ORACLE SQL & BRIDGING FROM POSTGRESQL
Key Differences & Core Syntax, Data Types - DUAL Table - ROWNUM Pseudo-column *(Oracle Specific)*, NULL Handling - Conditional Expressions *(Practice in Oracle)*, Comments: *Solutions*

Transitional SQL

May 26, 2025

Contents

1 Dataset

The dataset definition and population statements are assumed to have been provided with the exercises document and executed in your Oracle SQL environment. These solutions refer to tables `EmployeeRoster`, `ProductCatalog`, and `ProductSales` as defined therein.

```
-- Dataset defined in the exercises document.
-- Ensure tables EmployeeRoster, ProductCatalog, and ProductSales are
   created and populated.
```

Listing 1: Placeholder for Dataset Reference (Dataset provided with exercises)

2 Solutions: Meanings, Values, Relations, and Advantages

2.1 Solution 1.1: Understanding Oracle Data Types & Bridging from PostgreSQL

1. **VARCHAR2 vs. NVARCHAR2:**

   - Core Difference: `VARCHAR2` stores variable-length character strings using the database's default character set. `NVARCHAR2` stores variable-length character strings using a national character set, typically a Unicode encoding (e.g., AL16UTF16 or UTF8, depending on database configuration).
   - `bio` vs. `firstName`: `bio` is `NVARCHAR2` because it's intended to store potentially multilingual biographical text (like the Japanese スティーブン, Hindi □□□□, Greek αλέξανδρος, Hebrew □□ examples in the dataset), which requires Unicode support. `firstName` might be `VARCHAR2` if the application primarily deals with names fitting the database character set (e.g., ISO-8859-1 for predominantly Western European names). However, for global applications, even names benefit from `NVARCHAR2`.
   - Relation to PostgreSQL: Oracle's `VARCHAR2` is the standard character string type, analogous to PostgreSQL's `VARCHAR`. `TEXT` in PostgreSQL is for arbitrarily long strings; Oracle would use `CLOB` for very large character data, though `VARCHAR2` can go up to 32767 bytes in PL/SQL or as a table column if MAX_STRING_SIZE=EXTENDED (otherwise 4000 bytes). A key Oracle-specific consideration is `VARCHAR2` vs. the (almost never used) `VARCHAR` type which has slightly different semantics for future compatibility (Oracle recommends always using `VARCHAR2`). Also, `NLS_LENGTH_SEMANTICS` (BYTE vs. CHAR) is an Oracle-specific setting impacting `VARCHAR2` definitions.

2. **NUMBER Type:**

   - `NUMBER(6)` for `employeeId` implies an integer with up to 6 digits. `NUMBER(10,2)` for `salary` implies a number with up to 10 total digits, 2 of which are after the decimal point.
   - PostgreSQL equivalents: `employeeId` (`NUMBER(6)`) → `INTEGER` or `NUMERIC(6,0)`. `salary` (`NUMBER(10,2)`) → `NUMERIC(10,2)`.
   - Advantage of Oracle's `NUMBER`: Versatility. It can represent integers, fixed-point, and floating-point numbers within a single data type, simplifying choices. PostgreSQL has more specialized numeric types.

3. **DATE Type:**

```sql
SELECT employeeId, hireDate
FROM EmployeeRoster
WHERE firstName = 'Steven' AND lastName = 'King';
-- Output might be like: 100, 17-JUN-03 (depending on NLS_DATE_FORMAT)

-- Or using TO_CHAR for explicit format:
```

```sql
SELECT employeeId, TO_CHAR(hireDate, 'YYYY-MM-DD HH24:MI:SS') AS
    formattedHireDate
FROM EmployeeRoster
WHERE firstName = 'Steven' AND lastName = 'King';
-- Output: 100, 2003-06-17 00:00:00
```

Listing 2: Querying DATE type

- Oracle's `DATE` is most analogous to PostgreSQL's `TIMESTAMP WITHOUT TIME ZONE` (or `TIMESTAMP(0) WITHOUT TIME ZONE` as Oracle's `DATE` has no fractional seconds).
- Impact: If migrating `DATE` data from PostgreSQL to Oracle, it will fit into Oracle `DATE` but time components will be 00:00:00. If migrating Oracle `DATE` to PostgreSQL `DATE`, the time component from Oracle `DATE` will be truncated, potentially losing information if the time part was significant. Queries comparing dates must be aware (e.g., use `TRUNC()` in Oracle if only date part comparison is needed).

4. **TIMESTAMP Variations:**

```sql
ALTER SESSION SET NLS_TIMESTAMP_FORMAT = 'YYYY-MM-DD HH24:MI:SS.FF3';
ALTER SESSION SET NLS_TIMESTAMP_TZ_FORMAT = 'YYYY-MM-DD HH24:MI:SS.FF
    TZR TZD';

SELECT productName, lastStockCheck, nextShipmentDue, localEntryTime
FROM ProductCatalog
WHERE productName = 'Oracle Database 19c';
```

Listing 3: Querying TIMESTAMP variations

- `lastStockCheck` (`TIMESTAMP`): Advantageous for recording an event time where time zone is implicitly known (e.g., always server time, or a fixed operational timezone) and high precision (fractional seconds) is useful. Simpler than TZ types if TZ info isn't globally relevant for this specific event.
- `nextShipmentDue` (`TIMESTAMP WITH TIME ZONE`): Advantageous for events where the exact point in time, including its original time zone, is crucial (e.g., a deadline specified by a supplier in their local time zone). It preserves the TZ information.
- `localEntryTime` (`TIMESTAMP WITH LOCAL TIME ZONE`): Advantageous when data is entered from various locations/time zones, but needs to be stored consistently (normalized to DB time zone) and then displayed relative to each user's local session time zone. This simplifies application logic for displaying times correctly to global users.

2.2 Solution 1.2: DUAL Table and NULL Handling (NVL, NVL2, COALESCE)

1. **DUAL Table:**

- `DUAL` is a special, public, single-row, single-column (`DUMMY VARCHAR2(1)`) table in Oracle.

- Common use cases:
  (a) Selecting pseudo-columns like `SYSDATE` or `USER`: `SELECT SYSDATE FROM DUAL;`
  (b) Testing functions or expressions: `SELECT (10*5)+3 FROM DUAL;`
  (c) Getting sequence next value: `SELECT my_sequence.NEXTVAL FROM DUAL;`
- In Oracle, `SELECT 1+1 FROM DUAL;` is required. Oracle's `SELECT` statement syntax mandates a `FROM` clause. `DUAL` serves as this placeholder when not querying actual tables.

2. **NVL Function:**

```sql
SELECT
    employeeId,
    firstName,
    salary,
    commissionRate,
    salary + (salary * NVL(commissionRate, 0)) AS guaranteedPay
FROM EmployeeRoster;
```

Listing 4: Using NVL Function

- `NVL(expr1, expr2)` is very similar to `COALESCE(expr1, expr2)`. Both return `expr2` if `expr1` is NULL, otherwise `expr1`. Key differences: `NVL` only takes two arguments. `COALESCE` can take multiple. `NVL` may have slightly different type conversion rules in some edge cases compared to `COALESCE`.

3. **NVL2 Function:**

```sql
SELECT
    employeeId,
    firstName,
    NVL2(commissionRate, 'Eligible for Commission Bonus', 'Salary Only') AS commissionStatus
FROM EmployeeRoster;
```

Listing 5: Using NVL2 Function

- Using `CASE` (known from PostgreSQL):

```sql
SELECT
    employeeId,
    firstName,
    CASE
        WHEN commissionRate IS NOT NULL THEN 'Eligible for Commission Bonus'
        ELSE 'Salary Only'
    END AS commissionStatus
FROM EmployeeRoster;
```

Listing 6: NVL2 logic with CASE

4. **COALESCE Function:**

- For the specific prompt: "If notes is NULL, show 'No additional notes'. If notes is NULL and supplierInfo also happens to be NULL, show 'Critical info missing'." This implies a priority. The `CASE` statement is clearer for this specific nested logic:

```sql
SELECT
    productId,
    productName,
    CASE
        WHEN notes IS NOT NULL THEN notes
        WHEN notes IS NULL AND supplierInfo IS NULL THEN 'Critical info missing'
        WHEN notes IS NULL THEN 'No additional notes'
        ELSE notes -- Should not be reached if notes is not null is first.
    END AS derivedNotes
FROM ProductCatalog;
```

Listing 7: Using CASE for specific COALESCE-like logic

- A simpler `COALESCE` demonstration for when any of the columns would suffice, or a final default: "display notes, if null display supplierInfo, if both null display 'Critical info missing'":

```sql
SELECT
    productId,
    productName,
    COALESCE(notes, supplierInfo, 'Critical info missing') AS effectiveInfo
FROM ProductCatalog;

-- Or, for the simpler part of the prompt:
SELECT
    productId,
    productName,
    COALESCE(notes, 'No additional notes provided') AS productNotesInfo
FROM ProductCatalog;
```

Listing 8: Simpler COALESCE demonstration

2.3 Solution 1.3: Conditional Logic (DECODE, CASE) & Comments

1. **DECODE vs. CASE:**

   - Syntactical Difference: `DECODE(expression, search1, result1, search2, result2, ..., default)` evaluates `expression` once and compares it to each `search` value. `CASE` has two forms: Simple `CASE expression WHEN value1 THEN result1 ... END` (similar to `DECODE`) and Searched `CASE WHEN condition1 THEN result1 ... END` (evaluates separate conditions).
   - Readability & Flexibility: `CASE` (especially searched `CASE`) is generally more readable and flexible for complex conditions (e.g., range checks, `OR` logic within a condition) than `DECODE`. `CASE` is also SQL standard.

2. **DECODE Function:**

```sql
SELECT
    firstName,
    jobTitle,
    DECODE(jobTitle,
           'President', 'Top Tier',
           'Administration VP', 'Mid Tier',
           'Finance Manager', 'Mid Tier',
           'Programmer', 'Staff',
           'Other') AS jobLevel
FROM EmployeeRoster;
```

Listing 9: Using DECODE Function

3. **CASE Expression:**

```sql
-- This query categorizes employees by job title into different levels
SELECT
    firstName,
    jobTitle,
    CASE
        WHEN jobTitle = 'President' THEN 'Top Tier'
        WHEN jobTitle IN ('Administration VP', 'Finance Manager') THEN
    'Mid Tier'
        WHEN jobTitle = 'Programmer' THEN 'Staff'
        ELSE 'Other'
    END AS jobLevel
FROM EmployeeRoster;
```

Listing 10: Using CASE Expression for jobLevel

```sql
SELECT
    productName,
    unitPrice,
    CASE
        WHEN unitPrice = 0 THEN 'Free'
        WHEN unitPrice > 0 AND unitPrice <= 100 THEN 'Affordable'
        WHEN unitPrice > 100 THEN 'Premium'
        ELSE 'Not Priced' -- Handles NULL or other unexpected
   unitPrice values
    END AS priceTag
FROM ProductCatalog;
```

Listing 11: Using CASE Expression for priceTag

4. **Comments:**

- Single-line comment example is shown above the first CASE expression query.
- Multi-line comment example (to be placed at top of a script file):

```sql
/*
  Oracle SQL Transitional Course Exercises
  Section: Key Differences, Data Types, DUAL, NULL Handling,
   Conditional Expressions, ROWNUM, Comments
  Focus: Practicing core Oracle syntax and specific features.
*/
```

Listing 12: Example of Multi-line Comment

2.4 Solution 1.4: ROWNUM Pseudo-column

1. **ROWNUM Basics:**
   - `ROWNUM` is an Oracle pseudo-column that assigns a sequential number (1, 2, 3, ...) to each row as it is selected from a table (or view/subquery result) and passes the `WHERE` clause conditions of that query block. Crucially, this assignment happens *before* any `ORDER BY` clause in the same query block is applied (unless the `ORDER BY` is in a subquery and `ROWNUM` is in the outer query).
   - Difference from PostgreSQL `LIMIT`: PostgreSQL's `LIMIT` clause is applied *after* the `ORDER BY` clause, making it straightforward to get the top N rows from an ordered set. With Oracle's `ROWNUM`, to get a true top-N based on an ordering, you typically need to `ORDER BY` in a subquery and then apply `WHERE ROWNUM <= N` in the outer query.

2. **Top-N Query:**

```sql
SELECT firstName, lastName, salary
FROM (
    SELECT firstName, lastName, salary
    FROM EmployeeRoster
    ORDER BY salary DESC
)
WHERE ROWNUM <= 3;
```

Listing 13: Correct Top-N Query with ROWNUM

(*This ensures that ordering by salary happens first, then the first 3 rows of that ordered set are taken.*)

3. **Pagination Emulation (Rows 4-5):** To select rows 4-5 using `ROWNUM`, you need a two-level subquery approach:
   (a) Inner subquery: Select all necessary columns and `ORDER BY` salary.
   (b) Middle subquery: Select from the inner subquery and assign `ROWNUM` to these ordered rows (aliased, e.g., as `rn`).
   (c) Outer subquery: Filter on the aliased row number `rn`.

```sql
SELECT firstName, lastName, salary
FROM (
    SELECT query_results.*, ROWNUM AS rn
    FROM (
        SELECT firstName, lastName, salary
        FROM EmployeeRoster
        ORDER BY salary DESC
    ) query_results
)
WHERE rn BETWEEN 4 AND 5;
-- Alternatively, and often more common for ROWNUM pagination:
-- WHERE rn >= 4 AND ROWNUM <= 2 (if you were selecting page 2 of size 2 after skipping 2)
-- This specific example (rn BETWEEN 4 AND 5) is for clarity on getting specific rows.
```

Listing 14: ROWNUM for Pagination

(*Explanation: The innermost query orders all employees. The middle query assigns* `ROWNUM` (*as* `rn`) *to this ordered set. The outermost query can then filter on* `rn` *using conditions like* `rn > X` *or* `rn BETWEEN X AND Y` *because* `rn` *is now a fixed attribute of each row from the middle subquery's result set. You cannot directly use* `WHERE ROWNUM > N` *in a single query block because ROWNUM must be 1 for a row to be considered initially.*)

3 Solutions: Disadvantages and Pitfalls

3.1 Solution 2.1: Data Type Pitfalls and Misunderstandings

1. **VARCHAR2 Size & Semantics:**
   - `VARCHAR2(10 BYTE)`:
     - 'Christophe' (10 chars, 10 bytes in ASCII): Inserts successfully.
     - 'René' (4 chars): If 'é' is 1 byte (e.g. ISO-8859-1 database character set), total 4 bytes, inserts. If 'é' is 2 bytes (e.g. AL32UTF8 database character set), total 5 bytes, inserts.
     - Pitfall with `BYTE` semantics: If the database character set is multi-byte (like AL32UTF8) and `NLS_LENGTH_SEMANTICS` for column definition is effectively `BYTE` (default for `VARCHAR2` definition unless instance/session parameter is `CHAR`), `VARCHAR2(10 BYTE)` means 10 bytes. A 5-character string where each character is 3 bytes would require 15 bytes and fail the insert with an "ORA-12899: value too large for column" error, even though it's only 5 characters. Using `VARCHAR2(X CHAR)` is often safer for multi-byte character sets to define storage by number of characters rather than bytes.
2. **NUMBER Precision/Scale:**
   - `NUMBER` (no precision/scale) for `salary`: If you inserted `12345.678912345`, it would be stored exactly as `12345.678912345` (up to Oracle's internal limit of 38 digits of precision). Pitfall: While flexible, for financial data, this allows varying decimal places, which can be problematic for calculations, consistency, and reporting. It also doesn't enforce limits on the total number of digits, potentially leading to unexpectedly large numbers.
   - `commissionRate NUMBER(4,2)`: This means 4 total significant digits, with 2 digits after the decimal point. This implies a range of -99.99 to 99.99.
     - Insert `0.125`: Will be rounded to `0.13`. (Oracle rounds, rather than truncates, when scale is exceeded).
     - Insert `10.50`: Inserts successfully as `10.50`.
     - Insert `123.45`: Fails. `ORA-01438: value larger than specified precision allowed for this column`. The value `123.45` has 5 significant digits (3 before, 2 after decimal), but `NUMBER(4,2)` allows for $4 - 2 = 2$ digits before the decimal if 2 are used after, or at most 4 total significant digits.
3. **Oracle DATE Time Component:**
   - `TO_DATE('2023-11-10', 'YYYY-MM-DD')` inserts `2023-11-10 00:00:00` into `hireDate`.
   - The query `WHERE hireDate = TO_DATE('2023-11-10 10:00:00', 'YYYY-MM-DD HH24:MI:SS')` will **not** find the record.
   - Why: Because `2023-11-10 00:00:00` is not equal to `2023-11-10 10:00:00`.

- Pitfall: Users expecting date-only comparisons (like in PostgreSQL `DATE`) might write equality checks that fail due to the time component. They should use `TRUNC(hireDate) = TO_DATE('2023-11-10', 'YYYY-MM-DD')` or a range condition like `hireDate >= TO_DATE('2023-11-10', 'YYYY-MM-DD') AND hireDate < TO_DATE('2023-11-11', 'YYYY-MM-DD')` for robust date-only comparisons.

4. **TIMESTAMP WITH LOCAL TIME ZONE (TSLTZ):**

   - Session A (NY, e.g., UTC-5 or UTC-4 depending on DST) inserts `TIMESTAMP '2023-11-10 10:00:00 America/New_York'`. This specific instant in time is, for example, `2023-11-10 15:00:00 UTC` (if NY is UTC-5). The value is normalized to the database's time zone (`DBTIMEZONE`) for storage, but the key is that it represents that absolute UTC instant.
   - Session B (London, e.g., UTC+0 or UTC+1 depending on DST) queries. Oracle converts the stored UTC-equivalent time to Session B's session time zone. If London is UTC+0 at that time, Session B will see `2023-11-10 15:00:00 Europe/London`.
   - Pitfall with `DBTIMEZONE` and `SYSTIMESTAMP`: `SYSTIMESTAMP` returns the date, time, and time zone of the database server's operating system. If this TSLTZ value is inserted into a TSLTZ column, it's then normalized to `DBTIMEZONE` for storage (conceptually, it's stored based on UTC and `DBTIMEZONE` might influence this representation or just be metadata). When retrieved, it's converted to the session's time zone. If `DBTIMEZONE` is different from the server OS time zone, the stored representation might seem unusual if inspected directly without considering the session time zone conversion. The critical aspect for TSLTZ is that it correctly converts to the *session's* time zone upon retrieval, regardless of `DBTIMEZONE` or server OS timezone, because it represents an absolute point in time. However, confusion can arise if developers expect `SYSTIMESTAMP` inserted values to look the same as the OS time when queried in a session with a different time zone or if they try to reason about the raw stored value without understanding the normalization and conversion processes.

3.2 Solution 2.2: NULL Handling Function Caveats

1. **NVL Type Conversion:**

   - `NVL(salary, 'Not Available')`: If `salary` (a `NUMBER`) is NOT NULL, it returns the `salary`. If `salary` IS NULL, Oracle attempts to convert the second argument, 'Not Available' (a `VARCHAR2`), to a `NUMBER` to match the data type of the first argument. This conversion will fail and raise an `ORA-01722: invalid number` error.
   - Pitfall: Implicit type conversion by `NVL` (where it tries to make the second argument's type match the first if the first is not null, or vice-versa if types are different across arguments) can lead to runtime errors if the types are incompatible and the conversion fails. The rule is that the datatype of the return value is the datatype of `expr1` if it is character, or if `expr1` is numeric/date and `expr2` can be converted.

- Correction for string output: `NVL(TO_CHAR(salary), 'Not Available')`. This explicitly converts `salary` to a string, so both potential return values are strings (or compatible).

2. **NVL2 Type Mismatch:**

   - `NVL2(hireDate, SYSDATE + 7, 'Not Hired Yet')`:
     - If `hireDate` (a `DATE`) is NOT NULL: result is `SYSDATE + 7` (a `DATE`).
     - If `hireDate` IS NULL: result is 'Not Hired Yet' (a `VARCHAR2`).
   - Potential Issue: The `NVL2` function must return a single data type. Oracle will try to implicitly convert one of the result expressions (`expr2` or `expr3`) to match the other. The data type of the return value is the data type of `expr2` if it's a character string. If `expr2` is numeric or date, then Oracle determines the data type with the higher precedence between `expr2` and `expr3` and converts the other to it. In this case, if `hireDate` is NOT NULL, 'Not Hired Yet' (`expr3`) would need to be converted to `DATE` to match `SYSDATE + 7` (`expr2`). This would cause an `ORA-01858: a non-numeric character was found where a numeric was expected` or similar date conversion error.
   - To avoid this, ensure `expr2` and `expr3` are of compatible types, often by explicit conversion to a common target type: `NVL2(hireDate, TO_CHAR(SYSDATE + 7, 'DD-MON-YYYY'), 'Not Hired Yet')`.

3. **COALESCE Argument Evaluation:**

   - `COALESCE(numericColumn, dateColumn, 'textFallback')`:
     - If `numericColumn` is NOT NULL: `COALESCE` returns `numericColumn` (a `NUMBER`). The function's determined return type becomes `NUMBER`.
     - If `numericColumn` IS NULL and `dateColumn` IS NOT NULL: `COALESCE` evaluates `dateColumn`. It then tries to convert `dateColumn` to the determined return type of the function (which might be influenced by the type of `numericColumn` even if it was NULL, or by the type of the first non-NULL argument encountered). If it attempts to convert `dateColumn` (a `DATE`) to a `NUMBER`, it will result in an `ORA-01722: invalid number` error.
   - The data type of the `COALESCE` result is determined by the data type of the first expression that is not null, if all expressions are of the same data type group (e.g., all numeric, all character). If types are mixed, Oracle applies precedence rules. It's best to ensure all arguments are of compatible types or use explicit conversions to a desired common type (usually `VARCHAR2` if mixing heavily, e.g., `COALESCE(TO_CHAR(numericColumn), TO_CHAR(dateColumn), 'textFallback')`).

3.3 Solution 2.3: DECODE and ROWNUM Logic Traps

1. **DECODE's NULL Handling:**

- If both `colA` and `colB` are `NULL`, `DECODE(colA, colB, 'Match', 'No Match')` returns **'Match'**. `DECODE` treats `NULL` as equal to `NULL` for its comparisons.
- `CASE WHEN colA = colB THEN 'Match' ELSE 'No Match' END` returns **'No Match'** if both `colA` and `colB` are `NULL`, because standard SQL comparison `NULL = NULL` evaluates to UNKNOWN, which the `WHEN` condition treats as false.
- Pitfall: `DECODE`'s `NULL` handling can be a pitfall if a developer expects standard SQL `NULL` comparison semantics (where `NULL` is not equal to anything, including itself). This can lead to logical errors if not understood, especially when migrating logic from systems or habits that use standard `NULL` checks (`IS NULL` or `IS NOT NULL`).

2. **ROWNUM for Pagination - Incorrect Attempt:**
   - The query `SELECT productName FROM ProductCatalog WHERE ROWNUM BETWEEN 3 AND 4 ORDER BY productId;` will return no rows.
   - Reason: `ROWNUM` is assigned to a row only if it passes the `WHERE` clause. The condition `ROWNUM BETWEEN 3 AND 4` means `ROWNUM >= 3 AND ROWNUM <= 4`.
     - Oracle fetches the first potential row. `ROWNUM` would become 1. Is `1 >= 3`? False. Row discarded.
     - Since no row ever satisfies the condition to become `ROWNUM = 1` (and thus be kept), no row can ever be assigned `ROWNUM = 2`, and consequently, no row can become `ROWNUM = 3`. The condition `ROWNUM >= 3` (or any `ROWNUM > 1`) effectively prevents any row from being selected if applied directly in the same query block where `ROWNUM` is generated.

3. **ROWNUM with ORDER BY - Misconception:**
   - Query: `SELECT productName, ROWNUM FROM ProductCatalog WHERE ROWNUM <= 2 ORDER BY productName DESC;`
   - Output: This query first selects the *first two rows* it physically encounters from `ProductCatalog` that satisfy any other `WHERE` conditions (here, only `ROWNUM <= 2`). Let's assume these are 'Oracle Database 19c' (assigned `ROWNUM 1`) and 'PostgreSQL 15' (assigned `ROWNUM 2`) based on their internal storage or access path. Then, *only these two selected rows* are ordered by `productName DESC`. So, the output might be:

     ```
     PRODUCTNAME           ROWNUM
     -------------------- ------
     PostgreSQL 15              2
     Oracle Database 19c        1
     ```

     (The actual two rows selected before ordering can vary if there's no explicit `ORDER BY` in a subquery controlling what `ROWNUM <= 2` picks).

- Guarantee: It is **NOT** guaranteed to be the two products whose names are last alphabetically from the entire table. `ROWNUM` is applied before the `ORDER BY` clause in the same query block. To get the two products whose names are last alphabetically, you'd need:

```sql
SELECT productName, ROWNUM -- This ROWNUM is from the outer query
FROM (
    SELECT productName
    FROM ProductCatalog
    ORDER BY productName DESC
)
WHERE ROWNUM <= 2;
```

Listing 15: Correct Top-N with ROWNUM and ORDER BY

4 Solutions: Contrasting with Inefficient Common Solutions

4.1 Solution 3.1: Suboptimal Logic vs. Oracle SQL Efficiency

1. **Client-Side NULL Handling:**

   - Efficient Oracle SQL:

```sql
SELECT
    firstName,
    commissionRate,
    -- Assuming commissionRate is a percentage, and we want to display the calculated commission amount
    NVL(TO_CHAR(salary * commissionRate, 'L999G999D00'), TO_CHAR(0, 'L999G999D00')) AS commissionAmountDisplay
    -- Or if just displaying the rate itself or a placeholder like 'N/A':
    -- NVL(TO_CHAR(commissionRate), 'N/A') AS commissionRateDisplay
FROM EmployeeRoster;
```

Listing 16: Efficient NULL Handling in SQL

   - Loss of advantage with client-side:
     - **Increased Network Traffic:** All rows, including those with `NULL` commission rates that will be processed/transformed by the client, are sent over the network. Transferring raw data is less efficient than transferring already formatted/processed data.
     - **Slower Client Processing:** The client application expends CPU cycles iterating and applying conditional logic for each row. For large datasets, this is significantly less efficient than letting the database handle it.
     - **Data Transformation Closer to Source:** Databases are highly optimized for set-based operations and data transformations. Performing this logic in SQL leverages these optimizations.
     - **Code Maintainability & Consistency:** Business logic for display or derived values can be centralized in SQL views or queries. If multiple client applications or reports need the same logic, defining it once in the database ensures consistency and easier maintenance compared to replicating the logic in each client.

2. **Client-Side Conditional Logic:**

   - Efficient Oracle SQL:

```sql
SELECT
    productName,
    productCategory,
    CASE productCategory
        WHEN 'Software' THEN 'Digital Good'
        WHEN 'Hardware' THEN 'Physical Good'
        ELSE 'Misc Good'
    END AS itemTypeDisplay
```

```
FROM ProductCatalog;
```

Listing 17: Efficient Conditional Logic in SQL

- Why SQL is better for reporting:
  - **Reduced Complexity in Reporting Tool/Client:** The reporting tool or client application receives pre-categorized data, simplifying report design or client-side display logic.
  - **Consistency:** The categorization logic is defined once in the SQL query or a view. All reports or clients using this query/view will use the exact same definitions, ensuring consistency.
  - **Performance:** The database can often perform such categorizations much faster than a client tool processing raw data row-by-row, especially if dealing with large volumes of data. This is because the database works on sets and can use indexes (e.g., on `productCategory`).
  - **Data Volume Reduction:** Only necessary, transformed data is sent to the client/reporting tool, not all raw data that would then require client-side processing. This reduces network load and client memory footprint.

4.2 Solution 3.2: Inefficient ROWNUM Usage and DUAL Misconceptions

1. **Inefficient DUAL Usage:**
   - Efficient Way:

```
SELECT SYSTIMESTAMP AS actionTimestamp, USER AS performingUser FROM
    DUAL;
```

Listing 18: Efficient DUAL Usage

   - Oracle value lost by the inefficient approach:
     - **Performance/Efficiency:** The inefficient approach makes two separate round trips to the database (parse, execute, fetch for each query). The efficient way makes only one. This significantly reduces network latency and database server overhead, especially if this logging occurs frequently. Each SQL execution, no matter how small, incurs a cost.
     - **Atomicity (Conceptual):** While `SYSTIMESTAMP` and `USER` are unlikely to change meaningfully between two extremely fast, consecutive queries, for other values or more complex scenarios, getting related pieces of information in a single query ensures they are all consistent with the same point in time or transactional context. Multiple queries open a (however small) window for state change.

2. **Incorrect Top-N with ROWNUM:**
   - Explain why it's not guaranteed: The query is:

```sql
SELECT productName, unitPrice
FROM ProductCatalog
WHERE unitPrice > 0 AND ROWNUM <= 3
ORDER BY unitPrice ASC;
```

The `WHERE unitPrice > 0 AND ROWNUM <= 3` clause is applied as Oracle fetches rows, *before* the `ORDER BY unitPrice ASC` is processed. `ROWNUM <= 3` will pick the first three rows it happens to encounter that also satisfy `unitPrice > 0`. The order in which Oracle "encounters" rows without an explicit `ORDER BY` in the same query block (or a subquery) is not guaranteed (it could be physical storage order, index order if an index is used for another predicate, etc.). Therefore, the subsequent `ORDER BY unitPrice ASC` will only sort these arbitrarily chosen three (non-free) rows by their price. It does not guarantee that these three are the *overall* three cheapest non-free products from the entire table.

- Efficient, correct Oracle-idiomatic way:

```sql
SELECT productName, unitPrice
FROM (
    SELECT productName, unitPrice
    FROM ProductCatalog
    WHERE unitPrice > 0
    ORDER BY unitPrice ASC -- Order first to identify the cheapest
)
WHERE ROWNUM <= 3; -- Then take the top 3 from the ordered set
```

Listing 19: Correct Top-N for Cheapest Non-Free Products

(*This first filters for non-free products and orders them by price (cheapest first) in a subquery. The outer query then takes the top 3 rows from this correctly ordered and filtered result set.*)

5 Solution: Hardcore Combined Problem

5.1 Solution 4.1: Multi-Concept Oracle Challenge for "Employee Performance Review Prep"

```sql
/*
  Employee Performance Review Prep Report
  Purpose: Identifies the top 2 longest-serving Programmers in the IT department
  for upcoming performance reviews, providing key details and a review focus.
*/
SELECT
    employeeId,
    employeeName,
    jobTitle,
    department,
    hireDateDisplay,
    yearsOfService,
    bioExtract,
    reviewFocus
FROM (
    SELECT
        employeeId,
        lastName || ', ' || firstName AS employeeName,
        jobTitle,
        departmentName AS department,
        TO_CHAR(hireDate, 'FMMonth DD, YYYY') AS hireDateDisplay, -- FM removes leading/trailing blanks from month
        ROUND(MONTHS_BETWEEN(SYSDATE, hireDate) / 12, 1) AS yearsOfService,
        NVL2(bio, SUBSTR(bio, 1, 30) || CASE WHEN LENGTH(bio) > 30 THEN '...' ELSE '' END, 'No Bio on File') AS bioExtract,
        CASE
            WHEN commissionRate IS NOT NULL THEN 'Sales & Technical Skills Review'
            ELSE DECODE(managerId,
                        102, 'Project Leadership Potential',
                        'Core Technical Deep Dive') -- Default for DECODE if managerId is not 102 or is NULL
        END AS reviewFocus,
        hireDate -- Keep for ordering in the subquery, not needed in final SELECT
    FROM EmployeeRoster
    WHERE departmentName = 'IT' AND jobTitle = 'Programmer'
    ORDER BY hireDate ASC -- Earliest hire date first (longest serving)
) sorted_programmers
WHERE ROWNUM <= 2; -- Selects the top 2 rows from the ordered subquery result
```

Listing 20: Employee Performance Review Prep Report Solution

Explanation of Solution Elements:

- **Data Types Used Implicitly/Explicitly:** The query interacts with NUMBER (`employeeId`, `commissionRate`, `managerId`), VARCHAR2 (`lastName`, `firstName`, `jobTitle`, `departmentName`), NVARCHAR2 (`bio`), and DATE (`hireDate`). Output columns are derived with appropriate types.

- **DUAL Table (Implicit):** `SYSDATE` is used for calculating `yearsOfService`. While not explicitly `SELECT SYSDATE FROM DUAL`, its usage in functions like `MONTHS_BETWEEN` leverages Oracle's ability to provide such system values.
- **NULL Handling:**
  - `NVL2(bio, ... , 'No Bio on File')` is used for `bioExtract`. If `bio` is NOT NULL, it takes the first 30 characters (adding '...' if longer). If `bio` IS NULL, it displays 'No Bio on File'.
  - The `CASE` statement checks `commissionRate IS NOT NULL`.
  - `DECODE` handles `managerId`; its default clause also covers cases where `managerId` might be NULL for a programmer (though not in current data for programmers, it's robust).
- **Conditional Expressions:**
  - A main `CASE` expression determines `reviewFocus` based on `commissionRate`.
  - A nested `DECODE` function is used within the `ELSE` part of the `CASE` to provide further conditions based on `managerId`.
  - A small `CASE` expression is used within the `NVL2` for `bioExtract` to conditionally append '...' only if the bio was actually longer than 30 characters.
- **ROWNUM Pseudo-column:** Applied in the outermost query: `WHERE ROWNUM <= 2`. This correctly fetches the top 2 employees *after* they have been filtered (Programmers in IT) and ordered by `hireDate ASC` (longest serving) in the inner subquery (`sorted_programmers`). This is the correct Oracle idiom for Top-N queries.
- **Comments:** A multi-line comment at the beginning explains the report's overall purpose. A single-line comment clarifies the `ROWNUM` logic.
- **Oracle Specific Functions/Syntax in Play:**
  - Data types: `VARCHAR2`, `NVARCHAR2`, `NUMBER`, `DATE`.
  - String concatenation: `||`.
  - String functions: `SUBSTR`, `LENGTH`.
  - Date functions: `TO_CHAR` (with `FMMonth` format model specifier to remove padding), `MONTHS_BETWEEN`, `SYSDATE`.
  - Numeric function: `ROUND`.
  - NULL handling: `NVL2`.
  - Conditional logic: `DECODE`, `CASE`.
  - Row limiting: `ROWNUM` with a subquery.
- **Bridging from PostgreSQL Considerations:**
  - **Date Calculation:** PostgreSQL would use `AGE(SYSDATE, hireDate)` and then extract parts or format it. Oracle's `MONTHS_BETWEEN(d1, d2) / 12` is a common way to get years.

- **NULL Handling:** PostgreSQL primarily uses `COALESCE`. Oracle's `NVL2` provides a compact IF-NOT-NULL-THEN-ELSE structure.
- **Conditional Logic:** `CASE` is very similar. `DECODE` is Oracle-specific and less flexible than `CASE` for complex conditions but can be concise for simple equality checks.
- **Top-N Queries:** PostgreSQL uses `ORDER BY ... LIMIT N`. The Oracle pattern of `ORDER BY` in a subquery, then `ROWNUM <= N` in the outer query, is a fundamental difference to internalize.
- **String Functions:** `SUBSTR` (or `SUBSTRING`) and `LENGTH` are quite standard. Concatenation with `||` is common to both.
- **Formatting:** `TO_CHAR` for dates is crucial in Oracle, and its format models are extensive. PostgreSQL also has `TO_CHAR`.