PL/SQL Day 2: Detailed Trainer & Learner Handbook – Variables, Collections, Anonymous Blocks

# Session Metadata

Date: [Fill Date]

Time: 09:30 AM – 05:45 PM

Location: Hybrid (Online + Classroom)

Facilitator: Niti Dwivedi

# 1. What is PL/SQL?

PL/SQL (Procedural Language/Structured Query Language) is Oracle Corporation's procedural extension for SQL and the Oracle relational database. It combines the data manipulation power of SQL with the processing power of procedural programming languages.  
PL/SQL allows declaration of constants and variables, procedures and functions, types and variables of those types, and triggers. It can handle exceptions (runtime errors).

# 2. Benefits of PL/SQL over SQL

- \*\*Modularity\*\*: Break logic into blocks, procedures, and functions.  
- \*\*Performance\*\*: Reduce network traffic by bundling multiple SQL operations.  
- \*\*Error Handling\*\*: Provides structured error trapping and handling.  
- \*\*Portability\*\*: Runs on any Oracle-supported platform.

Real-life Example:

Instead of executing 100 individual UPDATEs from a frontend app, we write a single PL/SQL block with a loop and send one request to the DB.

## ****2. Benefits of PL/SQL over SQL (Deep Dive with Use Case)****

### ****Traditional SQL (Before PL/SQL)****

In traditional SQL:

* Every statement (like INSERT, UPDATE, SELECT) is **sent separately** from the application to the database.
* There’s **no flow control** (e.g., no IF, LOOP, or EXCEPTION handling).
* Any logic had to be written **outside** the database (e.g., in Java, .NET, Python), increasing **network calls** and **tight coupling** between app and DB.

## ****How PL/SQL Improves on SQL****

Let’s explore the four main benefits in real-world context:

### 1️⃣ ****Modularity**** – “Divide & Rule”

#### Benefit:

You can create **procedures**, **functions**, and **packages** that:

* Group logic together
* Can be **reused** across apps
* Are easier to **test**, **debug**, and **maintain**

#### Example:

CREATE OR REPLACE PROCEDURE update\_bonus(p\_dept NUMBER) IS

BEGIN

UPDATE employees SET bonus = salary \* 0.10 WHERE dept\_id = p\_dept;

END;

💬 Now you can call *update\_bonus(10);* anytime without repeating the logic!

### 2️⃣ ****Performance**** – “One Trip, Many Tasks”

#### Benefit:

PL/SQL minimizes **network traffic** by sending **one request** to the database instead of many.

#### ⚠️ Drawback without PL/SQL:

Imagine a frontend application (e.g., in Java) doing:

for (int i = 0; i < 100; i++) {

runUpdateQuery(i);

}

This makes **100 round-trips** to the database. Very slow.

#### ✅ PL/SQL Solution:

BEGIN

FOR i IN 1..100 LOOP

UPDATE orders SET status = 'Processed' WHERE order\_id = i;

END LOOP;

END;

Just **one** round-trip — significantly **faster and efficient**.

Fantastic question! You’ve touched on a very important concept — and yes, **in both PL/SQL and Java, a FOR loop executes statements one by one**. But there are **two big performance and architecture differences** when you run this loop inside the **Oracle Database (PL/SQL)** vs. doing it from an **external application (like Java)**.

## Yes, PL/SQL Loop Updates One by One — But Here's Why It’s Better:

### ⚠️ Scenario 1: Loop in Java (Client-Side)

for (int i = 1; i <= 100; i++) {

Statement stmt = conn.createStatement();

stmt.executeUpdate("UPDATE orders SET status = 'Processed' WHERE order\_id = " + i);

}

### Problem:

* Each loop **sends a separate SQL statement to the database**.
* That means: **100 separate database calls**.
* **High network overhead** 🛜 between your app and DB.
* Slower execution.
* More pressure on DB connection pool.

### ✅ Scenario 2: Loop in PL/SQL (Server-Side)

BEGIN

FOR i IN 1..100 LOOP

UPDATE orders SET status = 'Processed' WHERE order\_id = i;

END LOOP;

END;

### ✅ Advantage:

* The **entire loop runs within the database engine**.
* No client-server round-trips.
* Only **one request is sent from the client (your app)** — the whole PL/SQL block.
* **Much faster** and efficient for bulk operations.

## Summary :

| **Feature** | **Java Loop (external)** | **PL/SQL Loop (internal)** |
| --- | --- | --- |
| Location of logic | In application | In the database |
| No. of DB calls | 100 calls (1 per iteration) | 1 call (entire loop inside DB) |
| Network usage | High | Very Low |
| Execution speed | Slower | Much faster |
| Reusability | App-specific | Stored procedure, reusable |

### 3️⃣ ****Error Handling**** – “Catch Before Crash”

#### Benefit:

PL/SQL allows **structured exception handling** using EXCEPTION blocks:

* Catch errors like division by zero, missing data, etc.
* Respond gracefully instead of crashing

#### Traditional SQL:

If an error occurs in one SQL statement — execution stops. You don’t know what went wrong or how to recover.

#### PL/SQL Solution:

sql

CopyEdit

BEGIN

SELECT salary INTO v\_salary FROM employees WHERE emp\_id = 101;

DBMS\_OUTPUT.PUT\_LINE('Salary is: ' || v\_salary);

EXCEPTION

WHEN NO\_DATA\_FOUND THEN

DBMS\_OUTPUT.PUT\_LINE('No such employee found!');

END;

You handle the issue, log it, and the program **continues safely**.

### 4️⃣ ****Portability**** – “Write Once, Use Anywhere in Oracle”

#### Benefit:

PL/SQL is consistent across all Oracle platforms:

* Oracle 11g, 12c, 19c, Cloud, etc.
* Code written once can be deployed on different Oracle databases

#### In Traditional SQL:

* Logic is split between app layers
* Changing platforms means rewriting the business logic in a new language

#### PL/SQL Solution:

Store your logic in the database as **procedures/functions**, and your apps can just call them via API or drivers — no logic duplication needed.

## ****Real-World Case Study: Payroll System****

### Scenario (Without PL/SQL):

You must:

* Get employees from the HR department
* Calculate bonus
* Log missing records

Using **SQL alone**, you'd:

* Send multiple SELECTs and UPDATEs from frontend
* Handle exceptions in Java/Python
* Result: **Too many trips**, **less reuse**, **hard to debug**

### PL/SQL Alternative:

CREATE OR REPLACE PROCEDURE calculate\_hr\_bonus IS

CURSOR emp\_cur IS SELECT emp\_id, salary FROM employees WHERE dept = 'HR';

v\_bonus NUMBER;

BEGIN

FOR emp IN emp\_cur LOOP

v\_bonus := emp.salary \* 0.10;

UPDATE employees SET bonus = v\_bonus WHERE emp\_id = emp.emp\_id;

END LOOP;

COMMIT;

EXCEPTION

WHEN OTHERS THEN

DBMS\_OUTPUT.PUT\_LINE('Something went wrong: ' || SQLERRM);

END;

All logic is bundled, reusable, and secure inside the database.

## Final Comparison Table

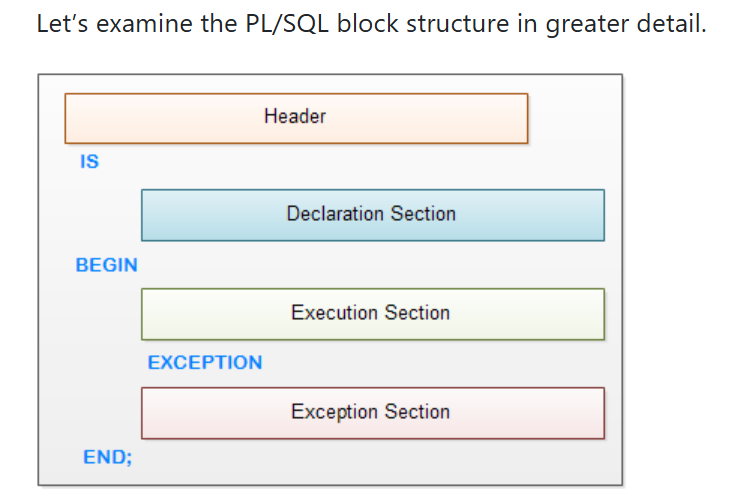
| **Feature** | **Traditional SQL** | **PL/SQL** |
| --- | --- | --- |
| Logic Handling | In external app | In database block (procedure) |
| Performance | Many round-trips | One round-trip |
| Reusability | Low (code duplication) | High (procedures/functions) |
| Error Handling | Not built-in | Built-in (EXCEPTION block) |
| Maintainability | Hard (spread out) | Easy (modular structure) |

# 3. PL/SQL Architecture

- \*\*Anonymous Block\*\*: Code block not stored permanently in the database.  
- \*\*Named Blocks\*\*: Includes procedures, functions, triggers.

<https://www.plsqltutorial.com/plsql-block-structure/>

## 🧱 PL/SQL Block Structure



The image shows the structure of a **PL/SQL block**, which consists of:

1. **Header** (optional — only in named blocks)
2. **Declaration Section** (DECLARE)
3. **Execution Section** (BEGIN ... END)
4. **Exception Section** (EXCEPTION)

Structure:

DECLARE  
 -- Variable declarations  
BEGIN  
 -- Executable statements  
EXCEPTION  
 -- Error handling logic  
END;

## 🔷 What is the ****Header**** in PL/SQL?

### 🔸 The ****Header**** is used only in ****named blocks****, such as:

* **Procedures**
* **Functions**
* **Triggers**
* **Packages**

It is the part where you define the **name** of the block and, optionally, **parameters**.

### The Header is ****not required**** in anonymous blocks.

### Examples

### 1️⃣ ****Anonymous Block (No Header)****

DECLARE

v\_msg VARCHAR2(50);

BEGIN

v\_msg := 'Hello';

DBMS\_OUTPUT.PUT\_LINE(v\_msg);

END;

/

Note if we write directly it will give error DBMS\_OUTPUT.PUT\_LINE(v\_msg);

It should be in begin and end block

|  |  |  |
| --- | --- | --- |
| **Section** | Mandatory? | Description |
| **Declaration** | Optional | Declare variables/constants/cursors |
| **Execution (BEGIN)** | Yes | Main logic happens here |
| **Exception** | Optional | Handles errors if they occur during execution |

### 2️⃣ ****Named Block with Header (Procedure)****

| **Section** | **Mandatory?** | **Description** |
| --- | --- | --- |
| **Header** | Optional | Used for named blocks like procedures/functions |

CREATE OR REPLACE PROCEDURE greet\_user(p\_name VARCHAR2)

IS -- << Header ends here

v\_msg VARCHAR2(100);

BEGIN

v\_msg := 'Hello, ' || p\_name;

DBMS\_OUTPUT.PUT\_LINE(v\_msg);

END;

/

🔹 Here, greet\_user(p\_name VARCHAR2) is the **header** of the procedure.

## 💡 Summary Table

| **Section** | **Mandatory?** | **Description** |
| --- | --- | --- |
| **Header** | Optional | Used for named blocks like procedures/functions |
| **Declaration** | Optional | Declare variables/constants/cursors |
| **Execution (BEGIN)** | Yes | Main logic happens here |
| **Exception** | Optional | Handles errors if they occur during execution |

### 🔍 Q: ****What is**** IS ****in the PL/SQL procedure?****

**Is it mandatory? What does it do?**

### ✅ Answer:

In a **named PL/SQL block** (like a **procedure**, **function**, or **trigger**), the keyword IS (or AS) is used to **begin the definition** of the block **after the header**.

### 📌 ****Syntax Pattern:****

CREATE [OR REPLACE] PROCEDURE procedure\_name (parameters)

IS

-- Declarations

BEGIN

-- Executable Statements

END;

or

CREATE [OR REPLACE] PROCEDURE procedure\_name (parameters)

AS

-- Declarations

BEGIN

-- Executable Statements

END;

### So, is IS mandatory?

**Yes — it is mandatory** in **named PL/SQL blocks**, such as:

* PROCEDURE
* FUNCTION
* TRIGGER

**But not used** in **anonymous blocks**.

### IS vs AS — Are they different?

They are **interchangeable** in most cases!

| **Keyword** | **Usage Context** | **Meaning** |
| --- | --- | --- |
| IS | More common in procedures | Start of the block body |
| AS | Also accepted (esp. in functions/triggers) | Same meaning |

Both IS and AS work — just **don’t skip them** in named blocks.

### Example with AS instead of IS:

CREATE OR REPLACE PROCEDURE greet\_user(p\_name VARCHAR2)

AS

v\_msg VARCHAR2(100);

BEGIN

v\_msg := 'Hello, ' || p\_name;

DBMS\_OUTPUT.PUT\_LINE(v\_msg);

END;

/

Same output, same functionality.

### Final Verdict:

* In **named blocks**, IS or AS is **mandatory**
* In **anonymous blocks**, you skip the header and **don't need IS**

Bottom of Form

## 🧪 ****PL/SQL Quiz: Understanding Block Structure****

### 📘 Part A: Multiple Choice (Choose the correct option)

**1. What is the purpose of the IS keyword in a PL/SQL procedure?**  
a) To initialize a variable  
b) To start the procedure body after the header  
c) To end the PL/SQL block  
d) To create a cursor

✅ **Answer:** b) To start the procedure body after the header

**2. Which of the following sections is mandatory in a PL/SQL block?**  
a) Declaration section  
b) Exception section  
c) Execution section  
d) Header section

✅ **Answer:** c) Execution section

**3. Which PL/SQL keyword is used to handle runtime errors?**  
a) HANDLE  
b) TRY  
c) EXCEPTION  
d) ERROR

✅ **Answer:** c) EXCEPTION

**4. Which keyword can be used interchangeably with IS in a named block?**  
a) BEGIN  
b) AS  
c) DEFINE  
d) WITH

✅ **Answer:** b) AS

**5. In which block type is the Header section not required?**  
a) Procedure  
b) Trigger  
c) Anonymous block  
d) Function

✅ **Answer:** c) Anonymous block

### 📘 Part B: Fill in the blanks

**6. The \_\_\_\_\_\_\_\_\_\_\_ section of a PL/SQL block is where we declare variables.**

✅ Answer: **Declaration**

**7. The DBMS\_OUTPUT.PUT\_LINE statement is used inside the \_\_\_\_\_\_\_\_\_\_\_ section to display output.**

✅ Answer: **Execution**

**8. A procedure must begin its logic body with the keyword \_\_\_\_\_\_\_\_\_\_ or \_\_\_\_\_\_\_\_\_\_.**

✅ Answer: **IS** or **AS**

**9. The last section of a PL/SQL block, used to handle exceptions, begins with the keyword \_\_\_\_\_\_\_\_\_\_.**

✅ Answer: **EXCEPTION**

**10. The block is always terminated with the keyword \_\_\_\_\_\_\_\_\_\_.**

✅ Answer: **END**

### 💡 Bonus Practical Exercise

Fill in the missing parts of this PL/SQL procedure:

CREATE OR REPLACE PROCEDURE show\_salary(p\_id NUMBER)

\_\_\_\_\_ -- (a)

v\_salary employees.salary%TYPE;

BEGIN

SELECT salary INTO v\_salary FROM employees WHERE emp\_id = p\_id;

DBMS\_OUTPUT.PUT\_LINE('Salary: ' || v\_salary);

\_\_\_\_\_\_\_\_ -- (b)

WHEN NO\_DATA\_FOUND THEN

DBMS\_OUTPUT.PUT\_LINE('Employee not found.');

END;

✅ Answers:

* (a): IS or AS
* (b): EXCEPTION

## PL/SQL block structure example

Let’s take a look at the simplest PL/SQL block that does nothing.

BEGIN

NULL;

END;Code language: SQL (Structured Query Language) (sql)

If you execute the above anonymous block in SQL\*Plus you will see that it issues a message saying:

PL/SQL procedure successfully completed.

Because the NULL statement does nothing.

To display database’s output on the screen, you need to:

* First, use the SET SERVEROUTPUT ON command to instruct SQL\*Plus to echo database’s output after executing the PL/SQL block. The SET SERVEROUTPUT ONis SQL\*Plus command, which is not related to PL/SQL.
* Second, use the DBMS\_OUTPUT.PUT\_LINE procedure to output a string on the screen.

The following example displays a message Hello PL/SQL on a screen using SQL\*Plus:

SET SERVEROUTPUT ON SIZE 1000000

BEGIN

DBMS\_OUTPUT.PUT\_LINE('Hello PL/SQL');

END;

/

## 🔍 Why it works ****without**** SET SERVEROUTPUT ON in ****Oracle Live SQL****

### ✅ In ****Oracle Live SQL****:

* SET SERVEROUTPUT ON is **enabled by default** in the **web-based Oracle Live SQL environment**.
* So, **you don’t need to manually turn it on** like you do in SQL\*Plus or SQL Developer CLI.

That's why this runs **without issues**:

sql

CopyEdit

BEGIN

DBMS\_OUTPUT.PUT\_LINE('Hello PL/SQL');

END;

And it **shows output** like:

sql

CopyEdit

Hello PL/SQL

PL/SQL procedure successfully completed.

## 🧠 But in other environments like:

### ❌ ****SQL\*Plus****

### ❌ ****SQL Developer (Worksheet)****

### ❌ ****Toad / DBeaver / CLI****

You must explicitly run:

sql

CopyEdit

SET SERVEROUTPUT ON;

**Otherwise, you won’t see DBMS\_OUTPUT text** even though your PL/SQL block executes successfully.

## 📌 Summary Table

| **Environment** | **Need SET SERVEROUTPUT ON?** | **Output visible by default?** |
| --- | --- | --- |
| Oracle Live SQL | ❌ No | ✅ Yes |
| SQL\*Plus (terminal) | ✅ Yes | ❌ No |
| Oracle SQL Developer | ✅ Yes (or click the Output tab) | ❌ No unless enabled |
| Toad / DBeaver / Others | ✅ Usually required | ❌ No unless configured |

## ✅ Trainer Tip:

When teaching students using Oracle Live SQL:

* **Mention that** **SET SERVEROUTPUT ON is auto-enabled**, but still **show them the command** — because in real-world tools, **they will need it**.

# Variables and Data Types

<https://www.plsqltutorial.com/plsql-variables/>

PL/SQL supports several scalar data types (VARCHAR2, NUMBER, DATE, etc.). You can also declare variables based on column types using `%TYPE`.

Example:

DECLARE  
 v\_name employees.emp\_name%TYPE;  
 v\_salary employees.salary%TYPE;  
BEGIN  
 SELECT emp\_name, salary INTO v\_name, v\_salary FROM employees WHERE emp\_id = 1;  
 DBMS\_OUTPUT.PUT\_LINE('Name: ' || v\_name || ', Salary: ' || v\_salary);  
END;

Use Case: Dynamically adjusting to column data types for future schema changes.

# 5. PL/SQL Collections

Collections are single-dimension data structures (like arrays). Types:  
- \*\*Associative Arrays\*\* (index-by tables): It is in-memory Unbounded sets of key-value pairs.

- \*\*VARRAYs\*\*: Fixed-size arrays.  
- \*\*Nested Tables\*\*: Variable-size arrays that can be stored in DB tables.

in PL/SQL, the keyword TABLE + INDEX BY = **Associative Array**

✅ This is the **official syntax for associative arrays** in PL/SQL:

sql

CopyEdit

TYPE <type\_name> IS TABLE OF <data\_type> INDEX BY <key\_type>;

It’s called "table" for historical reasons (like index-by tables) — but it is **not a SQL table**.

🧠 Important: Associative Arrays are the **only collection type** in PL/SQL that use INDEX BY.

## What is TYPE in PL/SQL?

### 🔹 It is a way to define ****user-defined (custom) data types**** — particularly:

* **Collections** (like arrays)
* **Records** (like structs or composite types)
* **Object types** (in advanced use)

## 🔷 1. ****"Associative Array is in-memory" — but where exactly?****

### ✅ Where is it stored?

Associative arrays in PL/SQL are stored in **RAM (Random Access Memory)** — specifically, in **the server-side memory allocated to your PL/SQL session**.

### 🔹 So:

* They **do not persist** in the database.
* They exist **only during the session** or block execution.
* Once your procedure/block ends, or session disconnects → the array is gone.

⚠️ **Not stored in ROM** (Read-Only Memory), and not written to disk unless you manually save the values into a table.

## 🔷 2. What does ****unbounded in size**** mean?

### ✅ Unbounded means:

There is **no fixed limit** or maximum number of elements for the collection.

### 💬 For Example:

plsql

CopyEdit

DECLARE

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

students student\_list;

BEGIN

FOR i IN 1..100000 LOOP

students(i) := 'Student ' || i;

END LOOP;

END;

* You didn’t declare a maximum size (VARRAY(1000)), but it can grow as long as memory allows.
* So, it is **only limited by available server memory** (RAM).

🟢 Unlike VARRAY, which is fixed (e.g., VARRAY(5)), **associative arrays grow as needed**.

## 🔷 3. What is PLS\_INTEGER?

### ✅ PLS\_INTEGER is a ****PL/SQL-only datatype**** (not a SQL datatype).

| **Feature** | **PLS\_INTEGER** |
| --- | --- |
| Type | Integer |
| Range | –2,147,483,648 to +2,147,483,647 |
| Speed | ✅ Fastest integer type in PL/SQL |
| Use | For indexing or fast math operations |
| Storage | Only in **PL/SQL memory**, not in tables |

It is faster than NUMBER in PL/SQL because it uses native machine arithmetic.

Note \* You can also index by string:

plsql

CopyEdit

TYPE student\_grades IS TABLE OF VARCHAR2(10) INDEX BY VARCHAR2(20);

Then:

plsql

CopyEdit

grades('EMP001') := 'A';

grades('EMP002') := 'B';

## 🔷 4. Is indexing done automatically in associative arrays?

### ⚠️ Not exactly "automatically" — ****you control the keys**** (indexes).

In associative arrays:

* **You decide the index values.**
* If you use INDEX BY PLS\_INTEGER, you insert like: my\_array(1) := 'value';
* If you use INDEX BY VARCHAR2(100), you insert like: my\_array('emp001') := 'John';

### Example:

plsql

CopyEdit

DECLARE

TYPE grade\_table IS TABLE OF VARCHAR2(10) INDEX BY PLS\_INTEGER;

grades grade\_table;

BEGIN

grades(1) := 'A';

grades(100) := 'B'; -- You decide the index!

grades(500) := 'C';

DBMS\_OUTPUT.PUT\_LINE(grades(100)); -- Output: B

END;

No default indexing like a normal array starting from 0 or 1. You manually assign keys.

## ✅ Summary

| **Concept** | **Explanation** |
| --- | --- |
| **In-memory?** | Yes, stored in **RAM (PGA memory)**, only during session |
| **Persistent?** | ❌ No, associative arrays disappear after block/session ends |
| **Unbounded?** | Yes, grows as needed — limited only by server memory |
| **PLS\_INTEGER?** | Fastest PL/SQL integer type; used for indexing |
| **Indexing?** | You provide keys (integer or string), nothing is auto-indexed |

## ✅ Summary of Keywords

| **Keyword** | **Role** |
| --- | --- |
| TYPE | Creates a **new PL/SQL type** (in this case, a collection) |
| IS TABLE OF | Declares a **collection of values** |
| VARCHAR2(10) | Each element will be a VARCHAR2 value (max 10 characters) |
| INDEX BY | Means you're defining an **Associative Array** |
| PLS\_INTEGER | The datatype for the **index/key** (faster than NUMBER) |

## 5.1 Associative Arrays – Example Use Case

## ✅ Example 1: Defining a ****custom collection**** type (Associative Array)

sql

CopyEdit

DECLARE

TYPE grade\_table IS TABLE OF VARCHAR2(10) INDEX BY PLS\_INTEGER;

grades grade\_table;

BEGIN

grades(1) := 'A';

grades(2) := 'B';

DBMS\_OUTPUT.PUT\_LINE(grades(2));

END;

* TYPE grade\_table → You are creating a **new custom data type**
* grades grade\_table; → You’re declaring a variable **of that custom type**

### 🔹 3. Why is it still ****unbounded****?

Even though you say VARCHAR2(10), you're **defining the data type of each element**, not the number of elements.

So:

* 'VARCHAR2(10)' means: each value in the array can be **up to 10 characters long**
* **It does NOT mean** the array is limited to 10 elements

✅ Associative arrays are unbounded because:

* You can keep inserting any number of values
* You can use any valid key (1, 100, 500, 999999, etc.)
* There’s **no fixed limit** or MAXSIZE like VARRAY

## 🔍 What does VARCHAR2(10) mean here?

* It means **each value in the associative array can hold up to 10 characters.**
* So you’re telling Oracle:

“Every element of my collection is a string, with max length = 10 characters.”

## 🔎 Why would we restrict size like this?

To:

1. **Optimize memory**: If you're storing codes, grades, short strings (like 'Pass', 'Fail', 'Grade A+'), you don’t need 100 or 1000 characters.
2. **Prevent overflows**: You want to avoid storing unintended long text.

### ✅ Real-World Example:

Imagine you're storing **student grades**, and the values can be like:

* "A+"
* "B-"
* "Distinction" → that's 11 characters ❌ (would raise an error if you insert into a VARCHAR2(10) field)

## 🧪 Demonstration Code:

plsql

CopyEdit

DECLARE

TYPE grade\_table IS TABLE OF VARCHAR2(10) INDEX BY PLS\_INTEGER;

grades grade\_table;

BEGIN

grades(1) := 'A+'; -- ✅ OK (2 characters)

grades(2) := 'Distinction'; -- ❌ Error: string length exceeds 10

DBMS\_OUTPUT.PUT\_LINE(grades(1));

DBMS\_OUTPUT.PUT\_LINE(grades(2)); -- Will not run due to the above error

END;

### 🛑 Error you’ll get:

sql

CopyEdit

ORA-06502: PL/SQL: numeric or value error: character string buffer too small

## ✅ Can we store both ****int**** and ****string**** values in the same associative array?

❌ **No.** You can only store **one data type per collection** in PL/SQL.

## 🔧 Example (Correct Usage — Single Type):

plsql

CopyEdit

DECLARE

TYPE grades\_type IS TABLE OF VARCHAR2(10) INDEX BY PLS\_INTEGER;

grades grades\_type;

BEGIN

grades(1) := 'A+';

grades(2) := 'B';

-- grades(3) := 100; -- ❌ Error: assigning a number to VARCHAR2

DBMS\_OUTPUT.PUT\_LINE(grades(1));

END;

🔴 grades(3) := 100; will raise **ORA-06550** because you're assigning a NUMBER to a VARCHAR2 array.

## ✅ Syntax: Associative Array for Integer Values

plsql

CopyEdit

TYPE array\_name IS TABLE OF <numeric\_datatype> INDEX BY PLS\_INTEGER;

* NUMBER → General numeric type (can be integer or decimal)
* PLS\_INTEGER → Optimized integer type for PL/SQL (faster in memory)

### 🔹 Example 1: Storing Marks (Using NUMBER)

plsql

CopyEdit

DECLARE

TYPE marks\_table IS TABLE OF NUMBER INDEX BY PLS\_INTEGER;

student\_marks marks\_table;

BEGIN

student\_marks(101) := 85;

student\_marks(102) := 92;

student\_marks(103) := 77;

DBMS\_OUTPUT.PUT\_LINE('Student 102 Score: ' || student\_marks(102));

END;

🟢 Output:

yaml

CopyEdit

Student 102 Score: 92

### 🔹 Example 2: Using PLS\_INTEGER for Better Performance

plsql

CopyEdit

DECLARE

TYPE int\_array IS TABLE OF PLS\_INTEGER INDEX BY PLS\_INTEGER;

values\_array int\_array;

BEGIN

values\_array(1) := 10;

values\_array(2) := 20;

values\_array(5) := 100;

DBMS\_OUTPUT.PUT\_LINE('Value at index 5: ' || values\_array(5));

END;

## ⚡ PLS\_INTEGER vs. NUMBER

| **Feature** | **PLS\_INTEGER** | **NUMBER** |
| --- | --- | --- |
| Type | Whole numbers only | Supports decimals too |
| Performance | ✅ Fast (native) | Slower in PL/SQL |
| Range | ±2 billion | Much wider |
| Use For | Loop counters, indexes | Prices, salaries, etc. |

## 📝 Summary

To store integers in associative arrays:

| **Use Case** | **Data Type to Use** | **Example** |
| --- | --- | --- |
| Any numeric (int/float) | NUMBER | TABLE OF NUMBER INDEX BY ... |
| Fast integer-only ops | PLS\_INTEGER | TABLE OF PLS\_INTEGER INDEX BY ... |

### ✅ Option 2: ****Use two separate arrays****

plsql

CopyEdit

DECLARE

TYPE name\_table IS TABLE OF VARCHAR2(100) INDEX BY PLS\_INTEGER;

TYPE score\_table IS TABLE OF NUMBER INDEX BY PLS\_INTEGER;

names name\_table;

scores score\_table;

BEGIN

names(1) := 'Alice';

scores(1) := 85;

names(2) := 'Bob';

scores(2) := 90;

DBMS\_OUTPUT.PUT\_LINE(names(1) || ': ' || scores(1));

END;

Exercise for learners

Create a associative array to store learners outstanding performance as excellent ,average and poor

For eg:

DECLARE

TYPE performance\_table IS TABLE OF VARCHAR2(10) INDEX BY PLS\_INTEGER; -- declaring or creating a custom defined type of an associative array

performance performance\_table; --  This declares a **variable** grades of the new collection type you

defined above.

 performance is now your associative array.

BEGIN

performance(1) := 'Excellent ';

performance(100) := ‘Average';

performance(500) := 'Poor';

DBMS\_OUTPUT.PUT\_LINE(performance(100));

END;

Now see a record type

## ✅ Example 2: Defining a ****custom record**** type

sql

CopyEdit

DECLARE

TYPE emp\_record\_type IS RECORD (

emp\_id NUMBER,

emp\_name VARCHAR2(50)

);

v\_emp emp\_record\_type;

BEGIN

v\_emp.emp\_id := 101;

v\_emp.emp\_name := 'John';

DBMS\_OUTPUT.PUT\_LINE(v\_emp.emp\_id || ' - ' || v\_emp.emp\_name);

END;

Here, emp\_record\_type is a **custom composite datatype**, and v\_emp is a variable of that record type.

## 📘 So Yes — TYPE in PL/SQL:

| **Concept** | **Explanation** |
| --- | --- |
| Custom Data Type | ✅ Yes, created using TYPE keyword |
| Reusable? | ✅ Yes, within the block/package where it’s defined |
| Used For | Collections, records, object types |
| Similar To | Struct in C/C++, Class without methods, Table definition |

## 🔍 ****In Associative Arrays (INDEX BY)**** — Do indexes start from 0 or follow any order?

### ❌ ****No, Associative Arrays in PL/SQL do NOT require sequential indexing.****

## 🔹 Associative Arrays are ****sparse**** and ****flexible****:

* You can use **any valid integer as a key** — positive or negative.
* The array **does not require keys to start at 0 or 1**.
* You can have keys like: grades(-5), grades(100), grades(9999).

## ✅ Real Examples:

plsql

CopyEdit

DECLARE

TYPE grade\_table IS TABLE OF VARCHAR2(10) INDEX BY PLS\_INTEGER;

grades grade\_table;

BEGIN

grades(100) := 'A';

grades(5000) := 'B';

grades(-3) := 'F';

DBMS\_OUTPUT.PUT\_LINE(grades(100)); -- A

DBMS\_OUTPUT.PUT\_LINE(grades(-3)); -- F

END;

✅ You inserted values at keys 100, 5000, and –3 — and it worked fine.

## 📌 So Why is it Called "Index"?

In PL/SQL, the **index** just means the **key** used to refer to a value in the array.

But unlike regular arrays in other languages:

* It’s **not automatically assigned**
* It’s **not required to be sequential**

## 🔍 Can I start from 0?

Yes. You can use 0 if you like:

plsql

CopyEdit

grades(0) := 'Pass';

PL/SQL does **not assume a default starting index** — you decide what to use.

## 🔁 Summary Table

| **Feature** | **Associative Array in PL/SQL** |
| --- | --- |
| Starts from index 0? | ❌ No, you decide the index |
| Must be sequential? | ❌ No, can be 1, 100, –10, 5000 |
| Must be unique keys? | ✅ Yes, each index must be unique |
| Can use negative keys? | ✅ Yes |
| Can access with .FIRST? | ✅ Yes (returns the lowest used index) |

## Bonus Tip 🎯: Use .FIRST and .LAST

To loop over dynamic associative arrays:

plsql

CopyEdit

FOR i IN grades.FIRST .. grades.LAST LOOP

IF grades.EXISTS(i) THEN

DBMS\_OUTPUT.PUT\_LINE(i || ' => ' || grades(i));

END IF;

END LOOP;

✅ .EXISTS(i) helps skip over unused indices.

Use Case: Store last employee entered for each department. We will see when we will learn for and loop

Example:

DECLARE  
 TYPE emp\_table\_type IS TABLE OF VARCHAR2(100) INDEX BY PLS\_INTEGER;  
 v\_emps emp\_table\_type;  
BEGIN  
 FOR r IN (SELECT dept\_id, emp\_name FROM employees) LOOP  
 v\_emps(r.dept\_id) := r.emp\_name;  
 END LOOP;  
 DBMS\_OUTPUT.PUT\_LINE('Dept 10 Employee: ' || v\_emps(10));  
END;

## 5.2 VARRAYs – Example Use Case

## ✅ What is a VARRAY in PL/SQL?

A **VARRAY** (Variable-Size Array) is a **collection type** in PL/SQL used to **store a fixed number of elements** of the same data type — like a list of grades, scores, names, etc.

## 🔎 Syntax to Declare a VARRAY

plsql

CopyEdit

TYPE <type\_name> IS VARRAY(<max\_size>) OF <datatype>;

Then you create a variable of that type:

plsql

CopyEdit

<variable\_name> <type\_name> := <type\_name>();

## ✅ Simple Example: Store 3 recent hire names

plsql

CopyEdit

DECLARE

TYPE name\_list IS VARRAY(3) OF VARCHAR2(50); -- max 3 elements

v\_names name\_list := name\_list(); -- initialize

BEGIN

v\_names.EXTEND;

v\_names(1) := 'Alice';

v\_names.EXTEND;

v\_names(2) := 'Bob';

v\_names.EXTEND;

v\_names(3) := 'Charlie';

DBMS\_OUTPUT.PUT\_LINE(v\_names(2)); -- Output: Bob

END;

## 🔹 Answers to Common Questions (Like Associative Arrays)

### 🔸 1. ****Where is VARRAY stored?****

* If used in a **PL/SQL block**, it is stored **in memory (RAM)** like associative arrays.
* But VARRAYs can also be used as **columns in database tables** — unlike associative arrays.

### 🔸 2. Is it ****bounded or unbounded****?

❌ **Bounded** → You must declare the **maximum number of elements** at the time of type creation.

plsql

CopyEdit

TYPE my\_varray IS VARRAY(5) OF VARCHAR2(50); -- Only 5 allowed

If you try to insert 6 values — you'll get an error:

bash

CopyEdit

ORA-06532: Subscript outside of limit

### 🔸 3. What is the VARCHAR2(50) doing here?

Just like associative arrays:

* It **defines the maximum size of each value** (element), NOT the number of elements.
* So if you write VARCHAR2(10), each string in the array must be **10 characters or fewer**.

🧪 Example:

plsql

CopyEdit

TYPE colors IS VARRAY(3) OF VARCHAR2(5); -- Max 3 items, each up to 5 chars

### 🔸 4. Does VARRAY have default index values?

✅ Yes — VARRAY indexes always start from 1 to COUNT, in order.

So if your VARRAY has 3 elements:

pl

CopyEdit

v\_array(1), v\_array(2), v\_array(3)

You can’t skip an index or use non-sequential keys like you can in associative arrays.

### 🔸 5. Can I store both int and string in one VARRAY?

❌ No — just like associative arrays, VARRAYs are **homogeneous**:  
They store **only one data type**.

But you can define a RECORD to store multiple fields (string + number), and then store those in a VARRAY.

## 🧠 Comparison with Associative Array

| **Feature** | **VARRAY** | **Associative Array** |
| --- | --- | --- |
| Indexing | Always starts from 1 | You choose the index |
| Size Limit | ❌ Fixed size (bounded) | ✅ Unlimited (unbounded) |
| Memory or DB column? | ✅ Can be used in DB columns | ❌ Only in memory (PL/SQL block) |
| Use Case | Top N records, fixed size lists | Fast lookup, sparse data structure |
| Key type | Always sequential integer | Integer or string (PLS\_INTEGER, VARCHAR2) |
| Declaration | VARRAY(n) OF datatype | TABLE OF datatype INDEX BY ... |

## 🧪 Bonus Example: Storing Top 3 Salaries

plsql

CopyEdit

DECLARE

TYPE salary\_array IS VARRAY(3) OF NUMBER;

top\_salaries salary\_array := salary\_array();

BEGIN

top\_salaries.EXTEND(3);

top\_salaries(1) := 80000;

top\_salaries(2) := 75000;

top\_salaries(3) := 70000;

FOR i IN 1 .. top\_salaries.COUNT LOOP

DBMS\_OUTPUT.PUT\_LINE('Salary ' || i || ': ' || top\_salaries(i));

END LOOP;

END;

## 📝 Summary

| **Concept** | **VARRAY Explanation** |
| --- | --- |
| Fixed Size? | ✅ Yes (declared using VARRAY(n)) |
| Value Data Type? | You define it (VARCHAR2(50), NUMBER, etc.) |
| Can store mixed data types? | ❌ No (but you can use a RECORD type) |
| Indexing type? | Always sequential (1 to COUNT) |
| Where used? | In memory or in database table columns |
| Stored in DB? | ✅ Yes (can be used in tables, unlike associative arrays) |
| Must be initialized? | ✅ Yes (:= type\_name();) |
| Error on overfilling? | ✅ Yes (ORA-06532) |

Would you like:

* A .sql practice lab to experiment with VARRAYs?
* A variant showing a RECORD inside a VARRAY (like name + salary together)?

Let me know and I’ll send them right away.

##### You said:

DECLARE TYPE name\_list IS VARRAY(3) OF VARCHAR2(50); -- max 3 elements v\_names name\_list := name\_list(); -- initialize BEGIN v\_names.EXTEND; v\_names(1) := 'Alice'; v\_names.EXTEND; v\_names(2) := 'Bob'; v\_names.EXTEND; v\_names(3) := 'Charlie'; DBMS\_OUTPUT.PUT\_LINE(v\_names(2)); -- Output: Bob END;so in this example what is v\_names.Extend , why we use and is it mandatory in syntax and varray can also be created as integer like Number or PLS\_index

###### ChatGPT said:

Great! Let’s break this down step-by-step so it's super clear for you — and it answers both:

* ✅ What v\_names.EXTEND is doing
* ✅ Whether it’s mandatory
* ✅ And whether VARRAYs can hold NUMBER or PLS\_INTEGER

## ✅ First: What is v\_names.EXTEND?

### 🔹 Definition:

In PL/SQL, **.EXTEND is a method used to increase the size of a collection**, one element at a time.

### 🔸 Why do we use it in VARRAYs?

Because **VARRAYs start empty**, and you must **explicitly grow them** before you assign values.

#### 👉 Example:

pl

CopyEdit

v\_names.EXTEND;

v\_names(1) := 'Alice'; -- Only works AFTER extend

If you don’t EXTEND, this line will throw an error:

makefile

CopyEdit

ORA-06533: Subscript beyond count

### 🔸 Is EXTEND mandatory?

✅ Yes, in VARRAYs — you **must call EXTEND** to allocate space before using an index.

Unlike associative arrays (which can accept any index directly), VARRAYs are **dense and ordered**, and **must be filled in order**.

## ✅ Then: Can a VARRAY hold integers like NUMBER or PLS\_INTEGER?

✅ Yes!

You can define a VARRAY to store **any scalar data type**, including:

* NUMBER → for salaries, marks, IDs
* PLS\_INTEGER → for loop counters or integer values
* DATE → for storing dates

### 🔸 Example: VARRAY of NUMBER

plsql

CopyEdit

DECLARE

TYPE num\_list IS VARRAY(3) OF NUMBER;

v\_nums num\_list := num\_list(); -- initialize

BEGIN

v\_nums.EXTEND(3);

v\_nums(1) := 100;

v\_nums(2) := 200;

v\_nums(3) := 300;

DBMS\_OUTPUT.PUT\_LINE(v\_nums(2)); -- Output: 200

END;

✅ Works perfectly — VARRAYs are **flexible in the type of data** they store, but all elements must be of **the same type**.

## 🧠 Summary Table

| **Concept** | **Explanation** |
| --- | --- |
| EXTEND in VARRAY | Grows the VARRAY by 1 (or n) elements so you can store values |
| Mandatory? | ✅ Yes — You **must** extend before assigning a value |
| Default size? | ❌ No — You start with 0 elements even if VARRAY(3) is declared |
| Can store integers? | ✅ Yes (use NUMBER or PLS\_INTEGER) |
| Dense or Sparse? | ✅ Dense (no skipping indexes like 1, 100) |
| Indexed From | Always from 1 to COUNT |

## ✅ Bonus Tip: Extend in Bulk

You can grow all 3 positions at once:

plsql

CopyEdit

v\_names.EXTEND(3); -- Allocates space for 3 elements at once

Would you like a .sql practice sheet that includes VARRAYs of NUMBER, VARCHAR2, and a case where forgetting EXTEND throws an error?

##### You said:

and here VARRAY(3) means what the size of character of values stored or what because how we can say varray is fixed as every time we are extending it

###### ChatGPT said:

Excellent question! You're catching the **subtle but important** difference between:

* **Number of elements in the VARRAY**
* vs.
* **Size of each value inside those elements**

Let’s explain it step-by-step in very clear terms.

## 🔹 VARRAY(3) — What does this ****number 3**** mean?

✅ It means the **maximum number of elements** the array can hold — not the size of characters.

pl

CopyEdit

TYPE name\_list IS VARRAY(3) OF VARCHAR2(50);

### 🟩 Break it down:

| **Part** | **Meaning** |
| --- | --- |
| VARRAY(3) | The array can store **up to 3 items** total (fixed!) |
| OF VARCHAR2(50) | Each item (element) can store a **string of up to 50 characters** |

## 🔸 Then why do we use .EXTEND if the size is fixed?

Great question! Here's the answer:

* VARRAYs are **fixed in their maximum capacity** (in this case, max 3 items).
* But **they start empty** (COUNT = 0) until you explicitly add items.
* You use .EXTEND to **fill** them — up to their maximum limit.

You’re not changing the array’s size — you're just **allocating and filling it**, up to the allowed limit.

Use Case: Store exactly 3 recent hires for a report.we will do this use case when we will study loop and for etc.

DECLARE  
 TYPE recent\_hires IS VARRAY(3) OF VARCHAR2(100);  
 v\_list recent\_hires := recent\_hires();  
BEGIN  
 FOR r IN (SELECT emp\_name FROM employees ORDER BY hire\_date DESC FETCH FIRST 3 ROWS ONLY) LOOP  
 v\_list.EXTEND;  
 v\_list(v\_list.LAST) := r.emp\_name;  
 END LOOP;  
 FOR i IN 1..v\_list.COUNT LOOP  
 DBMS\_OUTPUT.PUT\_LINE(i || '. ' || v\_list(i));  
 END LOOP;  
END;

## 5.3 Nested Tables – Example Use Case

Absolutely! Let’s now break down **Nested Tables** in PL/SQL with the same clarity and simplicity as we did for Associative Arrays and VARRAYs.

## ✅ What is a ****Nested Table**** in PL/SQL?

A **Nested Table** is a **variable-size, ordered collection** — similar to a VARRAY — but more flexible:

* It can grow **dynamically** (unlike VARRAY which is bounded).
* It can also be stored in **database tables** as columns (unlike Associative Arrays).

## 🔸 Syntax for Declaring a Nested Table:

plsql

CopyEdit

TYPE <type\_name> IS TABLE OF <datatype>;

Then declare a variable of that type:

plsql

CopyEdit

<variable\_name> <type\_name> := <type\_name>();

✅ The array starts empty and must be initialized using the constructor := <type\_name>().

## 🧪 Example: Store a list of employee names

plsql

CopyEdit

DECLARE

TYPE emp\_list\_type IS TABLE OF VARCHAR2(100);

v\_emp\_list emp\_list\_type := emp\_list\_type(); -- Initialize the table

BEGIN

-- Manually adding values

v\_emp\_list.EXTEND;

v\_emp\_list(1) := 'Alice';

v\_emp\_list.EXTEND;

v\_emp\_list(2) := 'Bob';

v\_emp\_list.EXTEND;

v\_emp\_list(3) := 'Charlie';

FOR i IN 1 .. v\_emp\_list.COUNT LOOP

DBMS\_OUTPUT.PUT\_LINE('Employee: ' || v\_emp\_list(i));

END LOOP;

END;

✅ Output:

makefile

CopyEdit

Employee: Alice

Employee: Bob

Employee: Charlie

## 🔍 Nested Table vs VARRAY vs Associative Array

| **Feature** | **Nested Table** | **VARRAY** | **Associative Array** |
| --- | --- | --- | --- |
| Indexing | 1 to N (sequential) | 1 to N (bounded) | Any key (sparse, flexible) |
| Fixed size? | ❌ No (dynamic size) | ✅ Yes (fixed max elements) | ❌ No (unbounded) |
| Can be stored in DB? | ✅ Yes | ✅ Yes | ❌ No (PL/SQL only) |
| Allows gaps in index? | ✅ Yes (after delete) | ❌ No | ✅ Yes |
| Use case | Lists, dynamic groups | Top-N, ordered results | Key-based fast access |

## ✅ Real-Life Use Case

💡 **Scenario:** Get a list of employee names from the IT department and loop through them.

plsql

CopyEdit

DECLARE

TYPE name\_table IS TABLE OF VARCHAR2(100);

it\_employees name\_table;

BEGIN

SELECT emp\_name BULK COLLECT INTO it\_employees

FROM employees

WHERE department = 'IT';

FOR i IN 1..it\_employees.COUNT LOOP

DBMS\_OUTPUT.PUT\_LINE('IT Employee: ' || it\_employees(i));

END LOOP;

END;

✅ Uses BULK COLLECT to fetch rows directly into the nested table.

## 🔧 Common Methods with Nested Tables

| **Method** | **Use** |
| --- | --- |
| EXTEND(n) | Add n empty rows |
| TRIM | Remove rows from the end |
| DELETE | Delete all or a specific index (DELETE(3)) |
| COUNT | Total elements present |
| FIRST / LAST | First and last valid indexes |

## 🔥 Bonus Feature: Can be used in Table Columns

sql

CopyEdit

CREATE TYPE string\_list AS TABLE OF VARCHAR2(50);

Then in a table:

sql

CopyEdit

CREATE TABLE project\_teams (

team\_id NUMBER,

team\_members string\_list

) NESTED TABLE team\_members STORE AS team\_member\_tab;

✅ This allows one row to have a list of values (a team of members) — stored properly in DB.

## ✅ Summary

| **Feature** | **Explanation** |
| --- | --- |
| Collection Type | Ordered, dynamic-size collection |
| Declared as | TYPE name IS TABLE OF <type>; |
| Initialization | Required: := name(); |
| Storage in DB | ✅ Yes (can be used in DB tables as nested columns) |
| Size | Dynamic (not fixed like VARRAY) |
| Indexing | Sequential (1 to COUNT) |
| Can use BULK COLLECT? | ✅ Yes |

Would you like:

* A .sql file with nested table examples using EXTEND, TRIM, and BULK COLLECT?
* Or a case study (like student marks per subject) implemented with nested tables?

Let me know!

##### You said:

varray can also be stored in db just like table

###### ChatGPT said:

✅ Yes, you're absolutely right!

Both **VARRAYs** and **Nested Tables** can be **stored in database tables** — but they work **a bit differently**. Let me explain clearly:

## 🧠 Can VARRAYs Be Stored in Tables?

✅ **Yes**, a VARRAY can be used as a **column data type** in a database table — but with some **restrictions** compared to nested tables.

## 🔸 Step-by-Step: How to Store a VARRAY in a Table

### ✅ Step 1: Create a VARRAY type at the schema level

sql

CopyEdit

CREATE OR REPLACE TYPE phone\_array AS VARRAY(3) OF VARCHAR2(15);

This creates a **user-defined type** called phone\_array, which can store up to 3 phone numbers per row.

### ✅ Step 2: Use it in a table

sql

CopyEdit

CREATE TABLE customers (

customer\_id NUMBER PRIMARY KEY,

name VARCHAR2(50),

phones phone\_array -- column of VARRAY type

);

### ✅ Step 3: Insert data into it

sql

CopyEdit

INSERT INTO customers

VALUES (1, 'John Doe', phone\_array('9876543210', '9123456789', '9801234567'));

INSERT INTO customers

VALUES (2, 'Alice', phone\_array('9001122334'));

### ✅ Step 4: Query from it

sql

CopyEdit

SELECT name, phones(1) AS primary\_phone FROM customers;

You can access specific elements in the VARRAY using **phones(index)**.

## 📌 Comparison: VARRAY vs Nested Table in Tables

| **Feature** | **VARRAY** | **Nested Table** |
| --- | --- | --- |
| Fixed size | ✅ Yes | ❌ No (can grow dynamically) |
| Stored in DB column? | ✅ Yes | ✅ Yes |
| Indexed by | Integer (1 to N) | Integer (1 to N) |
| Can be joined easily? | ❌ Harder | ✅ Easier with NESTED TABLE clause |
| Requires storage table? | ❌ No | ✅ Yes (needs storage table) |

## 🚧 Limitations of VARRAY in Tables

| **Limitation** | **Explanation** |
| --- | --- |
| Fixed-size | Must define max elements (e.g., VARRAY(5)) |
| Max size constraint | Cannot grow beyond the limit |
| Not ideal for large collections | Best for small, fixed sets (like top 3 scores, phone numbers) |
| Hard to join | Cannot easily JOIN elements from within the VARRAY |

## 📝 Summary

| **Feature** | **VARRAY in Tables** |
| --- | --- |
| Allowed? | ✅ Yes |
| Easy to use? | ✅ For fixed small lists |
| Max Size? | ✅ Must define at creation |
| Grows dynamically? | ❌ No (unlike nested table) |
| Queryable? | ✅ Yes, using column(index) syntax |
| Use case example | List of up to 3 phone numbers, top 5 items, etc. |

Use Case: Load employees from a selected department dynamically.

DECLARE  
 TYPE emp\_list\_type IS TABLE OF VARCHAR2(100);  
 v\_emp\_list emp\_list\_type := emp\_list\_type();  
BEGIN  
 SELECT emp\_name BULK COLLECT INTO v\_emp\_list FROM employees WHERE dept\_id = 20;  
 FOR i IN 1..v\_emp\_list.COUNT LOOP  
 DBMS\_OUTPUT.PUT\_LINE('Employee: ' || v\_emp\_list(i));  
 END LOOP;  
END;

# 6. Case Study Data Setup

CREATE TABLE employees (  
 emp\_id NUMBER PRIMARY KEY,  
 emp\_name VARCHAR2(100),  
 salary NUMBER,  
 dept\_id NUMBER,  
 hire\_date DATE  
);  
  
INSERT INTO employees VALUES (1, 'John Doe', 5000, 10, DATE '2020-01-01');  
INSERT INTO employees VALUES (2, 'Jane Smith', 6000, 20, DATE '2019-03-15');  
INSERT INTO employees VALUES (3, 'Robert Brown', 4500, 10, DATE '2021-10-10');