**Understanding Context Switching**

When you execute a PL/SQL block, it runs within the PL/SQL engine. However, when that block encounters a SQL statement (like SELECT, INSERT, UPDATE, DELETE), control needs to be passed from the PL/SQL engine to the SQL engine for execution. Once the SQL engine completes its task, it returns control and any results back to the PL/SQL engine. This back-and-forth transfer of control is called a **context switch**.

For row-by-row processing (e.g., a FOR loop iterating over a cursor and performing an INSERT for each row), a context switch occurs for *every single row*. If you're processing thousands or millions of rows, this overhead can be enormous, leading to very poor performance.

BULK COLLECT and FORALL are the answers to this problem. They enable **bulk processing**, meaning they allow multiple rows to be processed in a single interaction (context switch) between the two engines.

**BULK COLLECT (For Retrieving Data)**

BULK COLLECT is used to fetch multiple rows from a SQL query into a PL/SQL collection (VARRAY, Nested Table, or Associative Array) in a single operation. Instead of fetching one row at a time, it fetches a "bulk" of rows.

**Purpose:**

To efficiently retrieve large result sets from the database into PL/SQL collections.

To drastically reduce the number of context switches when fetching data.

Syntax:

SELECT column1, column2, ...

BULK COLLECT INTO collection\_variable1, collection\_variable2, ...

FROM table\_name

WHERE condition;

**FORALL (For DML Operations)**

FORALL is used to perform a single DML (Data Manipulation Language) statement (INSERT, UPDATE, DELETE, or MERGE) multiple times, using values from a PL/SQL collection. Instead of executing the DML statement row by row, FORALL sends the entire batch of DML operations to the SQL engine in one go.

**Purpose:**

To efficiently perform bulk INSERT, UPDATE, DELETE, or MERGE operations.

To drastically reduce the number of context switches when modifying data.

Syntax:

FORALL index IN lower\_bound .. upper\_bound

DML\_statement;

**COLLECTIONS**

**Collection** is a composite data type that allows you to store and manipulate a group of elements of the same data type as a single logical unit.

 Collections are powerful, complex data types that allow developers to work with multiple data items as a **single entity**. These are extremely useful when handling large sets of data or when performing operations that involve multiple rows or values at once.

PL/SQL collections can be thought of as "**containers**" that hold multiple items of the same type. They offer flexibility and efficiency when working with large datasets, particularly when combined with Bulk Collect and FORALL operations for performance optimization.

**Why We Need Collections in SQL ?**

While the core of relational databases focuses on highly normalized tables where each piece of data is stored only once to avoid redundancy, real-world applications often need to work with groups of data together. This is where collections come in.

Interacting with the database is an I/O operation and involves context switching between the SQL engine and the procedural language. When you have to process many rows, executing SQL statements one by one in a loop can be very inefficient. Collections, especially when used with **BULK COLLECT** (for fetching multiple rows into a collection) and **FORALL** (for performing DML operations on multiple rows from a collection), allow you to process many rows in a single go, dramatically reducing the number of round-trips to the database and thus improving performance.

Collections provide a structured way to pass multiple values (e.g., a list of IDs, a set of objects) as a single parameter to stored procedures, functions, or between different blocks of code. This simplifies the interface and improves code readability.

**Processing Daily Sales Transactions**

Let's consider a scenario in an e-commerce system where you need to process a large number of sales transactions at the end of the day. For each transaction, you need to:

* Update the product\_inventory table by decreasing the quantity of each sold product.
* Insert a record into the sales\_history table.

without Collections (Row-by-Row Processing)

we have to fetch each transaction one by one and then execute separate UPDATE and INSERT statements for each item within that transaction.

DECLARE

CURSOR c\_daily\_transactions IS

SELECT transaction\_id, product\_id, quantity\_sold, customer\_id, unit\_price

FROM daily\_sales\_temp; -- A temporary table holding today's sales

v\_product\_id NUMBER;

v\_quantity\_sold NUMBER;

v\_customer\_id NUMBER;

v\_unit\_price NUMBER;

BEGIN

OPEN c\_daily\_transactions;

LOOP

FETCH c\_daily\_transactions INTO v\_transaction\_id, v\_product\_id, v\_quantity\_sold, v\_customer\_id, v\_unit\_price;

EXIT WHEN c\_daily\_transactions%NOTFOUND;

-- Update product inventory for EACH product, one by one

UPDATE product\_inventory

SET stock\_quantity = stock\_quantity - v\_quantity\_sold

WHERE product\_id = v\_product\_id;

-- Insert sales history record for EACH transaction, one by one

INSERT INTO sales\_history (transaction\_id, product\_id, quantity\_sold, sale\_date, customer\_id, total\_price)

VALUES (v\_transaction\_id, v\_product\_id, v\_quantity\_sold, SYSDATE, v\_customer\_id, v\_quantity\_sold \* v\_unit\_price);

END LOOP;

CLOSE c\_daily\_transactions;

COMMIT; -- Commit after all transactions

END;

/

In this there are three separate SQL statements being executed. If you have thousands or millions of sales transactions, this leads to an enormous number of context switches between the PL/SQL engine and the SQL engine, making the process very slow.

Now

**Scenario with Collections (Bulk Processing):**

DECLARE

TYPE num\_list IS TABLE OF NUMBER INDEX BY PLS\_INTEGER;

TYPE varchar\_list IS TABLE OF VARCHAR2(100) INDEX BY PLS\_INTEGER;

TYPE date\_list IS TABLE OF DATE INDEX BY PLS\_INTEGER;

v\_transaction\_ids num\_list;

v\_product\_ids num\_list;

v\_quantities num\_list;

v\_customer\_ids num\_list;

v\_unit\_prices num\_list;

-- Collections for sales\_history (calculated values)

v\_total\_prices num\_list;

BEGIN

-- 2. BULK COLLECT all daily transactions into collections in ONE round-trip

SELECT transaction\_id, product\_id, quantity\_sold, customer\_id, unit\_price

BULK COLLECT INTO v\_transaction\_ids, v\_product\_ids, v\_quantities, v\_customer\_ids, v\_unit\_prices

FROM daily\_sales\_temp;

-- 3. Prepare data for sales\_history (in-memory calculation)

IF v\_transaction\_ids.COUNT > 0 THEN

FOR i IN v\_transaction\_ids.FIRST .. v\_transaction\_ids.LAST LOOP

v\_total\_prices(i) := v\_quantities(i) \* v\_unit\_prices(i);

END LOOP;

-- 4. Perform bulk updates and inserts using FORALL

-- Update product inventory (ONE round-trip for all updates)

FORALL i IN v\_product\_ids.FIRST .. v\_product\_ids.LAST

UPDATE product\_inventory

SET stock\_quantity = stock\_quantity - v\_quantities(i)

WHERE product\_id = v\_product\_ids(i);

-- Insert sales history (ONE round-trip for all inserts)

FORALL i IN v\_transaction\_ids.FIRST .. v\_transaction\_ids.LAST

INSERT INTO sales\_history (transaction\_id, product\_id, quantity\_sold, sale\_date, customer\_id, total\_price)

VALUES (v\_transaction\_ids(i), v\_product\_ids(i), v\_quantities(i), SYSDATE, v\_customer\_ids(i), v\_total\_prices(i));

END IF;

COMMIT;

EXCEPTION

WHEN OTHERS THEN

ROLLBACK;

DBMS\_OUTPUT.PUT\_LINE('Error processing sales: ' || SQLERRM);

END;

/

Instead of potentially thousands or millions of individual UPDATE and INSERT statements, we're now performing just a few (one for BULK COLLECT, one for product inventory updates, one for sales history inserts). This drastically reduces context switching and I/O operations, making the batch process complete much faster.

The most common types of collections you'll encounter in SQL are:

1. **Varrays (Varying Arrays)**
2. **Nested Tables**
3. **Associative Arrays (Index-by Tables)**

**Varrays (Varying Arrays)**

Varrays are ordered collections of elements, similar to a fixed-size array in other programming languages.

The "varying" part implies that you define a maximum size for the array when you create its type, but the actual number of elements it holds at any given time can be less than or equal to that maximum.

**Why ?**

They are primarily used for storing a small, bounded number of elements where the order is important.

**When to Use ?**

1. When you know the **maximum number of elements** you'll ever need.
2. When the order of elements is important and you want  **numerically indexed** collection.

**Syntax**:

**CREATE [OR REPLACE] TYPE varray\_type\_name AS VARRAY(max\_size) OF element\_type;**

Create TYPE :: 🡺

This is the fundamental SQL command used to define a new **user-defined data type** in the database schema. It allows you to create custom types beyond the built-in ones.

**Ex:**

Imagine a Service catalog where user have taken multiple **services** , but the number of services per user is limited.

CREATE OR REPLACE TYPE services AS VARRAY(4) OF VARCHAR2(50);

CREATE TABLE Userss (

Userss\_id NUMBER PRIMARY KEY,

Userss\_name VARCHAR2(100),

Userss\_service services

);

INSERT INTO Userss VALUES (1, 'Cathy', services('Haircut', 'Hairwash', 'Hair Coloring'));

INSERT INTO Userss VALUES (2, 'Alan', services('Haircut', 'Beard Trimming', 'Hairwash', 'Hair Coloring'));

SELECT Userss\_name, Userss\_service FROM Userss;

**Nested Tables**

Nested tables are collections of elements, similar to a single-column table that can be "nested" within another table or used as a standalone PL/SQL variable.

Unlike varrays, nested tables do not have a predefined maximum size; they can grow dynamically.

They are particularly useful when you need to store a variable number of complex, multi-attribute items associated with a single row in a parent table.

**Why?**

Suitable for scenarios where the number of items is unknown**.**

Can store objects with multiple attributes, not just scalar values.

**When to Use ?**

1. When the number of elements is **unpredictable** and can grow or shrink dynamically.
2. When the order of elements is important, and you want **numerically indexed** collection (though gaps can be created by DELETE).
3. When you need to store the collection as a column in a database table. They are more flexible than varrays for this purpose.

**Syntax:**

**CREATE [OR REPLACE] TYPE nested\_table\_type\_name AS TABLE OF element\_type;**

**Ex:**

**Customer** placing multiple **orders**.

CREATE OR REPLACE TYPE order\_list AS TABLE OF VARCHAR2(100);

CREATE TABLE customers (

customer\_id NUMBER PRIMARY KEY,

customer\_name VARCHAR2(100),

orders order\_list

) NESTED TABLE orders STORE AS orders\_table;

INSERT INTO customers VALUES (1, 'John Doe', order\_list('Laptop', 'Smartphone', 'Tablet'));

SELECT \* FROM customers;

**Associative Arrays (Index-by Tables)**

Associative arrays, also known as index-by tables, are collections where you define the data type of the elements, and each element is associated with a unique "key" or "index."

This key can be a VARCHAR2 string or an INTEGER. They are primarily used within PL/SQL code as **temporary data structures** and **cannot be stored** directly as column types in a table.

**Why ?**

Provides very fast access to elements using their unique key.

Can grow or shrink as needed during runtime.

Useful when you don't need to store elements for every possible index (e.g., if you only have data for keys 1, 5, and 100, not 2, 3, 4, etc.)

**When to use ?**

1. When you need a **lookup table or a dictionary-like structure** where you access elements by a key (e.g., name, ID, code) rather than just a sequential index.
2. When the collection is **temporary** and only used within a PL/SQL block or package (they cannot be stored in the database).
3. When memory usage is a concern, as they only allocate space for elements you actually assign.

**Syntax**:

**TYPE associative\_array\_type\_name IS TABLE OF element\_type INDEX BY index\_type;**

**Ex:**

We can define employee’s salary based on their names using associative array.

DECLARE   
 TYPE employee\_salary IS TABLE OF NUMBER INDEX BY PLS\_INTEGER; //PLS\_INTEGER(signed integers)  
 emp\_salaries employee\_salary;  
BEGIN  
 emp\_salaries(101) := 50000;  
 emp\_salaries(102) := 60000;  
 emp\_salaries(103) := 55000;  
  
 -- Access and display the data  
 DBMS\_OUTPUT.PUT\_LINE('Salary of employee 101: ' || emp\_salaries(101));  
 DBMS\_OUTPUT.PUT\_LINE('Salary of employee 102: ' || emp\_salaries(102));  
 DBMS\_OUTPUT.PUT\_LINE('Salary of employee 103: ' || emp\_salaries(103));  
END;