET
---------- ------------------------- ----------
PROJECTS(PROJECT_NO, TITLE, COST)
-------------------------------------------------------------------
        10 Executive Administration    30000000
TYP_PROJECTLIST(TYP_PROJECT(1001, 'Travel Monitor', 400000),
TYP_PROJECT(1002, 'Open World', 10000000))
        20 Information Technology       5000000
TYP_PROJECTLIST(TYP_PROJECT(2001, 'DB11gR2', 900000))
```

Abfrage mit `TABLE` Expression:

```
SELECT d2.dept_id, d2.name, d1.*
FROM department d2, TABLE(d2.projects) d1;

DEPT_ID NAME                      PROJECT_NO TITLE              COST
------- ------------------------- ---------- --------------  --------
     10 Executive Administration       1001 Travel Monitor    400000
     10 Executive Administration       1002 Open World      10000000
     20 Information Technology         2001 DB11gR2           900000
```

Arbeiten mit Collections in PL/SQL

Können als Parameter und Rückgabewerte für Prozeduren und Funktionen genutzt werden.

```
CREATE OR REPLACE PACKAGE manage_dept_proj
AS
  PROCEDURE allocate_new_proj_list
    (p_dept_id NUMBER, p_name VARCHAR2, p_budget NUMBER);
  FUNCTION get_dept_project (p_dept_id NUMBER)
    RETURN typ_projectlist;
  PROCEDURE update_a_project
    (p_deptno NUMBER, p_new_project typ_Project,
     p_position NUMBER);
  FUNCTION manipulate_project (p_dept_id NUMBER)
    RETURN typ_projectlist;
  FUNCTION check_costs (p_project_list typ_projectlist)
    RETURN boolean;
END manage_dept_proj;
```

Initialisierung

3 Möglichkeiten:

- Konstruktor
- Füllen durch Datenbankabfrage (fetch)
- Direkte Zuweisung einer Collection-Variable

```
PROCEDURE allocate_new_proj_list
   (p_dept_id NUMBER, p_name VARCHAR2, p_budget NUMBER)
 IS
   v_accounting_project typ_projectlist;
 BEGIN
 -- this example uses a constructor
   v_accounting_project :=
     typ_ProjectList
       (typ_Project (1, 'Dsgn New Expense Rpt', 3250),
        typ_Project (2, 'Outsource Payroll', 12350),
        typ_Project (3, 'Audit Accounts Payable',1425));
   INSERT INTO department
     VALUES(p_dept_id, p_name, p_budget, v_accounting_project);
 END allocate_new_proj_list;
```

Initialisierung

```
FUNCTION get_dept_project (p_dept_id NUMBER)
   RETURN typ_projectlist
 IS
   v_accounting_project typ_projectlist;
 BEGIN  -- this example uses a fetch from the database
    SELECT  projects INTO  v_accounting_project
      FROM  department WHERE dept_id = p_dept_id;
    RETURN v_accounting_project;
 END get_dept_project;
```

```
FUNCTION manipulate_project (p_dept_id NUMBER)
   RETURN typ_projectlist
 IS
   v_accounting_project typ_projectlist;
   v_changed_list typ_projectlist;
 BEGIN
   SELECT  projects INTO  v_accounting_project
      FROM  department WHERE dept_id = p_dept_id;
 -- this example assigns one collection to another
   v_changed_list := v_accounting_project;
   RETURN v_changed_list;
 END manipulate_project;
```

Referenzierung über den Index

```
-- sample caller program to the manipulate_project function
DECLARE
  v_result_list typ_projectlist;
BEGIN
  v_result_list := manage_dept_proj.manipulate_project(10);
  FOR i IN 1..v_result_list.COUNT LOOP
    dbms_output.put_line('Project #: '
                                 ||v_result_list(i).project_no);
    dbms_output.put_line('Title: '||v_result_list(i).title);
    dbms_output.put_line('Cost: ' ||v_result_list(i).cost);
  END LOOP;

END;
```

```
Project #: 1001
Title: Travel Monitor
Cost: 400000
Project #: 1002
Title: Open World
Cost: 10000000
```

Collection Methoden: Beispiel - 1

Traversierung

```
FUNCTION check_costs (p_project_list typ_projectlist)
    RETURN boolean
  IS
    c_max_allowed        NUMBER := 10000000;
    i                    INTEGER;
    v_flag               BOOLEAN := FALSE;
  BEGIN
    i := p_project_list.FIRST ;
    WHILE i IS NOT NULL LOOP
      IF p_project_list(i).cost > c_max_allowed then
        v_flag := TRUE;
        dbms_output.put_line (p_project_list(i).title || '
                              exceeded allowable budget.');
        RETURN TRUE;
      END IF;
    i := p_project_list.NEXT(i);
    END LOOP;
    RETURN null;
  END check_costs;
```

Collection Methoden: Beispiel - 2

```
-- sample caller program to check_costs
set serverout on
DECLARE
  v_project_list typ_projectlist;
BEGIN
  v_project_list := typ_ProjectList(
    typ_Project (1,'Dsgn New Expense Rpt', 3250),
    typ_Project (2, 'Outsource Payroll', 120000),
    typ_Project (3, 'Audit Accounts Payable',14250000));
  IF manage_dept_proj.check_costs(v_project_list) THEN
    dbms_output.put_line('Project rejected: overbudget');
  ELSE
    dbms_output.put_line('Project accepted, fill out forms.');
  END IF;
END;
```

```
Audit Accounts Payable exceeded allowable budget.
Project rejected: overbudget
```

Verändern individueller Elemente

```
PROCEDURE update_a_project
    (p_deptno NUMBER, p_new_project typ_Project, p_position NUMBER)
  IS
     v_my_projects typ_ProjectList;
  BEGIN
    v_my_projects := get_dept_project (p_deptno);
    v_my_projects.EXTEND;   --make room for new project
    /* Move varray elements forward */
    FOR i IN REVERSE p_position..v_my_projects.LAST - 1 LOOP
      v_my_projects(i + 1) := v_my_projects(i);
    END LOOP;
    v_my_projects(p_position) := p_new_project; -- insert new one
   UPDATE department SET projects = v_my_projects
     WHERE dept_id = p_deptno;
  END update_a_project;
```

Vermeidung von Collection Exceptions

Häufige Exceptions:

- `COLLECTION_IS_NULL`
- `NO_DATA_FOUND`
- `SUBSCRIPT_BEYOND_COUNT`
- `SUBSCRIPT_OUTSIDE_LIMIT`
- `VALUE_ERROR`

Vermeidung von Exceptions: Beispiel

```
DECLARE
  TYPE NumList IS TABLE OF NUMBER;
  nums NumList;          -- atomically null
BEGIN
  /* Assume execution continues despite the raised
exceptions. */
  nums(1) := 1;          -- raises COLLECTION_IS_NULL
  nums := NumList(1,2); -- initialize table
  nums(NULL) := 3        -- raises VALUE_ERROR

  nums(0) := 3;          -- raises
SUBSCRIPT_OUTSIDE_LIMIT
  nums(3) := 3;          -- raises SUBSCRIPT_BEYOND_COUNT
  nums.DELETE(1);        -- delete element 1
  IF nums(1) = 1 THEN    -- raises NO_DATA_FOUND
...
```

Bulk Binding: Syntax

- Das `FORALL` Schlüsselwort sagt der PL/SQL Engine, die Input-Collection zusammen zu fassen, bevor diese an die SQL Engine übergeben wird.

```
FORALL index IN lower_bound .. upper_bound
  [SAVE EXCEPTIONS]
  sql_statement;
```

- Das `BULK COLLECT` Schlüsselwort weist die SQL Engine an, die Ergebnisse zusammen zu fassen, bevor diese an die PL/SQL Engine übergeben werden.

```
... BULK COLLECT INTO
      collection_name[,collection_name] ...
```

Bulk Binding FORALL: Beispiel

```
CREATE PROCEDURE raise_salary(p_percent NUMBER) IS
  TYPE numlist_type IS TABLE OF NUMBER
     INDEX BY BINARY_INTEGER;
  v_id  numlist_type; -- collection
BEGIN
 v_id(1):= 100; v_id(2):= 102; v_id(3):= 104; v_id(4) := 110;
 -- bulk-bind the PL/SQL table
  FORALL i IN v_id.FIRST .. v_id.LAST
    UPDATE employees
      SET salary = (1 + p_percent/100) * salary
      WHERE employee_id = v_id(i);
END;
/
```

```
EXECUTE raise_salary(10)

PL/SQL procedure successfully completed.
```

BULK COLLECT INTO mit Abfragen

The SELECT statement has been enhanced to support the BULK COLLECT INTO syntax.

```
CREATE PROCEDURE get_departments(p_loc NUMBER) IS
  TYPE dept_tab_type IS
    TABLE OF departments%ROWTYPE;
  v_depts dept_tab_type;
BEGIN
  SELECT * BULK COLLECT INTO v_depts
  FROM departments
  WHERE location_id = p_loc;
  FOR i IN 1 .. v_depts.COUNT LOOP
    DBMS_OUTPUT.PUT_LINE(v_depts(i).department_id
     ||' '|| v_depts(i).department_name);
  END LOOP;
END;
```

BULK COLLECT INTO mit Cursor

```
CREATE PROCEDURE get_departments(p_loc NUMBER) IS
  CURSOR cur_dept IS
    SELECT * FROM departments
    WHERE location_id = p_loc;
  TYPE dept_tab_type IS TABLE OF cur_dept%ROWTYPE;
  v_depts dept_tab_type;
BEGIN
  OPEN cur_dept;
  FETCH cur_dept BULK COLLECT INTO v_depts;
  CLOSE cur_dept;
 FOR i IN 1 .. v_depts.COUNT LOOP
    DBMS_OUTPUT.PUT_LINE(v_depts(i).department_id
     ||' '|| v_depts(i).department_name);
  END LOOP;
END;
```

BULK COLLECT INTO mit RETURNING Clause

```
CREATE PROCEDURE raise_salary(p_rate NUMBER) IS
   TYPE emplist_type IS TABLE OF NUMBER;
   TYPE numlist_type IS TABLE OF employees.salary%TYPE
     INDEX BY BINARY_INTEGER;
   v_emp_ids  emplist_type :=
   emplist_type(100,101,102,104);
   v_new_sals numlist_type;
BEGIN
  FORALL i IN v_emp_ids.FIRST .. v_emp_ids.LAST
    UPDATE employees
      SET commission_pct = p_rate * salary
    WHERE employee_id = v_emp_ids(i)

    RETURNING salary BULK COLLECT INTO v_new_sals;
  FOR i IN 1 ..   v_new_sals.COUNT LOOP ...
END;
```

FORALL Support für Sparse Collections

```
-- The new INDICES OF syntax allows the bound arrays
-- themselves to be sparse.

FORALL index_name IN INDICES OF sparse_array_name
      BETWEEN LOWER_BOUND AND UPPER_BOUND -- optional
      SAVE EXCEPTIONS  -- optional, but recommended
      INSERT INTO table_name VALUES
    sparse_array(index_name);
```

```
-- The new VALUES OF syntax lets you indicate a subset
-- of the binding arrays.

FORALL index_name IN VALUES OF index_array_name
      SAVE EXCEPTIONS -- optional,but recommended
      INSERT INTO table_name VALUES
    binding_array_name(index_name);
```

Bulk Bind mit Index Array

```
CREATE OR REPLACE PROCEDURE ins_emp2 AS
  TYPE emptab_type IS TABLE OF employees%ROWTYPE;
  v_emp emptab_type;
  TYPE values_of_tab_type IS TABLE OF PLS_INTEGER
      INDEX BY PLS_INTEGER;
  v_num   values_of_tab_type;
    . . .
BEGIN
    . . .
    FORALL k IN VALUES OF v_num
    INSERT INTO new_employees VALUES v_emp(k);
END;
```