Collections in PL/SQL

Dr. Frank Haney
Consultant
Jena

Schlüsselworte:

Datenbankentwicklung, PL/SQL, Collections

Zum Gegenstand

Mit der Version 7.0 (1992!) hat Oracle in Ergänzung der skalaren mengenwertige Variablen eingeführt, seinerzeit PL/SQL-Tabellen genannt, zeitgleich mit Stored Procedures und Triggern. Die Funktionalität wurde von Release zu Release immer weiter ausgebaut. Jetzt spricht man von Collections, von denen es mehrere Typen gibt, die sich teilweise auch bei der Datenmodellierung und damit auch in SQL einsetzen lassen. Trotz der langen Geschichte und der unabweisbaren Vorteile ihrer Verwendung ist das Wissen darum nicht besonders weit verbreitet, selbst bei Entwicklern. Immer wieder sieht man in der Praxis Code, der sich mit Collections eleganter und vor allem auch performanter lösen ließe. Der Vortrag möchte den Gegenstand stärker in den Fokus rücken und das Wissen darum auf den aktuellen Stand bringen helfen.

Warum Collections?

Jeder, der beginnt, sich mit relationalen Datenbanken zu beschäftigen wird als eines ihrer Grundprinzipien die Erste Normalform kennenlernen: Alle Attributwerte sollen atomar, Ausprägungen skalarer Standarddatentypen wie `CHAR`, `INTEGER` etc. sein. Mengenwertige Attribute, z.B. Listen skalarer Attributwerte, sind verboten. Aufgelöst wird das, indem für jedes skalare Element des mengenwertigen Attributs der Datensatz dupliziert wird. Im weiteren Fortgang der Normalisierung führt das dann zu über Fremdschlüsselbeziehungen verbundene Relationen (Tabellen). Das ist aber aus der Sicht der Datenmodellierung nicht immer geschickt, z.B. bei abhängigen Entitäten, die nur im Kontext des Masterdatensatzes existieren: Kinder und ihre Eltern, Kontoverbindungen eines Kunden etc. Das hat dazu geführt, daß moderne relationale Datenbankmanagementsysteme in der Regel neben den skalaren mengenwertige Datentypen (Collections) vorsehen, die dann auch vom jeweiligen SQL-Dialekt unterstützt werden.

In PL/SQL ist die Motivation etwas anders gelagert: SQL ist mengenorientiert, d.h. eine Abfrage an die Datenbank liefert im allgemeinen eine Menge von Datensätzen (Tupeln), wogegen PL/SQL als imperative Programmiersprache gleichzeitig immer nur einen Datensatz in einem Satz von Variablen oder einem Record (zusammengesetzter Datentyp) speichern kann. Cursor sind eingeführt worden, um 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 Cursors 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.

Für beide Probleme bieten Collections die Lösung.

Nach diesen allgemeinen Bemerkungen sollen jetzt die verschiedenen Möglichkeiten, Collections zu verwenden zunächst vorgestellt werden.

Übersicht der von Oracle angebotenen Collections

Es gibt drei verschiedene Arten von Collections, deren Eigenschaften die folgende Tabelle zusammenfasst:

Collection-Typ	PL/SQL-Tabelle	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)	dünn	dicht	dicht bis dünn
Indizierung	explizit	implizit	implizit

Im folgenden wollen wir uns schwerpunktmäßig mit der Verwendung in PL/SQL beschäftigen und nur am Rande auf die Verwendung als Schemaobjekt eingehen.

Bei der Verwendung der verschiedenen Arten von Collections in PL/SQL ist gemeinsam, daß immer erst ein entsprechender Typ definiert werden muss. Dann wird eine Variable des entsprechenden Typs deklariert, die im Ausführungsteil des Codes dann die Instanzen des jeweiligen Typs als gleichartige Elemente enthält. Jeder Instanz innerhalb der Collection korreliert implizit oder explizit ein Indexwert. Gemeinsam ist auch die Eindimensionalität. Es gibt keine mehrdimensionalen Collections. Collections als Elemente von Collections sind aber möglich, d.h. das Element einer Collection muss nicht zwangsläufig skalar sein.

Wo können Collections im PL/SQL-Code verwendet werden:
- Reguläre PL/SQL-Variable
- Prozedurparameter (IN und OUT)
- Feld eines Record
- Return eines Funktionsaufrufs
- RETURNING-Klausel einer DML-Anweisung

Assoziative Arrays (PL/SQL-Tabellen)

PL/SQL-Tabellen werden im Deklarationsteil deklariert und im Anweisungsteil verwendet. Die allgemeine Syntax ist:

Deklaration des Tabellentyps:

```
TYPE type_name IS TABLE OF
{datatype | objekt_type | variable%TYPE | table.column%TYPE |
record_type | table%ROWTYPE | record%TYPE} [NOT NULL]
INDEX BY {BINARY_INTEGER | VARCHAR2(n)};
```

Deklaration einer konkreten Tabellenvariable des zuvor deklarierten Typs:

```
v_table   type_name;
```

Anmerkung: Anstelle von `BINARY_INTEGER` kann auch einer der Subtypen wie `PLS_INTEGER` etc. verwendet werden. Für `VARCHAR2(n)` kann auch ein entsprechend deklarierter Typ, z.B. `table.column%TYPE` stehen.

Referenziert werden die Elemente (Zeilen) der PL/SQL-Tabelle über den Index, z.B. mit

```
v_table(index) := value;
```

wobei `index` je nach Art der Indizierung numerisch oder eine Zeichenkette sein kann.
Natürlich kann man auch umgekehrt einer skalaren Variablen den Wert einer bestimmten Zeile der PL/SQL-Tabelle zuweisen:

```
variable := v_table(index)
```

Wenn die Elemente der PL/SQL-Tabelle Records, also zeilenwertig sind, muß zusätzlich die Spalte, das Feld, angegeben warden, außer man referenziert den ganzen Record:

```
v_table(index).field := value;
```

In PL/SQL-Tabellen kann man Daten aus der Datenbank zur Weiterverarbeitung laden oder umgekehrt mit deren Inhalten manipulieren. Für eine einzelne Tabellenzeile sieht das dann am Beispiel so aus:

```
SELECT name INTO name_table(3) FROM mitarbeiter WHERE mitar-
beiter_nr=1003;
```

Allerdings muss sichergestellt werden, daß die Abfrage genau eine Zeile liefert, weil man sonst die Standard-Exceptions `TOO_MANY_ROWS` oder `NO_DATA_FOUND` bekommt. Falls das nicht geht, muss ein Cursor verwendet und die PL/SQL-Tabelle in einer Schleife befüllt werden:

```
declare
     type mit_t is table of mitarbeiter%rowtype index by
     pls_integer;
     mit_tab mit_t;
     cursor mit_curs is select * from mitarbeiter;
     zahl number;
begin
     open mit_curs;
```

```
     select count(*) into zahl from mitarbeiter;
     for i in 1..zahl loop
          fetch mit_curs into mit_tab(i);
          dbms_output.put_line(mit_tab(i).name);
     end loop;
     close mit_curs;
end;
/
```

Man sieht hier sehr schön, wie ineffizient dieses Verfahren für größere Datenmengen ist, weil mit jedem Schleifendurchlauf die Abarbeitung zwischen SQL und PL/SQL wechselt. Wie dem abzuhelfen ist, werden wir später sehen.
Es ist nicht notwendig, wie im Beispiel die Elemente des assoziativen Arrays fortlaufend mit Daten aus der Datenbank zu befüllen, da dieses ja nicht dicht besetzt sein muss. Die einzelnen Datensätze kann man völlig wahlfrei beliebigen Indexwerten zuordnen.

Methoden für assoziative Arrays

Um effizient und fehlerfrei auf nicht dicht besetzte PL/SQL-Tabellen zugreifen zu können, sind verschiedene Methoden implementiert:

Methode	**Beschreibung**
EXISTS(n)	Gibt TRUE zurück, wenn das Element mit dem Index *n* einer PL/SQL-Tabelle existiert.
COUNT	Gibt die aktuelle Anzahl der Elemente der PL/SQL-Tabelle zurück.
FIRST oder LAST	Gibt den niedrigsten oder höchsten verwendeten Index einer PL/SQL-Tabelle zurück, oder NULL, wenn die PL/SQL-Tabelle leer ist.
PRIOR(n) oder NEXT(n)	Gibt den (besetzten) Index zurück, der vor bzw. nach dem Index *n* in der PL/SQL-Tabelle liegt.
DELETE	DELETE löscht alle Elemente aus der PL/SQL-Tabelle. DELETE(n) löscht das Element mit dem Index *n* aus der PL/SQL-Tabelle. DELETE(m, n) löscht alle Elemente aus der PL/SQL-Tabelle deren Index von *m* bis *n* reicht.
TRIM(n)	Entfernt am Ende ein bzw. n Element(e).

Verwendet werden die Methoden allgemein so:

`v_table.method`

z.B. `mit_tab.last` oder `mit_tab.delete(2,5)`

Noch ein Beispiel für die Methodenverwendung: Der folgende Code-Ausschnitt würde zu einer Fehlermeldung führen, weil die Collection nicht mehr dicht besetzt ist.

```
mit_tab.delete(2,5);
for i in mit_tab.first .. mit_tab.LAST loop
dbms_output.put_line(mit_tab(i).name);
end loop;
```

Dem lässt sich mit der Methode `NEXT` abhelfen, weil die immer zum nächsten besetzten Index springt.

```
mit_tab.delete(2,5);
zahl := mit_tab.first;
loop
exit when zahl is null;
dbms_output.put_line(mit_tab(zahl).name);
zahl := mit_tab.next(zahl);
end loop;
```

Effizientes Verarbeiten von Daten mit BULK Binding

Es wurde bereits angesprochen, dass eine Datenverarbeitung, bei der auf kleiner Granularität die Abarbeitung zwischen SQL und PL/SQL wechselt, negative Auswirkungen auf die Performance haben kann. Um das effizienter zu gestalten wurden zwei Konstrukte eingeführt:

- `BULK COLLECT` für das Laden von Daten aus der Datenbank in Collections
- `FORALL` für das Durchführen von DML-Operationen mit den Inhalten von Collections

Verwendet werden diese Konstrukte folgendermaßen:

`BULK COLLECT`
mit implizitem Cursor
`SELECT … BULK COLLECT INTO v_table FROM table …;`
mit explizitem Cursor
`FETCH v_curs BULK COLLECT INTO v_table;`
Das ist schon alles. Wir haben nur einen Kontextwechsel zwischen SQL und PL/SQL. Unter Performancegesichtspunkten muß aber noch beachtet werden, dass das Befüllen von PL/SQL-Tabellen eine Operation ist, die im Speicher gehalten werden muss. Im Gegensatz zu temporären Tabellen kann das nicht in den temporären Tablespace ausgelagert werden. Wenn also die PL/SQL-Tabellen zu groß werden und gleichzeitig nicht genügend PGA zur Verfügung steht, kann es zu Performanceproblemen kommen. Bei Verwendung des Parameter `PGA_AGGREGATE_LIMIT` (neu in Oracle 12*c*) gibt es keine Performanceprobleme, sondern eine Fehlermeldung, wenn diese Grenze für die Summe aller PGAs erreicht ist. Mit der zusätzlichen Klausel `LIMIT(N)` lässt sich aber die Anzahl der Zeilen begrenzen, die in einem `FETCH` in die PL/SQL-Tabelle geladen werden.

`FORALL`
Mit dieser Klausel wird eine Menge von Elementen einer Collection an die SQL-Engine übergeben. Die Verwendung geht so:
`FOR ALL index IN untergrenze .. obergrenze`

gefolgt von einem INSERT, UPDATE oder DELETE unter Verwendung der Collection.

Ein vereinfachtes Beispiel ist:

```
FORALL i IN v_mit.FIRST..v_mit.LAST
INSERT INTO mitarbeiter VALUES (v_mit(i).mitarbeiter_nr,
v_mit(i).name, …);
```

Diese Konstruktion geht allerdings davon aus, daß die Collection dicht besetzt ist. Ansonsten gibt es eine Fehlermeldung. Um dem abzuhelfen, gibt es zwei Möglichkeiten:

- Mit der Klausel `SAVE EXCEPTIONS` wird die Verarbeitung nach einem Fehler fortgesetzt. Die Fehlermeldungen werden in dem mengenwertigen Cursor-Attribut `SQL%BULK_EXCEPTIONS` gespeichert.
- Mit den Klauseln `INDICES OF` oder `VALUES OF` lässt sich die Verarbeitung auf wirklich vorhandene Elemente eingrenzen. Das soll hier aus Platzgründen nicht weiter ausgeführt werden.

Nested Tables

Dieser Typ von Collections ist sowohl als Schemaobjekt als auch als PL/SQL-Konstrukt möglich. Als Schemaobjekt wird er folgendermaßen deklariert:

```
CREATE OR REPLACE TYPE nt_type AS TABLE OF ele-
ment_declaration;
```

Dieser tabellenwertige Typ kann nun bei Schematabellen als Datentyp zur persistenten Speicherung von Daten eingesetzt werden. Die Daten werden Out-Of-Line in einer speziellen Storage-Tabelle gespeichert.

In PL/SQL können Variablen dieses Typs deklariert werden:

```
v_nt nt_type;
```

Es ist aber auch möglich Nested Tables direkt in PL/SQL zu definieren. Die Definition erfolgt genauso wie die von assoziativen Arrays, nur daß die Angabe `INDEX BY` weggelassen wird:

```
TYPE nt_type AS TABLE OF element_declaration;
v_nt nt_type;
```

Als Elementdeklaration sind skalare Datentypen, Records oder Objekt-Typen sein. Die Indizierung von Nested Tables erfolgt implizit fortlaufend. Deswegen sind Nested Tables auch zu Beginn zwangsläufig dicht besetzt, was sich aber im Verlauf der Verarbeitung ändern kann.

Nested Tables können mit der impliziten Konstuktormethode initialisiert werden, die den gleichen Namen wie der Typ hat. Das soll für den Fall, dass die Elemente vom skalaren Datentyp `CHAR(1)` sind, kurz erläutert werden:

```
TYPE nt_type AS TABLE OF CHAR(1);
v_nt nt_type := nt_type ('A', 'B', 'C');
```

Diese 3 initialisierten Elemente werden implizit mit (1, 2, 3) indiziert.

VARRAYs (Variable-Size Arrays)

Dieser Typ von Collections ist ähnlich wie Nested Tables mit ein paar Unterschieden:

- Die maximale Anzahl der Elemente ist festgelegt und kann in PL/SQL nicht erweitert werden.
- Die Reihenfolge der Elemente folgt der einmal festgelegten Ordnung.
- Es können keine einzelnen Elemente aus der Collection gelöscht werden.

Auch diese Art von Collections kann als Objekt auf Schemaebene und in PL/SQL verwendet werden.

```
CREATE OR REPLACE TYPE va_type AS VARRAY(n) OF ele-
ment_declaration;
```

Dieser tabellenwertige Typ kann nun bei Schematabellen als Datentyp eingesetzt werden. Außerdem können innerhalb PL/SQL Variablen dieses Typs deklariert werden:

```
v_va va_type;
```

Es ist aber auch möglich Varrays direkt in PL/SQL zu definieren:

```
TYPE va_type AS VARRAY(n) OF element_declaration;
v_va va_type;
```

Das `n` steht für die maximale Anzahl der Elemente des Arrays. Diese kann auch mit der Methode `EXTEND` nicht erhöht werden. Man sollte beim Einsatz von VARRAYs schon eine Vorstellung haben, wie viele Elemente man braucht. (Auf Schemaebene kann man mit `ALTER TYPE … MODIFY LIMIT` die Obergrenze ändern.)
Bei der Verwendung als Schemaobjekt kommt eine weitere Einschränkung hinzu: Es ist keine DML auf einzelne Elemente des Arrays möglich, man muss immer das ganze Array anfassen, auch wenn nur ein Element geändert werden soll.

Auch VARRAYs können mit der impliziten Konstruktormethode initialisiert werden, z.B.

```
TYPE va_num AS VARRAY(3) OF NUMBER;
v_num va_num := va_num (3, 5, 7);
```

Auch VARRAYs sind nach der Initialisierung dicht besetzt. Vor der Initialisierung sind die Elemente von VARRAYs genauso wie die von Nested Tables NULL, im Gegensatz zu asso-

ziativen Arrays, die einfach leer sind. Wenn VARRAYs als Schemaobjekt verwendet werden, dann erfolgt die Speicherung ihrer Daten In-Line, zusammen mit den relationalen Daten.

Einsatzgebiete der verschiedenen Collections

Nicht jede Art von Collection ist für jeden Einsatzzweck geeignet.

Assoziative Arrays sind besonders geeignet, wenn

- die Collection nicht dicht besetzt sein soll (muss)
- Zuweisungen auf negative Indexwerte erfolgen sollen

Nested Tables sollten verwendet werden, wenn

- größere Mengen von Daten persistent in einer Collection gespeichert werden sollen
- komplexe Mengenoperationen auf den Daten ausgeführt werden müssen

VARRAYs sind sinnvoll, wenn

- es um kleine Datenmengen geht (ein Datenblock, um Row Chaining zu vermeiden)
- eine Maximalanzahl der Element erzwungen werden soll
- die Ordnung der Elemente eine Rolle spielt

Kontaktadresse:
Dr. Frank Haney
Anna-Siemsen-Str. 5
D-07745 Jena

Telefon: +49(0)3641-210224
E-Mail: info@haney.it
Internet: http://www.haney.it