### **Project Context Recap: IT Helpdesk Ticketing System**

This system is designed to manage technical support requests in an organization. Each ticket logs:

* The issue reported
* Its severity
* The assigned engineer
* Its current status

We aim to **simulate real-time ticket data processing** using PL/SQL constructs that are also applied in **production-grade ITSM platforms** like ServiceNow, Freshdesk

## Objective:

Automate IT ticket logging, categorization, engineer assignment, and status tracking using PL/SQL constructs.

## Components Used:

* Scalar & Anchored Variables
* Collections: Associative Array, Nested Table, VARRAY
* Object Types & Records
* Views & Inline Views
* Control Structures (IF, CASE, Loops)
* Exception Handling: PRAGMA, Predefined, User-defined
* Cursors: Implicit, Explicit, Parameterized

## Step 1: Create Supporting Tables

CREATE TABLE engineers (  
 engineer\_id NUMBER PRIMARY KEY,  
 engineer\_name VARCHAR2(100),  
 department VARCHAR2(50)  
);  
  
CREATE TABLE tickets (  
 ticket\_id NUMBER PRIMARY KEY,  
 issue\_desc VARCHAR2(200),  
 severity CHAR(1),  
 assigned\_to NUMBER,  
 created\_on DATE,  
 status VARCHAR2(20)  
);  
  
CREATE SEQUENCE ticket\_seq START WITH 1001 INCREMENT BY 1;

## Step 2: Insert Data Using Scalar & Anchored Types

DECLARE  
 v\_id NUMBER := ticket\_seq.NEXTVAL;  
 v\_desc tickets.issue\_desc%TYPE := 'System Crash';  
 v\_status VARCHAR2(20) := 'OPEN';  
BEGIN  
 INSERT INTO tickets VALUES (v\_id, v\_desc, 'H', 101, SYSDATE, v\_status);  
 DBMS\_OUTPUT.PUT\_LINE('Ticket created with ID ' || v\_id);  
END;

## Step 3: Object Type, VARRAY, and CASE

ticket\_obj Defined a class-like datatype

ticket\_list Created a fixed-size array of ticket\_obj

ticket\_obj(...) Instantiated individual objects

ticket\_list(...) Used constructor to store those objects in array form

v\_list Is now a fully ready-to-loop list of in-memory tickets

-- Defined a class-like datatype

CREATE OR REPLACE TYPE ticket\_obj AS OBJECT (  
 id NUMBER,  
 desc VARCHAR2(100),  
 severity CHAR(1)  
);

-- Created a fixed-size array of ticket\_obj  
CREATE OR REPLACE TYPE ticket\_list IS VARRAY(5) OF ticket\_obj;  
  
DECLARE

-- Used list constructor to store those objects in array form   
 v\_list ticket\_list := ticket\_list(

-- Instantiated individual objects

ticket\_obj(1011, 'VPN Fail', 'M'),

ticket\_obj(1012, 'Email Down', 'H'));

BEGIN

-- Is now a fully ready-to-loop list of in-memory tickets  
 FOR i IN 1..v\_list.COUNT LOOP  
 CASE v\_list(i).severity  
 WHEN 'H' THEN DBMS\_OUTPUT.PUT\_LINE('High Severity: ' || v\_list(i).desc);  
 ELSE DBMS\_OUTPUT.PUT\_LINE('Other Severity: ' || v\_list(i).desc);  
 END CASE;  
 END LOOP;  
END;

When support teams interact with tickets, they often:

* View batches of issues
* Sort or group them by **severity**
* Prioritize high-impact problems like system crashes or network failures

So in our implementation:

* We simulate a **list of recent tickets**
* Use CASE to handle **severity-based logic**
* Structure data cleanly using **object and collection types**

So, Can We Populate It from the tickets Table?

DECLARE

v\_list ticket\_list := ticket\_list();

BEGIN

FOR r IN (

SELECT ticket\_id, issue\_desc, severity

FROM tickets

WHERE ROWNUM <= 5

) LOOP

v\_list.EXTEND(3);

v\_list(v\_list.LAST) := ticket\_obj(r.ticket\_id, r.issue\_desc, r.severity);

END LOOP;

END;

V\_list is a collection (usually a varray or a nested table ) that stores objects of type ticket\_obj

So v\_list holds a list of ticket\_obj values , but you can’t assign data to a collection slot unless it exists.

And extend allocates a new slot in the collection --- ( Add a new row to my array )

Count tells how many elements currently exist in the collection so after extend ,count will point to the last (newly added ) index which means assign a new ticket object to the last (new) position just created.

**Insert multiple new ticket records** into the actual tickets table using:

* An **object type** to define ticket structure
* A **VARRAY or Nested Table** to hold the bulk tickets
* A **FORALL** statement for efficient bulk insertion

### Step 1: Define the Object and Collection Type

-- 1. Object Type representing a Ticket

CREATE OR REPLACE TYPE ticket\_obj AS OBJECT (

issue\_desc VARCHAR2(100),

severity CHAR(1),

assigned\_to NUMBER,

created\_on DATE,

status VARCHAR2(20)

);

-- 2. Nested Table Type (for bulk insert use)

CREATE OR REPLACE TYPE ticket\_ntt IS TABLE OF ticket\_obj;

We're using a **nested table type** here because it's ideal for dynamic/bulk operations and usable with FORALL.

### Step 2: PL/SQL Block to Bulk Insert into tickets Table

DECLARE

-- 1. Collection of ticket\_obj

v\_tickets ticket\_ntt := ticket\_ntt(

ticket\_obj('Internet not working', 'H', 101, SYSDATE, 'OPEN'),

ticket\_obj('Email failure', 'M', 102, SYSDATE, 'OPEN'),

ticket\_obj('VPN Login issue', 'L', 103, SYSDATE, 'OPEN')

);

BEGIN

-- 2. Bulk Insert using FORALL

FORALL i IN v\_tickets.FIRST .. v\_tickets.LAST

INSERT INTO tickets (

ticket\_id,

issue\_desc,

severity,

assigned\_to,

created\_on,

status

) VALUES (

ticket\_seq.NEXTVAL,

v\_tickets(i).issue\_desc,

v\_tickets(i).severity,

v\_tickets(i).assigned\_to,

v\_tickets(i).created\_on,

v\_tickets(i).status

);

DBMS\_OUTPUT.PUT\_LINE('Inserted ' || v\_tickets.COUNT || ' tickets successfully.');

END;

In our **IT Helpdesk Ticketing System**, we must **dynamically reference engineer names** based on their IDs for ticket assignments, dashboards, and quick lookups.

### Why We Need This:

* In real applications (like in HCL or Infosys ticket systems), you don’t always want to query the DB again and again to get a name for an ID — **performance will suffer**.
* Instead, we **load engineer names once into memory** and then access them in-memory like a dictionary or key-value pair → this is exactly what **associative arrays** help us do in PL/SQL.

### Where It Fits in the Architecture:

* **Memory-resident lookup**: like in-memory cache → no need for repeated DB round-trips.
* Ideal in **batch processing**, **email alerts**, or **ticket report generation** when you need to refer to multiple engineers many times.
* Data read-only after load — ensures consistency during execution.

## Step 4: Associative Array for Engineer Mapping

DECLARE  
 TYPE eng\_map IS TABLE OF VARCHAR2(100) INDEX BY PLS\_INTEGER;  
 eng\_names eng\_map;  
BEGIN  
 SELECT engineer\_name BULK COLLECT INTO eng\_names FROM engineers;  
 DBMS\_OUTPUT.PUT\_LINE('Engineer: ' || eng\_names(1));  
END;

|  |  |
| --- | --- |
| TYPE eng\_map IS TABLE OF VARCHAR2(100) INDEX BY PLS\_INTEGER; | Declares an **associative array** (key-value structure). Key is number, value is engineer name. |
| eng\_names eng\_map; | Instantiates the associative array. |
| SELECT ... BULK COLLECT INTO eng\_names | Loads multiple rows at once into memory (**bulk fetch**) — faster than row-by-row. |
| eng\_names(1) | Accesses the value at index 1 (first engineer name). |

Scenario

In the real-world Helpdesk application, multiple tickets might need to be inserted at once:

* After a **network outage**, many users might report issues like:
  + Login failure
  + Slow internet
  + Printer not connecting
* These are **common issues** often logged in **bulk** by an IT Admin.

Instead of inserting them one-by-one manually, we **store them in a PL/SQL collection** and insert them efficiently via a loop.

Use case : Let’s say you want to retrieve all the tickets assigned to a particular enginner and display one by one

## Step 6: Cursor with Record and PRAGMA Exception

DECLARE  
 CURSOR cur\_tickets IS

SELECT ticket\_id, issue\_desc FROM tickets WHERE assigned\_to = 101;  
 r\_tickets cur\_tickets%ROWTYPE;  
 e\_no\_ticket EXCEPTION;  
 PRAGMA EXCEPTION\_INIT(e\_no\_ticket, -20001);  
 v\_count NUMBER := 0;  
BEGIN  
 OPEN cur\_tickets;  
 LOOP  
 FETCH cur\_tickets INTO r\_tickets;  
 EXIT WHEN cur\_tickets%NOTFOUND;  
 v\_count := v\_count + 1;  
 DBMS\_OUTPUT.PUT\_LINE('Ticket ID: ' || r\_tickets.ticket\_id);  
 END LOOP;  
 CLOSE cur\_tickets;  
  
 IF v\_count = 0 THEN  
 RAISE\_APPLICATION\_ERROR(-20001, 'No Tickets Found');  
 END IF;  
EXCEPTION  
 WHEN e\_no\_ticket THEN  
 DBMS\_OUTPUT.PUT\_LINE('Handled: ' || SQLERRM);  
 WHEN OTHERS THEN  
 DBMS\_OUTPUT.PUT\_LINE('Unexpected Error: ' || SQLERRM);  
END;

## Step 7: Views and Inline Views for Dashboard

CREATE OR REPLACE VIEW ticket\_summary AS  
SELECT assigned\_to, COUNT(\*) AS total  
FROM tickets  
GROUP BY assigned\_to;  
  
-- Inline View  
SELECT \* FROM (  
 SELECT issue\_desc, created\_on FROM tickets ORDER BY created\_on DESC  
) WHERE ROWNUM <= 3;

Explanation:

## 🔹 What’s the Real-World Purpose?

Imagine you're managing a Helpdesk System. On your dashboard, you'd want:

* **How many tickets** are assigned to each engineer → summary view
* **Latest 3 raised tickets** → recent activity feed

This step allows us to build a **read-only reporting layer** for such insights using **Views and Inline Views**.