Oracle AI Database 26ai: Complete Practice Guide

Introduction

Oracle AI Database 26ai represents a transformational upgrade from Oracle 23ai, positioning AI as a native, first-class citizen within the database rather than an add-on capability[1]. This comprehensive guide provides hands-on practice exercises with complete working code for all major new features introduced in Oracle 26ai[2].

Learning Objectives:

- Master AI Vector Search functionality
- Implement advanced SQL syntax enhancements
- Work with agentic AI workflows
- Practice new developer-centric features
- Understand transaction and performance improvements

Part 1: AI Vector Search (Core Feature)

1.1 Introduction to Vector Data Types

Vector data types allow you to store AI embeddings directly in the database. These embeddings represent text, images, audio, or other data as numerical arrays for semantic similarity searches[3].

1.2 Creating Vector Columns

Create a sample vector table:

```
-- Create table with vector columns
CREATE TABLE product_embeddings (
product_id NUMBER PRIMARY KEY,
product_name VARCHAR2(255) NOT NULL,
description VARCHAR2(1000),
embedding VECTOR,
metadata JSON,
created_date TIMESTAMP DEFAULT SYSDATE
);

-- Add comments for documentation
COMMENT ON TABLE product_embeddings IS 'Stores product data with AI embeddings for similarity search';
COMMENT ON COLUMN product_embeddings.embedding IS 'Vector embedding (1536 dimensions for OpenAI models)';
```

1.3 Inserting Vector Data

Insert sample product embeddings:

```
-- Insert products with vector embeddings
-- Note: In production, embeddings would come from an LLM API
INSERT INTO product_embeddings (product_id, product_name, description, embedding,
metadata)
VALUES (1, 'Laptop Pro 15', 'High-performance laptop with 16GB RAM and SSD storage',
TO_VECTOR('[0.123, 0.456, 0.789, 0.234, 0.567, 0.890, 0.111, 0.222, 0.333, 0.444]'),
JSON('{"category": "electronics", "price": 1299, "brand": "TechCorp"}'));

INSERT INTO product_embeddings (product_id, product_name, description, embedding,
metadata)
VALUES (2, 'Wireless Keyboard', 'Ergonomic wireless keyboard with mechanical switches',
TO_VECTOR('[0.121, 0.454, 0.785, 0.232, 0.565, 0.888, 0.112, 0.224, 0.335, 0.443]'),
JSON('{"category": "accessories", "price": 89, "brand": "InputMaster"}'));

INSERT INTO product_embeddings (product_id, product_name, description, embedding,
metadata)
VALUES (3, 'USB-C Hub', 'Multi-port USB-C hub with HDMI and card reader',
TO_VECTOR('[0.125, 0.458, 0.792, 0.236, 0.569, 0.892, 0.115, 0.226, 0.337, 0.446]'),
JSON('{"category": "accessories", "price": 49, "brand": "ConnectPlus"}'));

COMMIT;
```

1.4 Creating Vector Indexes

Create specialized vector indexes for faster similarity searches:

```
-- Create HNSW (Hierarchical Navigable Small World) Index - best for most use cases
CREATE VECTOR INDEX product_embedding_hnsw
ON product_embeddings(embedding)
ORGANIZATION HNSW
WITH TARGET ACCURACY 95;

-- Alternative: IVF-Flat Index - good for large datasets
-- CREATE VECTOR INDEX product_embedding_ivf
-- ON product_embeddings(embedding)
-- ORGANIZATION IVF
-- WITH PARAMETERS (neighbor_count=8, groups=100);

-- Alternative: Hybrid Index - combines benefits
-- CREATE VECTOR INDEX product_embedding_hybrid
-- ON product_embeddings(embedding)
-- ORGANIZATION HYBRID;
```

1.5 Vector Similarity Search

Query products using vector similarity:

```
-- Find products similar to a search query embedding
-- Assume we have a query embedding vector
DECLARE
query_vector VECTOR := TO_VECTOR('[0.122, 0.455, 0.788, 0.233, 0.566, 0.889, 0.113, 0.223, 0.334, 0.445]');
BEGIN
-- Euclidean distance (L2 norm)
SELECT product_id, product_name, description,
ROUND(VECTOR_DISTANCE(embedding, query_vector, EUCLIDEAN), 4) as similarity_score
FROM product_embeddings
ORDER BY VECTOR_DISTANCE(embedding, query_vector, EUCLIDEAN)
FETCH FIRST 5 ROWS ONLY;
END;
/

-- Cosine similarity (most common for text embeddings)
SELECT product_id, product_name,
ROUND(VECTOR_DISTANCE(embedding, TO_VECTOR('[0.122, 0.455, 0.788, 0.233, 0.566]'), COSINE), 4) as cosine_distance
FROM product_embeddings
ORDER BY VECTOR_DISTANCE(embedding, TO_VECTOR('[0.122, 0.455, 0.788, 0.233, 0.566]'), COSINE)
WHERE VECTOR_DISTANCE(embedding, TO_VECTOR('[0.122, 0.455, 0.788, 0.233, 0.566]'), COSINE) < 0.5;

-- Inner product distance
SELECT product_id, product_name,
ROUND(VECTOR_DISTANCE(embedding, TO_VECTOR('[0.122, 0.455, 0.788, 0.233, 0.566]'), DOT_PRODUCT), 4) as dot_product
FROM product_embeddings
ORDER BY VECTOR_DISTANCE(embedding, TO_VECTOR('[0.122, 0.455, 0.788, 0.233, 0.566]'), DOT_PRODUCT) DESC
FETCH FIRST 3 ROWS ONLY;
```

Part 2: Advanced SQL Syntax Enhancements

2.1 SELECT Expression-Only Queries (No FROM Clause)

Execute queries without a FROM clause:

```
-- Simple mathematical expressions
SELECT 10 + 20 AS sum_result, 100 * 5 AS multiply_result, SYSDATE AS current_date;

-- String concatenation
SELECT 'Oracle' || ' ' || 'Database' || ' ' || '26ai' AS database_version;

-- Function calls without table reference
SELECT ROUND(3.14159, 2) AS pi_rounded,
```

```
UPPER('hello world') AS uppercase_text,
LENGTH('Oracle26ai') AS string_length;

-- UUID generation
SELECT SYS_GUID() AS unique_id,
DBMS_CRYPTO.HASH(TO_CLOB('test'), 2) AS hash_value;

-- Date arithmetic
SELECT TRUNC(SYSDATE) AS today,
SYSDATE + 7 AS next_week,
LAST_DAY(SYSDATE) AS end_of_month;
```

2.2 BOOLEAN Data Type

Work with native boolean columns:

```
-- Create table with BOOLEAN columns
CREATE TABLE feature_flags (
flag_id NUMBER PRIMARY KEY,
feature_name VARCHAR2(100),
is_enabled BOOLEAN,
is_beta BOOL,
requires_auth BOOLEAN DEFAULT TRUE
);

-- Insert BOOLEAN values (multiple syntaxes supported)
INSERT INTO feature_flags (flag_id, feature_name, is_enabled, is_beta, requires_auth)
VALUES (1, 'Vector Search', TRUE, FALSE, TRUE);

INSERT INTO feature_flags (flag_id, feature_name, is_enabled, is_beta, requires_auth)
VALUES (2, 'Agentic AI', true, false, true);

INSERT INTO feature_flags (flag_id, feature_name, is_enabled, is_beta, requires_auth)
VALUES (3, 'Graph Database', FALSE, 1, 0);

INSERT INTO feature_flags (flag_id, feature_name, is_enabled, is_beta, requires_auth)
VALUES (4, 'AI Text Search', 'yes', 'no', 'on');

INSERT INTO feature_flags (flag_id, feature_name, is_enabled, is_beta, requires_auth)
VALUES (5, 'Advanced Analytics', 'off', 'false', 1);

COMMIT;

-- Query BOOLEAN columns
SELECT feature_name, is_enabled, is_beta, requires_auth
FROM feature_flags
WHERE is_enabled = TRUE AND requires_auth = TRUE;

-- Logical operations
SELECT feature_name
FROM feature_flags
WHERE is_enabled AND NOT is_beta;
```

```
SELECT feature_name
FROM feature_flags
WHERE (is_enabled = TRUE OR is_beta = TRUE) AND requires_auth = FALSE;

-- COUNT BOOLEAN values
SELECT
COUNT(CASE WHEN is_enabled THEN 1 END) AS enabled_features,
COUNT(CASE WHEN NOT is_enabled THEN 1 END) AS disabled_features
FROM feature_flags;
```

2.3 IF EXISTS and IF NOT EXISTS

Safer DDL operations:

```
-- Create table only if it doesn't exist
CREATE TABLE IF NOT EXISTS audit_log (
log_id NUMBER PRIMARY KEY,
table_name VARCHAR2(100),
operation VARCHAR2(50),
operation_time TIMESTAMP,
user_name VARCHAR2(100)
);

-- Drop table only if it exists (no error if table doesn't exist)
DROP TABLE IF EXISTS temp_staging;

-- Drop column only if it exists
ALTER TABLE product_embeddings
DROP COLUMN IF EXISTS legacy_field;

-- Add column only if it doesn't exist
ALTER TABLE product_embeddings
ADD IF NOT EXISTS last_updated TIMESTAMP DEFAULT SYSDATE;

-- Drop index if it exists
DROP INDEX IF EXISTS idx_old_products;

-- Create index only if it doesn't exist
CREATE INDEX IF NOT EXISTS idx_product_name
ON product_embeddings(product_name);
```

2.4 VALUES Clause for Materialized Rows

Create virtual rows without a table:

```
-- INSERT using VALUES
CREATE TABLE sales_data (
sale_id NUMBER PRIMARY KEY,
product_id NUMBER,
quantity NUMBER,
sale_date DATE
);
```

```
INSERT INTO sales_data (sale_id, product_id, quantity, sale_date)
VALUES (1, 101, 5, TRUNC(SYSDATE)),
(2, 102, 3, TRUNC(SYSDATE) - 1),
(3, 103, 8, TRUNC(SYSDATE) - 2);

-- SELECT using VALUES clause
SELECT * FROM (
VALUES (1, 'Q1 2025', 100000),
(2, 'Q2 2025', 150000),
(3, 'Q3 2025', 200000),
(4, 'Q4 2025', 180000)
) AS quarterly_targets(quarter_num, period, target_amount)
WHERE quarter_num > 2;

-- MERGE with VALUES clause
MERGE INTO sales_data t
USING (
VALUES (4, 104, 2, TRUNC(SYSDATE)),
(5, 105, 7, TRUNC(SYSDATE))
) s(sale_id, product_id, quantity, sale_date)
ON (t.sale_id = s.sale_id)
WHEN MATCHED THEN
UPDATE SET t.quantity = s.quantity
WHEN NOT MATCHED THEN
INSERT (t.sale_id, t.product_id, t.quantity, t.sale_date)
VALUES (s.sale_id, s.product_id, s.quantity, s.sale_date);

COMMIT;
```

2.5 RETURNING INTO: Old and New Values

Capture both before and after values:

```
-- Create table for demonstration
CREATE TABLE employee_audit (
emp_id NUMBER PRIMARY KEY,
emp_name VARCHAR2(100),
salary NUMBER(10,2),
department VARCHAR2(50),
old_salary NUMBER(10,2),
new_salary NUMBER(10,2),
old_dept VARCHAR2(50),
new_dept VARCHAR2(50)
);

-- INSERT with RETURNING old and new values
DECLARE
v_emp_id NUMBER;
v_emp_name VARCHAR2(100);
BEGIN
INSERT INTO employees (emp_id, emp_name, salary, department)
```

```
VALUES (1001, 'John Smith', 75000, 'IT')
RETURNING emp_id, emp_name INTO v_emp_id, v_emp_name;

  DBMS_OUTPUT.PUT_LINE('Inserted Employee: ' || v_emp_name || ' (ID: ' || v_e

END;
/

-- UPDATE with RETURNING old and new values
DECLARE
v_old_salary NUMBER(10,2);
v_new_salary NUMBER(10,2);
v_old_name VARCHAR2(100);
v_new_name VARCHAR2(100);
BEGIN
UPDATE employees
SET salary = salary * 1.1,
emp_name = 'Jane Doe'
WHERE emp_id = 1001
RETURNING salary, emp_name, PRIOR salary, PRIOR emp_name
INTO v_new_salary, v_new_name, v_old_salary, v_old_name;

  DBMS_OUTPUT.PUT_LINE('Updated Employee:');
  DBMS_OUTPUT.PUT_LINE('  Name: ' || v_old_name || ' -> ' || v_new_name);
  DBMS_OUTPUT.PUT_LINE('  Salary: ' || v_old_salary || ' -> ' || v_new_salary);

END;
/

-- DELETE with RETURNING
DECLARE
v_deleted_id NUMBER;
v_deleted_name VARCHAR2(100);
BEGIN
DELETE FROM employees
WHERE emp_id = 1001
RETURNING emp_id, emp_name INTO v_deleted_id, v_deleted_name;

  DBMS_OUTPUT.PUT_LINE('Deleted: ' || v_deleted_name || ' (ID: ' || v_deleted_i

END;
/
```

2.6 GROUP BY and HAVING with Column Aliases

Use column aliases in GROUP BY clauses:

```
-- Create sample data
CREATE TABLE sales_transactions (
transaction_id NUMBER PRIMARY KEY,
product_id NUMBER,
sale_amount NUMBER(10,2),
sale_date DATE,
region VARCHAR2(50)
);

INSERT INTO sales_transactions VALUES (1, 101, 1000, TRUNC(SYSDATE), 'North');
INSERT INTO sales_transactions VALUES (2, 102, 1500, TRUNC(SYSDATE), 'South');
INSERT INTO sales_transactions VALUES (3, 101, 1200, TRUNC(SYSDATE) - 1, 'North');
INSERT INTO sales_transactions VALUES (4, 103, 2000, TRUNC(SYSDATE) - 1, 'East');
COMMIT;

-- GROUP BY with column alias (enhanced feature)
SELECT region AS sales_region,
SUM(sale_amount) AS total_sales,
COUNT(*) AS transaction_count,
ROUND(AVG(sale_amount), 2) AS avg_transaction
FROM sales_transactions
GROUP BY sales_region
HAVING SUM(sale_amount) > 1000
ORDER BY total_sales DESC;

-- GROUP BY CUBE with aliases
SELECT region AS region_name,
EXTRACT(MONTH FROM sale_date) AS month_num,
SUM(sale_amount) AS revenue
FROM sales_transactions
GROUP BY CUBE(region, EXTRACT(MONTH FROM sale_date))
ORDER BY region_name, month_num;

-- GROUP BY ROLLUP with column position
SELECT region,
EXTRACT(YEAR FROM sale_date) AS sale_year,
SUM(sale_amount) AS yearly_revenue
FROM sales_transactions
GROUP BY ROLLUP(region, EXTRACT(YEAR FROM sale_date));

-- GROUP BY GROUPING SETS
SELECT region,
product_id,
SUM(sale_amount) AS total_amount
FROM sales_transactions
GROUP BY GROUPING SETS (region, product_id, ());
```

2.7 UPDATE and DELETE with FROM Clause

Join tables in UPDATE and DELETE statements:

```
-- Create example tables
CREATE TABLE products (
product_id NUMBER PRIMARY KEY,
product_name VARCHAR2(100),
category VARCHAR2(50),
price NUMBER(10,2)
);

CREATE TABLE inventory (
product_id NUMBER PRIMARY KEY,
stock_quantity NUMBER,
reorder_level NUMBER,
last_restocked DATE
);

INSERT INTO products VALUES (1, 'Laptop', 'Electronics', 1200);
INSERT INTO products VALUES (2, 'Mouse', 'Accessories', 50);
INSERT INTO products VALUES (3, 'Monitor', 'Electronics', 400);

INSERT INTO inventory VALUES (1, 5, 10, TRUNC(SYSDATE) - 30);
INSERT INTO inventory VALUES (2, 100, 20, TRUNC(SYSDATE) - 5);
INSERT INTO inventory VALUES (3, 8, 15, TRUNC(SYSDATE) - 20);
COMMIT;

-- UPDATE using FROM clause to join tables
UPDATE inventory i
SET i.reorder_level = i.reorder_level * 1.2,
i.last_restocked = SYSDATE
FROM products p
WHERE i.product_id = p.product_id
AND p.category = 'Electronics'
AND i.stock_quantity < i.reorder_level;

-- DELETE using FROM clause
DELETE FROM inventory i
USING products p
WHERE i.product_id = p.product_id
AND p.category = 'Accessories'
AND i.stock_quantity = 0;

COMMIT;

-- SELECT to verify updates
SELECT p.product_name, i.stock_quantity, i.reorder_level, i.last_restocked
FROM inventory i
JOIN products p ON i.product_id = p.product_id
ORDER BY p.product_name;
```

2.8 INSERT with SET Clause

Simplified INSERT syntax with named columns:

```
-- Traditional INSERT
INSERT INTO products (product_id, product_name, category, price)
VALUES (4, 'Keyboard', 'Accessories', 120);

-- New SET clause syntax (more readable)
INSERT INTO products
SET product_id = 5,
product_name = 'Headphones',
category = 'Accessories',
price = 150;

-- INSERT with BY NAME clause for subquery
INSERT INTO products (product_name, price, product_id, category)
SELECT product_name, price, product_id, category
FROM (
SELECT 6 AS product_id, 'USB Hub' AS product_name,
'Accessories' AS category, 75 AS price
FROM dual
) src
BY NAME;

-- INSERT with subquery values
INSERT INTO products
SET product_id = 7,
product_name = (SELECT 'Display Port Cable' FROM dual),
category = 'Accessories',
price = 30;

COMMIT;
```

Part 3: Time Series and Temporal Operations

3.1 Time Bucketing

Aggregate time series data into fixed intervals:

```
-- Create time series data
CREATE TABLE sensor_readings (
reading_id NUMBER PRIMARY KEY,
sensor_id NUMBER,
reading_value NUMBER(10,2),
reading_time TIMESTAMP
);

INSERT INTO sensor_readings VALUES (1, 101, 23.5, SYSDATE - INTERVAL '1' MINUTE);
INSERT INTO sensor_readings VALUES (2, 101, 23.7, SYSDATE - INTERVAL '2' MINUTE);
INSERT INTO sensor_readings VALUES (3, 101, 23.6, SYSDATE - INTERVAL '3' MINUTE);
INSERT INTO sensor_readings VALUES (4, 101, 23.9, SYSDATE - INTERVAL '4' MINUTE);
```

```
INSERT INTO sensor_readings VALUES (5, 101, 24.1, SYSDATE - INTERVAL '5' MINUTE);
COMMIT;

-- Time bucket to 5-minute intervals
SELECT TRUNC(reading_time, 'MI') +
TRUNC((EXTRACT(MINUTE FROM reading_time) / 5)) * INTERVAL '5' MINUTE AS
bucket_time,
AVG(reading_value) AS avg_reading,
MIN(reading_value) AS min_reading,
MAX(reading_value) AS max_reading,
COUNT(*) AS reading_count
FROM sensor_readings
GROUP BY TRUNC(reading_time, 'MI') +
TRUNC((EXTRACT(MINUTE FROM reading_time) / 5)) * INTERVAL '5' MINUTE
ORDER BY bucket_time;

-- Alternative: Bucket by hour
SELECT TRUNC(reading_time, 'HH') AS hour_bucket,
AVG(reading_value) AS hourly_avg,
STDDEV(reading_value) AS hourly_stddev,
COUNT(*) AS total_readings
FROM sensor_readings
GROUP BY TRUNC(reading_time, 'HH')
ORDER BY hour_bucket;
```

3.2 INTERVAL Calculations

Enhanced support for INTERVAL arithmetic:

```
-- Calculate totals and averages over INTERVAL values
CREATE TABLE event_durations (
event_id NUMBER PRIMARY KEY,
event_name VARCHAR2(100),
duration INTERVAL DAY TO SECOND,
scheduled_date DATE
);

INSERT INTO event_durations VALUES (1, 'Training Session', INTERVAL '2' HOUR +
INTERVAL '30' MINUTE, SYSDATE);
INSERT INTO event_durations VALUES (2, 'Conference', INTERVAL '3' DAY, SYSDATE + 7);
INSERT INTO event_durations VALUES (3, 'Workshop', INTERVAL '4' HOUR, SYSDATE + 14);
COMMIT;

-- Aggregate INTERVAL columns
SELECT AVG(EXTRACT(HOUR FROM duration)) AS avg_hours,
SUM(EXTRACT(DAY FROM duration)) AS total_days,
MAX(duration) AS longest_duration,
MIN(duration) AS shortest_duration
FROM event_durations;

-- Work with INTERVAL values
SELECT event_name,
```

```
duration,
duration * 2 AS doubled_duration,
duration + INTERVAL '1' HOUR AS extended_duration,
EXTRACT(HOUR FROM duration) AS hours_only,
CAST(EXTRACT(DAY FROM duration) AS NUMBER) * 24 +
CAST(EXTRACT(HOUR FROM duration) AS NUMBER) AS total_hours
FROM event_durations;
```

Part 4: SQL Macros and Advanced Features

4.1 Creating Scalar SQL Macros

Define reusable SQL expressions:

```
-- Create a scalar macro for tax calculation
CREATE OR REPLACE FUNCTION calculate_tax(amount IN NUMBER) RETURN NUMBER IS
BEGIN
RETURN amount * 0.15;
END;
/

-- Create a scalar macro for discount
CREATE OR REPLACE FUNCTION apply_discount(original_price IN NUMBER, discount_pct
IN NUMBER)
RETURN NUMBER IS
BEGIN
RETURN original_price * (1 - discount_pct / 100);
END;
/

-- Use macros in SELECT
SELECT product_id,
product_name,
price,
calculate_tax(price) AS tax_amount,
apply_discount(price, 10) AS discounted_price,
price + calculate_tax(price) AS total_with_tax
FROM products
WHERE price > 100;

-- Table-valued macro for common filtering
CREATE OR REPLACE FUNCTION expensive_products(min_price IN NUMBER)
RETURN TABLE PIPELINED AS
BEGIN
FOR rec IN (
SELECT product_id, product_name, price
FROM products
WHERE price >= min_price
ORDER BY price DESC
) LOOP
PIPE ROW(rec);
```

```
END LOOP;
RETURN;
END;
/

-- Use table-valued macro
SELECT * FROM expensive_products(500);
```

4.2 PL/SQL to SQL Transpilation

Automatic conversion of PL/SQL functions to SQL:

```
-- PL/SQL function that gets transpiled to SQL
CREATE OR REPLACE FUNCTION calculate_bonus(salary IN NUMBER, years_employed IN
NUMBER)
RETURN NUMBER IS
BEGIN
IF years_employed < 1 THEN
RETURN salary * 0.05;
ELSIF years_employed < 3 THEN
RETURN salary * 0.10;
ELSE
RETURN salary * 0.15;
END IF;
END;
/

-- Oracle 26ai automatically transpiles this to SQL when used in SELECT
-- (Improves performance by avoiding context switches)
SELECT emp_id,
emp_name,
salary,
calculate_bonus(salary, EXTRACT(YEAR FROM SYSDATE) - EXTRACT(YEAR FROM
hire_date)) AS bonus_amount
FROM employees
WHERE salary > 50000;

-- Transpilable PL/SQL function for string operations
CREATE OR REPLACE FUNCTION format_phone(phone_number IN VARCHAR2)
RETURN VARCHAR2 IS
BEGIN
RETURN '(' || SUBSTR(phone_number, 1, 3) || ') ' ||
SUBSTR(phone_number, 4, 3) || '-' ||
SUBSTR(phone_number, 7, 4);
END;
/

-- Used in SQL (gets transpiled)
SELECT employee_id,
emp_name,
format_phone(phone) AS formatted_phone
```

```
FROM employees
WHERE phone IS NOT NULL;
```

4.3 SQL Annotations

Add metadata to SQL statements:

```
-- Create a table with annotations
CREATE TABLE annotated_transactions (
transaction_id NUMBER PRIMARY KEY,
customer_id NUMBER,
amount NUMBER(10,2),
transaction_date DATE
);

-- Insert with SQL annotations for traceability
/*+ TRACE('financial_module', 'batch_import_v2.3', 'async') */
INSERT INTO annotated_transactions
VALUES (1, 101, 1500.00, SYSDATE);

-- Query with annotations for debugging
/*+ TRACE('reporting', 'daily_summary', 'finance_team')
COST_PRIORITY(HIGH)
EXECUTION_MODE(PARALLEL) /
SELECT customer_id,
SUM(amount) AS total_spent,
COUNT() AS transaction_count,
ROUND(AVG(amount), 2) AS avg_amount
FROM annotated_transactions
GROUP BY customer_id
HAVING SUM(amount) > 5000;

-- Annotation for performance monitoring
/+ TRACE('batch_processing', 'hourly_reconciliation', 'auto')
/+ PARALLEL(4) */ */
UPDATE annotated_transactions
SET transaction_date = TRUNC(SYSDATE)
WHERE transaction_date IS NULL;
```

Part 5: Advanced Data Ingestion

5.1 Memoptimized Rowstore Fast Ingest

High-performance data insertion:

```
-- Create table optimized for fast ingest
CREATE TABLE fast_ingest_table (
data_id NUMBER PRIMARY KEY,
data_value VARCHAR2(500),
received_time TIMESTAMP DEFAULT SYSDATE,
category VARCHAR2(50)
) MEMOPTIMIZED FAST INGEST ROW STORE;
```

```
-- Batch insert with high throughput
BEGIN
FOR i IN 1..10000 LOOP
INSERT INTO fast_ingest_table (data_id, data_value, category)
VALUES (i, 'Data_' || i, MOD(i, 10));

    -- Periodic commits for recovery and batching
    IF MOD(i, 1000) = 0 THEN
       COMMIT;
    END IF;
  END LOOP;
  COMMIT;

END;
/

-- Verify inserted data
SELECT COUNT(*) AS total_records,
COUNT(DISTINCT category) AS unique_categories,
MIN(received_time) AS first_insert,
MAX(received_time) AS last_insert
FROM fast_ingest_table;

-- Query with performance monitoring
SELECT category, COUNT(*) AS record_count
FROM fast_ingest_table
GROUP BY category
ORDER BY record_count DESC;
```

5.2 Pipelining for High-Volume Operations

Send multiple requests without waiting for responses:

```
-- Pipelining is typically used in application code
-- Here's a PL/SQL example simulating pipelined batch operations

CREATE TABLE pipelined_results (
operation_id NUMBER PRIMARY KEY,
status VARCHAR2(20),
result_value VARCHAR2(1000),
execution_time TIMESTAMP
);

-- Simulate pipelined batch inserts
DECLARE
TYPE result_table IS TABLE OF pipelined_results%ROWTYPE;
v_results result_table := result_table();
BEGIN
-- Queue multiple insert operations
```

```
FOR i IN 1..100 LOOP
v_results.EXTEND;
v_results(i).operation_id := i;
v_results(i).status := 'QUEUED';
v_results(i).result_value := 'Operation_' || i;
v_results(i).execution_time := SYSDATE;
END LOOP;
```

```
-- Bulk insert (processed as pipeline)
FORALL i IN 1..v_results.COUNT
   INSERT INTO pipelined_results VALUES v_results(i);

COMMIT;
DBMS_OUTPUT.PUT_LINE('Pipelined ' || v_results.COUNT || ' operations');
```

```
END;
/

-- Verify results
SELECT COUNT(*) AS total_operations,
COUNT(CASE WHEN status = 'QUEUED' THEN 1 END) AS queued_ops
FROM pipelined_results;
```

Part 6: Transaction Management and Priorities

6.1 Transaction Priority and Automatic Rollback

Prevent blocking of high-priority transactions:

```
-- Enable transaction priority management
ALTER SYSTEM SET max_priority_transactions = 10;

-- Set transaction priority for a session
BEGIN
DBMS_SESSION.SET_TRANSACTION_PRIORITY('HIGH');
END;
/

-- Low-priority transaction (vulnerable to rollback)
BEGIN
DBMS_SESSION.SET_TRANSACTION_PRIORITY('LOW');
UPDATE sales_data
SET quantity = quantity - 1
WHERE product_id = 101;
DBMS_LOCK.SLEEP(30); -- Simulate long-running operation
COMMIT;
EXCEPTION
WHEN OTHERS THEN
```

```
DBMS_OUTPUT.PUT_LINE('Transaction rolled back: ' || SQLERRM);
ROLLBACK;
END;
/

-- High-priority transaction (protected)
BEGIN
DBMS_SESSION.SET_TRANSACTION_PRIORITY('HIGH');
UPDATE sales_data
SET quantity = quantity + 10
WHERE product_id = 102;
COMMIT;
DBMS_OUTPUT.PUT_LINE('High-priority transaction committed successfully');
EXCEPTION
WHEN OTHERS THEN
DBMS_OUTPUT.PUT_LINE('Error: ' || SQLERRM);
END;
/
```

6.2 Sessionless Transactions

Microservice-friendly transaction handling:

```
-- Sessionless transaction example (application-level)
-- Typically used with REST or microservices

CREATE TABLE sessionless_ops (
op_id NUMBER PRIMARY KEY,
operation_type VARCHAR2(50),
status VARCHAR2(20),
created_time TIMESTAMP DEFAULT SYSDATE
);

-- Simulate sessionless transaction pattern
DECLARE
v_op_id NUMBER := 1001;
BEGIN
-- Begin session-independent transaction
INSERT INTO sessionless_ops (op_id, operation_type, status)
VALUES (v_op_id, 'API_CALL', 'INITIATED');
COMMIT;
```

```
-- Release connection
-- Connection can be reused for other requests

-- Resume transaction with same operation_id
INSERT INTO sessionless_ops (op_id, operation_type, status)
VALUES (v_op_id + 1, 'API_CALL', 'COMPLETED');
COMMIT;
```

```
DBMS_OUTPUT.PUT_LINE('Sessionless transaction completed: Op ' || v_op_id);
```

END;
/

Part 7: Graph Database Capabilities

7.1 Property Graph Operations

Work with graph data structures:

```
-- Create property graph tables
CREATE TABLE graph_vertices (
vertex_id VARCHAR2(100) PRIMARY KEY,
vertex_label VARCHAR2(100),
properties JSON
);

CREATE TABLE graph_edges (
edge_id VARCHAR2(200) PRIMARY KEY,
source_vertex_id VARCHAR2(100),
target_vertex_id VARCHAR2(100),
edge_label VARCHAR2(100),
properties JSON,
FOREIGN KEY (source_vertex_id) REFERENCES graph_vertices(vertex_id),
FOREIGN KEY (target_vertex_id) REFERENCES graph_vertices(vertex_id)
);

-- Insert graph vertices
INSERT INTO graph_vertices VALUES ('person_1', 'Person',
JSON('{"name": "Alice", "age": 30, "city": "New York"}'));
INSERT INTO graph_vertices VALUES ('person_2', 'Person',
JSON('{"name": "Bob", "age": 28, "city": "Boston"}'));
INSERT INTO graph_vertices VALUES ('person_3', 'Person',
JSON('{"name": "Charlie", "age": 35, "city": "New York"}'));

-- Insert graph edges
INSERT INTO graph_edges VALUES ('edge_1', 'person_1', 'person_2', 'KNOWS',
JSON('{"strength": 8, "since": "2020"}'));
INSERT INTO graph_edges VALUES ('edge_2', 'person_2', 'person_3', 'KNOWS',
JSON('{"strength": 6, "since": "2021"}'));
INSERT INTO graph_edges VALUES ('edge_3', 'person_1', 'person_3', 'WORKS_WITH',
JSON('{"projects": 5}'));
COMMIT;

-- Query graph relationships
SELECT v1.properties.name AS person1,
v2.properties.name AS person2,
e.edge_label AS relationship,
```

```
e.properties.strength AS strength_score
FROM graph_edges e
JOIN graph_vertices v1 ON e.source_vertex_id = v1.vertex_id
JOIN graph_vertices v2 ON e.target_vertex_id = v2.vertex_id
WHERE e.edge_label = 'KNOWS'
ORDER BY strength_score DESC;

-- Find connected components (friend network)
SELECT DISTINCT v1.properties.name AS person,
COUNT(*) OVER (PARTITION BY v1.vertex_id) AS connections
FROM graph_edges e
JOIN graph_vertices v1 ON e.source_vertex_id = v1.vertex_id
ORDER BY connections DESC;
```

Part 8: Ubiquitous Search with DBMS_SEARCH

8.1 Creating Universal Search Indexes

Search across multiple tables simultaneously:

```
-- Create comprehensive search index
CREATE INDEX universal_search_idx
ON products(product_name, description)
INDEXTYPE IS CTXSYS.CONTEXT
PARAMETERS('SYNC(ON COMMIT)');

-- Or use DBMS_SEARCH for easier multi-table indexing
BEGIN
DBMS_SEARCH.CREATE_INDEX(
index_name => 'product_universal_search',
table_name => 'products',
columns => 'product_name,description'
);
END;
/

-- Full-text search across multiple columns
SELECT product_id, product_name, description, price
FROM products
WHERE CONTAINS(product_name, 'laptop') > 0
OR CONTAINS(description, 'high-performance') > 0;

-- Ranked search results
SELECT product_id, product_name,
SCORE(1) AS relevance_score
FROM products
WHERE CONTAINS(description, '(vector NEAR ai)', 1) > 0
ORDER BY SCORE(1) DESC;
```

Practice Exercises

Exercise 1: Vector Search Application

Create a complete vector search scenario for product recommendations:

- Store product embeddings
- Implement similarity search
- Rank results by relevance

Tasks:

1. Extend product_embeddings table with more products
2. Create multiple vector indexes (HNSW, IVF, Hybrid)
3. Write queries comparing different distance metrics
4. Implement RAG-style product recommendations

Exercise 2: Advanced SQL Transformations

Build a data transformation pipeline using new syntax:

- Use VALUES clauses for data loading
- Implement UPDATE with FROM joins
- Apply GROUP BY aliases in complex queries
- Capture before/after values in RETURNING clauses

Tasks:

1. Load test data with VALUES clauses
2. Perform multi-table updates
3. Create aggregation reports
4. Audit data changes

Exercise 3: Time Series Analytics

Develop time series analysis with interval operations:

- Create sensor data with timestamps
- Perform time bucketing for aggregation
- Calculate interval-based metrics
- Implement rolling window functions

Tasks:

1. Insert time series data
2. Aggregate into 5-min, hourly, daily buckets
3. Calculate moving averages
4. Detect anomalies

Performance Optimization Tips

Vector Search Performance

- **Choose appropriate index type:** HNSW for most cases, IVF-Flat for very large datasets
- **Set TARGET ACCURACY** based on use case (95% is typical)
- **Use distance metrics wisely:** Cosine for text embeddings, Euclidean for dense vectors
- **Batch similarity searches** for better throughput

SQL Optimization

- **Use column aliases in GROUP BY** for clearer query plans
- **Leverage RETURNING clause** to avoid additional SELECT statements
- **Use VALUES clause** instead of multiple UNION queries
- **Index columns used in FROM joins** for UPDATE/DELETE operations

Transaction Management

- **Set appropriate priorities** for competing workloads
- **Use sessionless transactions** for microservices architectures
- **Monitor long-running transactions** to prevent blocking
- **Implement automatic rollback** for low-priority transactions

Data Ingestion

- **Enable MEMOPTIMIZED FAST INGEST** for high-volume loading
- **Use pipelining** for batch operations
- **Batch commits** every 1000-5000 rows
- **Monitor ingest throughput** with performance views

Conclusion

Oracle AI Database 26ai represents a paradigm shift in enterprise database architecture. By embedding AI as a native capability alongside traditional SQL and advanced developer features, Oracle enables organizations to:

- Build intelligent applications without external systems
- Maintain strong data governance and security
- Optimize performance for modern microservices
- Implement sophisticated analytics and search capabilities[4]

The features covered in this guide provide the foundation for modern data-driven applications that leverage AI at the database layer.

References

[1] Dataconomy. (2025, November 18). Oracle Database 26ai New Features. https://dataconomy.com/2025/11/19/oracle-database-26ai-new-features/

[2] Oracle. (2025, October 13). Oracle AI Database 26ai: Feature Highlights. https://www.oracle.com/database/26ai/

[3] Oracle. (2025, October 15). Oracle AI Database 26ai Powers the AI for Data Revolution. https://www.oracle.com/news/announcement/ai-world-database-26ai-powers-the-ai-for-data-revolution-2025-10-14/

[4] ChatDBA. (2025). Oracle 26ai: 15 Key Developer Features Unveiled. https://www.chatdba.com/blog/oracle-26ai-15-key-developer-features-unveiled