ST2ADB - Advanced Databases

Course Handout

I2 - Data Science (DAI, DE, BIA, BIO)

Doryan Denis

Instructor: Diana Allam

Academic Year 2025

This handout integrates both lecture content and laboratory exercises, providing a unified reference for SQL, PL/SQL, and Distributed Databases.

Contents

1 PL/SQL **1**

1.1 What is PL/SQL? . . . 1
1.2 PL/SQL Block Structure . . . 2
1.3 Variables & Types . . . 5
1.4 Assignments & `SELECT ... INTO` . . . 9
1.5 Using Variables in DML . . . 11
1.6 Control Structures . . . 14
1.7 Exception Handling . . . 18
1.8 Triggers . . . 23
1.9 Subprograms: Procedures & Functions . . . 29
1.10 Packages . . . 32
1.11 Cursors . . . 37

2 Transactions **42**

2.1 Transaction Basics . . . 42
2.2 ACID Properties . . . 45
2.3 Interleaving & Serializability . . . 49

3 Object-Relational Databases **53**

3.1 ER Model Refresher & Inheritance Mapping . . . 53
3.2 SQL3 Object Types & Inheritance in Oracle . . . 56
3.3 Object Tables & Constraints . . . 61

4 Views, Indexes, and Access Rights **66**

4.1 Views . . . 66
4.2 Indexes . . . 70
4.3 Access Rights . . . 74

5 Distributed Databases & NoSQL Framing **78**

5.1 Why Distribution? . . . 78
5.2 Transparency Types . . . 81
5.3 Fragmentation . . . 85
5.4 Replication . . . 90
5.5 Naming & Allocation . . . 94
5.6 Distributed Query Processing . . . 97
5.7 Distributed Transactions . . . 101
5.8 Architectures . . . 105

1 PL/SQL

1.1 What is PL/SQL?

Definition

PL/SQL (Procedural Language for SQL) is Oracle's procedural extension of the SQL language. It allows programmers to combine the power of *SQL* for data manipulation with the capabilities of a *procedural language* such as loops, conditions, and exception handling.

In other words, PL/SQL bridges the gap between declarative database operations and algorithmic control structures. It enables us to write full programs that interact directly with the Oracle Database, encapsulating logic close to the data rather than relying on external application code.

Purpose. While SQL alone can query and modify data, it lacks constructs for procedural logic (such as `IF`, `LOOP`, and `FUNCTIONS`). PL/SQL extends SQL by allowing:

- the creation of **blocks of code** executed as a single unit (anonymous blocks or stored programs);
- the use of **variables** and **constants** to store intermediate results;
- the definition of **procedures**, **functions**, and **packages** for modular programming;
- the handling of **exceptions** to manage runtime errors gracefully;
- the implementation of **triggers** that automatically respond to events such as insertions, updates, or deletions.

Integration with SQL. PL/SQL is *tightly integrated* with SQL:

- All standard SQL statements (`SELECT`, `INSERT`, `UPDATE`, `DELETE`) can be executed inside PL/SQL blocks.
- SQL and PL/SQL share the same **data types** and can exchange data seamlessly through variables.
- SQL queries can directly assign values to PL/SQL variables using `SELECT ... INTO`.

Advantages over pure SQL. The procedural nature of PL/SQL provides significant advantages when compared to traditional SQL scripting:

- **Algorithmic control:** the ability to execute conditional and iterative logic directly within the database.

- **Code reusability:** procedures and functions can be stored in the database and reused by multiple applications.
- **Performance:** PL/SQL reduces the number of round-trips between the application and the database by executing logic server-side.
- **Error handling:** built-in exception management allows developers to handle database errors programmatically.
- **Security and encapsulation:** sensitive logic can be stored securely within the database, protecting business rules from external tampering.

Example of use

A simple example of a PL/SQL block:

```
BEGIN
    UPDATE employees
    SET salary = salary * 1.05
    WHERE department_id = 10;
    DBMS_OUTPUT.PUT_LINE('Salaries updated successfully.');
END;
/
```

This block increases all salaries in department 10 by 5% and prints a confirmation message. The symbol `/` at the end executes the block in tools such as SQL*Plus or SQL Developer.

Key takeaway

PL/SQL is not a separate programming language but an *extension* of SQL designed to bring procedural logic, modularity, and error control directly into the Oracle database environment. It forms the foundation for all advanced Oracle features such as triggers, stored procedures, and packages.

1.2 PL/SQL Block Structure

Definition

A **PL/SQL block** is the fundamental unit of program organization in Oracle. Every PL/SQL program—whether an anonymous block, a procedure, a function, or a trigger—is composed of one or several blocks. Each block groups together declarations, executable instructions, and optional exception-handling logic.

General syntax

A complete block follows this general structure:

```
DECLARE
    -- Declarations of variables, constants, cursors, and user-defined types
BEGIN
    -- Executable statements
EXCEPTION
    -- Exception-handling statements
END;
/
```

- The `DECLARE` section is *optional*. It introduces variables, constants, and cursors.
- The `BEGIN` section contains executable statements. At least one executable statement is required.
- The `EXCEPTION` section (optional) defines how runtime errors are handled.
- The `END;` keyword terminates the block. In SQL*Plus or SQL Developer, a forward slash (`/`) on a new line is required to execute the block.

Minimal block. A minimal block can contain a single null instruction, useful for testing the structure or placeholders:

```
BEGIN
    NULL;
END;
/
```

Here, the `NULL;` statement explicitly means "do nothing," ensuring the block remains syntactically valid.

Nested blocks and scope. Blocks can be nested inside one another, creating local variable scopes. Each inner block can:

- access variables declared in its parent block (outer scope);
- declare new variables of its own, which remain local to that inner block.

If a variable of the same name is declared both in an inner and outer block, the inner one *shadows* the outer variable within its scope.

```
DECLARE
    v_message VARCHAR2(30) := 'Outer block';
BEGIN
    DECLARE
        v_message VARCHAR2(30) := 'Inner block';
    BEGIN
        DBMS_OUTPUT.PUT_LINE(v_message); -- Prints: Inner block
    END;
    DBMS_OUTPUT.PUT_LINE(v_message);     -- Prints: Outer block
END;
/
```

This example shows how variable scope is hierarchical and that variable names should be chosen carefully to avoid ambiguity.

Comment styles. PL/SQL supports two types of comments:

- **Single-line comments:** introduced by `--`, valid until the end of the line.
- **Multi-line comments:** enclosed between `/*` and `*/`.

```
-- This is a single-line comment

/*
   This is a multi-line comment
   spanning several lines.
*/
```

Proper commenting improves readability and maintainability of code, especially in nested or long PL/SQL blocks.

Traps & Keys

- In **SQL*Plus** or similar Oracle clients, the forward slash `/` on a new line is *mandatory* to execute a standalone block. Without it, the block remains buffered but not executed.
- Nested blocks follow **scope rules**: inner variables can hide (shadow) outer variables of the same name. This can lead to unintentional behavior if naming is inconsistent.
- Each block is self-contained: variables declared within it cease to exist once the block finishes executing.

Key takeaway

Every PL/SQL program is composed of one or more blocks, which can contain declarations, executable logic, and error handling. Understanding block structure and variable scope is essential for writing clear, predictable, and maintainable PL/SQL code.

1.3 Variables & Types

Definition

In PL/SQL, **variables** are memory locations used to store temporary data during program execution. They enable computations, comparisons, and the dynamic handling of data retrieved from the database. Every variable must be declared before it can be used, and each declaration must specify a valid **data type**.

Syntax of a declaration

A variable declaration follows this structure:

```
variable_name datatype [CONSTANT] [NOT NULL] [:= initial_value];
```

- **variable_name** — must begin with a letter and may include letters, digits, and underscores.
- **datatype** — defines the type and size of data that can be stored.
- **CONSTANT** — specifies that the variable's value cannot be changed after initialization.
- **NOT NULL** — enforces that the variable must always have a value (an initial value is therefore required).
- **:= initial_value** — assigns a default value at declaration.

Example

```
DECLARE
    v_name      VARCHAR2(30) := 'Alice';
    v_salary    NUMBER(7,2)  := 2500.50;
    v_bonus     CONSTANT NUMBER(4,2) := 250.00;
    v_hire_date DATE := SYSDATE;
BEGIN
```

```
        DBMS_OUTPUT.PUT_LINE('Employee: ' || v_name);
        DBMS_OUTPUT.PUT_LINE('Salary: ' || v_salary);
    END;
    /
```

This block declares several variables of different data types, initializes them, and displays their values.

Base data types. PL/SQL supports a rich variety of scalar data types, compatible with those in SQL:

- **VARCHAR2(n)** — variable-length character string of up to n characters.
- **CHAR(n)** — fixed-length character string, right-padded with spaces.
- **NUMBER(p,s)** — numeric value with precision p and scale s.
- **INTEGER** — equivalent to `NUMBER(38,0)` (integer type).
- **DATE** — stores date and time values.
- **BOOLEAN** — can hold `TRUE`, `FALSE`, or `NULL` (cannot be used directly in SQL statements).

Type inheritance with `%TYPE`. The `%TYPE` attribute allows a variable to inherit the data type of a table column or another variable. This ensures automatic consistency between variable definitions and database schema, avoiding redundancy and type mismatch.

```
DECLARE
    v_emp_name employees.last_name%TYPE;
    v_emp_sal  employees.salary%TYPE;
BEGIN
    SELECT last_name, salary
    INTO v_emp_name, v_emp_sal
    FROM employees
    WHERE employee_id = 100;
END;
/
```

Here, `v_emp_name` and `v_emp_sal` automatically adopt the data types of the `employees` table columns.

Row inheritance with %ROWTYPE. The %ROWTYPE attribute declares a record that represents a full row from a table or a cursor. This allows handling multiple columns together as a single structured variable.

```
DECLARE
    v_emp employees%ROWTYPE;
BEGIN
    SELECT * INTO v_emp
    FROM employees
    WHERE employee_id = 200;
    DBMS_OUTPUT.PUT_LINE(v_emp.last_name || ' - ' || v_emp.salary);
END;
/
```

Using %ROWTYPE simplifies code maintenance and provides automatic synchronization with table definitions.

Host (global) variables in SQL*Plus. When using SQL*Plus or similar command-line tools, **host variables** allow interaction between PL/SQL blocks and the client environment. Host variables are prefixed with a colon (:) and are defined at the SQL*Plus level, not inside the PL/SQL block.

```
-- Define a host variable in SQL*Plus
VARIABLE g_bonus NUMBER;

BEGIN
    :g_bonus := 500;
END;
/

PRINT g_bonus;
```

Host variables are particularly useful for exchanging values between multiple PL/SQL blocks or scripts executed sequentially in the same session.

From Lab: Environment & Tooling. In practice, PL/SQL can be executed in different environments. The two main options used in this course are:

- **Oracle LiveSQL (Web-based platform):**
  - Allows running SQL and PL/SQL directly in the browser.
  - Displays DBMS_OUTPUT automatically — no need for SET SERVEROUTPUT ON.

- Limited support for SQL*Plus commands (`ACCEPT`, `HOST`, `SPOOL`) and administrative operations.
- Temporary session: databases and scripts disappear after logout or inactivity.

- **Local Oracle XE installation (via Docker container):**
  - Provides a complete local Oracle Database environment suitable for advanced tasks (triggers, transactions, users, concurrency).
  - Basic setup command:

```
docker run -d \
  --name oracle-xe \
  -p 1521:1521 -p 5500:5500 \
  -e ORACLE_PASSWORD=MySecret123 \
  -v oracle-data:/opt/oracle/oradata \
  gvenzl/oracle-xe:21-slim
```

  - Connect using:
    - **SQL Developer (GUI):** Host: `localhost`, Port: `1521`, Service: `XEPDB1`, User: `SYSTEM`.
    - **SQL*Plus inside container:**

```
docker exec -it oracle-xe bash
sqlplus system/MySecret123@XEPDB1
```

  - Start and stop commands:

```
docker stop oracle-xe
docker start oracle-xe
```

Tip. When using Oracle LiveSQL, always **save your scripts locally**, as sessions are ephemeral and any created objects (tables, triggers, procedures) will be lost once disconnected. For persistent development and debugging of PL/SQL programs, a local Oracle XE setup is strongly recommended.

Key takeaway

Proper declaration of variables and careful management of data types are essential to reliable PL/SQL programming. Attributes such as `%TYPE` and `%ROWTYPE` provide strong consistency with database schema, while environment choice determines which features (such as persistence and output control) are available for testing and debugging.

1.4 Assignments & SELECT ... INTO

Definition

In PL/SQL, an **assignment** stores a value into a variable. Assignments can be made explicitly with the operator := or implicitly through a SELECT ... INTO statement that retrieves values from the database.

Assignments are essential for transferring values between expressions, computations, and query results within a PL/SQL block.

Explicit assignment syntax

The basic assignment form is:

```
variable_name := expression;
```

Example

```
DECLARE
    v_counter INTEGER := 0;
BEGIN
    v_counter := v_counter + 1;
    DBMS_OUTPUT.PUT_LINE('Counter = ' || v_counter);
END;
/
```

This assigns the result of the expression on the right-hand side to the variable on the left-hand side.

Implicit assignment with SELECT ... INTO. The SELECT ... INTO clause allows the direct assignment of query results to one or more PL/SQL variables. It is used when the query returns exactly one row, with one or several columns.

```
DECLARE
    v_name     VARCHAR2(30);
    v_salary   NUMBER(7,2);
BEGIN
    SELECT last_name, salary
    INTO v_name, v_salary
    FROM employees
    WHERE employee_id = 100;

    DBMS_OUTPUT.PUT_LINE('Employee: ' || v_name);
```

```
    DBMS_OUTPUT.PUT_LINE('Salary: ' || v_salary);
END;
/
```

Rules

- The number of columns in the `SELECT` clause must exactly match the number of variables listed in the `INTO` clause.
- The data types must be compatible between selected columns and destination variables.
- The query must return **exactly one row**. If no row or more than one row is returned, PL/SQL raises an exception.

Multiple target example

`SELECT ... INTO` can retrieve several values at once:

```
DECLARE
    v_first_name employees.first_name%TYPE;
    v_last_name  employees.last_name%TYPE;
    v_job_id     employees.job_id%TYPE;
BEGIN
    SELECT first_name, last_name, job_id
    INTO v_first_name, v_last_name, v_job_id
    FROM employees
    WHERE employee_id = 150;

    DBMS_OUTPUT.PUT_LINE('Employee: ' || v_first_name || ' ' || v_last_name);
    DBMS_OUTPUT.PUT_LINE('Job ID: ' || v_job_id);
END;
/
```

This structure is useful for fetching entire records or key attributes from a table into local variables for computation or conditional logic.

Single-row requirement. The `SELECT ... INTO` statement is only valid when the query returns a single row. If multiple rows match the condition, Oracle cannot determine which row to assign to the variables, and an exception is raised. To handle cases where multiple rows may be returned, the query should use cursors or aggregate functions (e.g., `MAX`, `COUNT`, `AVG`) to ensure a single result.

Traps & Keys

- A `SELECT INTO` that returns **no rows** raises the predefined exception `NO_DATA_FOUND`.
- A `SELECT INTO` that returns **more than one row** raises the exception `TOO_MANY_ROWS`.
- Both exceptions can be handled in the `EXCEPTION` section of the PL/SQL block, for example:

```
DECLARE
    v_salary employees.salary%TYPE;
BEGIN
    SELECT salary INTO v_salary
    FROM employees
    WHERE department_id = 10;

    DBMS_OUTPUT.PUT_LINE('Salary: ' || v_salary);

EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('No employee found in department 10.');
    WHEN TOO_MANY_ROWS THEN
        DBMS_OUTPUT.PUT_LINE('Multiple employees found in department 10.');
END;
/
```

Key takeaway

The `SELECT ... INTO` statement provides a direct and efficient way to fetch data into PL/SQL variables. However, it is crucial to ensure that the query returns exactly one row; otherwise, exceptions must be handled to maintain robust program behavior.

1.5 Using Variables in DML

Definition

PL/SQL variables can be used directly inside **Data Manipulation Language (DML)** statements such as `INSERT`, `UPDATE`, `DELETE`, and `MERGE`. This integration between procedural logic and SQL commands allows dynamic data manipulation,

computed updates, and programmatic control over the contents of database tables.

Principle. Inside the `BEGIN ... END;` section of a PL/SQL block, variables and expressions can replace literal values in any SQL command. Oracle automatically substitutes the variable's current value at runtime.

Example: INSERT using variables

```
DECLARE
    v_id       NUMBER := 300;
    v_name     VARCHAR2(30) := 'New Student';
    v_year     NUMBER := 1;
BEGIN
    INSERT INTO students (student_id, name, year)
    VALUES (v_id, v_name, v_year);

    DBMS_OUTPUT.PUT_LINE('New record inserted successfully.');
END;
/
```

In this example, three local variables are used to insert a new record into the `students` table. Such use of variables enables insertion of values obtained from computations, user input, or other table queries.

Example: UPDATE using variables

```
DECLARE
    v_increase NUMBER := 10;
    v_spec     VARCHAR2(20) := 'Mathematics';
BEGIN
    UPDATE teachers
    SET current_salary = current_salary + v_increase
    WHERE specialty = v_spec;

    DBMS_OUTPUT.PUT_LINE('Salaries updated for specialty: ' || v_spec);
END;
/
```

The variable `v_increase` determines the adjustment amount, while `v_spec` specifies which teachers are affected. This approach provides flexibility and avoids hardcoding values inside DML statements.

Combining computation and DML. Variables can hold results of intermediate calculations before being used in DML commands. This allows complex updates that depend on multiple conditions or calculations.

```
DECLARE
    v_bonus NUMBER := 200;
    v_avg   NUMBER;
BEGIN
    SELECT AVG(salary)
    INTO v_avg
    FROM teachers;

    UPDATE teachers
    SET current_salary = current_salary + v_bonus
    WHERE current_salary < v_avg;

    DBMS_OUTPUT.PUT_LINE('Bonus applied to salaries below average.');
END;
/
```

Here, a variable is assigned the average salary from the table, and that value is used in an `UPDATE` statement to determine which rows receive a bonus.

Ensuring consistency with `%TYPE`. When assigning or manipulating values intended for a specific column, it is best practice to declare variables using the `%TYPE` attribute. This ensures perfect alignment with the column's data type and precision, avoiding mismatched types or overflow errors.

```
DECLARE
    v_emp_id   employees.employee_id%TYPE := 999;
    v_new_sal  employees.salary%TYPE := 4200;
BEGIN
    UPDATE employees
    SET salary = v_new_sal
    WHERE employee_id = v_emp_id;

    DBMS_OUTPUT.PUT_LINE('Salary updated for employee ID ' || v_emp_id);
END;
/
```

If the definition of the `salary` column changes in the database (e.g., different precision or scale), the variable will automatically adapt without needing modification in the PL/SQL

code. This mechanism ensures long-term consistency between procedural code and table schema.

Using variables with `INSERT` and `%TYPE`.

```
DECLARE
    v_stud_id   students.student_id%TYPE := 400;
    v_stud_name students.name%TYPE := 'Alice Dupont';
    v_stud_year students.year%TYPE := 3;
BEGIN
    INSERT INTO students (student_id, name, year)
    VALUES (v_stud_id, v_stud_name, v_stud_year);

    DBMS_OUTPUT.PUT_LINE('Inserted student: ' || v_stud_name);
END;
/
```

This example highlights how `%TYPE` ensures compatibility between variable declarations and table columns. It is a recommended practice when writing maintainable and schema-safe PL/SQL programs.

Key takeaway

Using variables inside DML statements brings flexibility, clarity, and adaptability to PL/SQL programs. Combining this with the `%TYPE` attribute ensures structural consistency with the database schema and prevents data type mismatches when tables evolve over time.

1.6 Control Structures

Definition

PL/SQL provides several **control structures** that allow conditional execution and iterative processing within programs. These include `IF` statements for decision-making, and loops (`LOOP`, `WHILE`, and `FOR`) for repeated execution of code blocks. They make PL/SQL a true procedural language capable of expressing algorithms directly inside the database engine.

Conditional structures: `IF / ELSIF / ELSE`

Conditional statements allow executing different instructions depending on logical expressions.
The general syntax is:

```
IF condition THEN
    statements;
ELSIF another_condition THEN
    other_statements;
ELSE
    default_statements;
END IF;
```

Example

```
DECLARE
    v_score NUMBER := 14;
BEGIN
    IF v_score < 10 THEN
        DBMS_OUTPUT.PUT_LINE('Fail');
    ELSIF v_score < 15 THEN
        DBMS_OUTPUT.PUT_LINE('Average');
    ELSE
        DBMS_OUTPUT.PUT_LINE('Good');
    END IF;
END;
/
```

The `IF` statement evaluates conditions in order. Once one condition is true, its corresponding block executes, and the others are skipped.

The `RETURN` statement inside functions. The `RETURN` statement immediately exits a function and provides a value to the calling environment. It can also be used within conditional structures to stop execution early.

```
CREATE OR REPLACE FUNCTION grade_message(p_score NUMBER)
RETURN VARCHAR2 IS
BEGIN
    IF p_score < 10 THEN
        RETURN 'Fail';
    ELSIF p_score < 15 THEN
        RETURN 'Average';
    ELSE
        RETURN 'Good';
    END IF;
END;
```

```
/
```

Unconditional loop (LOOP ... END LOOP)

The simplest loop structure repeatedly executes its body until an EXIT statement is encountered.

```
DECLARE
    v_counter NUMBER := 1;
BEGIN
    LOOP
        DBMS_OUTPUT.PUT_LINE('Iteration: ' || v_counter);
        v_counter := v_counter + 1;
        EXIT WHEN v_counter > 5;
    END LOOP;
END;
/
```

Here, the loop runs five times. The EXIT WHEN condition ensures termination; without it, the loop would continue indefinitely.

While loop (WHILE ... LOOP)

The WHILE loop repeats its body as long as the specified condition remains true. It is typically used when the number of iterations is not known in advance.

```
DECLARE
    v_n NUMBER := 1;
BEGIN
    WHILE v_n <= 3 LOOP
        DBMS_OUTPUT.PUT_LINE('Value: ' || v_n);
        v_n := v_n + 1;
    END LOOP;
END;
/
```

For loop (FOR ... LOOP)

The FOR loop is commonly used when the number of iterations is known. The loop counter is implicitly declared and automatically incremented or decremented by Oracle.

```
BEGIN
```

```
        FOR i IN 1..5 LOOP
            DBMS_OUTPUT.PUT_LINE('i = ' || i);
        END LOOP;
    END;
    /
```

Reverse iteration:

```
BEGIN
    FOR i IN REVERSE 1..3 LOOP
        DBMS_OUTPUT.PUT_LINE('Countdown: ' || i);
    END LOOP;
END;
/
```

The `REVERSE` keyword allows iterating backward through the range.

Labels and exiting outer loops. Labels are identifiers assigned to loops, allowing precise control when multiple loops are nested. By prefixing a loop with a label and using `EXIT label_name`, we can exit a specific outer loop.

```
DECLARE
    v_i NUMBER;
    v_j NUMBER;
BEGIN
    <<outer_loop>>
    FOR v_i IN 1..3 LOOP
        FOR v_j IN 1..3 LOOP
            DBMS_OUTPUT.PUT_LINE('i=' || v_i || ', j=' || v_j);
            EXIT outer_loop WHEN v_i = v_j;
        END LOOP;
    END LOOP;
END;
/
```

In this example, when the values of `v_i` and `v_j` are equal, the program exits both loops simultaneously using the label.

Common pitfalls

- **Off-by-one errors:** forgetting that `FOR i IN 1..N` includes both endpoints. The loop runs N times, not *N-1*.

- **Infinite loops:** omitting an `EXIT` condition inside a `LOOP` results in endless execution.
- **Unclear termination logic:** mixing `EXIT` and `RETURN` can make program flow difficult to follow—prefer explicit control structures.

Key takeaway

Control structures in PL/SQL make it possible to express complete procedural logic directly within the database engine. Proper use of `IF` conditions and loops allows complex data manipulation while maintaining readability and preventing runtime anomalies.

1.7 Exception Handling

Definition

An **exception** in PL/SQL is a runtime error or abnormal condition that interrupts the normal flow of a program. Oracle provides a powerful mechanism to detect, handle, and recover from such situations, ensuring that applications behave predictably even in the presence of unexpected events.

Each PL/SQL block can include an optional `EXCEPTION` section, where different errors are trapped and processed through dedicated handlers.

General structure

```
BEGIN
    -- Executable statements
EXCEPTION
    WHEN exception_name1 THEN
        -- Handling instructions
    WHEN exception_name2 THEN
        -- Alternative handling
    WHEN OTHERS THEN
        -- Catch-all handler
END;
/
```

The `WHEN OTHERS` clause is optional and acts as a default case to handle any unanticipated exception not explicitly listed.

Predefined exceptions. Oracle defines a set of built-in exceptions for common runtime errors. These exceptions are automatically raised when the corresponding condition occurs.

Exception	Raised when...
NO_DATA_FOUND	A SELECT INTO statement returns no rows.
TOO_MANY_ROWS	A SELECT INTO statement returns more than one row.
ZERO_DIVIDE	A division by zero occurs.
INVALID_NUMBER	An attempt is made to convert a string to a number that is invalid.
VALUE_ERROR	An arithmetic, conversion, or constraint error occurs.
DUP_VAL_ON_INDEX	An attempt is made to insert a duplicate value in a unique column or primary key.

Example

```
DECLARE
    v_result NUMBER;
BEGIN
    v_result := 100 / 0;
EXCEPTION
    WHEN ZERO_DIVIDE THEN
        DBMS_OUTPUT.PUT_LINE('Error: Division by zero.');
END;
/
```

Non-predefined exceptions with PRAGMA EXCEPTION_INIT. Some Oracle error codes do not have predefined names. We can associate a specific Oracle error number (ORA-xxxxx) with a user-defined exception name using the PRAGMA EXCEPTION_INIT directive.

```
DECLARE
    e_foreign_key EXCEPTION;
    PRAGMA EXCEPTION_INIT(e_foreign_key, -2292);
    -- ORA-02292: child record found
BEGIN
    DELETE FROM departments WHERE department_id = 10;
EXCEPTION
    WHEN e_foreign_key THEN
        DBMS_OUTPUT.PUT_LINE('Cannot delete: existing related records.');
END;
/
```

This association improves code readability and allows handling of database constraint violations or other Oracle errors explicitly.

User-defined exceptions. Developers can define their own exceptions to handle specific business rules or logical conditions that are not inherently errors in SQL.

```
DECLARE
    e_invalid_salary EXCEPTION;
    v_salary NUMBER := -100;
BEGIN
    IF v_salary < 0 THEN
        RAISE e_invalid_salary;
    END IF;
EXCEPTION
    WHEN e_invalid_salary THEN
        DBMS_OUTPUT.PUT_LINE('Salary cannot be negative.');
END;
/
```

The `RAISE` statement triggers the exception, transferring control immediately to the `EXCEPTION` section.

Using `RAISE_APPLICATION_ERROR`. For custom messages intended for the end user or external applications, Oracle provides the `RAISE_APPLICATION_ERROR` procedure. It allows raising exceptions with a custom error code in the range `-20000` to `-20999`.

```
IF v_salary < v_base_salary THEN
    RAISE_APPLICATION_ERROR(-20001, 'Error: Salary cannot be less than base salary.')
END IF;
```

This mechanism is particularly useful in triggers and stored procedures to enforce business logic directly at the database level.

Retrieving error information: `SQLCODE` and `SQLERRM`. Oracle provides two functions to obtain information about the most recent error:

- `SQLCODE` returns the numeric error code (negative for Oracle errors).
- `SQLERRM` returns the full text message associated with the error.

Example

```
BEGIN
    DELETE FROM teachers WHERE specialty = 'Physics';
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('Error code: ' || SQLCODE);
        DBMS_OUTPUT.PUT_LINE('Error message: ' || SQLERRM);
END;
/
```

These functions are often used in audit or logging tables to keep track of errors for debugging and traceability.

From Lab: Auditing & business rules via triggers and exceptions. In the lab environment, exception handling is combined with **triggers** to enforce constraints and maintain an audit trail of database modifications.

1. Enforcing business rules: "Salary cannot decrease." The following trigger checks that a teacher's salary never decreases. If an update attempts to lower the salary, it raises an exception using `RAISE_APPLICATION_ERROR`.

```
CREATE OR REPLACE TRIGGER trg_check_salary
BEFORE UPDATE OF current_salary ON teachers
FOR EACH ROW
BEGIN
    IF :NEW.current_salary < :OLD.current_salary THEN
        RAISE_APPLICATION_ERROR(-20002, 'Error: Salary cannot decrease.');
    END IF;
END;
/
```

This trigger prevents invalid updates at the database level, ensuring that salary changes follow the defined business rule.

2. Auditing modifications on student scores. The audit mechanism records all changes made to the `RESULTS` table, including who performed them and when. An `AUDIT_RESULTS` table stores this information.

```
CREATE TABLE audit_results (
    v_user     VARCHAR2(50),
    date_maj   DATE,
```

```
    desc_maj   VARCHAR2(20),
    student_id NUMBER,
    course_id  NUMBER,
    points     NUMBER
);
```

A trigger then automatically logs every modification to `RESULTS`:

```
CREATE OR REPLACE TRIGGER trg_audit_results
AFTER INSERT OR DELETE OR UPDATE ON results
FOR EACH ROW
BEGIN
    IF INSERTING THEN
        INSERT INTO audit_results
        VALUES (USER, SYSDATE, 'INSERT',
                :NEW.student_id, :NEW.course_id, :NEW.points);

    ELSIF DELETING THEN
        INSERT INTO audit_results
        VALUES (USER, SYSDATE, 'DELETE',
                :OLD.student_id, :OLD.course_id, :OLD.points);

    ELSIF UPDATING THEN
        INSERT INTO audit_results
        VALUES (USER, SYSDATE, 'UPDATE',
                :NEW.student_id, :NEW.course_id, :NEW.points);
    END IF;
END;
/
```

This trigger provides full traceability of operations on student results, storing:

- the username of the person performing the change (`USER`);
- the date of modification (`SYSDATE`);
- the type of operation (`'INSERT'`, `'DELETE'`, or `'UPDATE'`);
- and the affected data (`student_id`, `course_id`, `points`).

Key takeaway

Exception handling ensures robust and secure PL/SQL programs. By combining predefined and user-defined exceptions, developers can manage both technical errors and business rule violations gracefully. In database triggers, exceptions not only prevent invalid updates but can also trigger auditing mechanisms that preserve accountability and traceability across all data modifications.

1.8 Triggers

Definition

A **trigger** is a named PL/SQL block automatically executed by Oracle in response to a specific event on a database table or view. Triggers are used to enforce business rules, maintain data integrity, automate redundancy, or log activity for auditing purposes.

Each trigger is associated with one or more DML events (`INSERT`, `UPDATE`, or `DELETE`) and can be executed either **before** or **after** the triggering event.

General syntax

```
CREATE [OR REPLACE] TRIGGER trigger_name
{BEFORE | AFTER} {INSERT | UPDATE | DELETE}
[OF column_name]
ON table_name
[FOR EACH ROW]
[WHEN (condition)]
BEGIN
    -- Trigger body
END;
/
```

Trigger timing and event combinations.

- **BEFORE triggers** execute prior to the DML operation and can modify data before it is written to the table.
- **AFTER triggers** execute after the DML operation has been completed.
- Multiple events can be combined, for example:

```
CREATE OR REPLACE TRIGGER trg_example
AFTER INSERT OR UPDATE OR DELETE ON teachers
BEGIN
```

```
    DBMS_OUTPUT.PUT_LINE('Teachers table modified.');
END;
/
```

Row-level vs statement-level triggers. A trigger can execute:

- **Once per statement** — a *statement-level* trigger, declared *without* `FOR EACH ROW`.
- **Once per affected row** — a *row-level* trigger, declared *with* `FOR EACH ROW`.

Row-level triggers can access two pseudorecords:

- `:OLD` — values of the row before the modification.
- `:NEW` — values of the row after the modification.

Example (row-level trigger)

```
CREATE OR REPLACE TRIGGER trg_salary_check
BEFORE UPDATE OF salary ON employees
FOR EACH ROW
BEGIN
    IF :NEW.salary < :OLD.salary THEN
        RAISE_APPLICATION_ERROR(-20010, 'Salary cannot decrease.');
    END IF;
END;
/
```

Here, `:OLD.salary` refers to the previous value, and `:NEW.salary` refers to the updated one. Both records are only available inside row-level triggers.

Conditional triggers with the `WHEN` clause. The `WHEN` clause allows a trigger to execute only when a given condition is true. It can only reference `:NEW` and `:OLD` values (not prefixed by colon inside the condition).

Example

```
CREATE OR REPLACE TRIGGER trg_bonus_update
BEFORE UPDATE OF salary ON employees
FOR EACH ROW
WHEN (NEW.salary > 5000)
BEGIN
    DBMS_OUTPUT.PUT_LINE('High-salary update detected.');
END;
```

```
/
```

Event predicates. Within a trigger body, three Boolean functions can be used to determine which event caused activation:

- `INSERTING`
- `UPDATING`
- `DELETING`

These allow a single trigger to handle multiple event types cleanly.

Example

```
CREATE OR REPLACE TRIGGER trg_multi_event
AFTER INSERT OR UPDATE OR DELETE ON students
FOR EACH ROW
BEGIN
    IF INSERTING THEN
        DBMS_OUTPUT.PUT_LINE('New student inserted.');
    ELSIF UPDATING THEN
        DBMS_OUTPUT.PUT_LINE('Student record updated.');
    ELSIF DELETING THEN
        DBMS_OUTPUT.PUT_LINE('Student record deleted.');
    END IF;
END;
/
```

Trigger management: enable, disable, drop.

- Disable a trigger:

```
ALTER TRIGGER trg_example DISABLE;
```

- Enable a trigger:

```
ALTER TRIGGER trg_example ENABLE;
```

- Drop a trigger:

```
DROP TRIGGER trg_example;
```

- List triggers for a table:

```
SELECT trigger_name, status
FROM user_triggers
WHERE table_name = 'TEACHERS';
```

From Lab: Operational patterns. In the laboratory work, triggers are used to enforce operational consistency and auditability within the database. Below are three essential trigger patterns explored in practice.

1. Automatic redundancy maintenance. A redundancy table `TEACHER_SPECIALTY(SPECIALTY, NB_TEACHERS)` stores the number of teachers for each specialty. The following trigger set maintains this redundancy automatically for all `INSERT`, `UPDATE`, and `DELETE` operations on the `TEACHERS` table.

```
CREATE OR REPLACE TRIGGER trg_teacher_specialty
AFTER INSERT OR UPDATE OR DELETE ON teachers
FOR EACH ROW
BEGIN
    IF INSERTING THEN
        UPDATE teacher_specialty
        SET nb_teachers = nb_teachers + 1
        WHERE specialty = :NEW.specialty;

    ELSIF DELETING THEN
        UPDATE teacher_specialty
        SET nb_teachers = nb_teachers - 1
        WHERE specialty = :OLD.specialty;

    ELSIF UPDATING THEN
        IF :NEW.specialty != :OLD.specialty THEN
            UPDATE teacher_specialty
            SET nb_teachers = nb_teachers - 1
            WHERE specialty = :OLD.specialty;

            UPDATE teacher_specialty
            SET nb_teachers = nb_teachers + 1
            WHERE specialty = :NEW.specialty;
        END IF;
    END IF;
```

```
END;
/
```

This ensures that the redundancy table remains synchronized without manual intervention.

2. Cascaded update: maintaining consistency on deletion. When a teacher is deleted or renumbered, associated workloads must remain consistent. The following trigger cascades the update or handles cleanup automatically.
Example (delete cascade):

```
CREATE OR REPLACE TRIGGER trg_delete_workload
AFTER DELETE ON teachers
FOR EACH ROW
BEGIN
    DELETE FROM workload
    WHERE teacher_id = :OLD.teacher_id;
END;
/
```

Example (update cascade on renumbering):

```
CREATE OR REPLACE TRIGGER trg_update_workload
AFTER UPDATE OF teacher_id ON teachers
FOR EACH ROW
BEGIN
    UPDATE workload
    SET teacher_id = :NEW.teacher_id
    WHERE teacher_id = :OLD.teacher_id;
END;
/
```

These ensure relational consistency when a teacher's record is modified or deleted.

3. Audit security: tracking score changes. The audit mechanism records who performed changes, when they occurred, and what data was affected. An `AUDIT_SCORES` table stores this information:

```
CREATE TABLE audit_scores (
    v_user     VARCHAR2(50),
    operation  VARCHAR2(10),
    date_maj   DATE,
    student_id NUMBER,
    course_id  NUMBER,
```

```
    old_points NUMBER,
    new_points NUMBER
);
```

The corresponding trigger:

```
CREATE OR REPLACE TRIGGER trg_audit_scores
AFTER INSERT OR DELETE OR UPDATE ON results
FOR EACH ROW
BEGIN
    IF INSERTING THEN
        INSERT INTO audit_scores
        VALUES (USER, 'INSERT', SYSDATE,
                :NEW.student_id, :NEW.course_id,
                NULL, :NEW.points);

    ELSIF DELETING THEN
        INSERT INTO audit_scores
        VALUES (USER, 'DELETE', SYSDATE,
                :OLD.student_id, :OLD.course_id,
                :OLD.points, NULL);

    ELSIF UPDATING THEN
        INSERT INTO audit_scores
        VALUES (USER, 'UPDATE', SYSDATE,
                :NEW.student_id, :NEW.course_id,
                :OLD.points, :NEW.points);
    END IF;
END;
/
```

This structure guarantees full auditability of all modifications to scores, identifying:

- the operation type (`INSERT`, `DELETE`, `UPDATE`);
- the user who performed the operation (`USER`);
- the timestamp of the change (`SYSDATE`);
- and both the old and new values of the modified fields.

Key takeaway

Triggers allow the database to act autonomously, reacting to data events with consistency rules, cascades, and auditing. Through proper use of timing, event predicates, and pseudorecords, they provide powerful tools to enforce business logic and maintain data integrity directly within the Oracle engine.

1.9 Subprograms: Procedures & Functions

Definition

PL/SQL allows the definition of reusable program units called **subprograms**. A subprogram is a named PL/SQL block that performs a specific task and can be called from other PL/SQL code or directly from SQL*Plus (in the case of procedures).

There are two types of subprograms:

- **Procedures** — perform an action; they do not return a value directly.
- **Functions** — perform computations and return a value.

Subprograms promote modular design, reusability, and maintainability by encapsulating logic into named units.

General syntax

Procedure:

```
CREATE [OR REPLACE] PROCEDURE procedure_name
[(parameter [mode] datatype [, ...])]
IS
BEGIN
    -- Executable statements
END [procedure_name];
/
```

Function:

```
CREATE [OR REPLACE] FUNCTION function_name
[(parameter [mode] datatype [, ...])]
RETURN return_datatype
IS
BEGIN
    -- Computation and logic
    RETURN value;
```

```
END [function_name];
/
```

Parameter modes. Subprogram parameters define how data is passed between the caller and the subprogram. There are three modes:

- **IN** — (default) data is passed *into* the subprogram; it cannot be modified inside.
- **OUT** — used to return values *from* the subprogram; its initial value is undefined.
- **IN OUT** — allows both input and output; the parameter is initialized by the caller and may be modified within the subprogram.

Example (parameter mode usage)

```
CREATE OR REPLACE PROCEDURE adjust_salary(
    p_emp_id   IN employees.employee_id%TYPE,
    p_factor   IN NUMBER,
    p_new_sal  OUT NUMBER)
IS
BEGIN
    UPDATE employees
    SET salary = salary * p_factor
    WHERE employee_id = p_emp_id
    RETURNING salary INTO p_new_sal;

    DBMS_OUTPUT.PUT_LINE('New salary applied.');
END;
/
```

Procedure example (stored)

The following procedure increases the salary of all employees in a given department by 10%:

```
CREATE OR REPLACE PROCEDURE raise_department_salary(
    p_dept_id IN employees.department_id%TYPE)
IS
BEGIN
    UPDATE employees
    SET salary = salary * 1.10
    WHERE department_id = p_dept_id;
```

```
    DBMS_OUTPUT.PUT_LINE('Salaries raised by 10% for department ' || p_dept_id);
END;
/
```

Execution in SQL*Plus:

```
EXEC raise_department_salary(30);
```

Execution inside another PL/SQL block:

```
BEGIN
    raise_department_salary(30);
END;
/
```

Function example (stored)

Functions return values and can be used in expressions, assignments, or even SQL queries (provided they are deterministic and free of side effects).

```
CREATE OR REPLACE FUNCTION annual_bonus(
    p_salary IN NUMBER)
RETURN NUMBER
IS
    v_bonus NUMBER;
BEGIN
    v_bonus := p_salary * 0.15;
    RETURN v_bonus;
END;
/
```

Execution from SQL*Plus:

```
VARIABLE g_bonus NUMBER;
EXEC :g_bonus := annual_bonus(4000);
PRINT g_bonus;
```

Usage within a PL/SQL block:

```
DECLARE
    v_salary NUMBER := 4000;
    v_bonus  NUMBER;
BEGIN
```

```
    v_bonus := annual_bonus(v_salary);
    DBMS_OUTPUT.PUT_LINE('Calculated bonus = ' || v_bonus);
END;
/
```

Anonymous vs stored subprograms.

- **Anonymous subprograms** are defined and executed immediately within a PL/SQL block; they are not saved in the database.
- **Stored subprograms** (created using `CREATE OR REPLACE`) are compiled and stored in the data dictionary for reuse by other programs and users.

Stored subprograms can be listed and managed using system views:

```
SELECT object_name, object_type, status
FROM user_objects
WHERE object_type IN ('PROCEDURE', 'FUNCTION');
```

Key takeaway

Procedures and functions allow modularization of logic within the database. Procedures are action-oriented and may return results through `OUT` parameters, while functions compute and return a single value. Using stored subprograms enhances performance, maintainability, and security by centralizing application logic directly in the Oracle engine.

1.10 Packages

Definition

A **package** is a collection of logically related PL/SQL elements — procedures, functions, variables, cursors, and types — grouped together under a single name. Packages promote modular programming, code reuse, and encapsulation, by separating public declarations (interface) from private implementations (logic).

Each package consists of two distinct parts:

- the **Specification** (public interface);
- the **Body** (implementation of procedures, functions, and internal logic).

Structure of a package

1. **Package specification (spec):**

```
CREATE OR REPLACE PACKAGE package_name IS
    -- Declarations visible to users
    PROCEDURE procedure_name (parameters);
    FUNCTION function_name (parameters) RETURN datatype;
END package_name;
/
```

2. **Package body:**

```
CREATE OR REPLACE PACKAGE BODY package_name IS
    -- Implementation of declared procedures/functions
    PROCEDURE procedure_name (parameters) IS
    BEGIN
        -- Statements
    END;

    FUNCTION function_name (parameters) RETURN datatype IS
    BEGIN
        -- Computation
        RETURN value;
    END;
END package_name;
/
```

Execution. Once created, package elements can be executed directly:

```
EXEC package_name.procedure_name(arguments);
```

or called within PL/SQL blocks:

```
BEGIN
    package_name.procedure_name(arguments);
END;
/
```

Advantages.

- **Encapsulation:** internal details (variables, helper routines) remain hidden in the package body.

- **Reusability:** multiple programs can share the same logic through a unified interface.
- **Improved performance:** packages are loaded into memory once per session, reducing context switching.
- **Maintainability:** changes in internal logic require recompilation of the package body only — not the specification.

Example

A simple package grouping salary-related operations:

```
CREATE OR REPLACE PACKAGE pkg_salary IS
    PROCEDURE raise_all(p_factor NUMBER);
    FUNCTION annual_bonus(p_salary NUMBER) RETURN NUMBER;
END pkg_salary;
/

CREATE OR REPLACE PACKAGE BODY pkg_salary IS
    PROCEDURE raise_all(p_factor NUMBER) IS
    BEGIN
        UPDATE employees SET salary = salary * p_factor;
    END;

    FUNCTION annual_bonus(p_salary NUMBER) RETURN NUMBER IS
    BEGIN
        RETURN p_salary * 0.15;
    END;
END pkg_salary;
/
```

From Lab: Packaging assessment routines. In the lab, packages are used to structure related routines into coherent modules. The example below shows a package that handles student assessment, combining both a function that computes averages and a procedure that assigns qualitative results based on predefined performance bands.

1. Function: computing the average score. The function `fn_average` computes the average grade for a given student based on the `RESULTS` table.

```
CREATE OR REPLACE FUNCTION fn_average(p_student_id IN NUMBER)
RETURN NUMBER
IS
    v_avg NUMBER;
```

```
BEGIN
    SELECT AVG(points)
    INTO v_avg
    FROM results
    WHERE student_id = p_student_id;

    RETURN v_avg;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        RETURN NULL;
END;
/
```

2. Procedure: producing qualitative results. The procedure `pr_result` classifies a student's average into performance bands.

```
CREATE OR REPLACE PROCEDURE pr_result(p_student_id IN NUMBER)
IS
    v_avg NUMBER;
    v_comment VARCHAR2(30);
BEGIN
    v_avg := fn_average(p_student_id);

    IF v_avg IS NULL THEN
        v_comment := 'No results available';
    ELSIF v_avg < 10 THEN
        v_comment := 'Fail';
    ELSIF v_avg < 12 THEN
        v_comment := 'Average';
    ELSIF v_avg < 14 THEN
        v_comment := 'Pretty good';
    ELSIF v_avg < 16 THEN
        v_comment := 'Good';
    ELSE
        v_comment := 'Very good';
    END IF;

    DBMS_OUTPUT.PUT_LINE('Student ' || p_student_id || ': ' || v_comment);
END;
/
```

3. Packaging both routines together. Both routines can be grouped into a single package called `pkg_assessment`, simplifying their reuse and version control.

```
CREATE OR REPLACE PACKAGE pkg_assessment IS
    FUNCTION fn_average(p_student_id IN NUMBER) RETURN NUMBER;
    PROCEDURE pr_result(p_student_id IN NUMBER);
END pkg_assessment;
/

CREATE OR REPLACE PACKAGE BODY pkg_assessment IS
    FUNCTION fn_average(p_student_id IN NUMBER) RETURN NUMBER IS
        v_avg NUMBER;
    BEGIN
        SELECT AVG(points)
        INTO v_avg
        FROM results
        WHERE student_id = p_student_id;

        RETURN v_avg;
    EXCEPTION
        WHEN NO_DATA_FOUND THEN
            RETURN NULL;
    END;

    PROCEDURE pr_result(p_student_id IN NUMBER) IS
        v_avg NUMBER;
        v_comment VARCHAR2(30);
    BEGIN
        v_avg := fn_average(p_student_id);

        IF v_avg IS NULL THEN
            v_comment := 'No results available';
        ELSIF v_avg < 10 THEN
            v_comment := 'Fail';
        ELSIF v_avg < 12 THEN
            v_comment := 'Average';
        ELSIF v_avg < 14 THEN
            v_comment := 'Pretty good';
        ELSIF v_avg < 16 THEN
            v_comment := 'Good';
        ELSE
```

```
            v_comment := 'Very good';
        END IF;

        DBMS_OUTPUT.PUT_LINE('Student ' || p_student_id || ': ' || v_comment);
    END;
END pkg_assessment;
/
```

Usage. Once compiled, both elements are accessible through the package name:

```
EXEC pkg_assessment.pr_result(101);
```

`fn_average` may also be used directly in expressions:

```
SELECT pkg_assessment.fn_average(101) AS average_score FROM dual;
```

Key takeaway

Packages allow developers to organize related logic into cohesive modules, improving code readability, performance, and reusability. By separating interface (specification) from implementation (body), packages facilitate collaborative development and controlled evolution of complex PL/SQL systems.

1.11 Cursors

Definition

A **cursor** is a pointer to a private memory area in the Oracle server that holds the result set of a SQL query. In PL/SQL, cursors allow row-by-row processing of query results, especially when multiple rows are returned — something not possible with a simple `SELECT INTO` statement.

Cursors are essential when sequentially iterating through query results, applying logic to each row individually.

Implicit vs explicit cursors. PL/SQL uses two types of cursors:

- **Implicit cursors** are automatically created by Oracle for all SQL statements that return a single row (e.g., `INSERT`, `UPDATE`, `DELETE`, and `SELECT INTO`).
- **Explicit cursors** are declared manually by the programmer to handle multi-row query results explicitly.

Implicit cursor attributes. Oracle provides predefined attributes for the implicit cursor `SQL`, which refer to the most recent SQL statement executed.

Attribute	Description
`SQL%FOUND`	Returns TRUE if one or more rows were affected.
`SQL%NOTFOUND`	Returns TRUE if no rows were affected.
`SQL%ROWCOUNT`	Returns the number of rows affected by the last SQL operation.
`SQL%ISOPEN`	Always FALSE for implicit cursors (they are automatically closed).

Example

```
BEGIN
    UPDATE teachers
    SET current_salary = current_salary * 1.05
    WHERE specialty = 'Mathematics';

    IF SQL%FOUND THEN
        DBMS_OUTPUT.PUT_LINE(SQL%ROWCOUNT || ' rows updated.');
    ELSE
        DBMS_OUTPUT.PUT_LINE('No matching teachers found.');
    END IF;
END;
/
```

Explicit cursors. When a query returns multiple rows, an explicit cursor allows controlled navigation through the result set. The programmer defines, opens, fetches, and closes the cursor manually.

Life cycle of an explicit cursor:

1. **Declare** the cursor with a SQL query.
2. **Open** the cursor — the query is executed and the result set is created.
3. **Fetch** rows — retrieve one row at a time into variables.
4. **Close** the cursor to release memory.

Example: manual explicit cursor

```
DECLARE
    CURSOR c_teachers IS
        SELECT teacher_id, specialty, current_salary FROM teachers;
```

```
    v_id teachers.teacher_id%TYPE;
    v_spec teachers.specialty%TYPE;
    v_sal teachers.current_salary%TYPE;
BEGIN
    OPEN c_teachers;
    LOOP
        FETCH c_teachers INTO v_id, v_spec, v_sal;
        EXIT WHEN c_teachers%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE('Teacher ' || v_id || ' (' || v_spec ||
                             ') earns ' || v_sal);
    END LOOP;
    CLOSE c_teachers;
END;
/
```

This block iterates over every teacher, printing their ID, specialty, and salary.

Cursor attributes for explicit cursors. Just like implicit cursors, explicit cursors also provide four built-in attributes:

Attribute	Meaning
c_name%FOUND	TRUE if the last fetch returned a row.
c_name%NOTFOUND	TRUE if the last fetch did not return a row.
c_name%ROWCOUNT	Number of rows fetched so far.
c_name%ISOPEN	TRUE if the cursor is currently open.

Simplified iteration with cursor FOR loop. The explicit open, fetch, and close steps can be automatically handled by a `FOR` loop, greatly simplifying syntax and improving readability.

Example

```
DECLARE
    CURSOR c_results IS
        SELECT student_id, points FROM results WHERE course_id = 'DB';
BEGIN
    FOR r IN c_results LOOP
        DBMS_OUTPUT.PUT_LINE('Student ' || r.student_id ||
                             ' scored ' || r.points);
    END LOOP;
END;
```

```
/
```

In this form, Oracle automatically opens, fetches, and closes the cursor behind the scenes.

Parameterized cursors. Cursors can take parameters, allowing flexible reuse of the same logic with different query criteria.

Example

```
DECLARE
    CURSOR c_student(p_course_id results.course_id%TYPE) IS
        SELECT student_id, points
        FROM results
        WHERE course_id = p_course_id;
BEGIN
    FOR rec IN c_student('SM503M') LOOP
        DBMS_OUTPUT.PUT_LINE('Student ' || rec.student_id ||
                             ' scored ' || rec.points);
    END LOOP;
END;
/
```

From Lab: Cursor-based reporting. In the lab, cursors are used to generate dynamic reports, iterate over result sets, and perform custom aggregations. For instance, a cursor can be used to calculate average scores per student or to display workload details per teacher.

Example 1 – reporting on results:

```
DECLARE
    CURSOR c_student_results IS
        SELECT student_id, AVG(points) AS avg_points
        FROM results
        GROUP BY student_id;
BEGIN
    FOR r IN c_student_results LOOP
        DBMS_OUTPUT.PUT_LINE('Student ' || r.student_id ||
                             ' average = ' || ROUND(r.avg_points, 2));
    END LOOP;
END;
/
```

Example 2 – workload iteration:

```
DECLARE
    CURSOR c_workload IS
        SELECT teacher_id, SUM(hours) AS total_hours
        FROM workload
        GROUP BY teacher_id;
BEGIN
    FOR r IN c_workload LOOP
        DBMS_OUTPUT.PUT_LINE('Teacher ' || r.teacher_id ||
                             ' total workload = ' || r.total_hours || 'h');
    END LOOP;
END;
/
```

These examples demonstrate how cursors allow fine-grained control over multi-row query processing within PL/SQL.

Common pitfalls

- Forgetting to `CLOSE` a cursor after use — leads to unnecessary memory usage.
- Fetching past the last row without checking `%NOTFOUND`.
- Assuming order of rows — unless explicitly sorted, cursor order is undefined.
- Using cursors for operations better handled with set-based SQL — leading to performance degradation.

Key takeaway

Cursors allow row-by-row access to query results and provide procedural control over data traversal. Explicit cursors offer flexibility for complex reporting or data validation, while cursor `FOR` loops simplify iteration for most practical applications.

2 Transactions

2.1 Transaction Basics

Definition

A **transaction** is a logical unit of work that transforms a database from one consistent state to another. It groups one or more SQL statements into a single operation that must be executed entirely or not at all. In other words, all statements within a transaction succeed or all are rolled back if any fail.

A transaction is said to be *committed* when all its changes are permanently stored in the database, and *rolled back* when all uncommitted changes are canceled.

Basic control commands. Transaction control is achieved through three essential SQL commands:

- `COMMIT` — validates all pending changes made in the current transaction and makes them permanent.
- `ROLLBACK` — cancels all uncommitted changes since the last `COMMIT`.
- `SAVEPOINT` — creates a checkpoint within a transaction, allowing partial rollback to that point.

Example

```
-- Start an implicit transaction
UPDATE students SET year = year + 1 WHERE student_id = 1001;

SAVEPOINT before_bonus;

UPDATE results SET points = points + 2 WHERE student_id = 1001;

-- Cancel only the second update
ROLLBACK TO before_bonus;

-- Validate the first update
COMMIT;
```

Explicit vs implicit validation. In Oracle, every data manipulation statement (`INSERT`, `UPDATE`, `DELETE`) is executed within a transaction. By default, transactions start implicitly

when the first DML statement is executed and continue until an explicit `COMMIT` or `ROLLBACK` is issued.
However, validation may also occur **implicitly**:

- Executing any `DDL` statement (`CREATE`, `DROP`, `ALTER`) causes an *automatic commit* both before and after the statement.
- Exiting a SQL*Plus or LiveSQL session without committing causes all uncommitted changes to be rolled back.

Thus, Oracle provides full transaction control but expects the user to explicitly manage validation when required.

Autocommit differences (Oracle vs MySQL).

- In **Oracle**, the `AUTOCOMMIT` feature is **OFF** by default — users must explicitly commit their changes.
- In **MySQL**, the default is `AUTOCOMMIT = ON`, meaning that each DML statement is automatically committed once executed unless explicitly disabled.

Example

```
-- Disable autocommit in MySQL for transactional control
SET autocommit = 0;
START TRANSACTION;

UPDATE customers SET balance = balance - 500 WHERE id = 1;
UPDATE customers SET balance = balance + 500 WHERE id = 2;

COMMIT;
```

When transactions start and end automatically. A transaction in Oracle begins automatically when:

- A DML statement (INSERT, UPDATE, DELETE) is executed.

and ends when:

- The user issues `COMMIT` or `ROLLBACK`.
- A DDL statement executes (implicit commit).
- The session terminates (implicit rollback if no commit).

From Lab: Practical understanding of transactional control. In the laboratory session, students explored how Oracle manages transactions implicitly. Each modification statement starts a new transaction, even if no `BEGIN TRANSACTION` command is issued.

1. Observing commit behavior. In LiveSQL or a local Docker Oracle XE instance:

```
UPDATE results SET points = 15 WHERE student_id = 3;
SELECT points FROM results WHERE student_id = 3;
-- (Value appears changed in your session)
ROLLBACK;
SELECT points FROM results WHERE student_id = 3;
-- (Value reverts to original)
```

This demonstrates that changes remain temporary until a `COMMIT` is issued.

2. Identifying anomalies. When multiple users or sessions interact concurrently without proper transaction control, different anomalies can occur:

- **Dirty Read** — reading uncommitted changes made by another transaction.
- **Non-Repeatable Read** — reading the same row twice and seeing different values.
- **Lost Update** — two concurrent updates overwrite each other.
- **Phantom Read** — new rows appear in a repeated query due to concurrent insertions.

These anomalies motivated the introduction of formal isolation levels studied later in Section 2.3.

3. Hands-on: commit vs rollback. Students verified persistence behavior by executing:

```
-- First session
UPDATE teachers SET current_salary = current_salary * 1.1 WHERE specialty = 'DB';
COMMIT;

-- Second session
UPDATE teachers SET current_salary = current_salary * 0.9 WHERE specialty = 'DB';
ROLLBACK;
```

After the rollback, the second session saw its changes undone, while the committed increase from the first session remained visible to all users.

Traps & Keys

- Oracle automatically performs a **commit before and after DDL** statements (`CREATE`, `DROP`, `ALTER`).
- Failing to issue `COMMIT` causes all pending changes to be lost when the session ends.
- In MySQL, the default mode is `AUTOCOMMIT = ON`; each statement is validated instantly unless explicitly disabled.
- A `ROLLBACK` undoes all uncommitted changes — not just the last statement — unless a `SAVEPOINT` is used.
- If we have a plain `ROLLBACK` without a specified savepoint, even if a savepoint was created earlier, all changes since the beginning of the transaction are undone.

Key takeaway

Transaction control is fundamental to data integrity. Understanding when commits occur, how rollback behaves, and how different systems handle autocommit is essential before tackling concurrency and isolation mechanisms.

2.2 ACID Properties

Definition

Reliable transaction processing relies on the **ACID** properties — *Atomicity, Consistency, Isolation, Durability*. Together, they ensure that database operations behave predictably even in the presence of errors, concurrency, or system failures.

Each property addresses a distinct aspect of correctness and robustness in transaction management.

Atomicity. **Atomicity** guarantees that a transaction's operations are treated as a single, indivisible unit of work. Either all modifications are applied, or none are applied. If any statement within a transaction fails, all preceding changes are rolled back automatically to preserve integrity.

Example (bank transfer)

```
UPDATE accounts SET balance = balance - 500 WHERE id = 1;  -- debit
UPDATE accounts SET balance = balance + 500 WHERE id = 2;  -- credit
```

```
COMMIT;
```

If the system crashes after the debit but before the credit, Atomicity ensures that neither update persists — the entire transaction is rolled back, avoiding partial transfers.

Consistency. **Consistency** ensures that every transaction moves the database from one valid state to another, respecting all declared integrity constraints (primary keys, foreign keys, check constraints) and business rules. A transaction that violates referential integrity or a business invariant (e.g., negative stock quantities) is automatically rolled back.

Example

In a student database, a trigger preventing grades outside the range [0, 20] ensures consistency:

```
CREATE OR REPLACE TRIGGER check_score
BEFORE INSERT OR UPDATE ON results
FOR EACH ROW
BEGIN
    IF :NEW.points < 0 OR :NEW.points > 20 THEN
        RAISE_APPLICATION_ERROR(-20001, 'Invalid score value');
    END IF;
END;
/
```

If an invalid grade is inserted, the trigger raises an exception, and the transaction fails — maintaining a consistent database state.

Isolation. **Isolation** ensures that concurrent transactions behave as if they were executed sequentially. Even when multiple users modify data simultaneously, each transaction must act as if it were running alone, without seeing intermediate (uncommitted) changes from others.

Lack of proper isolation leads to anomalies such as:

- **Dirty Read:** one transaction reads data modified by another uncommitted transaction.
- **Non-Repeatable Read:** a transaction re-reads the same row and obtains a different value.
- **Lost Update:** concurrent updates overwrite each other.
- **Phantom Read:** new rows appear between two identical queries.

Example (lost update)

```
-- T1
SELECT balance FROM accounts WHERE id = 1;  -- returns 1000
-- T2
SELECT balance FROM accounts WHERE id = 1;  -- returns 1000
-- T1
UPDATE accounts SET balance = 900 WHERE id = 1; COMMIT;
-- T2
UPDATE accounts SET balance = 950 WHERE id = 1; COMMIT;  -- overwrites T1
```

Proper isolation levels (Section 2.3) prevent such interference.

Durability. **Durability** guarantees that once a transaction is committed, its changes survive permanently — even if the system crashes immediately afterward. The DBMS ensures this property through mechanisms like redo logs, journaling, and write-ahead logging.

Example

If a power failure occurs right after a successful `COMMIT`, Oracle will recover all committed transactions from its redo logs upon restart, preserving every confirmed update.

Crash and failure illustrations.

- **Atomicity failure:** The system crashes after half of a multi-step transfer (only the debit recorded). After recovery, both debit and credit are undone.
- **Consistency failure:** An application inserts a row violating a constraint (e.g., invalid foreign key). Oracle rejects the operation and restores the previous state.
- **Isolation failure:** Two concurrent updates produce incorrect total stock counts due to missing locks.
- **Durability failure:** A non-logged file system or disabled redo logging causes committed data to be lost after crash — prevented by Oracle's redo and undo mechanisms.

From Lab: Mapping ACID to real-world failures. In **Lab Exercise 2**, students analyzed six scenarios (A–F) to identify which ACID property was violated in each case.

Scenario	Description	Violated Property
A	System crash mid-update; only part of data visible.	Atomicity
B	Another transaction reads account balance before transfer completes.	Isolation
C	Committed transaction lost after restart.	Durability
D	Transaction reads uncommitted change from another.	Isolation
E	Salary updated but related tax deduction not adjusted.	Consistency
F	Negative stock recorded after sale due to missing constraint.	Consistency

Demonstrations:

- **Atomicity violation:** simulate a partial update without commit, then crash or rollback — only the complete transaction should persist.
- **Durability violation:** disable redo logs (theoretically) and observe loss of committed changes after restart.
- **Isolation violation:** execute concurrent sessions reading uncommitted data, showing dirty or non-repeatable reads.

Oracle mechanisms.

- **Redo logs** ensure *Durability* — all changes are recorded before being acknowledged as committed.
- **Constraints, triggers, and exceptions** enforce *Consistency* by rejecting invalid transactions.

Traps & Keys

- Do not confuse **Consistency** with simple referential integrity: it also includes compliance with application-level rules and invariants.
- In Oracle, a `COMMIT` ensures **Durability** even if the system crashes immediately afterward — thanks to redo log persistence.
- Different DBMSs offer varying default isolation guarantees. Oracle provides strong consistency and durability but defaults to medium isolation (`READ COMMITTED`).

Key takeaway

The ACID properties form the foundation of reliable transaction processing. Oracle enforces them through internal logging, rollback segments, and integrity constraints — ensuring that every committed transaction is both permanent and consistent, regardless of concurrent activity or system failure.

2.3 Interleaving & Serializability

Definition

In multi-user database systems, several transactions often execute concurrently to improve overall performance and resource utilization. This is called **interleaving** — the overlapping execution of operations from different transactions on shared data items. While interleaving enhances throughput, it also introduces the risk of inconsistent outcomes when transactions interfere improperly.

Good vs bad interleavings. A **good interleaving** preserves the same result as some serial (one-after-another) execution of transactions. A **bad interleaving** produces a result that could never occur in any serial order.

Example (banking context)

```
-- Transaction T1: Transfer 100€ from A to B
READ(A); A := A - 100; WRITE(A);
READ(B); B := B + 100; WRITE(B);

-- Transaction T2: Compute total balance A + B
READ(A); READ(B); SUM := A + B;
```

If `T2` reads `A` after T1's debit but reads `B` before T1's credit, the total becomes inconsistent (-100€ discrepancy). Proper scheduling or locking avoids such invalid states.

Conflicts and conflict equivalence. A **conflict** occurs when two operations from different transactions access the same data item and at least one is a write.
Types of conflicts:

- **Read–Write (RW):** one transaction reads data another later writes.
- **Write–Read (WR):** one writes, another later reads.
- **Write–Write (WW):** both write the same item.

Two schedules are **conflict-equivalent** if:

1. They contain the same operations on the same data items.
2. The relative order of all conflicting operations is identical.

If a concurrent schedule is conflict-equivalent to a serial schedule, it is called **conflict-serializable**.

Precedence (serialization) graphs. To test whether a schedule is conflict-serializable, we construct a **precedence graph** (or serialization graph).
Algorithm:

1. Each transaction is represented as a node.
2. For every pair of conflicting operations, draw a directed edge: $\texttt{T}_\texttt{i} \rightarrow \texttt{T}_\texttt{j}$ if an operation in $\texttt{T}_\texttt{i}$ precedes and conflicts with one in $\texttt{T}_\texttt{j}$.
3. If the graph is **acyclic**, the schedule is conflict-serializable.

Example

Given the schedule:

```
T1:  READ(A) T3:  READ(A) T1:  WRITE(A) T3:  WRITE(A) T2:  READ(B) T3:
                  READ(B) T2:  WRITE(B) T3:  WRITE(B)
```

Conflicts:

- On `A`: T1 → T3 (WRITE after READ), T3 → T1 (WRITE after WRITE)
- On `B`: T2 → T3, T3 → T2

The resulting graph has cycles ⇒ not serializable.

Two-Phase Locking (2PL). **Two-Phase Locking (2PL)** is a concurrency control protocol ensuring conflict serializability by requiring that all locks be acquired before any are released.
Phases:

- **Growing phase:** a transaction may acquire locks but not release any.
- **Shrinking phase:** once a lock is released, no new locks may be obtained.

Strict 2PL (used in most DBMS, including Oracle) extends this rule:

- All exclusive (write) locks are held until the transaction commits or rolls back.

This guarantees recoverability and avoids cascading rollbacks.

Deadlocks. When two or more transactions wait indefinitely for each other's locked resources, a **deadlock** occurs.
Wait-for graph:

- Each transaction is a node.
- An edge $T_i \rightarrow T_j$ means T_i is waiting for a resource held by T_j.
- A cycle in this graph $\Rightarrow$ deadlock detected.

Strategies:

- **Prevention:** enforce ordering on resource acquisition (e.g., lock ordering, timeout policy).
- **Detection:** allow deadlocks to occur, then periodically detect cycles and abort one transaction (Oracle's strategy).

From Lab: Practical concurrency control and isolation. In **Lab Exercises 3–5**, students implemented all theoretical notions through direct experimentation.

1. Constructing precedence graphs. Using the provided schedule for T1, T2, and T3:

```
1. T1: READ(A)
2. T3: READ(A)
3. T1: WRITE(A)
4. T3: WRITE(A)
5. T2: READ(B)
6. T3: READ(B)
7. T2: WRITE(B)
8. T3: WRITE(B)
```

Students identified all conflicts and drew the precedence graph, then verified its cyclic nature — confirming the schedule was **not conflict-serializable**.

2. Isolation levels in theory and practice. Four standard isolation levels were analyzed and compared:

Isolation Level	Description
Read Uncommitted	Transactions may read uncommitted data (dirty reads possible).
Read Committed	Only committed data is visible; prevents dirty reads (Oracle default).
Repeatable Read	Ensures consistent rereads of rows, but new rows may appear (phantoms).
Serializable	Full isolation; transactions behave as if executed sequentially.

Oracle vs MySQL:

- Oracle supports only `READ COMMITTED` and `SERIALIZABLE`.
- MySQL (InnoDB) supports all four levels.

3. Banking case study. A two-transaction scenario was used:

- T1 transfers funds between accounts.
- T2 generates a balance report.

At low isolation (Read Committed), the report could read mixed states (after debit, before credit). At Serializable, T2 waits until T1 finishes, ensuring a consistent total.

4. Oracle hands-on: anomaly simulation. Students practiced reproducing and resolving concurrency anomalies:

- **Unrepeatable read:** one session updates a row while another rereads it.
- **Phantom read:** one session inserts a row visible to another after repeating a query.
- **Lost update:** concurrent updates overwrite one another.

After switching to `SERIALIZABLE`, Oracle prevented these anomalies by locking read sets or raising:

```
ORA-08177: can't serialize access for this transaction
```

5. Deadlock construction. Students also created a deadlock situation by updating two tables in opposite order:

```
-- Session 1
UPDATE accounts SET balance = balance + 100 WHERE id = 1;
UPDATE accounts SET balance = balance + 100 WHERE id = 2;
```

```
-- Session 2
UPDATE accounts SET balance = balance + 100 WHERE id = 2;
UPDATE accounts SET balance = balance + 100 WHERE id = 1;
```

Oracle automatically detected the deadlock and terminated one transaction:

```
ORA-00060: deadlock detected while waiting for resource
```

The surviving transaction completed successfully, demonstrating Oracle's internal deadlock detection mechanism.

Traps & Keys

- Oracle's default isolation level is **READ COMMITTED** — dirty reads are impossible, but non-repeatable and phantom reads may occur.
- Oracle does not implement **READ UNCOMMITTED**.
- Under **SERIALIZABLE**, Oracle prevents lost updates by raising `ORA-08177: can't serialize access for this transaction`.
- Deadlock prevention is not guaranteed; Oracle performs dynamic **deadlock detection** and aborts one transaction to resolve it.
- High isolation increases correctness but reduces concurrency — a key trade-off in database performance.

Key takeaway

Transaction interleaving improves efficiency but must be carefully controlled to preserve consistency. Serializability provides a theoretical benchmark for safe concurrency, while practical systems like Oracle enforce it via locking and error-based detection — ensuring that even concurrent workloads remain logically equivalent to a serial execution.

3 Object-Relational Databases

3.1 ER Model Refresher & Inheritance Mapping

Recap of the Entity-Relationship model. The **Entity-Relationship (ER) model** provides the conceptual foundation for relational database design. It defines:

- **Entities** — objects or concepts represented in the database (e.g., `Person`, `Course`);
- **Attributes** — properties describing entities (e.g., `name`, `birth_date`);

- **Relationships** — logical associations between entities (e.g., a `Student` *enrolls in* a `Course`);
- **Keys** — unique identifiers such as primary keys (PK) and foreign keys (FK).

The ER model forms the basis for relational design, where entities and relationships are transformed into normalized tables.

Normalization and relational decomposition. Normalization ensures that data is organized efficiently to eliminate redundancy and update anomalies. Each table represents one type of entity or relationship, and attributes are placed in appropriate tables according to functional dependencies.
Typical stages of normalization:

1. **1NF:** remove repeating groups — ensure atomic attribute values;
2. **2NF:** remove partial dependencies on composite keys;
3. **3NF:** remove transitive dependencies between non-key attributes.

Through decomposition, complex structures are split into smaller, related tables joined by primary–foreign key relationships.

From relational to object-relational logic. Traditional relational databases treat data as simple tuples of primitive values. However, modern applications require representing complex entities (e.g., addresses, multimedia, nested structures). The **Object-Relational (OR) model** extends SQL with:

- user-defined **object types** (structured attributes);
- **inheritance** between types;
- **methods** associated with data structures;
- support for **collections** (arrays, nested tables).

This hybrid model bridges the gap between relational storage and object-oriented programming logic, offering better expressiveness and reusability.

Inheritance mapping strategies. Inheritance introduces a challenge: how to represent hierarchies (`Person` $\rightarrow$ `Teacher`/`Student`) in a relational or object-relational schema. Two classical strategies are used:

1. Vertical mapping (table-per-subtype). Each subtype is stored in its own table. The supertype's attributes are placed in a base table, and subtype tables reference it via primary–foreign key links.

Example

```
PERSON(person_id, name, birth_date)
TEACHER(person_id, specialty, hire_date)
STUDENT(person_id, year, weight)
```

Queries retrieving full information require `JOIN` operations between the base and subtype tables. However, this design avoids null columns and cleanly separates type-specific attributes.

2. Horizontal mapping (single-table inheritance). All attributes (common and specific) are placed in one wide table. Subtype identification relies on a discriminator column (e.g., `role = 'T' | 'S'`).

Example

```
PERSON(
    person_id, name, birth_date, role,
    specialty, hire_date, year, weight
)
```

This model simplifies querying but leads to redundancy and many null values when attributes apply only to certain subtypes.

Trade-offs between mapping strategies. The choice between vertical and horizontal mapping depends on the application context:

Criterion	**Vertical mapping**	**Horizontal mapping**
Storage efficiency	Compact; no nulls	Redundant; many nulls
Query performance	Requires joins	Simpler single-table queries
Integrity	Strong separation of subtypes	Risk of inconsistent subtype data
Schema flexibility	Easier to extend with new subtypes	Requires altering main table

Vertical mapping aligns naturally with SQL3 inheritance, while horizontal mapping suits simpler schemas or reporting-oriented workloads.

From Lab: Applying inheritance mapping to the School schema. In the laboratory, students revisited the conceptual **School schema** containing the following key entities:

- `Person` (abstract supertype);

- `Teacher` (specialty, hire date, salary);
- `Student` (birth date, year, weight, address).

1. Mapping exercise. Each group transformed the conceptual model into two relational designs:

1. **Vertical mapping:** base `Person` table + subtype tables `Teacher` and `Student`;
2. **Horizontal mapping:** unified `Person` table with a discriminator column.

Students compared the resulting SQL definitions, highlighting redundancy and maintenance trade-offs.

2. Choice of mapping strategy. After analysis, the lab adopted **vertical mapping**, as it corresponds directly to Oracle's object-relational model. Each subtype (teacher/student) inherits from the `Person` type and is stored in a distinct object table. This choice improves schema extensibility and aligns with SQL3 inheritance semantics.

Traps & Keys

- **Vertical mapping** avoids null-filled columns but requires joins when retrieving complete subtype data.
- **Horizontal mapping** yields simpler queries but introduces redundancy and more complex update logic.
- In **SQL3 and Oracle**, inheritance propagation differs from object-oriented languages:
  - Constraints and methods are *not automatically inherited*;
  - Each subtype may redefine its own rules and behaviors explicitly.

Key takeaway

Understanding inheritance mapping is crucial before implementing object-relational databases. The vertical strategy, used in the lab's School schema, provides a clean structure compatible with Oracle's `CREATE TYPE` and `UNDER` mechanisms, forming the foundation for object typing in Section 3.2.

3.2 SQL3 Object Types & Inheritance in Oracle

Introduction. The SQL:1999 (SQL3) standard extended classical SQL with object-oriented features, introducing the concept of **user-defined structured types**. These

allow the definition of composite objects with attributes, optional methods, and inheritance hierarchies — bridging the gap between relational and object-oriented paradigms.
In Oracle, these features are implemented through the `CREATE TYPE` and `CREATE TABLE OF` mechanisms, forming the core of the Object-Relational Database (ORD) model.

Type definition syntax

A **type** defines the structure (and optionally the behavior) of an object. Each type may contain attributes and, if required, member functions or procedures. Oracle distinguishes between *root types* and *subtypes* that inherit from existing ones.
Basic syntax:

```
CREATE TYPE type_name AS OBJECT (
    attribute_name1 datatype [NOT NULL],
    attribute_name2 datatype,
    ...
) [NOT FINAL];
/
```

Key elements:

- **AS OBJECT** — defines a structured type with attributes.
- **NOT FINAL** — allows other types to inherit from it (if omitted, the type is final by default).
- **Methods** can be added to define behavior:

```
MEMBER FUNCTION get_age RETURN NUMBER;
```

Inheritance syntax

Subtypes extend existing object types using the **UNDER** keyword.
Syntax:

```
CREATE TYPE subtype_name UNDER supertype_name (
    -- additional attributes or methods
);
/
```

All attributes and methods of the supertype are inherited by the subtype, while the subtype may define new ones or override existing methods (if declared `NOT FINAL`).

Example: STAFF and TEACHER types

To illustrate type inheritance, consider a staff hierarchy in the SCHOOL database.
Base type definition:

```
CREATE TYPE staff_type AS OBJECT (
    staff_id      NUMBER,
    name          VARCHAR2(50),
    hire_date     DATE
) NOT FINAL;
/
```

Subtype extending the base type:

```
CREATE TYPE teacher_type UNDER staff_type (
    specialty     VARCHAR2(30),
    base_salary   NUMBER(8,2),
    current_salary NUMBER(8,2)
);
/
```

Here, `teacher_type` inherits all attributes from `staff_type` (ID, name, hire date) and adds new ones specific to teachers.
Verification:

```
DESC teacher_type
```

will show all inherited and extended attributes, confirming the propagation of structure.

Constructors and methods. Oracle automatically creates a **system-defined constructor function** for every object type, allowing users to instantiate objects easily.

Example

```
teacher_type(101, 'Alice', DATE '2021-09-01', 'Mathematics', 2500, 2700)
```

User-defined methods can also be declared within the type definition and implemented in a separate `CREATE TYPE BODY` block.

Example with method

```
CREATE TYPE staff_type AS OBJECT (
    staff_id NUMBER,
    name VARCHAR2(50),
```

```
    hire_date DATE,
    MEMBER FUNCTION years_of_service RETURN NUMBER
) NOT FINAL;
/

CREATE TYPE BODY staff_type AS
    MEMBER FUNCTION years_of_service RETURN NUMBER IS
    BEGIN
        RETURN ROUND(MONTHS_BETWEEN(SYSDATE, hire_date) / 12, 1);
    END;
END;
/
```

Methods behave similarly to object-oriented member functions and can be invoked using dot notation:

```
SELECT s.name, s.years_of_service() FROM staff_table s;
```

From Lab: Implementation of object types in the SCHOOL project. In the laboratory, students implemented the SCHOOL schema using Oracle's object-relational features, defining all entity types and their relationships.

1. **Address type.** Defines a structured composite for storing location data.

```
CREATE TYPE address_type AS OBJECT (
    street       VARCHAR2(50),
    city         VARCHAR2(30),
    postal_code  VARCHAR2(10),
    country      VARCHAR2(30)
);
/
```

2. **Person type (supertype).** Represents the base structure common to all individuals.

```
CREATE TYPE person_type AS OBJECT (
    person_number  NUMBER,
    name           VARCHAR2(40),
    gender         CHAR(1)
) NOT FINAL;
/
```

3. Teacher subtype. Extends `person_type` with employment-related attributes.

```
CREATE TYPE teacher_type UNDER person_type (
    specialty       VARCHAR2(30),
    hire_date       DATE,
    base_salary     NUMBER(8,2),
    current_salary  NUMBER(8,2)
);
/
```

4. Student subtype. Extends `person_type` with educational and personal details, including a nested object attribute for address.

```
CREATE TYPE student_type UNDER person_type (
    birth_date  DATE,
    weight      NUMBER(4,1),
    year        NUMBER,
    address     address_type
);
/
```

5. Verification and exploration. Students verified the inheritance structure using `DESC` in SQL*Plus or SQL Developer:

```
DESC student_type
DESC teacher_type
```

They observed that all inherited attributes from `person_type` appear automatically in the subtype definition, confirming correct propagation of structure.

Traps & Keys

- Oracle requires that all dependent tables be dropped before altering a type's inheritance status — for example, changing a type from `FINAL` to `NOT FINAL`.
- Constraints and methods are **not automatically inherited** by subtypes; they must be redefined explicitly if needed.
- Using `UNDER` automatically inherits all attributes from the supertype but not constraints, indexes, or triggers defined at the table level.
- Types are stored in the Oracle Data Dictionary; dropping or modifying them affects all dependent tables or views.

Key takeaway

SQL3 object types in Oracle enable modular and extensible data modeling. By defining `NOT FINAL` supertypes and extending them with `UNDER`, developers can represent inheritance directly in the database, paving the way for structured object tables and constraints in Section 3.3.

3.3 Object Tables & Constraints

Definition

In Oracle's Object-Relational model, object types can be materialized directly as tables, allowing the storage of complete object instances rather than simple tuples. Such tables are called **object tables**. Each row in an object table is a structured object with attributes defined by its underlying type.

Creating object tables. An object table is created using the syntax:

```
CREATE TABLE table_name OF type_name
    [ (PRIMARY KEY (...), FOREIGN KEY (...) REFERENCES ..., ...) ];
```

Unlike classical relational tables, the columns of an object table are implicitly defined by the attributes of the associated object type.

Example

```
CREATE TYPE staff_type AS OBJECT (
    staff_id    NUMBER,
    name        VARCHAR2(40),
    hire_date   DATE
) NOT FINAL;
/

CREATE TYPE teacher_type UNDER staff_type (
    specialty      VARCHAR2(30),
    base_salary    NUMBER(8,2),
    current_salary NUMBER(8,2)
);
/

CREATE TABLE teachers OF teacher_type (
    PRIMARY KEY (staff_id)
```

```
);
```

Here, each record in `teachers` is a complete instance of `teacher_type`. Attributes are accessed using dot notation, for example:

```
SELECT t.name, t.specialty, t.current_salary FROM teachers t;
```

Storage and representation. Each row in an object table corresponds to an object instance, internally identified by an **Object Identifier (OID)** automatically generated by Oracle. This mechanism allows referencing entire objects, not just scalar columns, enabling object references (REFs) between tables.
Objects stored this way preserve their hierarchical structure and may contain other object types or collections as nested attributes.

Constraints on object tables. Standard relational constraints (PK, FK, CHECK) can be declared directly on object tables, but their scope and propagation differ from the relational model.

- **Primary Keys (PK):** can be declared on one or more scalar attributes of the object type.
- **Foreign Keys (FK):** can reference attributes or REFs to objects in another table.
- **Check constraints:** apply to scalar attributes of the object.

Example

```
CREATE TABLE persons OF person_type (
    CONSTRAINT pk_person PRIMARY KEY (person_number)
);

CREATE TABLE teachers OF teacher_type (
    CONSTRAINT pk_teacher PRIMARY KEY (person_number),
    CONSTRAINT fk_teacher_person FOREIGN KEY (person_number)
        REFERENCES persons(person_number)
);
```

However, constraints defined on a supertype are not automatically replicated in subtype tables — they must be explicitly redeclared.

Behavior across inheritance hierarchies. In Oracle, constraints reside at the table level, not within the type hierarchy. Thus:

- Constraints applied to a supertype's table do not automatically propagate to subtypes.
- A subtype table inherits attributes structurally but not their relational integrity rules.

For this reason, object hierarchies often require manual replication of primary and foreign key constraints on each subtype table.

Duplication, disjointness, and coverage. When modeling inheritance, two integrity issues arise:

- **Disjointness:** each real-world entity (e.g., a person) must belong to exactly one subtype.
- **Coverage:** every supertype instance must appear in at least one subtype.

SQL3 does not enforce these rules automatically — they must be implemented via triggers or application logic.

From Lab: Full School Object-Relational implementation. The laboratory exercise completed the School database's migration from conceptual design to a full Object-Relational implementation.

1. Object tables. Students created distinct tables for each object type:

```
CREATE TABLE persons OF person_type (
    CONSTRAINT pk_person PRIMARY KEY (person_number)
);

CREATE TABLE teachers OF teacher_type (
    CONSTRAINT pk_teacher PRIMARY KEY (person_number),
    CONSTRAINT fk_teacher_person FOREIGN KEY (person_number)
        REFERENCES persons(person_number)
);

CREATE TABLE students OF student_type (
    CONSTRAINT pk_student PRIMARY KEY (person_number),
    CONSTRAINT fk_student_person FOREIGN KEY (person_number)
        REFERENCES persons(person_number)
);
```

Additional tables were also defined for `activities` and `courses`, both modeled as independent object tables.

2. Enforcing partition integrity. To guarantee that each person appears in only one subtype (disjointness), a trigger was implemented:

```
CREATE OR REPLACE TRIGGER trg_check_disjointness
BEFORE INSERT ON teachers
FOR EACH ROW
DECLARE
    v_exists NUMBER;
BEGIN
    SELECT COUNT(*) INTO v_exists
    FROM students
    WHERE person_number = :NEW.person_number;

    IF v_exists > 0 THEN
        RAISE_APPLICATION_ERROR(-20010,
          'Person already exists in STUDENTS table.');
    END IF;
END;
/
```

A symmetric trigger can be created on `students` to check for presence in `teachers`, ensuring strict disjointness of partitioned instances.

3. Collections within object types. Collections introduce multi-valued attributes directly within an object type. Two major collection mechanisms are used in Oracle: **VARRAYs** and **NESTED TABLES**.
Example 1 – VARRAY of UE identifiers:

```
CREATE TYPE ue_type AS VARRAY(5) OF VARCHAR2(10);
/
```

Example 2 – Nested table of results:

```
CREATE TYPE result_type AS OBJECT (
    course_name VARCHAR2(30),
    grade       NUMBER(4,1)
);
/

CREATE TYPE result_table_type AS TABLE OF result_type;
/
```

The `student_type` was then extended to include these collections:

```
CREATE OR REPLACE TYPE student_type UNDER person_type (
    birth_date  DATE,
    weight      NUMBER(4,1),
    year        NUMBER,
    address     address_type,
    ues         ue_type,
    results     result_table_type
);
/
```

Note: the `NESTED TABLE` attribute requires explicit storage specification when used in an object table:

```
CREATE TABLE students OF student_type (
    PRIMARY KEY (person_number),
    NESTED TABLE results STORE AS results_storage
);
```

4. Inspecting object structures. Students verified their schema using Oracle's `DESC` command to visualize hierarchical and nested structures:

```
DESC student_type
DESC students
```

They also tested object constructors for insertion:

```
INSERT INTO persons VALUES (person_type(1, 'Alice', 'F'));
INSERT INTO teachers VALUES (
    teacher_type(1, 'Alice', 'F', 'Mathematics',
                 DATE '2022-09-01', 2500, 2700)
);
```

Traps & Keys

- Constraints (`PRIMARY KEY`, `FOREIGN KEY`, `CHECK`) must be declared on the supertype's object table — they are **not automatically duplicated** in sub-tables.
- Dropping or modifying a supertype (`ALTER TYPE` / `DROP TYPE`) cascades to all subtypes and dependent tables; dependencies must be removed first.
- When altering an object type, all dependent tables must be dropped and recreated before the new version can be compiled.
- Nested tables require explicit `STORE AS` clauses for physical storage definition.

Key takeaway

Object tables in Oracle provide a direct way to persist structured objects while maintaining relational integrity. Their flexibility enables rich modeling through inheritance and collections, but careful constraint management and dependency handling are essential to maintain schema consistency.

4 Views, Indexes, and Access Rights

4.1 Views

Definition

A **view** is a virtual table whose content is defined by a stored SQL query. It does not physically store data (unless materialized) but dynamically displays the result of the underlying query each time it is accessed.

Views act as a logical abstraction layer over the base tables, simplifying access, enforcing security, and encapsulating complex joins or filters.

Virtual vs materialized views. Two main types of views exist in modern relational systems:

- **Virtual views:** computed on demand at query time. They provide always up-to-date results, as data is fetched directly from the base tables.
- **Materialized views:** physically store the query result. They improve performance for heavy aggregations but must be periodically refreshed to remain consistent.

Example syntax

```
-- Virtual view
CREATE VIEW v_employees AS
SELECT emp_id, name, department, salary
FROM employee;

-- Materialized view (Oracle)
CREATE MATERIALIZED VIEW mv_sales
REFRESH FAST ON COMMIT
AS SELECT dept_id, SUM(amount) AS total_sales
FROM sales
GROUP BY dept_id;
```

Benefits of views. Views provide several key advantages:

- **Abstraction:** hide query complexity and internal structure of base tables.
- **Security:** expose only selected attributes or rows to specific users.
- **Simplicity:** allow users to query data through predefined virtual tables.
- **Reusability:** centralize business logic and simplify maintenance.

Syntax and semantics

The standard form for view creation is:

```
CREATE [OR REPLACE] [FORCE | NOFORCE] VIEW view_name AS
SELECT columns
FROM base_tables
[WHERE conditions]
[WITH CHECK OPTION [CONSTRAINT constraint_name]]
[WITH READ ONLY];
```

- `CREATE OR REPLACE` — recreates the view if it already exists.
- `FORCE` — allows creation even if base tables do not yet exist.
- `WITH CHECK OPTION` — enforces that any inserted or updated rows still satisfy the view's condition.
- `WITH READ ONLY` — prevents any modification through the view.

Updatable views. A view is considered **updatable** if each row of the view maps to exactly one row in a base table. Oracle allows updates, inserts, and deletes on simple views when:

- The view is based on a single base table;
- It contains no `GROUP BY`, `DISTINCT`, `UNION`, or aggregation functions;
- All columns being modified come from the same underlying table.

Complex views (joins, unions, derived columns) are generally **non-updatable**.

The `WITH CHECK OPTION` clause. When defining updatable views, the `WITH CHECK OPTION` ensures that any modification performed through the view does not violate its selection condition.

```
CREATE VIEW v_active_employees AS
SELECT * FROM employee
WHERE status = 'ACTIVE'
WITH CHECK OPTION;
```

Attempting to insert or update a record that does not satisfy `status = 'ACTIVE'` will raise an error.
Two modes exist:

- **LOCAL:** checks only the immediate view condition.
- **CASCADED:** checks conditions in all underlying nested views.

Example 1: Simple view

A straightforward projection of a base table:

```
CREATE VIEW v_employee_basic AS
SELECT emp_id, name, department
FROM employee;
```

Example 2: Join view

A combination of multiple tables to present unified information:

```
CREATE VIEW v_department_summary AS
SELECT d.dept_id, d.name AS department_name,
       e.name AS manager_name
FROM department d
JOIN employee e ON e.emp_id = d.manager_id;
```

Example 3: Inheritance-based view (object tables)

Applied in object-relational models, views can present unified access to hierarchies of object tables.

```
CREATE VIEW v_persons AS
SELECT p.person_number, p.name, 'Person' AS role FROM persons p
UNION ALL
SELECT s.person_number, s.name, 'Student' FROM students s
UNION ALL
SELECT t.person_number, t.name, 'Teacher' FROM teachers t;
```

This view abstracts the inheritance hierarchy (`Person` → `Student`/`Teacher`) into a single queryable interface.

From Lab: Implementation of views in the Enterprise_Lambda schema. In the laboratory session, students implemented several practical examples of views within the `Enterprise_Lambda` schema to illustrate abstraction, access control, and updatability.

1. Managerial view (`v_resp`). A join view displaying departments and their responsible managers:

```
CREATE VIEW v_resp AS
SELECT d.dept_id, d.name AS department_name,
       e.emp_id AS manager_id, e.name AS manager_name
FROM department d
JOIN employee e ON e.emp_id = d.manager_id;
```

2. Mission overview view (`v_missions`). A multi-table join aggregating data from `mission`, `employee`, and `department`:

```
CREATE VIEW v_missions AS
SELECT m.mission_id, m.mission_name,
       e.name AS assigned_employee,
       d.name AS department_name,
       m.start_date, m.end_date
FROM mission m
JOIN employee e ON m.emp_id = e.emp_id
JOIN department d ON e.dept_id = d.dept_id;
```

3. Restricted employee view (`v_employee_public`). To ensure confidentiality, a view was created to display employee information excluding sensitive columns (salary, commission).

```
CREATE VIEW v_employee_public AS
SELECT emp_id, name, department, hire_date
FROM employee;
```

Access control:

```
GRANT SELECT ON v_employee_public TO employee;
```

This allows the user `employee` to query public information but not see financial data.

4. Validation: view updatability. Students tested whether views could be updated:

```
UPDATE v_employee_public
SET department = 'Logistics'
WHERE emp_id = 105;
```

Oracle allowed updates because the view is based on a single base table with no aggregation. Conversely, attempting to update `v_missions` (a join view) raised an error, since Oracle cannot determine which base table to modify.

Traps & Keys

- Views built on **aggregations**, **joins**, or **renamed columns** are **not inherently updatable**.
- Complex updates on non-updatable views can be enabled only with **INSTEAD OF triggers** (beyond current scope).
- The clause `WITH CHECK OPTION` ensures that updates or inserts made through the view do not violate its filtering condition; the `CASCADED` form enforces the rule across multiple nested views.
- Dropping or modifying a base table automatically invalidates all dependent views.

Key takeaway

Views serve as a powerful mechanism for abstraction, modularization, and access control in database design. In the Enterprise_Lambda project, they provided both managerial visibility (joins) and security filtering (restricted user views), illustrating the dual purpose of clarity and protection in relational schemas.

4.2 Indexes

Motivation

An **index** is a data structure that accelerates data retrieval operations by reducing the number of disk I/O operations needed to locate rows. Instead of scanning entire tables, the database engine can quickly find matching rows through indexed access paths.
Indexes are particularly effective for:

- Columns frequently used in `WHERE` clauses and joins;
- Foreign keys or unique constraints;
- Sorting and grouping operations.

However, they introduce maintenance overhead during insertions, updates, and deletions, as the index itself must also be modified whenever the underlying data changes.

Conceptual basis: B-tree structures. Most relational databases (including Oracle, MySQL, and PostgreSQL) use a variation of the **B-tree** data structure for indexing. A B-tree (Balanced Tree) maintains sorted keys and ensures that all leaf nodes are at the same depth, thereby guaranteeing logarithmic search time.

- Each node contains an ordered list of keys and pointers to child nodes.
- Searching proceeds by traversing from root to leaf, following key intervals.
- The number of keys per node is determined by the tree's **degree**.

Properties of B-trees:

- Balanced height → ensures consistent search cost.
- Ordered keys → enable range queries.
- Efficient insertion/deletion → local rebalancing maintains structure.

B-tree operations: simplified explanation.

1. **Insertion:** A key is inserted in sorted order within a node. If the node exceeds its capacity, it **splits** into two nodes, and the middle key is promoted to the parent. This propagation can continue up to the root, possibly increasing the tree's height.
2. **Deletion:** The key is removed, and if a node falls below minimum capacity, it may **borrow** a key from a sibling or **merge** with it. Balancing ensures minimal restructuring while maintaining sorted order.
3. **Rebalancing:** Insertions and deletions trigger local rebalancing, keeping access paths efficient.

In Oracle and MySQL, the B-tree structure is further optimized as a **B+Tree**, where all data rows (or pointers to them) are stored only at the leaf level, and internal nodes contain only key values.

Clustered vs non-clustered indexes.

- A **clustered index** determines the physical order of data rows in the table itself. Each table can have only one clustered index because rows can only be physically sorted in one way.
- A **non-clustered index** is a separate structure that stores key values and pointers (ROWIDs) to actual table rows. Multiple non-clustered indexes can exist on different columns.

Oracle note: Oracle does not implement clustered indexes in the same sense as SQL Server; instead, it uses **index-organized tables (IOTs)** to achieve similar functionality.

Syntax examples

Oracle.

```
-- Simple index
CREATE INDEX idx_emp_name ON employee(name);

-- Unique index
CREATE UNIQUE INDEX idx_emp_email ON employee(email);

-- Index on multiple columns
CREATE INDEX idx_emp_dept_job ON employee(dept_id, job_id);
```

MySQL.

```
-- Create a composite index
CREATE INDEX idx_customers_name_city ON customers(last_name, city);

-- Drop an index
DROP INDEX idx_customers_name_city ON customers;
```

Indexes are automatically created for:

- Primary key constraints (`PRIMARY KEY`);
- Unique constraints (`UNIQUE`);
- Foreign keys (implicitly indexed for referential checks in most DBMSs).

Impact on performance. Advantages:

- Significantly improves performance of `SELECT` queries and joins.
- Enables efficient execution of `ORDER BY` and `GROUP BY` operations.
- Reduces response time for lookups involving selective predicates.

Drawbacks:

- Slower `INSERT`, `UPDATE`, and `DELETE` operations because indexes must be maintained.
- Additional disk space for index structures.
- Possible redundant or unused indexes that increase maintenance cost.

From Lab: B+Tree practice exercise. In the laboratory, students explored the internal structure and behavior of B+Trees through an online visualizer (`https://www.cs.usfca.edu/~galles/visualization/BPlusTree.html`).

1. Simulation of insertions. Students simulated the insertion of ten customer names (Anderson, Becker, Carter, Davidson, Evans, Fisher, Green, Hall, Irving, Johnson) into a degree-3 B+Tree. After each insertion, they observed how nodes split and propagated upwards, maintaining a balanced tree with minimal height.
Observation: The B+Tree remains balanced regardless of insertion order — ensuring logarithmic access time.

2. Comparison with sorted list. Students compared the same dataset stored as a sorted list. Sequential scans on the list required linear time ($O(n)$), while indexed lookups in the B+Tree achieved logarithmic time ($O(\log n)$) for searches and range queries.

3. Update and deletion behavior. Next, deletions and updates were performed to observe key reorganization. Merging and borrowing operations maintained structural balance, demonstrating how B+Trees automatically adjust to maintain optimal search paths.

4. When to create indexes. The exercise concluded with a discussion on index design choices:

- Use indexes on columns frequently appearing in `WHERE`, `JOIN`, or `ORDER BY` clauses.
- Index foreign key columns to speed up joins.
- Avoid indexing columns with low selectivity (e.g., gender, boolean flags).
- Avoid indexing frequently updated columns (e.g., salary, balance) to minimize maintenance cost.

Traps & Keys

- Indexes **improve read performance** but **degrade write operations**; each modification requires index maintenance.
- Analyze column **selectivity**: indexes are beneficial when a column has many distinct values relative to total rows.
- Primary keys automatically create unique indexes — adding redundant indexes wastes space and processing.
- Over-indexing leads to increased disk usage and longer update times; review index necessity periodically.

Key takeaway

Indexes are a cornerstone of query optimization in relational databases. The lab's B+Tree simulation concretely demonstrated how balanced tree structures maintain efficient access paths, and why real-world index design requires trade-offs between read speed and write cost.

4.3 Access Rights

Definition

Database management systems enforce access control through a combination of privileges and roles. Privileges determine what operations a user can perform, while roles serve as reusable collections of privileges for easier administration. This system ensures data security, separation of responsibilities, and controlled delegation of authority.

Object-level privileges. **Object privileges** apply to specific database objects (tables, views, procedures, etc.) and regulate how users may interact with them.

- `SELECT` – allows reading data from a table or view.
- `INSERT` – allows inserting new rows.
- `UPDATE` – allows modifying existing data.
- `DELETE` – allows removing rows.
- `REFERENCES` – allows defining foreign keys referencing a table.
- `EXECUTE` – allows running stored procedures or functions.

Example

```
GRANT SELECT, INSERT, UPDATE ON employees TO hr_assistant;
REVOKE DELETE ON employees FROM hr_assistant;
```

These commands respectively grant and revoke privileges for a specific user. Privileges may also be propagated further using the `WITH GRANT OPTION` clause.

System privileges. **System privileges** control the ability to create, modify, or manage database structures. They are broader than object-level privileges and typically reserved for administrators.

- `CREATE TABLE`, `ALTER TABLE`, `DROP TABLE`
- `CREATE USER`, `DROP USER`, `CREATE ROLE`
- `CREATE VIEW`, `CREATE PROCEDURE`, `CREATE SESSION`

Example (Oracle):

```
GRANT CREATE USER, DROP USER, CREATE ROLE TO admin_user;
```

System privileges are often aggregated into predefined roles such as `DBA`, `RESOURCE`, or `CONNECT`.

Roles. A **role** is a named group of privileges that can be granted to users or other roles. Roles simplify administration by allowing the reuse of privilege sets instead of managing them individually per user.

Syntax

```
CREATE ROLE role_name;
GRANT privilege [, privilege ...] TO role_name;
GRANT role_name TO user_name;
REVOKE privilege FROM role_name;
```

Example

```
CREATE ROLE accountant;
GRANT SELECT, UPDATE ON invoices TO accountant;
GRANT accountant TO alice;
```

Users can activate or deactivate roles during sessions depending on security policies.

Differences between Oracle and PostgreSQL roles.

- In **Oracle**, roles and users are distinct entities. A user represents a schema owner, whereas a role groups privileges that can be granted to one or more users.
- In **PostgreSQL**, there is no distinction — a **role** can act both as a user and as a group. Roles with the `LOGIN` attribute behave like users, while others function purely as groups.
- Oracle's roles cannot directly own objects (tables, views); PostgreSQL roles can.

User and role creation workflow. In practice, user setup follows a standard administrative pattern.

Example (Oracle)

```
-- 1. Create a user with connection privileges
CREATE USER john IDENTIFIED BY john123;
GRANT CREATE SESSION TO john;

-- 2. Create roles
CREATE ROLE analyst;
CREATE ROLE editor;

-- 3. Grant privileges to roles
GRANT SELECT ON sales TO analyst;
GRANT INSERT, UPDATE ON sales TO editor;

-- 4. Assign roles to user
GRANT analyst, editor TO john;

-- 5. Revoke role or privilege
REVOKE editor FROM john;
```

Example (PostgreSQL)

```
CREATE ROLE alice LOGIN PASSWORD 'pass123';
GRANT analyst TO alice;
```

From Lab: Enterprise_Lambda administrative simulation. In the laboratory, students implemented a full access-control model based on the `Entreprise_Lambda` database, using Oracle's user, role, and privilege management commands.

1. Role-based hierarchy. Different user roles were designed to reflect realistic responsibilities:

- **Chef** – complete administrative privileges: `SELECT`, `INSERT`, `UPDATE`, `DELETE`, and DDL creation rights.
- **Kitchen Assistant** – restricted write permissions on operational tables such as `Mission` and `Stock`.
- **Classic User** – read-only privileges limited to non-sensitive views (e.g., public mission list).
- **Premium User** – full read privileges across all views, but no modification rights.

Example:

```
CREATE ROLE chef;
GRANT ALL PRIVILEGES TO chef;

CREATE ROLE kitchen_assistant;
GRANT SELECT, INSERT, UPDATE ON mission TO kitchen_assistant;

CREATE ROLE classic_user;
GRANT SELECT ON v_missions TO classic_user;

CREATE ROLE premium;
GRANT SELECT ON v_resp, v_missions, v_employee_public TO premium;
```

2. User creation and assignment. New users were created to simulate actual employees and linked to the corresponding roles:

```
CREATE USER sarah IDENTIFIED BY sarah123;
CREATE USER tom IDENTIFIED BY tom123;
CREATE USER laura IDENTIFIED BY laura123;

GRANT chef TO sarah;
GRANT kitchen_assistant TO tom;
GRANT premium TO laura;
```

Each user's privileges were verified via `USER_TAB_PRIVS` and `USER_ROLE_PRIVS` data dictionary views.

3. Restricted user access (employee). To ensure confidentiality, a specific user `employee` was created with access limited to a sanitized view excluding salary and commission data:

```
CREATE USER employee IDENTIFIED BY emp123;
GRANT CREATE SESSION TO employee;
GRANT SELECT ON v_employee_public TO employee;
```

Privilege verification: When the `employee` user attempted:

```
UPDATE v_employee_public
SET hire_date = SYSDATE
WHERE emp_id = 105;
```

Oracle returned an error indicating insufficient privileges, confirming the access restriction.

4. Privilege revocation and escalation testing. Students tested the effects of revoking privileges and improper delegation:

```
-- Remove editing rights
REVOKE UPDATE ON mission FROM kitchen_assistant;

-- Test unauthorized update
UPDATE mission SET city = 'Paris' WHERE mission_id = 102;
-- Result: ORA-01031: insufficient privileges
```

The exercise demonstrated how Oracle enforces privilege hierarchy and prevents privilege escalation across users and roles.

Traps & Keys

- Privileges on **views** are independent of those on base tables — revoking base-table access does **not** automatically revoke view privileges.
- Roles simplify administration but can cause confusion when nested or cyclically granted; always document role dependencies.
- The clauses `WITH ADMIN OPTION` (for roles) and `WITH GRANT OPTION` (for privileges) allow delegation — use with caution to prevent unintended propagation.
- Dropping a user automatically revokes all grants; dependent objects owned by that user must be reassigned or removed beforehand.

Key takeaway

Access control mechanisms define who can do what within a database. Through the Enterprise_Lambda lab, students experienced how **roles**, **views**, and **granular privileges** combine to achieve secure, structured, and auditable data management in multi-user environments.

5 Distributed Databases & NoSQL Framing

5.1 Why Distribution?

Definition and motivation

A **distributed database system** is a collection of logically related data that is physically distributed across multiple sites connected by a network. Each site has its own local database and processing capabilities, but the system aims to appear as a single, unified database to the end user.

Distribution arises naturally from organizational and technical needs for:

- **Locality:** bringing data closer to where it is most frequently accessed or modified (e.g., a regional office storing its local customers or trips).
- **Parallelism:** allowing multiple sites to execute queries or transactions concurrently, thereby improving performance and scalability.
- **Availability and fault tolerance:** ensuring that a system continues to function even if one site fails, by maintaining redundant or replicated data.
- **Autonomy:** enabling regional databases to operate semi-independently while still cooperating within a global framework.

The goal is to optimize access time and system load distribution while maintaining logical consistency across all participating sites.

Locality. Data is often accessed by geographically distributed users. Placing fragments of data near their main consumers reduces communication delays and network load. For instance, European customers should ideally be handled by a European database rather than querying a central server located overseas.

Example

A travel company could maintain:

- a `North` database for trips and accommodations in Northern regions,
- a `South` database for Southern data,
- a `East` database for Eastern destinations.

Local queries remain fast, while global coordination allows consolidated reporting.

Parallelism. Distribution enables **parallel query execution**, dividing tasks across multiple nodes. Large operations such as aggregations or joins can be partitioned, executed simultaneously at each site, and then merged centrally. This reduces overall response time and balances system load.

Example

A global query calculating total tourist bookings per region can be decomposed into regional subqueries (`SUM(booking_count)` in North, South, East) and combined centrally.

Availability and replication basics. To ensure continuous service, distributed systems often maintain **replicated data**. Replication keeps copies of critical or reference tables at multiple sites, allowing local access even if another site becomes unavailable.
Replication can be:

- **Synchronous:** all copies updated simultaneously — high consistency, higher latency.
- **Asynchronous:** updates propagated later — lower latency, potential temporary divergence.

A good distributed design balances replication benefits (read availability) with its cost (update coordination).

From Lab: Enterprise Travel System simulation. In the laboratory, students simulated a distributed travel management system within Oracle, representing different geographical regions and a central coordination schema.

1. Regional schemas. Five Oracle schemas were defined:

- `north_user`, `south_user`, `east_user`: regional databases holding local information.
- `central_user`: coordination layer providing global views and control logic.
- `data_user`: shared schema for reference data common to all sites.

Each regional schema stored its local subset of entities such as `Trips`, `Guides`, and `Accommodations`, while the central schema unified access via database links and views.

2. Demonstrating locality and parallelism. Students executed equivalent queries in two ways:

1. Directly on local regional schemas (e.g., `SELECT * FROM north_user.Trips;`);
2. Through a global view in `central_user` combining all regions (e.g., `SELECT * FROM All_Trips;`);

They observed that local queries returned faster due to data proximity, while global queries provided integrated visibility at the cost of increased communication overhead. This illustrated how distribution improves local responsiveness and parallelism but introduces coordination complexity.

3. Early replication example. Certain **reference tables**, such as `CulturalEvents`, were duplicated across all regional schemas to ensure that local users could access cultural event details even if the central schema was unavailable.

Example replication setup:

```
CREATE TABLE data_user.CulturalEvents (...);
CREATE TABLE north_user.CulturalEvents AS SELECT * FROM data_user.CulturalEvents;
CREATE TABLE south_user.CulturalEvents AS SELECT * FROM data_user.CulturalEvents;
CREATE TABLE east_user.CulturalEvents  AS SELECT * FROM data_user.CulturalEvents;
```

Each site maintained its own copy for faster reads; synchronization (if updates occur) was discussed later in the replication section (5.4).

Traps & Keys

- Distribution improves responsiveness and scalability but increases coordination and maintenance complexity.
- The Oracle lab does **not perform physical distribution**: data remains on the same instance, and "distribution" is simulated via schema isolation and database links.
- Network latency, connection failures, and replication lag are unavoidable trade-offs in real distributed systems.

Key takeaway

Distribution aims to balance **performance**, **availability**, and **consistency**. The Enterprise Travel System lab provided a controlled environment to visualize these principles through region-based schemas, early replication, and global coordination views.

5.2 Transparency Types

Definition

In a distributed database, **transparency** refers to the ability for users and applications to interact with data as if it were stored in a single, centralized system. Transparency hides the complexity of distribution, ensuring that the location, fragmentation, and replication of data remain invisible to end users.

The main goal is to provide **logical unity over physical distribution**: regardless of where data is stored or how it is fragmented, queries and updates behave as if there were only one database.

Types of transparency. Distributed systems aim to support several complementary forms of transparency:

- **Fragmentation transparency:** The user is unaware that a table is divided into fragments across several sites. Queries automatically access the relevant fragments and reassemble results.
- **Replication transparency:** The system hides the existence of multiple copies of the same data. Users see a single logical relation, while updates are propagated to all replicas.
- **Location transparency:** Users need not know the physical location (server, schema, or site) of data. The database system automatically routes queries to the correct node.

Example

A global query such as:

```
SELECT * FROM Trips WHERE Region = 'North';
```

should automatically access the appropriate fragment (e.g., stored in the North site) without the user having to specify which database to contact.

Fragmentation transparency. When a relation is fragmented (horizontally, vertically, or mixed), the system must automatically reconstruct it when queried globally.

Example

If the `Trips` table is divided into three regional fragments:

- `Trips_North`
- `Trips_South`
- `Trips_East`

then a global query should behave as if a single logical table existed.
Reconstruction:

```
CREATE VIEW All_Trips AS
SELECT * FROM north_user.Trips_North
UNION ALL
SELECT * FROM south_user.Trips_South
UNION ALL
SELECT * FROM east_user.Trips_East;
```

Replication transparency. When identical data is replicated across several sites, users should see one unified version. Replication transparency ensures that a query returning, for example, all cultural events, does not duplicate results from different replicas.

Example

```
CREATE VIEW All_CulturalEvents AS
SELECT DISTINCT * FROM north_user.CulturalEvents
UNION ALL
SELECT DISTINCT * FROM south_user.CulturalEvents
UNION ALL
SELECT DISTINCT * FROM east_user.CulturalEvents;
```

The system ensures that replicas remain logically consistent, even if they are physically separate.

Location transparency. Location transparency abstracts the physical location of data objects. A user or application references a logical name, and the system resolves the appropriate site through **database links** or similar routing mechanisms.

Example (Oracle)

```
CREATE DATABASE LINK north_link
  CONNECT TO north_user IDENTIFIED BY pass
  USING 'north_db';

SELECT * FROM Trips@north_link;
```

Here, the suffix `@north_link` identifies a remote site. By hiding such details behind global views, location transparency becomes complete — users query `All_Trips` without knowing which schema or server the data resides in.

From Lab: Implementing transparency in the Enterprise_Lambda system. The laboratory session extended the distributed travel database (introduced in 5.1) to achieve full logical transparency across regions. Students practiced creating database links and global integration views to unify all fragments.

1. Database links between regional schemas. Each region was connected to the central schema (`central_user`) via authenticated database links:

```
CREATE DATABASE LINK north_link
  CONNECT TO north_user IDENTIFIED BY north_pwd
  USING 'north_db';
```

```
CREATE DATABASE LINK south_link
  CONNECT TO south_user IDENTIFIED BY south_pwd
  USING 'south_db';

CREATE DATABASE LINK east_link
  CONNECT TO east_user IDENTIFIED BY east_pwd
  USING 'east_db';
```

These links allowed the central user to query remote tables as if they were local.

2. Global views for unified access. Students then created global views combining data from each region to ensure full fragmentation and location transparency.
Examples:

```
CREATE VIEW All_Guides AS
SELECT * FROM Guides@north_link
UNION ALL
SELECT * FROM Guides@south_link
UNION ALL
SELECT * FROM Guides@east_link;

CREATE VIEW All_Accommodations AS
SELECT * FROM Accommodations@north_link
UNION ALL
SELECT * FROM Accommodations@south_link
UNION ALL
SELECT * FROM Accommodations@east_link;
```

Similar logic was applied for `All_CulturalEvents`, `All_Tourists`, and `All_Bookings`, yielding a unified schema at the central site.

3. Transparency validation. Students tested the transparency by executing identical SQL queries:

```
SELECT COUNT(*) FROM All_Trips;
SELECT DISTINCT Region FROM All_Guides;
```

Both queries returned integrated results without any manual reference to regional schemas. From a user's perspective, the distributed system behaved like a centralized database.

Traps & Keys

- In Oracle, transparency is purely **logical**: data is not physically distributed; rather, database links and views simulate the illusion of distribution.
- Database links must be correctly authenticated and maintained; incorrect credentials or dropped links cause query failures.
- Circular links (where two databases link to each other) may create dependency loops and slow query resolution.
- Latency can occur when global views query multiple sites sequentially — particularly when one site responds slowly or is offline.

Key takeaway

Transparency is the cornerstone of distributed databases: it allows seamless access to partitioned and replicated data. In the lab, Oracle's database links and global integration views effectively demonstrated how logical transparency can approximate physical distribution in practice.

5.3 Fragmentation

Definition

Fragmentation is the process of dividing a global relation (table) into smaller, more manageable pieces — called **fragments** — distributed across multiple sites. Each fragment represents a logical subset of the original relation, designed to optimize locality of access, performance, and autonomy, while maintaining the illusion of a unified database.

The fragments collectively represent the full relation and must satisfy:

- **Completeness:** all data of the global table must appear in at least one fragment;
- **Disjointness:** no data should be duplicated between fragments (for horizontal fragmentation);
- **Reconstructability:** it must be possible to reconstruct the global table from its fragments using relational operators.

Types of fragmentation. Three main fragmentation strategies are used in distributed database design.

1. Horizontal fragmentation. The table is split by rows according to a selection predicate. Each fragment contains a subset of tuples satisfying a regional or logical condition.

Example

```
CREATE TABLE Trips_North AS
SELECT * FROM Trips WHERE Region = 'North';

CREATE TABLE Trips_South AS
SELECT * FROM Trips WHERE Region = 'South';

CREATE TABLE Trips_East AS
SELECT * FROM Trips WHERE Region = 'East';
```

Each fragment stores local data for its region. The original relation can be reconstructed via:

```
CREATE VIEW All_Trips AS
SELECT * FROM Trips_North
UNION ALL
SELECT * FROM Trips_South
UNION ALL
SELECT * FROM Trips_East;
```

Horizontal fragmentation is the most common approach for geographic or organizational partitioning.

2. Vertical fragmentation. The table is split by columns, often separating sensitive or rarely used attributes from frequently accessed ones. Each fragment retains the same primary key to allow reconstruction by joining on the key.

Example

```
CREATE TABLE Tourist_Basic AS
SELECT Tourist_ID, Name, Country, Gender
FROM Tourist;

CREATE TABLE Tourist_Contact AS
SELECT Tourist_ID, Email, Phone, Address
FROM Tourist;
```

Reconstruction is done through a natural join on the common key:

```
CREATE VIEW All_Tourists AS
SELECT *
FROM Tourist_Basic tb
JOIN Tourist_Contact tc
ON tb.Tourist_ID = tc.Tourist_ID;
```

Vertical fragmentation is typically used to separate personal or confidential information from public data, improving data privacy and modular access.

3. Mixed fragmentation. Mixed (or hybrid) fragmentation combines both horizontal and vertical strategies. It can first fragment a table by region (horizontal) and then by attribute (vertical), or vice versa.

Example

```
-- Regional information stored locally
CREATE TABLE Booking_Info_North AS
SELECT Booking_ID, Tourist_ID, Trip_ID, Date_Booking
FROM Booking
WHERE Region = 'North';

-- Financial data stored centrally
CREATE TABLE Booking_Amount AS
SELECT Booking_ID, Total_Amount, Currency
FROM Booking;
```

The global table can be reconstructed by joining on the common key (`Booking_ID`) and uniting all regional components:

```
CREATE VIEW All_Bookings AS
SELECT bi.Booking_ID, bi.Tourist_ID, bi.Trip_ID, bi.Date_Booking,
       ba.Total_Amount, ba.Currency
FROM Booking_Amount ba
JOIN (
    SELECT * FROM Booking_Info_North
    UNION ALL
    SELECT * FROM Booking_Info_South
    UNION ALL
    SELECT * FROM Booking_Info_East
) bi
ON ba.Booking_ID = bi.Booking_ID;
```

This design combines locality (regional access) with centralized financial control.

From Lab: Implementation in the Enterprise_Lambda distributed schema. In the laboratory project, students implemented all three fragmentation types on the distributed travel system, using Oracle schemas to represent physical regions.

1. Horizontal fragmentation. Horizontal fragments were created for several main entities:

- `Trips_North`, `Trips_South`, `Trips_East`;
- `Guides_North`, `Guides_South`, `Guides_East`;
- `Accommodations_North`, `Accommodations_South`, `Accommodations_East`;
- `CulturalEvents_North`, `CulturalEvents_South`, `CulturalEvents_East`.

Each regional schema (`north_user`, `south_user`, `east_user`) stored its corresponding fragment. The global integration view in `central_user` unified all of them transparently:

```
CREATE VIEW All_Trips AS
SELECT * FROM Trips@north_link
UNION ALL
SELECT * FROM Trips@south_link
UNION ALL
SELECT * FROM Trips@east_link;
```

2. Vertical fragmentation. The `Tourist` table was divided into two fragments based on data sensitivity:

- `Tourist_Basic` containing public identification data;
- `Tourist_Contact` storing private contact information.

Example:

```
CREATE TABLE Tourist_Basic AS
SELECT Tourist_ID, Name, Country, Gender
FROM data_user.Tourist;

CREATE TABLE Tourist_Contact AS
SELECT Tourist_ID, Email, Phone, Address
FROM data_user.Tourist;
```

Reconstruction view in `central_user`:

```
CREATE VIEW All_Tourists AS
SELECT *
FROM Tourist_Basic tb
JOIN Tourist_Contact tc
ON tb.Tourist_ID = tc.Tourist_ID;
```

This approach allowed controlled exposure of sensitive attributes through role-based privileges.

3. Mixed fragmentation. The `Booking` relation was fragmented both by region and by attribute:

- Regional fragments stored by each user schema (`north_user`, `south_user`, `east_user`) under `Booking_Info`;
- Financial data centralized in `data_user.Booking_Amount`.

Example:

```
CREATE TABLE north_user.Booking_Info AS
SELECT Booking_ID, Tourist_ID, Trip_ID, Date_Booking
FROM Booking WHERE Region = 'North';

CREATE TABLE data_user.Booking_Amount AS
SELECT Booking_ID, Total_Amount, Currency
FROM Booking;
```

Recomposition was handled by a unified global view:

```
CREATE VIEW All_Bookings AS
SELECT bi.Booking_ID, bi.Tourist_ID, bi.Trip_ID,
       bi.Date_Booking, ba.Total_Amount, ba.Currency
FROM Booking_Amount ba
JOIN (
    SELECT * FROM Booking_Info@north_link
    UNION ALL
    SELECT * FROM Booking_Info@south_link
    UNION ALL
    SELECT * FROM Booking_Info@east_link
) bi
ON ba.Booking_ID = bi.Booking_ID;
```

This structure provided both locality for bookings and centralization for accounting.

Traps & Keys

- Each horizontal fragment must be **disjoint** (no overlapping tuples) and collectively **complete** to guarantee correct reconstruction of the global relation.
- Joins between vertically fragmented tables must use a shared primary key (e.g., `Tourist_ID`) to avoid duplication or data loss.
- Oracle does not enforce fragmentation rules automatically — validation must be ensured manually or through PL/SQL triggers.
- Excessive fragmentation may harm performance by requiring complex recomposition joins and cross-site queries.

Key takeaway

Fragmentation allows logical data partitioning for improved performance, privacy, and locality. In the Enterprise_Lambda lab, horizontal, vertical, and mixed fragmentations were implemented and unified through global views, achieving full transparency at the logical level while preserving modular control over regional and sensitive data.

5.4 Replication

Definition

Replication is the process of maintaining copies of the same data at multiple sites in a distributed database system. Its goal is to improve data **availability**, enhance **read performance**, and ensure **fault tolerance** in case of site failure.

When one copy of a dataset becomes temporarily inaccessible, other replicas can still respond to queries, allowing the system to continue operating. Replication, however, introduces the challenge of keeping copies consistent when updates occur.

Types of replication. Replication strategies vary according to synchronization mode and control model.

1. Synchronous replication. All replicas are updated simultaneously within the same transaction. The system guarantees that all copies remain consistent at all times, but at the cost of slower response times and reduced availability if any site is offline.
Characteristics:

- Updates are propagated immediately.
- Ensures strong consistency.

- Sensitive to network latency and site failures.

Example

```
UPDATE Trips_North SET Price = 1200 WHERE Trip_ID = 101;
UPDATE Trips_North_Replica SET Price = 1200 WHERE Trip_ID = 101;
COMMIT;
```

Both copies are modified within the same transaction to maintain perfect consistency.

2. Asynchronous replication. Updates are propagated after the original transaction commits. Replicas may temporarily diverge, but the system achieves eventual consistency once synchronization completes.

Characteristics:

- Non-blocking and higher performance.
- Risk of short-term inconsistency between replicas.
- Ideal for read-intensive and geographically distributed systems.

Example

```
UPDATE Trips_North SET Price = 1200 WHERE Trip_ID = 101;
COMMIT;
-- Later, replication process synchronizes the replica
UPDATE Trips_North_Replica
SET Price = 1200 WHERE Trip_ID = 101;
```

Primary-copy model. In many replication architectures, one site is designated as the **primary copy** responsible for updates, while other sites maintain read-only or delayed replicas.

Rules:

- All writes are directed to the primary.
- Replicas are refreshed either synchronously or asynchronously.
- If the primary fails, a new primary may be elected or manually reassigned.

This model simplifies consistency management at the cost of potentially reduced fault tolerance during primary downtime.

Conflict resolution strategies. When multiple sites can update replicated data independently, conflicts may occur. Several strategies exist to reconcile differences:

- **Timestamp-based:** the most recent update (based on `LastUpdated` column) prevails — known as the "last writer wins" rule.
- **Quorum-based:** updates require acknowledgment from a majority of replicas before being committed.
- **Application-specific:** custom reconciliation logic based on business rules.

Conflict resolution must be deterministic to guarantee convergent replicas after reconciliation.

From Lab: Simulated replication in the Enterprise_Lambda system. In the laboratory, students implemented a simplified form of replication using Oracle schemas and views to simulate redundancy and fault tolerance.

1. Creating a replica table. A replica was created for the northern regional fragment:

```
CREATE TABLE Trips_North_Replica AS
SELECT * FROM Trips_North;
```

This secondary copy represented a local backup of the northern trips dataset, allowing continued operation in case of temporary inaccessibility of the primary fragment.

2. Building a fault-tolerant query procedure. A PL/SQL procedure was written to automatically fallback to the replica if the main table became unreachable due to a DB link or schema issue.
Example:

```
CREATE OR REPLACE PROCEDURE Query_North_Trips AS
BEGIN
    SELECT * FROM Trips_North;
EXCEPTION
    WHEN OTHERS THEN
        SELECT * FROM Trips_North_Replica;
END;
/
```

This logic ensures that queries always return results, even when the primary site is unavailable.
Students simulated failures by disabling access to the original table, confirming that the procedure correctly redirected queries to the replica.

3. Conflict resolution and synchronization. To handle update divergence, a timestamp-based reconciliation mechanism was adopted using a `LastUpdated` column.
Conflict resolution rule: "last writer wins."
Example:

```
CREATE VIEW Reconciled_Trips AS
SELECT * FROM Trips_North
UNION ALL
SELECT * FROM Trips_North_Replica
WHERE LastUpdated >
      (SELECT LastUpdated FROM Trips_North
       WHERE Trips_North.Trip_ID = Trips_North_Replica.Trip_ID);
```

This approach ensures that, for each conflicting row, the most recently updated version is preserved in the global view.

4. Failover and recovery discussion. During the lab, students simulated a temporary disconnection of the northern schema by disabling the related DB link. Queries automatically redirected to the replica via the PL/SQL fallback mechanism.
After restoring connectivity, synchronization was re-established manually by comparing `LastUpdated` timestamps and reinserting newer rows into the primary table. This exercise illustrated the importance of automated reconciliation in fault-tolerant distributed systems.

Traps & Keys

- The Oracle setup used in this lab provides only **logical replication** — there is no built-in physical synchronization mechanism. Replication is simulated through copies, triggers, and timestamps.
- Achieving true consistency across replicas requires distributed transaction protocols such as the **Two-Phase Commit (2PC)**.
- Asynchronous replication improves performance but can lead to temporary divergence across sites.
- Synchronous replication ensures consistency but reduces system availability when a site is slow or offline.

Key takeaway

Replication increases availability and fault tolerance but introduces complexity in synchronization and conflict management. Through the Enterprise_Lambda lab, students implemented a simplified logical replication mechanism demonstrating failover behavior, timestamp-based reconciliation, and the trade-off between consistency and

performance.

5.5 Naming & Allocation

Definition

In a distributed database, **naming** and **allocation** determine how data fragments are identified, located, and accessed across different sites. These mechanisms ensure that users can reference data objects consistently, regardless of their physical storage location.

Naming provides a unified logical identifier for each distributed object, while allocation defines where these objects are physically stored to optimize access performance and reliability.

Naming mechanisms. Distributed systems employ several strategies for identifying data fragments.

- **Centralized catalog:** A single naming server maintains all object-to-site mappings. Each client consults the central catalog to locate data before issuing queries. This approach simplifies management but creates a potential single point of failure.
- **Alias-based naming:** Each site defines logical aliases (or database links) that reference remote schemas. Applications use these aliases transparently without knowing physical details.
- **Distributed directory:** A more advanced system maintains replicated naming catalogs across multiple sites, ensuring both availability and scalability at the cost of more complex synchronization.

The chosen strategy directly affects system reliability, query routing latency, and administrative complexity.

Allocation principles. **Allocation** decides where each fragment or replica of a relation should reside. The decision is typically guided by several criteria:

- **Locality of access:** Place data near the users or applications that access it most frequently.
- **Access frequency:** Highly queried data should reside on high-performance or local nodes.
- **Update patterns:** Frequently updated fragments should minimize network propagation cost.

- **Storage and maintenance cost:** Consider hardware capacity, data replication overhead, and administrative effort.

Allocation can be:

- **Static:** fixed at design time, rarely modified;
- **Dynamic:** adjusted at runtime based on load, locality, or cost metrics.

Trade-offs.

- A **centralized catalog** offers simple lookup and management but becomes a bottleneck and single point of failure.
- A **distributed naming system** provides robustness and load distribution but complicates consistency and synchronization.
- Alias-based references (e.g., Oracle database links) provide flexibility but depend on secure and consistent credential management.

The optimal solution balances transparency, reliability, and performance according to organizational needs.

From Lab: Implementing naming and allocation in the Enterprise_Lambda simulation. In the laboratory, students explored how logical naming and physical allocation interact within the Oracle-based distributed travel system. They simulated both centralized and alias-based approaches to achieve unified access across regions.

1. Creating aliases through database links. Each regional schema (`north_user`, `south_user`, `east_user`) was referenced in the central schema (`central_user`) through database links acting as **naming aliases**.
Example:

```
CREATE DATABASE LINK north_link
  CONNECT TO north_user IDENTIFIED BY pass
  USING 'north_db';

CREATE DATABASE LINK south_link
  CONNECT TO south_user IDENTIFIED BY pass
  USING 'south_db';

CREATE DATABASE LINK east_link
  CONNECT TO east_user IDENTIFIED BY pass
  USING 'east_db';
```

Each link served as a logical name for a remote site, allowing transparent queries such as:

```
SELECT * FROM Trips@north_link;
```

without requiring the user to know the actual database host or schema details.

2. Logical abstraction through global views. Global views defined in `central_user` used these database links to provide a unified logical representation of distributed fragments. **Example:**

```
CREATE VIEW All_Guides AS
SELECT * FROM Guides@north_link
UNION ALL
SELECT * FROM Guides@south_link
UNION ALL
SELECT * FROM Guides@east_link;
```

Thus, the view `All_Guides` acted as a logical alias aggregating distributed data into a single interface. This illustrates how naming (via links) and allocation (via site assignment) jointly achieve full transparency.

3. Central naming catalog. To track fragment locations, students implemented a manual mapping table in the central schema — effectively simulating a **name server**. **Example:**

```
CREATE TABLE Data_Location (
    ObjectName VARCHAR2(50),
    SiteName   VARCHAR2(50)
);

INSERT INTO Data_Location VALUES ('Trips_North', 'north_db');
INSERT INTO Data_Location VALUES ('Trips_South', 'south_db');
INSERT INTO Data_Location VALUES ('Trips_East',  'east_db');
INSERT INTO Data_Location VALUES ('Tourist_Basic', 'data_db');
```

This catalog allowed diagnostic queries such as:

```
SELECT SiteName FROM Data_Location WHERE ObjectName = 'Trips_South';
```

Such catalogs are useful for administrators to verify allocation consistency and detect misconfigurations in distributed environments.

4. Physical allocation considerations. Students analyzed allocation strategies based on dataset characteristics:

- Large, region-specific data (e.g., `Trips`, `Accommodations`) stored at regional sites for locality.
- Shared reference data (e.g., `CulturalEvents`) replicated at all sites for availability.
- Sensitive or financial data (e.g., `Booking_Amount`) centralized in `data_user` for consistency and security.

This hybrid allocation maximized query efficiency while preserving control over critical information.

Traps & Keys

- Alias mismatches or incorrect database link definitions often cause "`ORA-02085: database link name does not match`" or "`invalid link`" errors. Consistent naming conventions across schemas are essential.
- Centralized catalogs simplify lookup but represent a single point of failure — redundancy or replication of metadata is recommended.
- Oracle database links store authentication credentials in system metadata. They must be managed carefully to avoid security vulnerabilities (use encrypted wallet or secure password management).

Key takeaway

Naming and allocation provide the foundation of distributed database transparency. Through database links, logical views, and a central mapping catalog, the Enterprise_Lambda lab demonstrated how a logically unified database can be maintained across physically distinct schemas while balancing locality, accessibility, and security.

5.6 Distributed Query Processing

Definition

Distributed Query Processing (DQP) refers to the set of techniques used to execute queries efficiently across multiple, physically separated sites in a distributed database system. Its goal is to ensure that the user perceives a single, unified database while minimizing communication cost, latency, and data transfer overhead.

In a distributed environment, query execution must consider both the *logical correctness* of results and the *physical efficiency* of data movement.

Processing phases. A distributed query is processed in several distinct stages that correspond to the system's internal query execution workflow.

1. **Query decomposition:** The SQL query is parsed and rewritten into an equivalent relational algebra expression. This step validates the query's semantics and identifies candidate subqueries or predicates.
2. **Data localization:** Each relation or fragment referenced in the query is mapped to its physical site using the system catalog or naming service. Localization transforms the logical plan into a distributed plan.
3. **Optimization:** The optimizer determines the most efficient strategy to execute the query. It selects between local and remote execution, join order, and data movement patterns.
4. **Local execution:** Subqueries are sent to remote sites (if needed) and executed locally on the appropriate fragments.
5. **Result assembly:** Partial results are transmitted to the initiating site, combined (e.g., via joins, unions, or aggregations), and presented as a unified output.

Each stage affects overall latency and resource utilization. In practice, the optimizer must balance computation cost against communication cost — the latter being dominant in distributed contexts.

Execution strategies. Two principal strategies are used to coordinate computation and data movement.

1. Data shipping. Data from remote sites is transferred to the local site, where the query or join operation is executed. This approach is simple but potentially inefficient for large datasets.

Example

```
SELECT * FROM Trips@north_link
WHERE Region = 'North';
-- Data shipped from the North schema to the central user.
```

Advantages:

- Requires minimal remote computation capability.
- Simple to implement for small datasets or infrequent queries.

Disadvantages:

- High data transfer cost.
- Increased latency for large or complex results.

2. Query shipping. Instead of transferring data, the central site sends the query itself to the remote site, where the computation is executed locally. Only the final results are transmitted back, drastically reducing communication cost.

Example

```
SELECT COUNT(*) FROM (SELECT * FROM Trips@north_link
                      WHERE Region = 'North');
```

Here, the aggregation runs directly on the remote node (`north_user`), and only a single number (the count) is returned.

Advantages:

- Reduces network load.
- Exploits remote system processing power.

Disadvantages:

- Requires query execution capability on all remote sites.
- Harder to coordinate across heterogeneous systems.

Communication optimization techniques. One of the key challenges in DQP is minimizing data transfer between sites. A classic approach is the use of **semijoins**.

Semijoins. A semijoin transmits only the join attribute values from one site to another to filter remote data before performing the full join.

Example

1. Central site sends the set of `TripID`s it needs to the remote site.
2. Remote site filters its rows and sends back only the matching tuples.

Benefit: Reduces the amount of data exchanged, especially when the selection is selective.

Parallelism in distributed execution. Distributed databases can exploit parallelism in two dimensions:

- **Inter-query parallelism:** multiple independent queries processed concurrently across sites.
- **Intra-query parallelism:** sub-operations of a single query (e.g., joins or aggregations) executed in parallel across fragments.

This parallel execution can greatly reduce response time, provided the network and remote nodes are adequately provisioned. However, it also introduces synchronization overhead and complex failure handling.

From Lab: Distributed query execution in Oracle. In the Enterprise_Lambda simulation, students executed queries across **global views** combining several DB links to observe distributed execution behavior.

1. Global queries over multiple links. The `All_Trips` and `All_Bookings` views were used to execute queries spanning three regional schemas:

```
SELECT t.Name, b.Amount, trips.Region
FROM All_Tourists t
JOIN All_Bookings b ON t.TouristID = b.TouristID
JOIN All_Trips trips ON b.TripID = trips.TripID;
```

Students analyzed the execution plan in SQL Developer, observing that Oracle sequentially fetched results from each remote link.

2. Mini-performance experiment. Students compared the performance of:

- Querying each regional site separately (direct DB link queries);
- Querying the centralized global view (`All_Trips`) aggregating all fragments.

Observation: The centralized query was more convenient but slower, as Oracle fetched all regional data even when only one region matched the filter.

3. Execution behavior and delegation. Oracle's distributed engine can push down certain predicates or aggregations to remote sites (query shipping). However, it does not perform cost-based distributed optimization across links: all fragments in a UNION view may still be scanned.

Traps & Keys

- **DB links are not cost-aware:** Oracle executes remote subqueries independently without optimizing global join order.
- **Parallel execution** must be explicitly enabled using session parameters (e.g., `ALTER SESSION ENABLE PARALLEL DML`); otherwise, distributed fetches occur sequentially.
- **Data transfer dominates cost:** minimizing network I/O is often more critical than reducing local computation.

- Semijoins and query shipping are powerful but require query rewriting and careful site capability analysis.

Key takeaway

Distributed query processing transforms global queries into localized subqueries, balancing computation and communication cost. In the lab, students observed how Oracle executes distributed queries through DB links and how performance depends heavily on data shipping strategy and query decomposition efficiency.

5.7 Distributed Transactions

Definition

A **distributed transaction** is a single logical unit of work that spans multiple databases or sites. Each participating site must agree on whether to commit or roll back the transaction, ensuring overall consistency across the distributed system.

Such coordination requires specialized components and protocols to maintain **atomicity** and **durability** despite potential failures or network partitions.

Roles in distributed transaction management. Distributed commit coordination typically involves two key roles:

- **Transaction Manager (TM):** Controls the execution flow of the transaction at the initiating site. It coordinates communication with remote sites, tracks transaction state, and handles recovery.
- **Transaction Coordinator (TC):** Supervises the commit protocol. It ensures that all participating sites either commit or roll back consistently. In many architectures, the TM and TC are combined at the initiating site.

Each participating database is called a **resource manager (RM)**, responsible for locally preparing and committing data updates.

The Two-Phase Commit (2PC) protocol. The **2PC protocol** is the standard algorithm used to ensure atomic commitment across distributed sites.
Phase 1 – Prepare phase:

1. The coordinator sends a `PREPARE TO COMMIT` message to all participants.
2. Each participant performs local checks, locks necessary resources, and writes a "prepared" log entry.

3. Each participant replies either `VOTE COMMIT` or `VOTE ABORT`.

Phase 2 – Commit phase:

1. If all participants vote `COMMIT`, the coordinator sends a global `COMMIT` message.
2. If any participant votes `ABORT`, the coordinator sends a global `ROLLBACK`.
3. Each participant executes the corresponding operation and acknowledges completion.

Guarantee: All participants reach the same final decision (commit or abort). However, 2PC can block indefinitely if the coordinator fails during the commit phase.

The Three-Phase Commit (3PC) protocol. To mitigate the blocking problem of 2PC, the **3PC protocol** introduces an additional intermediate phase, ensuring that no site remains in uncertainty.
Phases:

1. **CanCommit:** The coordinator asks whether participants are ready to commit.
2. **PreCommit:** If all agree, the coordinator sends a `PRECOMMIT` message, and participants record the intention to commit.
3. **DoCommit:** Once all acknowledgments are received, the coordinator issues a final `COMMIT`.

Limitation: Although 3PC avoids blocking, it assumes bounded communication delays and reliable clocks — unrealistic in most real-world distributed systems.

Consensus algorithms overview. Modern distributed systems such as `Google Spanner`, `CockroachDB`, and `etcd` often replace commit protocols with consensus algorithms like **Paxos** or **Raft**.

- These protocols replicate a log of agreed-upon operations across all nodes.
- Consensus guarantees that all healthy nodes eventually agree on the same sequence of updates, ensuring consistency even under network partitions.
- However, they trade off performance and availability (CAP theorem) for stronger fault tolerance and safety.

Deadlock detection in distributed systems. Deadlocks can occur when transactions across sites wait for resources locked by each other. Detection can be organized in three main architectures:

- **Centralized detection:** A single global detector gathers all local wait-for graphs and checks for cycles.
- **Hierarchical detection:** Local detectors handle site-level cycles and report unresolved dependencies to higher levels.
- **Timeout-based detection:** Transactions abort automatically after exceeding a predefined waiting time — simple but potentially wasteful.

Distributed deadlock detection requires coordination and consistent global state snapshots, which can be expensive to maintain.

Failure modes and recovery. Failures in distributed transactions can occur due to:

- **Coordinator failure:** leaves participants uncertain whether to commit or rollback.
- **Participant failure:** causes unacknowledged votes or missed final commits.
- **Network partition:** isolates sites, risking inconsistent local decisions.

Recovery mechanisms include:

- Transaction logs for replaying or undoing incomplete actions.
- Coordinator election (in consensus-based systems).
- Compensation logic to manually revert partial commits when atomicity is lost.

From Lab: Distributed PL/SQL procedure simulation. In the Oracle-based Enterprise_Lambda simulation, students implemented and analyzed a distributed PL/SQL procedure operating across multiple schemas through database links.

1. Multi-site PL/SQL procedure. The procedure `Generate_Tourist_Itinerary(p_TouristID)` reads from global views (aggregating distributed data) and performs updates on regional fragments.
Simplified structure:

```
CREATE OR REPLACE PROCEDURE Generate_Tourist_Itinerary(p_TouristID NUMBER) AS
BEGIN
    INSERT INTO Trips_North@north_link (...)
        SELECT ... FROM All_Trips WHERE Region = 'North' AND TouristID = p_Touris
```

```
    INSERT INTO Trips_South@south_link (...)
        SELECT ... FROM All_Trips WHERE Region = 'South' AND TouristID = p_Touris

    COMMIT;  -- Simulated multi-site commit
END;
/
```

This design illustrates how distributed transactions conceptually span multiple sites, though each linked site commits independently.

2. Simulating two-phase commit. Students manually replicated the 2PC logic by:

1. Opening multiple sessions connected to each schema.
2. Executing updates on all fragments.
3. Issuing a coordinated `COMMIT` only when all sub-operations succeeded.

This demonstrated the principle of atomic commitment — all sites must agree before finalizing the transaction.

3. Handling partial failures. To simulate a partial commit scenario, one site was disconnected mid-operation. Students observed that Oracle's linked-session commits propagate per site — meaning successful sites committed their data even if others failed. They discussed the need for manual **compensation transactions** (rollback scripts) to restore consistency, reflecting the lack of automatic atomic coordination through DB links.

4. Failure analysis. Students tested Oracle's behavior when a DB link failed during a commit. The initiating transaction raised a communication exception, but previously completed commits on remote sites were not reverted. This emphasized the non-atomic nature of distributed commits in Oracle without a dedicated transaction coordinator.

Traps & Keys

- Oracle DB links do **not** implement true distributed atomic commit — each remote site commits independently.
- If a sub-commit fails, the system cannot roll back other sites automatically; manual compensation is required.
- 2PC guarantees atomicity but can block indefinitely if the coordinator crashes before decision propagation.
- Perfect atomicity is impossible under network partition (CAP theorem): systems must choose between availability and consistency.

- Consensus protocols (Paxos, Raft) offer stronger safety but with higher latency and complexity.

Key takeaway

Distributed transactions are essential for maintaining global consistency but inherently complex. Through the lab, students visualized these challenges by simulating multi-site commits, analyzing partial failure behavior, and understanding why real-world systems often rely on consensus-based replication rather than strict 2PC enforcement.

5.8 Architectures

Definition

The architecture of a distributed or parallel database determines how data, computation, and control are organized across multiple nodes. It influences performance, scalability, fault tolerance, and the degree of transparency offered to users. Architectures can be analyzed both in terms of **physical topology** (how resources are shared) and **logical organization** (how systems cooperate).

Parallel database topologies. Parallel databases distribute query processing tasks over multiple processors or nodes to increase throughput. Four principal hardware and data-sharing models exist:

- **Shared memory:** All processors access a single, centralized memory and disk. Communication occurs via shared variables. This model simplifies coordination but does not scale well beyond a few processors due to contention.

  Example: SMP (Symmetric Multiprocessing) systems.

- **Shared disk:** Each processor has its own memory but shares access to a common disk subsystem. Coordination is handled through disk-level locks and cache consistency mechanisms.

  Example: Oracle RAC (Real Application Clusters).

- **Shared nothing:** Each node owns its memory, CPU, and disk; nodes communicate only via messages. This model scales efficiently and forms the foundation of most modern distributed databases.

  Example: MongoDB, Google Spanner, CockroachDB.

- **Hybrid:** Combines aspects of shared-disk and shared-nothing systems, often for specialized workloads or cloud-based clusters.

Example: hybrid clusters with local disks and network-attached shared storage.

Each model trades coordination complexity for performance scalability. Shared-nothing systems dominate large-scale deployments because they eliminate central bottlenecks and enable high availability.

Logical architectures. Beyond physical topology, distributed databases can be classified logically based on how components interact.

- **Client/Server:** Clients issue queries, while servers store and process data. Oracle's DB link-based federation model follows this paradigm: the central schema acts as the client, and regional schemas as servers.
- **Peer-to-peer:** Each node is both a client and a server, capable of initiating queries and storing data. Peer-to-peer systems enable higher autonomy but require decentralized coordination.
- **Middleware-based:** A middleware layer provides a unified interface over multiple, possibly heterogeneous databases. It ensures transparency and query routing between systems with different data models or vendors.

Homogeneous vs heterogeneous systems. Homogeneous distributed databases consist of nodes running the same DBMS and schema definitions. They simplify management and allow consistent query language and optimization strategies.
Example: Oracle-to-Oracle distributed setup using DB links.
Heterogeneous distributed databases, on the other hand, integrate multiple DBMS (e.g., Oracle, MongoDB, PostgreSQL). They require middleware or adapters to reconcile query syntax, data types, and transaction semantics.
Example: A data warehouse federating Oracle relational data and MongoDB document collections.

From Lab: Comparative architectural study.

1. Oracle simulation — homogeneous, centralized control. In the **Enterprise_Lambda** lab, Oracle simulated distribution through multiple schemas (`north_user`, `south_user`, `east_user`) coordinated by a central schema (`central_user`). Each schema resided on the same physical instance, connected via database links.

- This setup represents a **homogeneous, logically distributed** system.
- Distribution is simulated: there is only one actual database engine, but logical transparency is achieved via global views.
- The central schema acts as a middleware layer, querying remote links as if they were autonomous databases.

2. MongoDB cluster — shared-nothing architecture. Students then analyzed a real distributed system using a **MongoDB sharded cluster**, which physically separates data across independent nodes.
Cluster setup (via Docker Compose):

- One **Config Server Replica Set (CSRS)** — stores cluster metadata.
- Three **Shard Replica Sets** (`shard1-rs`, `shard2-rs`, `shard3-rs`) — each holding part of the dataset.
- One **Router** (`mongos`) — routes client queries to the correct shard(s).

Initialization commands:

```
sh.enableSharding("ecommerce");
sh.shardCollection("ecommerce.users", { region: 1 });
```

This enables horizontal fragmentation by region, distributing documents across shards according to their `region` field.

3. Locality control via tag ranges. To align with the Oracle regional schema logic, zones were defined:

```
sh.addShardTag("shard1-rs", "Asia");
sh.addShardTag("shard2-rs", "Europe");
sh.addShardTag("shard3-rs", "America");
sh.addTagRange("ecommerce.users",
               { region: "Asia" }, { region: "Europe" }, "Asia");
```

This ensured that documents belonging to specific regions remained co-located on the corresponding shard, improving locality and reducing scatter-gather query behavior.

4. Fault-tolerance drill. Students simulated failure by stopping one shard container and observing MongoDB's routing behavior. Queries targeting unaffected shards continued successfully, while those needing unavailable data returned partial results or errors. After restarting the shard, the router automatically rebalanced data according to the defined shard key and tag ranges.
This exercise highlighted **automatic failover** and **self-healing** behavior — features absent from Oracle's logical federation model.

5. Homework extension. Students replicated Oracle's `North/South/East Trips` scenario within MongoDB by creating a `culturalTrips` collection. Each trip document included a `region` attribute used as the shard key.
They then analyzed physical data placement with:

```
db.culturalTrips.getShardDistribution();
```

and compared query plans using:

```
db.culturalTrips.find({ region: "North" }).explain("executionStats");
```

Students observed how the router performed **targeted queries** when the shard key was specified, and **scatter-gather queries** when it was not — a clear parallel to Oracle's global view performance analysis.

Traps & Keys

- Oracle's so-called "distributed" setup is an **emulation**: data remains centralized; only logical transparency is achieved through DB links.
- MongoDB's **shared-nothing architecture** provides genuine distribution — each shard has independent storage, logs, and replication.
- Performance and fault tolerance in MongoDB depend critically on the **shard key design**: poor keys cause scatter-gather queries that hit all shards.
- Small datasets may not trigger automatic chunk splitting, leading to unrepresentative balancing behavior in lab simulations.
- Oracle favors strong consistency with synchronous control; MongoDB favors scalability and availability with eventual consistency.

Key takeaway

Different architectures serve distinct goals: Oracle emphasizes logical transparency and integrity in homogeneous environments, while MongoDB exemplifies physical distribution, parallelism, and fault-tolerant shared-nothing design. Together, they illustrate the continuum from centralized federation to fully decentralized distributed databases.