Collections in PL/SQL

Dr. Frank Haney

DOAG-Konferenz 2016, Nürnberg

Vorstellung

Selbständiger Oracle-Berater seit 2002

- OCP DBA
- Oracle University zertifizierter Trainer
- Auditleiter für geprüfte IT-Sicherheit

Administration

- Implementierung und Test
- Hochverfügbarkeit
- Backup und Recovery
- Migration und Upgrade
- Performance Diagnostik und Tuning
- Sicherheit

Applikationsentwicklung

- Design
- **SQL und PL/SQL**
- Oracle Text
- Oracle und XML
- Apex
- Objektrelationale Erweiterungen

Inhalt

- Collections in PL/SQL und SQL - eine Übersicht
- Assoziative Arrays (PL/SQL-Tabellen) - Deklaration und Verwendung
- Nested Tables als Alternative zu PL/SQL-Tabellen
- Nested Tabes im Datenmodell und in SQL
- Die Besonderheiten von VARRAYs (array varying)
- Methoden für PL/SQL-Tabellen
- Datenabfrage und -manipulation mit Collections
- Performanceüberlegungen
- Stärken und Schwächen für verschiedene Einsatzgebiete

Motivation für Collections

- **Erste Normalform im relationalen Datenmodell**: Alle Attributwerte sind atomar, Ausprägungen skalarer Standarddatentypen wie CHAR, INTEGER etc. sein, mengenwertige Attribute, z.B. Listen skalarer Attributwerte, sind nicht erlaubt. Das ist aber nicht immer geschickt, z.B. bei abhängigen Entitäten, die nur im Kontext des Masterdatensatzes existieren: Kinder und ihre Eltern etc.
- **SQL ist mengenorientiert**, d.h. eine Abfrage an die Datenbank liefert im allgemeinen eine Menge von Datensätzen (Tupeln), PL/SQL kann gleichzeitig immer nur einen Datensatz in einem Satz von Variablen speichern. Cursor dienen dazu, eine satzweise Verarbeitung einer Menge von Datensätzen zu ermöglichen. Damit sind zwei Probleme verbunden:
  - Die Abarbeitung erfolgt immer entsprechend dem Resultat der Anweisung, die als Cursor deklariert ist. Ein wahlfreier Zugriff auf einzelne Elemente des Cursor ist nicht vorgesehen.
  - Bei einer satzweisen Verarbeitung wechselt die Verarbeitung auf kleiner Granularität zwischen SQL- und PL/SQL-Engine. Das kann erhebliche Performanceprobleme mit sich bringen.

Collections

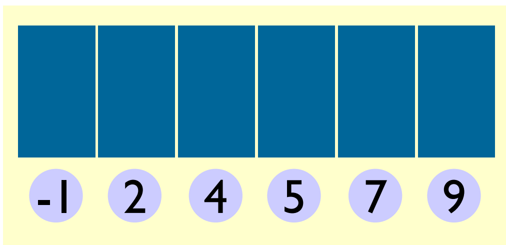
- In Collections sind Elemente desselben Typs zusammengefasst.
- Collections sind mengenwertigen Datentypen vergleichbar.
- Collections können Instanzen eines Objekttyps speichern und umgekehrt die Attribute eines Objekttyps sein.

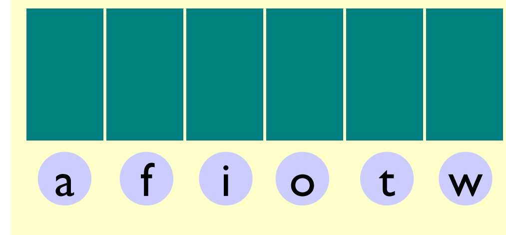
Collection-Typ	**PL/SQL-Tabelle** (Assoziatives Array)	**VARRAY**	**Nested Table**
Möglich in PL/SQL	ja	ja	ja
Als Schemaobjekt möglich	nein	ja	ja
Als Objektdatentyp möglich	nein	ja (in SQL)	ja (in SQL)
Speicherung	nicht persistent	persistent online (in SQL)	persistent offline (in SQL)
Anzahl der Elemente	offen	definiert	offen
Dimensionalität	eindimensional	eindimensional	eindimensional
Datendichte (Besetzung der Elemente) – engl. sparsity	dünn	immer dicht	dicht bis dünn
Indizierung	explizit	implizit	implizit

Collection-Typen

Assoziatives Array

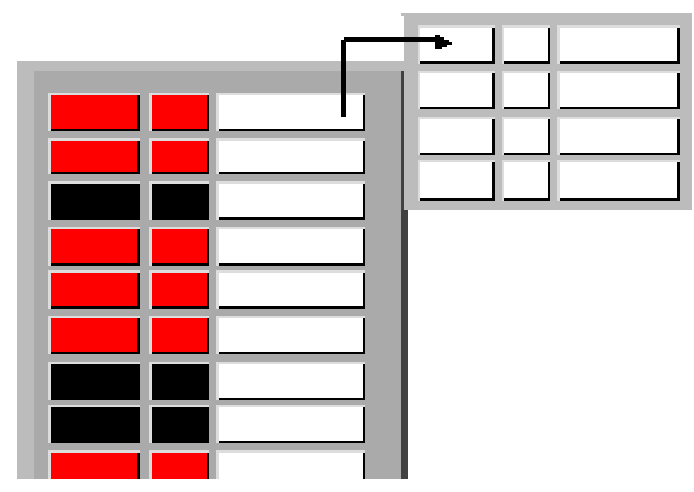
Durch
PLS_INTEGER
indiziert



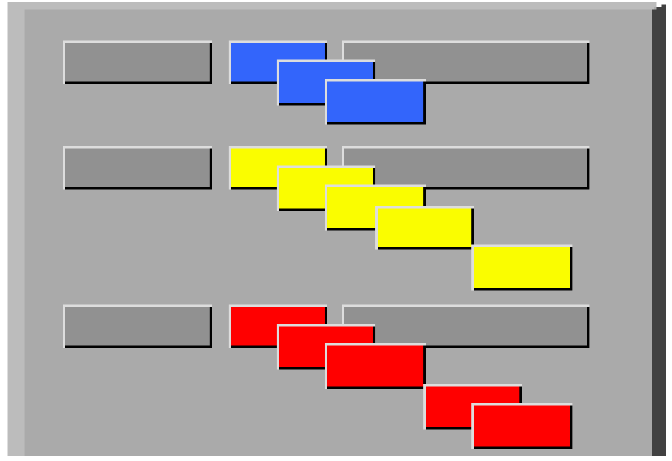


Durch
VARCHAR2
indiziert

Nested Table

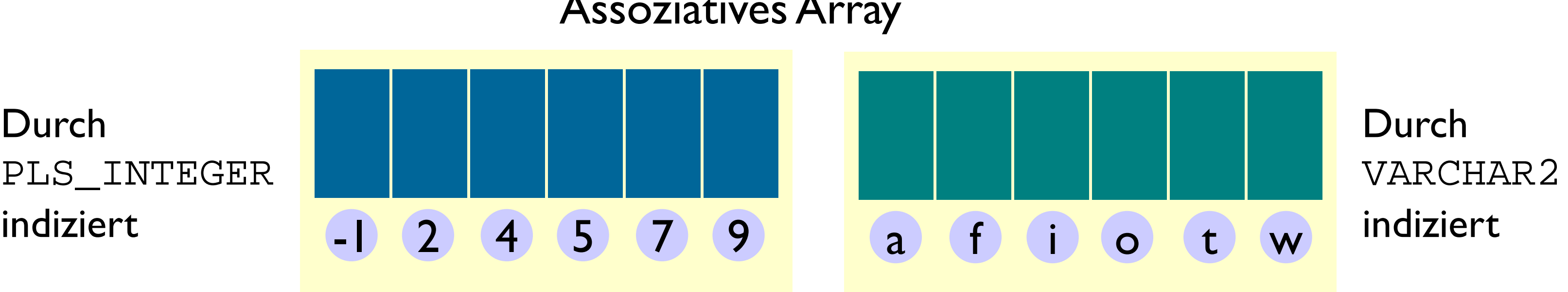


Varray



PL/SQL-Tabellen

- können die Performance der Verarbeitung verbessern
- sind reine Speicherstrukturen, die wesentlich schneller sind als Tabellen auf Schemaebene
- bieten sehr viel Flexibilität, z.B. wahlfreie Belegung der und Zugriff auf die Elemente
- bieten keine persistente Speicherung von Daten



Deklaration von PL/SQL-Tabellen

Eine PL/SQL-Tabelle (Objekt vom Typ TABLE – Index-by-table)

- ist einem Array ähnlich (auch assoziatives Array genannt) und
- muß zwei Komponenten enthalten:
  - einen Primärschlüssel des Datentyps `PLS_INTEGER` oder `VARCHAR2` (ab Oracle 9*i*), der die PL/SQL-Tabelle indiziert
    - Ein numerischer Index kann negativ sein
    - Bei Zeichenkettenindex Abhängigkeit von `NLS_SORT` und `NLS_COMP`
  - eine Spalte eines skalaren oder Recorddatentyps, welche die Elemente der PL/SQL-Tabelle speichert
- Der Typdeklaration muß die Deklaration einer Variable des Typs folgen.

```
TYPE typ_name IS TABLE OF          --Typdeklaration
      {datentyp | objekt_typ | variable%TYPE |
      tabelle.spalte%TYPE | record_type |
      tabelle%ROWTYPE | record%TYPE} [NOT NULL]
      INDEX BY {PLS_INTEGER | VARCHAR2(n)};
identifier   typ_name;             --Typzuweisung, keine Initialisierung
```

Beispiele von PL/SQL-Tabellen

- Tabellenspalte:

```
TYPE name_table_type IS TABLE OF mitarbeiter.name%TYPE
       INDEX BY BINARY_INTEGER;
name_table name_table_type;
```

- Record:

```
TYPE abt_rec_type IS RECORD
       ( abtnr abteilungen.abteilungs_nr%TYPE,
       abtname   abteilungen.abteilungsname%TYPE,
       ort abteilungen.ort%TYPE);
TYPE abt_table_type IS TABLE OF abt_rec_type
       INDEX BY BINARY_INTEGER;
abt_table abt_table_type;
```

- Record als ROWTYPE:

```
TYPE mit_table_type IS TABLE OF mitarbeiter%ROWTYPE
       INDEX BY PLS_INTEGER;
mit_table   mit_table_type;
```

PL/SQL-Tabellen füllen

```
-- direkte Zuweisung (Achtung: keine dichte Füllung)
DECLARE
TYPE mit_table_type IS TABLE OF mitarbeiter%ROWTYPE INDEX BY PLS_INTEGER;
mit_tab mit_table_type;
BEGIN
SELECT * INTO mit_tab(-312) FROM mitarbeiter WHERE mitarbeiter_nr=1001;
SELECT * INTO mit_tab(5) FROM mitarbeiter WHERE mitarbeiter_nr=1002;
END;

-- mit Schleife (zunächst dichte Füllung)
DECLARE
TYPE mit_table_type IS TABLE OF mitarbeiter%ROWTYPE INDEX BY PLS_INTEGER;
mit_tab mit_table_type;
BEGIN
FOR mit_rec IN (SELECT * FROM mitarbeiter) LOOP
mit_tab(mit_rec.mitarbeiter_nr) := mit_rec;
END LOOP;
END;
/
```

Referenzierung von PL/SQL-Tabellen

- Basierend auf Tabellenspalte:

identifier(index)

z.B.

```
SELECT name INTO name_table(3) FROM mitarbeiter
      WHERE mitarbeiter_nr=1003;
v_name := name_table(3);
```

- Basierend auf Record:

identifier(index).field

z.B.

```
SELECT * INTO abt_table(1) FROM abteilungen
      WHERE ort='BONN';
v_ort := abt_table(1).ort
```

Methoden für Collections

Methode	Beschreibung
EXISTS(n)	Gibt TRUE zurück, wenn das Element mit dem Index *n* einer Collection existiert.
COUNT	Gibt die aktuelle Anzahl der Elemente der Collection zurück.
FIRST oder LAST	Gibt den ersten oder letzten (niedrigsten oder höchsten) Index einer Collection zurück, oder NULL, wenn die Collection leer ist.
PRIOR(n)	Gibt den Index zurück, der vor dem Index *n* in der Collection liegt.
NEXT(n)	Gibt den Index zurück, der nach dem Index *n* in der Collection liegt.
DELETE	DELETE löscht alle Elemente. DELETE(n) löscht das Element mit dem Index *n*. DELETE(m, n) löscht alle Elemente im Bereich *m* bis *n*.
EXTEND(n)	Allokieren von neuen Zeilen
TRIM(n)	Entfernt am Ende ein bzw. n Element(e). (erneute Zuweisung nur nach vorherigem EXTEND möglich)

Syntax: `identifier.methode[(Parameter)]`

Tabellendaten verarbeiten

```
-- geht nicht weil keine dichte Füllung (NO_DATA_FOUND)
DECLARE
TYPE mit_table_type IS TABLE OF mitarbeiter%ROWTYPE INDEX BY PLS_INTEGER;
mit_tab mit_table_type;
BEGIN
SELECT * INTO mit_tab(-312) FROM mitarbeiter WHERE mitarbeiter_nr=1001;
SELECT * INTO mit_tab(5) FROM mitarbeiter WHERE mitarbeiter_nr=1002;
FOR i in mit_tab.FIRST..mit_tab.LAST
DBMS_OUTPUT.PUT_LINE(mit_tab(i).name);
END LOOP;
END;
/

-- aber
...
i := mit_tab.FIRST;
WHILE i IS NOT NULL LOOP
DBMS_OUTPUT.PUT_LINE(mit_tab(i).name);
i:=mit_tab.NEXT(i);
END LOOP;
END;
/
```

PL/SQL-Tabelle mit Zeichenkette indizieren

Assoziatives Array in PL/SQL (durch Zeichenfolge indiziert):

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

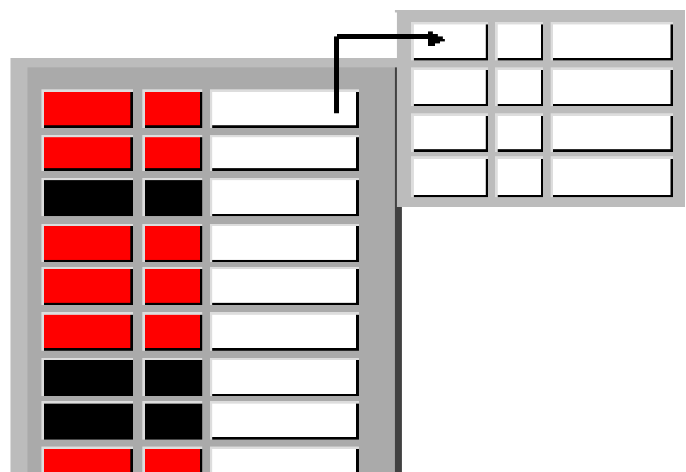
```
DECLARE
TYPE bevoelkerung IS TABLE OF NUMBER INDEX BY VARCHAR2(64);
stadt_bevoelkerung bevoelkerung;
i VARCHAR2(64);
BEGIN
    stadt_bevoelkerung('Nuernberg') := 509975;
    stadt_bevoelkerung('Jena') := 108207;
    stadt_bevoelkerung('Berlin') := 3520031;
    stadt_bevoelkerung('Jena') := 109527;
    i := stadt_bevoelkerung.FIRST;

WHILE i IS NOT NULL LOOP
    DBMS_OUTPUT.PUT_LINE ('Die Bevoelkerung von ' || i || ' ist ' ||
    stadt_bevoelkerung(i));
    i := stadt_bevoelkerung.NEXT(i);
END LOOP;
END;
/
```

Nested Tables

Eigenschaften:

- Eine Tabelle in einer Tabelle
- Ohne festes Format
- Sowohl in PL/SQL als auch als Datenbankobjekt verfügbar
- Array-ähnlicher Zugriff auf einzelne Zeilen

Nested Table:



Speicherung von Nested Tables

Nested Tables werden bei Verwendung als Schemaobjekt out-of-line in Speichertabellen gespeichert.

`pOrder` Nested Table:

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

Speichertabelle:

NESTED_TABLE_ID	PRODID	PRICE
←	55	555
←	56	566
←	57	577

NESTED_TABLE_ID	PRODID	PRICE
←	88	888

Typ für Nested Tables erstellen

Einen Nested Table Type in der Datenbank erstellen:

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

Einen Nested Table Type in PL/SQL erstellen:

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

Unterschied zu asoziativen Arrays: Keine explizite Angabe eines Index.

Nested Tables in SQL

- Zuerst einen Objekttyp definieren:

```
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
/
```

1

2

- Dann eine Spalte dieses Collection-Typs deklarieren:

```
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
/
```

3

Einfügen in Nested Tables

Verwendung der impliziten Konstruktormethode

```
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)));
```

1

2

pOrder Nested Table

ORDID	SUPPLIER	REQUESTER	ORDERED	ITEMS
500	50	5000	30-OCT-07	→ 1
800	80	8000	31-OCT-07	→ 2

1

PRODID	PRICE
55	555
56	566
57	577

2

PRODID	PRICE
88	888

Nested Tables in SQL abfragen

- Inhalte abfragen (Konstruktor):

```
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))
```

- Inhalte mit dem `TABLE`-Operator abfragen (unnesting):

```
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 Tables in PL/SQL

```
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;
  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;
```

Zuweisen auf Nested Tables in PL/SQL

```
DECLARE
  v_form_items  typ_item_nst:= typ_item_nst();
BEGIN
  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;
```

Variable `v_form_items`

PRODID	PRICE
1804	65
3172	42
3337	800
2144	14

Resultierende Daten in der `pOrder` Nested Table

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

PRODID	PRICE
1804	65
3172	42
3337	800
2144	14

Mengenoperationen mit Nested Tables

Nested Tables erlauben sogenannte *Multimengenoperationen*. Damit können Collections dieses Typs verglichen, Untermengen festgestellt, Duplikate ermittelt, bzw. eliminiert und die übliche Vereinigungs-, Durchschnitts- und Differenzbildung vollzogen werden. Beispiele sind:

- `MULTISET UNION [DISTINCT]`
- `MULTISET INTERSECT [DISTINCT]`
- `MULTISET EXCEPT [DISTINCT]`
- `SET`

VARRAYs anlegen

- Ein Varray als Datenbankobjekt erstellen:

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

- Ein Varray in PL/SQL erstellen:

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

VARRAYs müssen bei Bedarf explizit erweitert werden (Änderung von *size_limit*):

```
ALTER TYPE type_name LIMIT (new_size_limit);
```

VARRAYs als Datenbankobjekt verwenden

- Ein Varray als Datenbankobjekt erstellen:

```
CREATE OR REPLACE TYPE kinder AS VARRAY(10) OF VARCHAR2(10);
```

- Tabelle mit einer Spalte vom Typ des VARRAYs anlegen:

```
CREATE TABLE eltern(id NUMBER, vater VARCHAR2(20), mutter
VARCHAR2(20), sproesslinge kinder);
```

- Daten einfügen (Konstruktor):

```
INSERT INTO eltern VALUES (1, 'anton', 'baerbel',
kinder('claus', 'daniel'));

INSERT INTO eltern VALUES (2, 'emil', 'fritzi', kinder('gero',
'hanni', 'ilse', 'jens'));

COMMIT;
```

Das VARRAY als Tabellenspalte

Die Tabelle Eltern:

ID	Vater	Mutter	Sproesslinge
1	anton	bärbel	claus daniel
2	emil	fritzi	gero hanni ilse jens

Daten ändern (VARRAY kann nur als Ganzes geändert werden.):

```
UPDATE eltern SET sproesslinge=kinder('claus', 'daniel',
'nanni', 'olga') where vater='anton';
COMMIT;
```

VARRAYs abfragen

- Inhalte abfragen (Konstruktornotation):

```
SELECT * FROM eltern;
SELECT sproesslinge FROM eltern;
```

- Auflösung des Konstruktors und **Unnesting mit dem `TABLE`-Operator:**

```
SELECT e.id, e.vater, e.mutter, s.*
FROM eltern e, TABLE(e.sproesslinge) s;

SELECT e.id, e.vater, e.mutter, s.COLUMN_VALUE
FROM eltern e, TABLE(e.sproesslinge) s;

SELECT e.id, e.vater, e.mutter, s.COLUMN_VALUE
FROM eltern e, TABLE(e.sproesslinge) s
WHERE s.COLUMN_VALUE = 'claus';
```

Hinweis: Das Unnesting funktioniert de facto wie ein Join. Der Operator `COLUMN_VALUE` geht nur bei skalaren ARRAY-Feldern.

VARRAYs in PL/SQL

```
DECLARE
  TYPE quartett IS VARRAY(4) OF VARCHAR2(15);  -- VARRAY type
  -- varray variable mit konstruktor initialisieren:
  doppelkopfrunde quartett := quartett('Fritz', 'Franz', 'Albert', 'Hans');
  PROCEDURE print_spieler (ueberschrift VARCHAR2) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE(ueberschrift);
    FOR i IN 1..4 LOOP
      DBMS_OUTPUT.PUT_LINE(i || '.' || doppelkopfrunde(i));
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('---');
  END;
BEGIN
  print_spieler ('Spieler 2014:');
  -- zwei Element aendern (geht nicht in SQL)
  doppelkopfrunde(3) := 'Paul';
  doppelkopfrunde(4) := 'Yvonne';
  print_spieler('Spieler 2015:');
  -- mit dem konstruktor das komplette varray aendern:
  doppelkopfrunde := quartett('Hinz', 'Franz', 'Kunz', 'Yvonne');
  print_spieler('Spieler 2016:');
END;
/
```

Collections in PL/SQL-Packages verwenden

Man kann

- Collections als formale Parameter von Prozeduren und Funktionen deklarieren.
- Collections auf Collections zuweisen
- in der RETURN-Klausel einer Funktionsspezifikation einen Collection-Typ angeben.
- Collections in der RETURNING-Klausel von DML verwenden.

Dabei gelten für Collections die üblichen Regeln hinsichtlich Erstellung und Geltungsbereich von Variablen.

Exceptions bei Collections

Häufige Exceptions bei Collections:

- COLLECTION_IS_NULL
- NO_DATA_FOUND
- SUBSCRIPT_BEYOND_COUNT
- SUBSCRIPT_OUTSIDE_LIMIT
- VALUE_ERROR

Exceptions vermeiden

```
DECLARE
  TYPE NumList                IS TABLE OF NUMBER;
  nums NumList;             -- atomically null
BEGIN
  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
...
```

Effiziente Verarbeitung von Collections

- Das Problem der auf kleiner Granularität zwischen SQL und PL/SQL wechselnden Verarbeitung ist im Vorstehenden noch offen geblieben, für die Performance haben wir noch nichts erreicht. Mit jedem FETCH oder UPDATE einer Zeile wechsslt der Kontext von SQL zu PL/SQL und zurück.
- Die Lösung: BULK-Operationen
  - BULK COLLECT für das Füllen von Collections aus der Datenbank (SELECT)
  - FOR ALL für das Schreiben in die Datenbank (DML)

BULK COLLECT

- Syntax:
  `... BULK COLLECT INTO collection ...`
- Verwendbar in
  - `SELECT INTO` (impliziter Cursor)
  - `FETCH INTO` (expliziter Cursor)
  - `RETURNING INTO`
- Beispiel
  ```
  SELECT * BULK COLLECT INTO mit_tab FROM
  mitarbeiter;
  ```
- Bemerkungen
  - Anzahl der zu verarbeitenden Zeilen läßt sich mit `LIMIT n` begrenzen
  - Eine Cursor-FOR-Schleife wird seit Oracle 10g implizit wie ein `BULK COLLECT` verarbeitet.

FOR ALL

- Syntax
```
FORALL index IN untergrenze .. Obergrenze [SAVE
EXCEPTIONS] dml_statement;
```
- Beispiel:
```
FORALL i IN mit_tab.FIRST .. mit_tab.LAST
INSERT INTO mitarbeiter_neu VALUES
(mit_tab(index));
```
- Bemerkungen:
  - Das DML-Statement muß ein einzelnes Statement (`INSERT`, `UPDATE`, `DELETE` oder `MERGE`) sein.
  - Index ist implizit als `PLS_INTEGER` deklariert.
  - Für das Beispiel muß die Collection dicht gefüllt sein, sonst `ORA-22160`
  - `SAVE EXCEPTIONS` speichert alle vorkommenden Exceptions. Die Verarbeitung wird nach dem Fehler fortgesetzt.
- Es gibt zwei zusätzliche Cursor-Attribute:
  - `SQL%BULK_ROWCOUNT`: verarbeitete Zeilen
  - `SQL%BULK_EXCEPTIONS`: Pseudo-Collection mit den aufgetretenen Fehlern

Collections und die PGA

- Die Verarbeitung von Collections in PL/SQL erfolgt in der PGA und kann nicht ausgelagert werden wie Sortierung oder temporäre Tabellen.
- Extensive Verwendung solcher Collections kann die PGA-Allokierung weit über `PGA_AGGREGAT_TARGET` hinaustreiben und zu Swapping führen.
- Mit dem neuen Parameter `PGA_AGGREGATE_LIMIT` kommt es eventuell zu einer Fehlermeldung und Abbruch der Verarbeitung.
- Maßnahmen:
  - Keine zu großen PL/SQL-Tabellen
  - Klausel `LIMIT n` bei `BULK COLLECT`
  - Permanente Objekte für die Speicherung

Richtlinien für die Verwendung

- Assoziative Arrays
  - wahlfreie Zuweisung auf Indexschlüssel, bzw. negative Indexschlüssel
  - dünne Besetzung der Collection erforderlich
  - Elemente ohne `EXTEND` hinzufügen
- VARRAYs
  - wenn das Array zusammen mit den relationalen Daten in einen Block paßt, sonst Verkettung
  - für eine limitierte Anzahl von Einträgen
  - um die Reihenfolge von Elementen in der Collection-Spalte beizubehalten
  - lassen in SQL keine elementbezogenen Updates zu
- Nested Tables
  - für große Datenmengen
  - wenn Mengenoperationen nötig sind

Weiterlesen

- Dokumentation
  - 2 Day Developer's Guide
  - Development Guide
  - PL/SQL Language Reference
  - PL/SQL Packages and Types Reference
- Literatur
  - Steven Feuerstein, Bill Pribyl: Oracle PL/SQL Programming
  - Steven Feuerstein, Bill Pribyl: Oracle PL/SQL – kurz&gut
  - Steven Feuerstein: Oracle PL/SQL Best Practices
  - Jürgen Sieben: Oracle PL/SQL: Das umfassende Handbuch

Vielen Dank!

Q & A

Dr. Frank Haney
info@haney.it
Tel.: 03641-210224