8i AND 9i PL/SQL COLLECTIONS AND RECORDS

Bill Coulam, New Global Telecom

Modern programming languages allow the software artisan to model information about the elements in their problem domain, and store and manipulate those models in memory and other media. The models are often called *datatypes*. The storage and manipulation media are often called *data structures*.

Datatypes can be thought of as information templates, guiding and constraining the sort of information that is allowed in the transient and permanent data structures. There are simple types that cannot be decomposed any further, such as integer, date, byte and char. Oracle calls these *scalar* types. There are also complex types, made up of one or more simple and/or other complex types. Oracle calls these *composite* types. Oracle defines a third, the *reference* type, but it will not be covered here.

The data structures that can be built on these datatypes are illustrated below, along with comparables from other languages.

Data Structure	Oracle DB	Oracle PL/SQL	DB Version	Other Languages
Single, Scalar	Column	Constant, Literal, Simple variable	6.?	Same
Single, Composite	Row	Record	7.0	Struct, Object
List of Scalar	Multi-row column	Collection:		
		1) "PL/SQL Table"	7.0	Hash by integer
		"Index-By Table" (name change)	8.0	"
		"Associative Array" (name change)	9.1	Hash by integer or string
		2) Nested Table	8.0	Array, Queue
		3) Varray	8.0	Array, Stack
List of Composite	Multi-row columns (Table)	1) Index-By Table (Array) of Record	7.3.4	Array, Bag, Vector, Tree, Heap
		2) Nested Table of Record or Object	8.0	"
		3) Varray of Record or Object	8.0	"
List of List(s)	Parent-Child tables	1) Nested Table of Collection(s)	9.2	Multi-dimensional array (extensible)
		2) Varray of Collection(s)	9.2	Multi-dimensional array (bounded)

Table 1 (Oracle Database and PL/SQL Data Structures)

Something Table 1 fails to illustrate is data structure operations. Because PL/SQL collections can be dense or sparse; easily traversed, searched or accessed; and extended at will, a PL/SQL collection implements the best features of arrays, binary trees, heaps, vectors, hash tables and double-ended, doubly-linked lists…all at once!

Many of these powerful structures, and key operations on them, have not been available until recently. Since '94 I've been anxiously waiting for features that finally showed up in release 9.2. This paper is a basic introduction to the history, syntax and design concepts behind collections and records. It is a supplement to the in-depth tips and examples I'll provide in the presentation[1]. I anticipate the majority of my audience to be programmers or DBAs with at least a year of good PL/SQL experience. If this is you, skip right to the presentation. For the novices however, start here, then spend some time with Oracle's online documentation at http://docs.oracle.com (choose the 9i Database tab, then the Library link to the Online Documentation).

History

For those who wrote enterprise-class Oracle applications in PL/SQL before version 7, I salute you. I don't know how you did it. With version 7, Oracle finally introduced the PL/SQL Record and PL/SQL Table. With records, the contents of a programmer-defined structure or table row could be loaded into a variable that could be manipulated and used as a parameter to PL/SQL routines[2]. With PL/SQL Tables, lists of scalar values, and even lists of records, could be created and manipulated without the overhead of doing everything with tables and SQL statements. This greatly simplified programming in PL/SQL and enabled elegant, robust PL/SQL routines. Life was good.

[1] You may download the presentation and code examples from http://www.cbdba.net/papers.htm

[2] By *routines*, I mean packaged or stand-alone procedures and functions

Unfortunately, PL/SQL still resembled a 3-legged chair. Oracle programmers continued to run into limitations in their quest to simplify and speed up their routines. The significant limitations and when they were finally resolved are:

Limitations and Frustrations	Fixed
Cannot store a list of values related to an individual row. Alternative: create a separate attributive table to hold the row's list. These "temporary" child tables clutter the schema, add processing and storage overhead, and make code more complex.	8.0
Cannot use a collection in SQL statements. Alternative: create, populate and join to a temporary table.	8.0
Could not do array-like, or bulk SQL operations on collections. Alternative: None. Limited to one-by-one row processing. Shifting from the PL/SQL to SQL engine as each row was processed was a real drag on performance.	8.1
Cannot access list values using character-based keys. Alternative: create another kludge table to store the key:value pairs, or roll your own hash tables using a packaged PL/SQL table of record, hash algorithms and functions. Do-able, but not fun.	9.2
Cannot insert or update a table using a record. Alternative: None. Break out a record into its individual columns for insert/update. This effectively eliminated the benefits of abstraction and de-coupling that records were meant to provide.	9.2
Cannot use awesome BULK ops to get data directly into collections of record. Alternative: break every column down into its own collection of scalar. That was really unpleasant working with any more than a three-column table.	9.2

Table 2 (Past Oracle Limitations re: Collections and Records)

As you can see, with the arrival of Oracle 8, we were tossed a huge bone: Collections. They could be stored in a table, do almost everything the PL/SQL Table could do, and you could *perform SQL against them*. Oracle 8i introduced bulk operations with collections. This was a huge improvement in the performance arena, but didn't help simplicity much. It was still not possible to bulk fetch into or do bulk DML using a collection of records. Oracle 9i Release 2 (9.2) is my hero! It permitted multi-dimension arrays, record-based DML and string-keyed index-by tables. 9.2 also corrected the oversight introduced in 8i: you can now bulk collect directly into a collection of records!

Semantics and Syntax of Collections and Records

Records

A record is much like a row in a relational database table, composed of one or more individual fields, or columns. The structure of the record's type can be explicitly defined by the programmer, or implicitly defined by the %ROWTYPE attribute. %ROWTYPE abstracts the details of the underlying object. As columns are added, deleted and modified, your code doesn't have to change as often. Records allow you to assign, access, refer to and pass entire rows and tables as a single unit, as opposed to scores of individual variables or parameters.

We choose PL/SQL when we need to 1) batch up multiple SQL operations, or 2) improve speed by performing logic within the database, rather than moving the data over the network. Since both goals involve operations on rows of data that is/will be stored in relational tables, you should be using records most of the time! This is a common best practice in PL/SQL circles. Leaning heavily on records leaves your code simpler, cleaner, easier to read and maintain, and more robust.

Record Syntax

To define a record type *explicitly*, use the following syntax in the declaration section of a PL/SQL block:

```
TYPE type_name IS RECORD (field_declaration[,field_declaration]...);
```

where `field_declaration` means

```
field_name field_type [[NOT NULL] {:= | DEFAULT} expression]
```

where `field_type` is any PL/SQL datatype[3] except REF CURSOR[4] and where `expression` yields a value whose type is equal to `field_type`. Here's a hairy example that covers most of the possibilities:

```
TYPE tr_account IS RECORD (
   account_id   acc_types.t_account_id NOT NULL – packaged subtype, constrained
,  account_owner    acc_types.tr_account_owner – packaged record
,  related_accounts acc_types.tas – collection of disparate account IDs
,  created_dtm  TIMESTAMP WITH LOCAL TIMEZONE DEFAULT SYSTIMESTAMP – precise time w/default
,  active_flg   accounts.active_flg%TYPE – anchored to table column type
,  fraud_check  risk_types.to_fraud_check – packaged object type );
```

[3] All the scalars, including UROWID, BOOLEAN and RAW; and PL/SQL objects and REFs, collections, LOBs, and other records.

[4] The PL/SQL Bible, *Oracle PL/SQL Programming*, 3rd Ed, claims that records can contain ref cursors (p. 326). The 9.2 Oracle docs do not support this statement, and indeed you get error PLS-00989 if you try it.

A record type is defined *implicitly* by the existence of a table-like object (table, view, synonym, cursor, or cursor variable).

Once the record has been defined explicitly or implicitly, it simply remains to declare variables or parameters based on that underlying object or cursor:

```
PROCEDURE cancel_account(ir_account IN acc_types.tr_account) IS –parameter based on explicit type
   CURSOR c_mytab IS SELECT col1, col2, … FROM mytab WHERE …; --explicit cursor
   lr_mytab  c_mytab%ROWTYPE; --based on explicit cursor's ROWTYPE attribute

   TYPE trc_mytab IS REF CURSOR RETURN mytab%ROWTYPE; --define ref cursor
   lrc_mytab  trc_mytab; --declare ref cursor variable
   lr_mytab   lrc_mytab%ROWTYPE; --based on ref cursor's ROWTYPE attribute

   lr_myview  myview%ROWTYPE; --based on view's ROWTYPE attribute
   lr_mytab   mytab%ROWTYPE; --based on table's ROWTYPE attribute
   lr_theirtab  theirtab%ROWTYPE; --based on ROWTYPE attribute of private synonym's table
   lr_cust_account  acc_types.tr_account; --based on packaged, explicit record
BEGIN
   FOR lr_mytab --based on implicit cursor
      IN (SELECT col1, col2, … FROM mytab WHERE …) LOOP
      <do stuff>
   END LOOP;
```

Record Usage

Find a complete treatment of this subject in the Oracle docs and various excellent PL/SQL programming books. Briefly, however, when working with the whole record variable, you may:

```
   lr_account := ir_account; --copy one record to another (known as aggregate assignment)
   lr_old_account := NULL; --empty out a record by assigning NULL
   lr_new_account := lr_old_account; --empty by assigning an empty record

   -- Use the record as a parameter, or as the value returned by a function.
   lr_new_account := fr_create_checking_account(lr_cust_info, ls_cheezy_gift_choice);

   -- Direct SELECT or fetch into a record
   SELECT * INTO lr_account FROM accounts_vw WHERE account_id = in_account_id;
```

Individual fields within the record are handled just like regular variables, the only difference being the dot notation used to access that part of the record, e.g.

```
   lr_account.active_flg := 'A'; -- dot notation to record's field
   lar_accounts(i).active_flg := 'I'; --dot notation to one of the records in a collection
```

Collections

A collection is nothing more than a list, an ordered group of elements, all of the same type – much like a single-dimension array in C or a one-column relational table. Each element in a collection has a unique index, or key, that determines its position in the collection. If you know the key, you are able to perform random access to the value, which is faster than iteration and even faster than SQL against the same information in an index-organized or cached table.

Oracle provides three different collections: *associative arrays*, *nested tables* and *varrays*. "Associative" because the value is directly tied, or associated, with its numeric or character-based key. "Nested", because as a column type, it allows you to embed, or nest, a table within a column of a row. "Varray", short for "variable-size array", is mystifying. The name and the Oracle docs[5] suggest that varrays are more flexible than the other collection types, when in fact, the opposite is true. I have yet to find that a varray was better suited to my problem than a nested table.

Collection Syntax

Unlike implicit records, all collection types must be defined, either *locally* or *globally*. Locally means the collection type is only available inside PL/SQL, and unless it's declared in a package specification, it's only available to the current PL/SQL

[5] The PL/SQL User's Guide is poorly worded. It says varrays "hold a fixed number of elements (although you can change the number of elements at runtime)." It also implies varrays are extensible: "To increase the size of a nested table or varray, use EXTEND." It also showed a varray with a maxsize of 10, and then proceeded to explain that the uppder bound could be extended to "8, 9, 10, and so on." Um, no. You will get ORA-06532 if you try to extend beyond 10, its maximum size.

block. Globally means it is available for use within PL/SQL, SQL, object type definitions, and as a datatype for columns in database tables. *If you want to perform SQL against your collection, it must be globally defined.*

To define a collection type locally:

```
TYPE type_name IS TABLE OF element_type [NOT NULL]; --nested table
TYPE type_name IS {VARRAY | VARYING ARRAY} (size_limit) OF element_type [NOT NULL]; --varray
```

where `element_type` is any PL/SQL datatype except REF CURSOR[6] and `size_limit` is a positive integer indicating the maximum number of elements that will be allowed in the varray. Also varrays cannot contain LOBs.

To define a nested table collection type globally (from the SQL*Plus command line), just start the above TYPE definitions with the "**CREATE OR REPLACE**" clause. Unlike local collections, since global collections can be used as column datatypes, they share many of the same rules as columns, e.g. global collections cannot use the datatypes: BINARY_INTEGER, PLS_INTEGER, BOOLEAN, LONG, LONG RAW, NATURAL, NATURALN, POSITIVE, POSITIVEN, REF CURSOR, SIGNTYPE, or STRING.

Associative arrays cannot be used as column datatypes or in SQL statements. They can only be defined locally:

```
TYPE type_name IS TABLE OF element_type [NOT NULL]
   INDEX BY [BINARY_INTEGER | PLS_INTEGER | VARCHAR2(size_limit)];
```

where size_limit is a positive integer up to the VARCHAR2 datatype max size (32767);

Once a collection type has been defined, it may be used, as a PL/SQL variable

```
lnt_callback_dtm  dates.tnt_dtm := tnt_dtm(); --local nested table, initialized empty
las_comments  strings.tas; --local array of string (VARCHAR2)
lar_accounts  acc_types.tar_accounts; --local array of record
```

a parameter, or function return type

```
las_accounts := fas_brokerage_accounts(lar_accounts);
```

the type of a record field, or an object's attribute

```
CREATE TYPE to_cust IS OBJECT(cust_id NUMBER, related_accounts tnt_account_ids, … );
```

the type of a database table column[7]

```
CREATE TABLE branch (gid NUMBER PRIMARY KEY, accounts tnt_accounts, … )
   NESTED TABLE accounts STORE AS branch_accounts_nt (ORGANIZATION INDEX);
```

Note that nested tables and varrays must be initialized before they can be populated or read, unless the population occurs by direct or bulk fetch, or aggregate assignment. Oracle provides the following built-in functions to manipulate collections:

Built-In Function	Comments
EXISTS(n)	Returns TRUE if there is an element at *n*, otherwise returns FALSE.
COUNT	Returns 0 if the collection hasn't been initialized, or if it simply has no elements. Does not include elements that have been DELETEd in the count. Once the collection is sparse, COUNT no longer equals LAST.
FIRST, LAST	Returns NULL if collection is empty, otherwise returns the smallest/largest index numbers in the collection. If an associative array indexed by strings, uses the binary values of the strings to determine order.
PRIOR(n), NEXT(n)	Returns the index preceding/following the element at *n*. Returns NULL if *n* is the first or last element, respectively.
DELETE, DELETE(n), DELETE(a,z)	DELETE removes all elements from a collection. DELETE(n) removes the *n*th element from the collection. *n* can be a string key. May not use this form with varrays. DELETE(a,z) removes all elements between the range from *a* to z. *a* and *z* can be string keys. May not use this form with varrays.
EXTEND, EXTEND(n)	Crucial for nested tables and varrays. Associative arrays automatically increase in size when you assign a new key:value pair to them. Nested table and varrays do not. You must append a new null element to the collection with EXTEND before you can refer to the new subscript or a assign a value to its position. EXTEND(n) appends *n* null elements.
TRIM, TRIM(n)	TRIM removes one element from the end of the collection. The other form removes *n* elements from the end. TRIM includes deleted elements in its successfully trimmed count, so be careful. Treat nested tables as an array and use only DELETE, or treat them as a stack and use only TRIM and EXTEND.
LIMIT	For varrays. Returns the maximum size of the varray. Returns NULL for nested tables and associative arrays.

Table 3 (Collection Built-Ins)

[6] Local nested tables and varrays were still heavily limited even in Oracle 8.1.7. All the restrictions were lifted in 9.2.

[7] Specify the **STORE AS** table for ANY nested tables in columns, even if it is buried inside a column's object type attributes.

Collections in Data Design

Recall your experience and training in relational data modeling. One of the goals is to avoid redundancy. This aids data integrity and simplifies data maintenance and extension. A good data architect attempts to keep models simple and elegant using a technique called *normalization*.

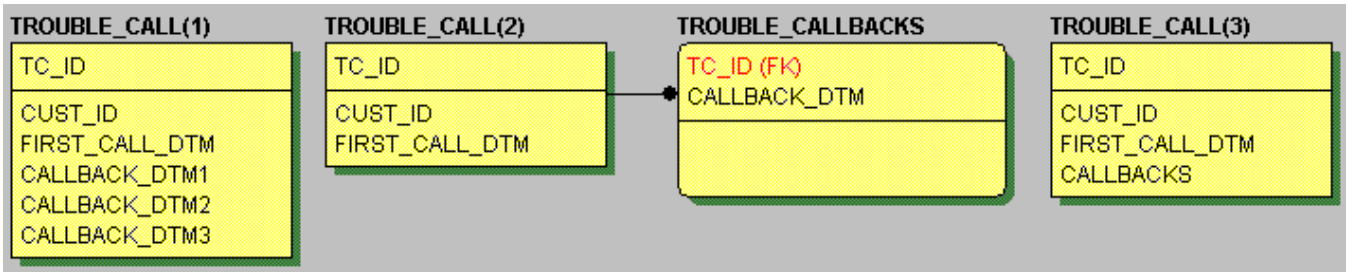

Figure 1 (Three Model Phases of Trouble Call Data)

TROUBLE_CALL(1) has a repeating attribute. In first normal form, that attribute becomes a child table whose primary key is a foreign key back to its parent, an attributive relationship. After first normalization, the model looks like TROUBLE_CALL(2). It's clean and it meets first normal form, but it's a pain. As a list of dates, the callbacks don't describe anything useful without joining to the parent. No other entity depends on them. I like to call these attributive tables "Clingons". It would be nice to stuff them in the parent as shown in TROUBLE_CALL(3), but normalization and most databases won't allow it.

What if you could virtually include the attributive Clingons in the parent record? It would avoid excessive joins, eliminate table clutter, and still preserve normalization. Cascading deletes would be a breeze too: delete the parent row, and Poof! all the child rows are gone too. Updating would be easier too since you wouldn't have to traverse several physical tables to update a table at the bottom of a hierarchy. In OO all this is accomplished using a technique known as object composition. In the relational world, it's known as heresy. Heresy or not, since Oracle 8, you can include collections with the parent row to which they belong. Tables like TROUBLE_CALL(3) are easy to create, and make a lot of sense for certain scenarios.

Collections for the DBA

To be able to recommend and enforce the right kind of collection, you should know the following key points:

- In practice, most varrays (per row) total less than 4000 bytes. This data will be stored RAW inline and makes varrays your best choice for speed and efficiency. If the data surpasses the 4000 byte mark, or if you specify a LOB storage clause for the varray, Oracle will create a LOB segment and move the data out of line.
- Varrays cannot be extended beyond their initially defined maxsize. You will need to drop and recreate the type and table if you miscalculated the maxsize during modeling.
- Varray columns can't be modified using SQL; you are reduced to using PL/SQL to insert, update or delete varrays. You can't just replace an element inside the varray either; you must replace the whole array.
- Physically, nested tables are implemented by Oracle as a child table, with a hidden column (**`nested_table_id`**), which is foreign-keyed to a surrogate, hidden column in the parent table. The child table may not be queried or modify independently of the parent.
- There is no inherent order to how data in nested tables is stored or returned, and foreign keys are not supported. If you want to impose order, you must add an attribute to the nested table for indexing or sorting.
- Unlike regular DML, when unnesting the Oracle doesn't allow the subquery to return zero or multiple collections. The subquery to get the row's nested table must bring back only one nested table.
- If you do create columns using nested tables, index the **`nested_table_id`** pseudocolumn. Since the child table created by the server to store your nested table has the tendency to de-cluster the ordered data you've put in there, it is also a good idea to define the nested tables as index-organized:

If performance is a concern and the collection maxsize **will never** need to be larger, and you won't be updating the values after the initial load, you may suggest or enforce the use of a varray. Other than that, stick with nested tables. However, even nested tables should be used judiciously, where the list is attributive, you'd never want to query or modify the collection independently of the parent, and your DB access technology can handle the non-scalar datatypes.

Collections and Records in Application Design

After lots of first-hand experience, and intriguing threads on Quest's PL/SQL Pipeline and Tom Kyte's site, the list of scenarios, tests and code examples I wanted to share was getting huge. I had to move them to the presentation instead. In the end, they all boiled down to a couple rules of thumb:

- You should be using records all the time and asking yourself, "Now where can I use simple variables?", not the other way around. Collections of records are essential if you want to build PL/SQL isolation layers (presentation, rule, lookup, data access API, etc.) It's not a question of when should you use records, it's a question of how much pain you can endure as you stick with individual fields for everything.
- If you need fast and random, bi-directional or multiple-pass access over a static set of data, either for lookups or comparisons, you can't beat a collection.

That said, here are a few random thoughts to help you decide when and how to use collections and records.

Types

Make a package, perhaps named "T" which contains all the most generic, enterprise-wide, common subtypes and collections. Include global variables in the T package spec that are based on those collection types. This provides empty collections that are useful for variable re-initialization and `DEFAULT` values for collection parameters. Then give it a public synonym and grant everyone execute privilege. You will also need to create some generic, global collection types upon which you can build collection utilities that require SQL statements, or object types. You can create public synonyms for these as well, and grant execute privileges, but if you're on 8i, you will need to create private copies of the types in each schema (bug disallowing synonyms to other schemas' types.)

If yours is a shop that builds multiple applications in a single schema or instance, it is now time to build more specific packages for storing types. For each subsystem, you will want another package specification dedicated to defining a number of subtypes, record types and collections of record based on the columns and tables in the subsystem. I think of this as the end of the database design phase. Now you can move on to the detailed design of your application's isolation layers, using your packaged types for most of the interfaces.

Records

Interfaces

As long as you are writing lower-level PL/SQL routines that will be called only by other PL/SQL routines, you should generally use records as the primary parameters. This is especially true and beneficial if you are working on the data access layer. Some of you don't have to imagine a table with 50, 80, even 100 columns or more. You live with such modeling horrors every day. Now imagine an INSERT procedure that uses 50 individual parameters. Consider the very real possibility that Marketing gets an awful promotional approved, and you're stuck figuring out how to model and codify that which defies logic. There's no budget to refactor your legacy model, so you end up tacking 5 more columns onto the end of that monster table. Now you've got to hunt down all dependents on that table and paste in a bunch more parameters and code.

Using a record reduces that entire headache to nothing. Since the records are tied to the table using %ROWTYPE, the code automatically adjusts. No parameters to add, delete or change type length. If you've followed best practices, you'll only have the insert and update statements to modify. If you're on 9i, you can even avoid that![8]

If the callers of your PL/SQL routine are written in other languages, avoid records. In some, like PERL, you can write anonymous blocks that can take advantage of PL/SQL-only datatypes, but that is the exception. Java, for example, cannot handle records, period. Your only alternative that comes close is to use object types instead, then convert the object's attributes to a record once the call has reached the PL/SQL level. Then there's the issue of driver support. Even if your driver supports protocol Z, version X, it doesn't mean they did it right. We ended up having to abandon the idea of passing objects or arrays to PL/SQL thanks to our application server's JDBC driver[9].

[8] 9.2 finally permits DML using entire records, and bulk operations on collections of records.

[9] The workaround was to pass in a delimited string of values. There are overloaded copies of the intended interface. The one matching the Java call is just a wrapper that tokenizes the strings into records and collections, which it passes along to the original interface.

DML

If you're on 9i, there's no question. Most of your DML should use records and collections of record. If you're on 8i, I still find records the only way to go for most every DML statement. Single-column SELECT statements are the only exception.

Collections

Which collection type?

There are a number of differences between the three collection types. The key differences to know are:

Nested Tables	Associative Arrays	Varrays
Globals available for SQL statements and DDL	May not be used in SQL statements or DDL	Globals available for SQL statements and DDL
Order not preserved in the store table. Sequential (1..N), but may become sparse.	No order imposed unless loaded with a fetch. Index can be negative and arbitrary. May be created sparsely or become sparse.	Order preserved in the row or out of line LOB store . Sequential (1..N). May not DELETE individual elements.
Initialized by explicit or implicit constructor.	Initialized on declaration. Useable immediately.	Initialized by explicit or implicit constructor.
Can increase in size via EXTEND.	Can increase in size by assignment using new key	Cannot increase in size beyond defined maxsize.
Use DML from SQL or PL/SQL to operate on whole lists, or individual elements.	N/A	Can only use DML from PL/SQL to operate on the entire varray, NOT individual elements.

Table 4 (Significant Collection Differences)

I've never needed or used a varray. However, I'm head-over-heels in love with nested tables and associative arrays. Here are some pointers on when and where to make use of these two sorts of collections.

Hash Tables

Many of the tables I design have simple, surrogate primary keys, which fit perfectly into arrays and can be used in generic array utility routines. If your keys are natural, concatenated, or the string and date-based sort, you may have to do a little concatenation, delimiting and maybe even hashing to get the primary key into that subscript.

If yours is a shop that keeps codes and app configuration key:value pairs in numerous datastores (code tables, proprietary-format or XML files, and LDAP entries) , centralize that into either lowest-common denominator XML files or Oracle tables. If the latter, use associative arrays in the initialization section of packages to load up static data at the start of a session, then access that data at blazing speed using random access. This would be especially beneficial for PL/SQL applications that are the "Transformation" part of ETL architectures. You often have PL/SQL loops doing row-by-row processing of millions or billions of rows. Instead, you can instantaneously look up values in an associative array.

Faster, Better, Stronger…

Many batch-oriented or data-warehouse queries and PL/SQL cursors have joins to static reference and dimension tables. No matter how well you have those static tables laid out and cached, several articles and books claim that joining to a nested table will always be faster by several orders of magnitude. Much depends on the size of the static data and how many concurrent sessions can be expected to load the same data into PGA. As long as it isn't unreasonably large, load static data into a nested table variable, and then join to that in your million-row cursor.

I don't know how many times I've ended up with critical data in an index-by table and wanted to use it in a SQL statement, usually with the IN or NOT IN operator. I'd sigh, create another temporary table and write all the necessary code to get my values in there and maintain the table, then use it in my join. Yikes! Nested tables solved that for me. They are a little more effort[10], but more than worth it. To me, nested tables and bulk ops are the shiney hammer and everything looks like a nail. I'm finding more and more places where looped DML, comparisons and queries were made simpler, faster and more readable using that unbeatable combination. Dive in! Find at least one routine that could benefit. You'll never look back.

Comparisons/Synchronization

I've recently worked on some synchronization code. MINUS would have been great, but I didn't have a table to join to, only in-memory data. The first draft solution had me massaging the in-memory data so it could be inserted into the core, message source table. Then I was able to turn around, and query that data again using MINUS to do my comparisons. The worst part was that the sync routine was running every 5 minutes, and 99% of the time nothing needed to change. So we were storing hundreds of thousands of extra rows each month, just to accommodate this. When I got some time to refactor in a later release, the solution of using two sets of nested tables in SQL dawned on me like a beautiful country morning. The routine sped up by 60% and the data storage needs of that monster table dropped by 80%.

[10] Unless filling them from a SQL statement which handles the initialization and EXTEND calls for you.

Bi-Directional Access

Certain customized, in-house PL/SQL algorithms could benefit from the ability to repeatedly go back and forth over a static set of keys and values. I recall writing one such procedure that for every value in a query, had to find that value in an array, then search backwards and forwards, deleting certain elements, and searching for other values and how far apart they were in relation to it. Maybe a programming god could have written that in SQL, but I doubt it. If you have a similar need, then collections are your ticket.

Collections and Records in SQL Statements

Performing SQL on collections and records is barely covered in Oracle and 3rd party documentation. The majority of the presentation will focus on in-depth examples to remedy that dearth of info. What follows is only a simple introduction to SQL using nested tables and bulk operations.

Collections

When working with collections, you're either dealing with the whole collection at once (*nesting*) or individual entries (*unnesting*). You use the TABLE, CURSOR, CAST, and MULTISET expressions to nest and unnest. As you look at the following code, think of unnesting the same as you would joining to a child table. It makes the TABLE() expression a little easier to digest. Refer to the presentation for more complete examples.

```
CREATE OR REPLACE TYPE tnt_str AS TABLE OF VARCHAR2(60);
CREATE OR REPLACE TYPE tnt_num AS TABLE OF NUMBER;

CREATE TABLE soul
(soul_id NUMBER PRIMARY KEY, soul_nm VARCHAR2(60),angel_of_death VARCHAR2(60), bodies tnt_str)
NESTED TABLE bodies STORE AS soul_bodies_nt; --STORE AS required for every nested table col/attribute

DECLARE
   lr_soul soul%ROWTYPE;

   lnt_bodies tnt_str := tnt_str('Lungfish','Human'); --declare and initialize with values
   lnt_bodies2 tnt_str := tnt_str(); --declare and just initialize
   TYPE tant IS TABLE OF tnt_str INDEX BY PLS_INTEGER; /*9i ONLY!*/ --multi-level collection type
   lant_all_bodies tant; --multi-level collection variable

   CURSOR cur_similar(int_tab1 IN tnt_str,int_tab2 IN tnt_str) IS
   SELECT o.column_value val
   FROM   TABLE(CAST(int_tab1 AS tnt_str)) o
   INTERSECT
   SELECT n.column_value val
   FROM   TABLE(CAST(int_tab2 AS tnt_str)) n;
BEGIN
   lnt_bodies2.EXTEND(12); lnt_bodies2 := tnt_str(
       'Housefly','Rabbit','Newt','Frog','Oyster','Cow','Fish','Flea','Ant','Human',
       'Bowl of Petunias','Bat'); --can initiliae like this, or one at a time, or in a loop

   -- whole nested table inserts
   INSERT INTO soul VALUES (1,'Beldar','Afkhan',tnt_str('Catfish','Conehead')); --initialized here
   INSERT INTO soul VALUES (2,'Agrajag','Arthur Dent',lnt_bodies2);
   INSERT INTO soul VALUES (3,'Bill','TBD',lnt_bodies);
   -- single nested table entry insert
   INSERT INTO TABLE(SELECT bodies FROM soul WHERE soul_nm = 'Beldar') VALUES('Eel');

   lnt_bodies.DELETE; lnt_bodies2.DELETE; --reinitialized

   -- whole nested table updates
   UPDATE soul SET bodies = tnt_str('Newt','Baboon','Conehead') WHERE soul_id = 1;
   UPDATE soul SET bodies = tnt_str() WHERE soul_nm = 'Bill';
   -- single nested table update
   UPDATE TABLE(SELECT bodies FROM soul WHERE soul_nm = 'Agrajag') b
      SET b.COLUMN_VALUE = 'Bat-like Atrocity' WHERE b.COLUMN_VALUE = 'Bat';

   DELETE TABLE(SELECT bodies FROM soul WHERE soul_nm = 'Bill') b
      WHERE b.COLUMN_VALUE != 'Human'; -- delete of single entry in a nested table

   SELECT bodies INTO lnt_bodies FROM soul WHERE soul_id = 1; --direct select
   SELECT bodies INTO lnt_bodies FROM soul WHERE soul_id = 2;
```

```
   OPEN cur_similar(lnt_bodies, lnt_bodies2);
   FETCH cur_similar BULK COLLECT INTO lnt_bodies; --direct select
   CLOSE cur_similar;

   SELECT bodies BULK COLLECT INTO lant_all_bodies FROM soul; /*9i ONLY!*/
   SELECT * INTO lr_soul FROM soul WHERE soul_id = 2;
   FOR i in 1..lr_soul.bodies.COUNT LOOP
      msg.p(lr_soul.bodies(i)); --access to unnamed column in parent record
   END LOOP;
EXCEPTION
   WHEN OTHERS THEN
      IF (cur_similar%ISOPEN) THEN CLOSE cur_similar; END IF; RAISE;
END;

DECLARE
   lan tnt_num := tnt_num(1,1,2,3,4,5,6,6,6,7,8,9);
BEGIN
FOR lr IN (SELECT a.COLUMN_VALUE MY_id, COUNT(*) id_count
                FROM TABLE(SELECT CAST(lan AS tnt_num) FROM dual) a
                GROUP BY a. COLUMN_VALUE  --added group by and order by for variety
                ORDER BY 1) LOOP
   msg.p(TO_CHAR(lr.MY_id), TO_CHAR(lr.id_count));
END LOOP;
END;
```

Records

The ability to do record-based inserts and updates has been sorely needed since 7.3. It's so wonderful, I feel like a stock trader that's been given a chance to travel back in time to 1994. The only significant restrictions are:

- The number and type of fields in the record must equal the number and type of columns listed in the INTO clause.
- It is not supported in Dynamic SQL.
- The record you feed the insert/update may not be a function that returns a record.

Insert

```
INSERT INTO accounts VALUES ir_account; --that saved me about 40 lines of code over 8i
```

Update

```
UPDATE accounts SET ROW = tr_account; --again, 40 lines saved over same routine in 8i
```

Bulk DML with Collections of Record (9i Only)

```
SELECT {*|column_list} BULK COLLECT INTO rec_array FROM ...

FETCH {cursor|cursor_var} BULK COLLECT INTO rec_array [LIMIT n];

FORALL index IN lower_bound..upper_bound
   INSERT INTO table_ref [(column_list)]
   VALUES (rec_array(index)) ...

FORALL index IN lower_bound..upper_bound
   UPDATE table_ref
   SET (column_list) = rec_array(index)...

…RETURNING {row_expression[, row_expression]...}
   BULK COLLECT INTO rec_array;
```

where rec_array is your collection of records, cursor_var could be a host cursor as well, lower_bound and upper_bound are usually rec_array.FIRST..rec_array.LAST, and table_ref could mean a table, view, synonym, even a nested table returned by the TABLE operator.

Tips

- Always use aliases for the tables and nested tables when unnesting in SQL queries. Prevents known bugs.

- Since the store table for a nested table column is a child table, treat it as such. For example, when retrieving rows that might not/don't have values in the collection, use the outer join. That's not intuitive unless you think of the nested table "column" as the child table that it really is.
- Always use coll.FIRST..coll.LAST when writing FORALL statements and FOR loops, not 1..coll.COUNT.
- Always check your collection first to see if it has any values before working on it (col.COUNT > 0).
- If the consumer of the table containing the nested table doesn't need to see the nested table data very often, and the nested table takes a large amount of resources to transmit to the data consumer, use RETURN AS LOCATOR which just returns a pointer to the data. When needed, the pointer can be dereferenced to get the nested table data.
- The keyword COLUMN_VALUE is provided as the pseudonym of the column in a simple collection. Create an alias for this column when unnesting.
- Index associative arrays using PLS_INTEGER (for numeric) and %TYPE (tying it to column or subtype definition).

Tricks

- When viewing the structure of a table in SQL*Plus, use "SET DESCRIBE DEPTH ALL" to allow DESC[RIBE] to show all the columns in the composite column types.
- Unsupported, undocumented /*+ NESTED_TABLE_GET_REFS */ hint to directly query and manipulate a nested store table.

Traps

- Can't use RETURNING with INSERT, despite what the docs say.
- Prior to 9.2, ORA-22863 if you reference a synonym to a collection type created in another schema. Bug 1879471.
- Be careful doing SQL against PL/SQL collections, often used with the IN and NOT IN operator. If any member of NOT IN list is NULL, the result in NULL. No error. No result set for you!
- Outside PL/SQL, support for querying and updating collections is minimal. If your software is two-tier, directly connecting to and accessing tables, you may very well break your applications by introducing columns that aren't single values[11]. If you do have the proper isolation layers in place, you'll need to augment the data layer to unnest and convert collections into a data structure supported by your DB access technology.
- PL/SQL objects and global collections cannot have an attribute whose type is RECORD.
- Record-based DML cannot be done using nested records, functions that return a record type, or dynamic SQL.
- You may not do bulk "RETURNING *" into a collection of records, yet. Must list out all columns.
- May not bulk collect into associative arrays indexed by strings.

References

Expert One-on-One Oracle, Thomas Kyte, 2001

Oracle PL/SQL Programming, 3rd Ed, Steven Feuerstein and Bill Pribyl, 2002

[11] That could be a good thing. You could convince management to buy or build an n-tier application framework that isn't so fragile.