

Rdbms1



RDBMS Notes

Made By : Kashif Sayyad

Module I—Introduction, Terminologies & Relational Model

◆ Concept of RDBMS

A **Relational Database Management System (RDBMS)** is a type of database management system that stores data in a structured format using rows and columns — organized into **tables** (also called relations). The relational model was proposed by **E.F. Codd** in 1970, and it remains the foundation of most modern database systems. RDBMS uses **SQL (Structured Query Language)** as the standard language for querying and manipulating data. Popular RDBMS systems include MySQL, PostgreSQL, Oracle, Microsoft SQL Server, and SQLite. Data in an RDBMS is stored in multiple related tables, and relationships between tables are established using **keys** (primary and foreign keys). The relational model ensures data consistency, reduces redundancy, and allows complex querying through joins and set operations.

◆ Features of RDBMS

RDBMS comes with a rich set of features that make it the preferred choice for enterprise-level data management. It supports **ACID properties** (Atomicity, Consistency, Isolation, Durability) which ensure reliable transaction processing. RDBMS enforces **data integrity** through constraints like primary keys, foreign keys, unique, not null, and check constraints. It provides **multi-user access** with proper concurrency control, allowing multiple users to access and modify data simultaneously without conflicts. **Data security** is ensured through user authentication, authorization, and role-based access control. RDBMS supports **backup and recovery** mechanisms to protect data from loss. It also provides a **data dictionary** (system catalog) that stores metadata about the database structure itself.

Feature	Description
ACID Compliance	Ensures reliable transactions
Data Integrity	Constraints prevent invalid data
Multi-user Access	Concurrent access with control
Security	Authentication & authorization
Query Language	SQL for data manipulation
Backup & Recovery	Data protection mechanisms

◆ Difference Between DBMS and RDBMS

Though both DBMS and RDBMS are used to manage databases, they differ significantly in capabilities and structure. A **DBMS** (Database Management System) stores data as files and does not necessarily follow the relational model — it may store data hierarchically or as a network. An **RDBMS** strictly follows the relational model, organizing data into tables with defined relationships between them. DBMS does not enforce relationships between data stored in different files, while RDBMS uses foreign keys to maintain referential integrity across tables. RDBMS supports powerful JOIN operations to combine data from multiple tables, which is not natively available in basic DBMS. RDBMS is better suited for complex, large-scale applications requiring data consistency, while DBMS may be used for simpler, smaller applications. Examples of DBMS include file systems and XML databases, while RDBMS examples include MySQL, Oracle, and PostgreSQL.

◆ Terminologies — Relation, Attribute, Domain, Tuple

Understanding the core terminology of RDBMS is essential before diving deeper into its concepts. A **Relation** is simply a table in the database — it has a name, rows, and columns. An **Attribute** is a column in a table that represents a specific property of the entity being stored (e.g., `StudentName`, `RollNo`). A **Domain** is the set of all possible valid values that an attribute can hold — for example, the domain of an `Age` attribute might be integers between 1 and 150. A **Tuple** is a single row in a table — it represents one complete record of an entity (e.g., one student's data). The **Degree** of a relation is the total number of attributes (columns) it has, while the **Cardinality** is the total number of tuples (rows). These terms form the mathematical foundation of the relational data model.

◆ Entities and Codd's Rules

An **Entity** in database terminology refers to a real-world object or concept that has data worth storing — for example, a Student, Product, or Employee. Entities are represented as tables in an RDBMS. **E.F. Codd** defined **12 rules** (actually 13, numbered 0–12) that a true RDBMS must follow to be considered fully relational. Some key rules include: Rule 1 — all data must be stored in tables; Rule 2 — every piece of data must be accessible using table name, primary key, and column name; Rule 6 — the system must support updating views where logically possible; Rule 9 — application programs must remain unaffected when changes are made to storage or access methods; and Rule 10 — integrity constraints must be stored in the database catalog, not in application programs. These rules set the standard for what constitutes a true relational database system, though most commercial RDBMS systems follow them partially.

◆ Relational Model — Structure of Relational Database

The **Relational Model** organizes data into one or more tables (relations), each with a unique name. Every table consists of rows (tuples) and columns (attributes), and each cell contains a single atomic value. The structure of a relational database is defined by its **schema** — the logical design that specifies the tables, their columns, data types, and constraints. A **primary key** uniquely identifies each tuple in a table, while a **foreign key** is an attribute in one table that references the primary key of another table, establishing a relationship between them. The relational model supports three types of relationships — **One-to-One**, **One-to-Many**, and **Many-to-Many**. Many-to-many relationships are typically resolved by creating a junction/bridge table. The relational model separates the logical structure of data from its physical storage, providing **data independence**.

◆ Concept of Relational Algebra

Relational Algebra is a procedural query language that provides a theoretical foundation for relational databases and SQL. It works on relations (tables) and produces new relations as output. The fundamental operations of relational algebra include **Selection (σ)**, **Projection (π)**, **Union (\cup)**, **Set Difference ($-$)**, **Cartesian Product (\times)**, and **Rename (ρ)**. Additional derived operations include **Join (\bowtie)**, **Intersection (\cap)**, and **Division (\div)**. **Selection** filters rows based on a condition, **Projection** selects specific columns, and **Join** combines two tables based on a related attribute. Relational algebra expressions can be combined to form complex queries. Understanding relational algebra helps in query optimization and understanding how SQL queries are internally processed by the RDBMS query engine.

Operation	Symbol	Description
Selection	σ	Filter rows by condition
Projection	π	Select specific columns

Operation	Symbol	Description
Union	\cup	Combine tuples from two relations
Intersection	\cap	Common tuples in both relations
Set Difference	$-$	Tuples in first but not second
Cartesian Product	\times	All combinations of tuples
Join	\bowtie	Combine related tuples

◆ Role and Responsibilities of DBA

A **Database Administrator (DBA)** is a professional responsible for the installation, configuration, maintenance, security, and performance of database systems in an organization. The DBA designs the database schema and ensures it follows normalization principles to reduce redundancy. They manage **user accounts and permissions**, granting or revoking access to ensure data security. DBAs monitor **database performance**, identify bottlenecks, and tune queries and indexes for optimal speed. They are responsible for **backup and recovery** planning — ensuring data can be restored in case of system failure or data corruption. DBAs also handle **database upgrades**, patch management, and migration between database versions or platforms. In large organizations, the DBA works closely with developers, system administrators, and business analysts to ensure the database meets application requirements effectively.

◆ Integrity Constraints

Integrity constraints are rules enforced by the RDBMS to ensure the accuracy, validity, and consistency of data stored in the database. **Domain Constraints** restrict the values an attribute can hold to its defined domain (e.g., Age must be a positive integer). **Entity Integrity** requires that every table must have a primary key and that the primary key value cannot be NULL. **Referential Integrity** ensures that a foreign key value in one table must either match a primary key value in the referenced table or be NULL — this prevents orphan records. **Key Constraints** ensure uniqueness of primary key values across all tuples. **Not Null Constraints** prevent NULL values in specified columns. **Check Constraints** allow custom conditions to be defined (e.g., $\text{Salary} > 0$). These constraints are the guardians of data quality in any relational database system.

◆ Relational Algebra & Calculus — Selection, Projection, Union, Joins, Aggregate Functions

Selection (σ) retrieves rows from a table that satisfy a given condition — equivalent to the `WHERE` clause in SQL. **Projection (π)** retrieves specific columns from a table — equivalent to the `SELECT` clause. **Union** combines the results of two compatible relations, eliminating duplicate tuples. **Joins** are among the most powerful operations — an **Inner Join** returns only matching tuples from both tables, a **Left Join** returns all from the left and matching from the right, and a **Full Outer Join** returns all tuples from both. **Relational Calculus** is a non-procedural (declarative) alternative to relational algebra — you specify *what* you want, not *how* to get it. **Aggregate Functions** like `COUNT()`, `SUM()`, `AVG()`, `MAX()`, and `MIN()` perform calculations on a set of values and return a single result — commonly used with `GROUP BY` to summarize data by categories.

Module II — Transaction Management & Concurrency Control

◆ ACID Properties

ACID is an acronym that defines the four key properties that guarantee reliable processing of database transactions. These properties ensure that even in the event of system failures, errors, or concurrent access, the database remains in a consistent and trustworthy state. **Atomicity** ensures that a transaction is treated as a single unit — either all its operations are completed successfully or none of them are applied. **Consistency** guarantees that a transaction brings the database from one valid state to another, respecting all integrity constraints. **Isolation** ensures that concurrently executing transactions do not interfere with each other — each transaction appears to execute in isolation. **Durability** guarantees that once a transaction is committed, its changes are permanently saved even if the system crashes immediately after. ACID properties are the cornerstone of trustworthy database systems and are what distinguish RDBMS from simpler storage solutions.

Property	Meaning	Example
Atomicity	All or nothing	Bank transfer — debit & credit both happen or neither
Consistency	Valid state to valid state	Balance never goes negative if constraint exists
Isolation	Transactions don't interfere	Two users booking same seat handled separately
Durability	Committed data is permanent	Data survives server crash after commit

◆ Transaction Concept

A **transaction** in a database is a logical unit of work that consists of one or more database operations (INSERT, UPDATE, DELETE, SELECT) that are executed as a single, indivisible unit. Transactions are used to ensure data integrity when multiple related operations need to succeed or fail together — for example, transferring money between two bank accounts requires both a debit and a credit operation to succeed together. A transaction begins with a BEGIN TRANSACTION statement and ends with either a COMMIT (to save all changes permanently) or a ROLLBACK (to undo all changes made during the transaction). Transactions provide a safe boundary around a group of operations, protecting the database from partial updates that could leave data in an inconsistent state. In SQL, transactions are controlled using TCL (Transaction Control Language) commands — COMMIT, ROLLBACK, and SAVEPOINT.

◆ Transaction State

A transaction goes through several well-defined states during its lifecycle, and understanding these states is crucial for database management. The **Active** state is the initial state where the transaction is being executed and operations are being performed. When the transaction completes its final operation, it enters the **Partially Committed** state — the operations have been executed but not yet permanently saved. If all operations succeed and there are no errors, the transaction moves to the **Committed** state where changes are permanently written to the database. If any operation fails during execution, the transaction enters the **Failed** state, triggering a rollback. After rollback is complete, the transaction reaches the **Aborted** state and the database is restored to its state before the transaction began. Finally, both committed and aborted transactions reach the **Terminated** state, ending the transaction lifecycle.

Active → Partially Committed → Committed → Terminated

↓

Failed → Aborted → Terminated

◆ Implementation of Atomicity and Durability

Atomicity is implemented in RDBMS primarily through a mechanism called the **Transaction Log** (also called Write-Ahead Log or WAL). Before any change is made to the actual database, the RDBMS writes a record of the intended change to the log. If a transaction fails midway, the system uses the log to **undo** all changes made by that transaction, restoring the database to its previous state. **Durability** is achieved by ensuring that committed transaction data is written to **non-volatile storage** (hard disk) before the commit is acknowledged. The **REDO log** is used during recovery to reapply committed transactions that may not have been fully written to disk before a crash. Techniques like **checkpointing** are used to periodically save the database

state to disk, reducing the amount of log that needs to be replayed during recovery. Together, undo and redo logging form the backbone of atomicity and durability implementation in all major RDBMS systems.

◆ Concurrent Executions

Concurrent execution means multiple transactions are executed overlappingly in time — the RDBMS interleaves the operations of different transactions to improve system throughput and reduce waiting time. Concurrency is essential in multi-user database environments like banking systems, e-commerce platforms, and hospital management systems where many users access and modify data simultaneously. Without proper control, concurrent execution can lead to serious problems — **Lost Update** (two transactions overwrite each other's changes), **Dirty Read** (reading uncommitted data from another transaction), **Non-Repeatable Read** (reading different values for the same data in the same transaction), and **Phantom Read** (new rows appearing in a repeated query). These problems are collectively called **concurrency anomalies**. To handle them, RDBMS uses concurrency control mechanisms like locking protocols and timestamp ordering to ensure that concurrent transactions produce results equivalent to some serial (sequential) execution.

◆ Anomalies Due to Interleaved Execution of Transactions

When transactions are interleaved without proper control, several data anomalies can occur that compromise data integrity. The **Lost Update Problem** occurs when two transactions read the same value, both modify it, and the second write overwrites the first — the first transaction's update is completely lost. The **Dirty Read Problem** (also called Temporary Update Problem) happens when Transaction T2 reads data that was modified by Transaction T1 but T1 has not yet committed — if T1 rolls back, T2 has read invalid data. The **Non-Repeatable Read Problem** occurs when a transaction reads the same data twice within the same transaction but gets different values because another transaction modified and committed it in between. The **Phantom Read Problem** occurs when a transaction executes the same query twice but gets different sets of rows because another transaction inserted or deleted rows in between. These anomalies are the motivation behind concurrency control protocols in RDBMS systems.

◆ Serializability

Serializability is the gold standard of correctness for concurrent transaction execution in databases. A schedule (sequence of operations from multiple transactions) is considered **serializable** if its outcome is equivalent to some serial schedule where transactions execute one after another without any interleaving. There are two types of serializability — **Conflict**

Serializability and **View Serializability**. Two operations **conflict** if they belong to different transactions, access the same data item, and at least one of them is a write operation. A schedule is **conflict serializable** if it can be transformed into a serial schedule by swapping non-conflicting operations. This is tested using a **Precedence Graph (Serialization Graph)** — if the graph has no cycle, the schedule is conflict serializable. Serializability ensures that even though transactions execute concurrently, the final database state is as if they ran one at a time, maintaining full data consistency.

◆ **Recoverability**

Recoverability is a property of transaction schedules that ensures the database can be restored to a consistent state after a transaction failure. A schedule is **recoverable** if no transaction commits before all transactions whose changes it has read have also committed. If Transaction T2 reads data written by T1 and T2 commits before T1, and then T1 rolls back — T2 has committed based on data that no longer exists, making recovery impossible.

Cascading Rollbacks occur when the rollback of one transaction forces the rollback of other transactions that have read its uncommitted data — this can cause a large chain of rollbacks.

Cascadeless Schedules (also called Avoiding Cascading Aborts) prevent this by only allowing transactions to read committed data — Transaction T2 can only read data written by T1 after T1 commits. **Strict Schedules** go further by also preventing writes to data written by uncommitted transactions. Most commercial RDBMS systems implement strict schedules to ensure both recoverability and avoid cascading rollbacks.

◆ **Implementation of Isolation**

Isolation ensures that the intermediate state of a transaction is not visible to other transactions — each transaction should execute as if it's the only one running on the database. SQL standard defines four **isolation levels** that balance data consistency against performance and concurrency. **Read Uncommitted** is the lowest level — transactions can read uncommitted changes of others (dirty reads allowed). **Read Committed** prevents dirty reads but allows non-repeatable reads. **Repeatable Read** prevents both dirty and non-repeatable reads but phantom reads may still occur. **Serializable** is the highest level — it prevents all anomalies and ensures full isolation. Higher isolation levels provide more consistency but reduce concurrency and may cause more blocking. Isolation is implemented using **locking mechanisms** (shared and exclusive locks) or **Multi-Version Concurrency Control (MVCC)**, where each transaction sees a snapshot of the database at a specific point in time.

Isolation Level	Dirty Read	Non-Repeatable Read	Phantom Read
Read Uncommitted	<input checked="" type="checkbox"/> Possible	<input checked="" type="checkbox"/> Possible	<input checked="" type="checkbox"/> Possible

Isolation Level	Dirty Read	Non-Repeatable Read	Phantom Read
Read Committed	✗ Prevented	✓ Possible	✓ Possible
Repeatable Read	✗ Prevented	✗ Prevented	✓ Possible
Serializable	✗ Prevented	✗ Prevented	✗ Prevented

◆ Concurrency Control — Lock Based Concurrency Control

Lock-based concurrency control is the most widely used mechanism to manage concurrent access to database resources. A **lock** is a mechanism that restricts access to a data item when a transaction is using it. There are two main types of locks — **Shared Lock (S-Lock)** allows multiple transactions to read a data item simultaneously but prevents writing, and **Exclusive Lock (X-Lock)** gives one transaction complete control over a data item, blocking both reads and writes by others. The **Lock Compatibility Matrix** defines which lock types can coexist — two shared locks are compatible, but an exclusive lock is incompatible with any other lock. Transactions must request a lock before accessing data and release it after use. The **Lock Manager** in the RDBMS maintains a lock table to track which transactions hold or are waiting for locks on which data items. Proper lock management prevents conflicting concurrent access and ensures data integrity.

◆ Two-Phase Locking (2PL) and Deadlocks

Two-Phase Locking (2PL) is a concurrency control protocol that guarantees conflict-serializability by dividing a transaction's execution into two phases. In the **Growing Phase**, the transaction acquires all the locks it needs and cannot release any. In the **Shrinking Phase**, the transaction releases its locks and cannot acquire new ones. The point where the transaction switches from growing to shrinking is called the **Lock Point**. 2PL ensures that the resulting schedule is conflict serializable. However, 2PL can lead to **Deadlocks** — a situation where two or more transactions are waiting for each other to release locks, creating a circular waiting chain where none can proceed. For example, T1 holds lock on A and waits for B, while T2 holds lock on B and waits for A. Deadlocks are detected using a **Wait-for Graph** — a cycle in this graph indicates a deadlock. RDBMS resolves deadlocks by **aborting** (rolling back) one of the transactions involved, which is called the **victim transaction**.

◆ Timestamp Based Methods

Timestamp-based concurrency control is an alternative to locking that uses timestamps to order transactions and ensure serializability without using locks. Each transaction is assigned a

unique **timestamp** when it starts — either using the system clock or a logical counter. Each data item stores two timestamps — **Read Timestamp (RTS)**: the timestamp of the last transaction that read it, and **Write Timestamp (WTS)**: the timestamp of the last transaction that wrote it. The **Thomas Write Rule** and basic timestamp protocol define when a transaction's read or write operation should be allowed or rejected. If an older transaction tries to read/write data that a newer transaction has already modified, the older transaction is rolled back and restarted with a new timestamp. Timestamp methods are **deadlock-free** since no transaction ever waits for another — it either proceeds or is restarted. However, they can cause more rollbacks (cascading aborts) compared to locking under high contention workloads.

Module III — PL/SQL

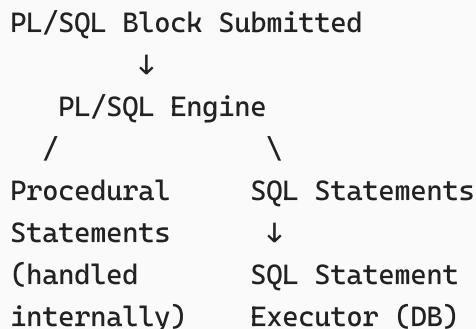
◆ Introduction to PL/SQL

PL/SQL (Procedural Language/Structured Query Language) is Oracle Corporation's procedural extension to SQL that combines the data manipulation power of SQL with the procedural capabilities of a programming language. It was developed to overcome the limitations of SQL, which is declarative and cannot handle complex logic, loops, or conditional processing on its own. PL/SQL allows developers to write complete programs — with variables, conditions, loops, functions, and procedures — that run directly inside the Oracle database engine. This means less data needs to travel between the application and the database, resulting in significantly improved performance. PL/SQL supports error handling, modular programming through procedures and packages, and can interact directly with Oracle's data dictionary. It is widely used in enterprise applications built on Oracle databases for writing business logic, data validation, and complex reporting.

◆ Architecture of PL/SQL

The PL/SQL engine has a well-defined architecture that determines how PL/SQL code is compiled and executed within the Oracle database environment. PL/SQL code is submitted to the **PL/SQL Engine**, which is embedded inside the Oracle Database Server. The engine separates the PL/SQL block into two parts — **procedural statements** (IF, LOOP, variable assignments) are handled by the PL/SQL engine itself, while **SQL statements** (SELECT, INSERT, UPDATE, DELETE) are passed to the **SQL Statement Executor** within the database. This separation allows the PL/SQL engine to process logic locally while delegating data operations to the optimized SQL engine. The **PL/SQL compiler** converts source code into **p-code** (intermediate bytecode), which is then executed by the PL/SQL runtime engine. This

architecture minimizes network traffic in client-server environments because multiple SQL statements can be bundled into a single PL/SQL block and sent to the server in one round trip.



◆ Data Types in PL/SQL

PL/SQL supports a rich set of data types for declaring variables, constants, and parameters in programs. **Scalar Data Types** hold a single value — these include `NUMBER` (for numeric values), `VARCHAR2` (for variable-length strings), `CHAR` (for fixed-length strings), `DATE` (for date and time values), `BOOLEAN` (TRUE, FALSE, or NULL), and `BINARY_INTEGER` (for integer arithmetic). **Composite Data Types** hold multiple values — `RECORD` holds a collection of fields (like a row of a table), and `TABLE` and `VARRAY` are collection types similar to arrays. The `%TYPE` attribute allows a variable to inherit the data type of a specific column or another variable — e.g., `v_salary employees.salary%TYPE` — this makes code more maintainable and adaptable to schema changes. The `%ROWTYPE` attribute declares a variable that can hold an entire row of a table or cursor result. Using these type attributes is considered best practice in PL/SQL development.

Data Type	Description	Example
<code>NUMBER</code>	Numeric values	<code>v_age NUMBER(3)</code>
<code>VARCHAR2</code>	Variable string	<code>v_name VARCHAR2(50)</code>
<code>CHAR</code>	Fixed string	<code>v_code CHAR(5)</code>
<code>DATE</code>	Date & time	<code>v_dob DATE</code>
<code>BOOLEAN</code>	True/False/NULL	<code>v_flag BOOLEAN</code>
<code>%TYPE</code>	Inherits column type	<code>v_sal emp.sal%TYPE</code>
<code>%ROWTYPE</code>	Inherits full row	<code>v_rec emp%ROWTYPE</code>

◆ Operators in PL/SQL

PL/SQL supports all standard operators needed to build expressions and logic within programs. **Arithmetic Operators** include + (addition), - (subtraction), * (multiplication), / (division), and ** (exponentiation — unique to PL/SQL). **Relational (Comparison) Operators** include =, != or <>, <, >, <=, >=, IS NULL, IS NOT NULL, LIKE, BETWEEN, and IN — these return Boolean results used in conditions. **Logical Operators** include AND, OR, and NOT — used to combine multiple conditions. **String Operator** || is used for string concatenation — e.g., 'Hello' || ' World' produces 'Hello World'. **Assignment Operator** in PL/SQL is := (not = like in SQL) — e.g., v_count := 0. The => operator is used for named notation in procedure and function calls. Understanding operator precedence is important — arithmetic operators have higher precedence than comparison operators, which have higher precedence than logical operators.

◆ Decision Making in PL/SQL

PL/SQL provides several conditional control structures that allow programs to make decisions and execute different code paths based on conditions. The **IF-THEN** statement executes a block of code only if the condition is TRUE. The **IF-THEN-ELSE** statement provides an alternative block that executes when the condition is FALSE. The **IF-THEN-ELSIF** (note: ELSIF, not ELSEIF) allows testing multiple conditions in sequence — the first TRUE condition's block is executed and the rest are skipped. The **CASE statement** is a cleaner alternative to multiple IF-ELSIF chains — it can be a **Simple CASE** (comparing one expression to multiple values) or a **Searched CASE** (evaluating multiple independent Boolean expressions). The **CASE expression** (different from CASE statement) can be used inline within SQL or assignment statements to return a value. NULL handling in conditions is important — any comparison with NULL returns NULL (not TRUE or FALSE), so IS NULL and IS NOT NULL operators must be used explicitly.

◆ Looping Statements in PL/SQL

PL/SQL offers three types of loop constructs that allow repeated execution of a block of code. The **Basic LOOP** is the simplest — it executes indefinitely until an EXIT or EXIT WHEN statement is encountered. It is useful when the number of iterations is not known in advance. The **WHILE LOOP** evaluates a condition before each iteration — the loop body executes only if the condition is TRUE, and stops as soon as it becomes FALSE. It is ideal when you want to loop while a condition holds. The **FOR LOOP** is used when the number of iterations is known — it automatically declares a loop counter variable and iterates from a lower bound to an upper bound (or in reverse using REVERSE). The EXIT statement immediately terminates a loop, while CONTINUE (available in Oracle 11g+) skips the rest of the current iteration and moves to the next. Nested loops are supported, and labels can be used to control which loop an EXIT or CONTINUE applies to.

```
-- Basic Loop Example
DECLARE v_count NUMBER := 1;
BEGIN
LOOP
    DBMS_OUTPUT.PUT_LINE(v_count);
    v_count := v_count + 1;
    EXIT WHEN v_count > 5;
END LOOP;
END;
```

◆ Simple PL/SQL Programmes

A PL/SQL program is organized into **blocks** — each block has three sections: **DECLARE** (optional, for variable declarations), **BEGIN** (mandatory, contains the executable code), and **EXCEPTION** (optional, for error handling). The simplest PL/SQL program just has a BEGIN-END block with some SQL or procedural statements. `DBMS_OUTPUT.PUT_LINE()` is the standard procedure used to print output to the console during testing and debugging. Variables are declared in the `DECLARE` section with a name, data type, and optional initial value. Constants are declared using the `CONSTANT` keyword and must be initialized at declaration. PL/SQL programs can include SQL DML statements (`SELECT INTO`, `INSERT`, `UPDATE`, `DELETE`) directly — `SELECT INTO` is used to fetch a single row's values into PL/SQL variables. Anonymous blocks (unnamed PL/SQL blocks) are executed once and not stored in the database, while named blocks (procedures, functions, packages) are stored and can be reused.

◆ Triggers in PL/SQL

A **trigger** is a named PL/SQL block that is automatically executed (fired) by the Oracle database in response to a specific event on a table, view, schema, or database. Triggers are powerful tools for enforcing business rules, auditing data changes, and maintaining derived data automatically. **DML Triggers** fire in response to `INSERT`, `UPDATE`, or `DELETE` operations on a table. They can be defined as **BEFORE** triggers (execute before the DML operation) or **AFTER** triggers (execute after the DML operation). **Row-Level Triggers** (using `FOR EACH ROW`) fire once for each row affected by the DML statement, while **Statement-Level Triggers** fire once per DML statement regardless of how many rows are affected. Inside a row-level trigger, `:NEW` refers to the new value being inserted/updated and `:OLD` refers to the old value being updated/deleted. **INSTEAD OF Triggers** are used on views to intercept DML operations that cannot be performed directly on complex views. Triggers should be used carefully as they can impact performance and make debugging more complex.

```
-- Example: Audit trigger
CREATE OR REPLACE TRIGGER trg_salary_audit
BEFORE UPDATE OF salary ON employees
FOR EACH ROW
BEGIN
    INSERT INTO salary_audit(emp_id, old_sal, new_sal, changed_on)
    VALUES(:OLD.employee_id, :OLD.salary, :NEW.salary, SYSDATE);
END;
```

◆ Cursors in PL/SQL

A **cursor** in PL/SQL is a pointer to the context area (memory area) that Oracle creates to process SQL statements. Since SQL operates on sets of rows but PL/SQL processes data procedurally (row by row), cursors bridge this gap by allowing programs to fetch and process one row at a time from a multi-row query result. **Implicit Cursors** are automatically created by Oracle for all DML statements and single-row SELECT INTO statements — you can check their status using attributes like `SQL%FOUND`, `SQL%NOTFOUND`, `SQL%ROWCOUNT`, and `SQL%ISOPEN`. **Explicit Cursors** are defined by the programmer for queries that return multiple rows — they go through four steps: **DECLARE** (define the cursor with a SELECT statement), **OPEN** (execute the query), **FETCH** (retrieve one row at a time into variables), and **CLOSE** (release the memory). **Cursor FOR LOOP** is a simplified way to use explicit cursors — it automatically opens, fetches, and closes the cursor. **Parameterized Cursors** accept parameters, making them reusable for different query conditions.

Cursor Attribute	Meaning
<code>%FOUND</code>	TRUE if last fetch returned a row
<code>%NOTFOUND</code>	TRUE if last fetch returned no row
<code>%ROWCOUNT</code>	Number of rows fetched so far
<code>%ISOPEN</code>	TRUE if cursor is open

◆ Handling Errors and Exceptions in PL/SQL

Exception handling in PL/SQL allows programs to gracefully deal with runtime errors instead of crashing abruptly. The EXCEPTION section at the end of a PL/SQL block contains handlers for specific error conditions. **Predefined Exceptions** are named exceptions that Oracle automatically raises for common errors — examples include `NO_DATA_FOUND` (SELECT INTO returns no rows), `TOO_MANY_ROWS` (SELECT INTO returns more than one row), `ZERO_DIVIDE` (division by zero), `VALUE_ERROR` (type conversion error), and `DUP_VAL_ON_INDEX` (duplicate

value in unique index). **User-Defined Exceptions** are declared in the DECLARE section using the EXCEPTION data type and raised explicitly using the RAISE statement when a business rule is violated. **RAISE_APPLICATION_ERROR** procedure allows raising custom error messages with error numbers between -20000 and -20999. The **WHEN OTHERS** handler acts as a catch-all for any unhandled exception — SQLCODE returns the error number and SQLERRM returns the error message. Proper exception handling is critical for writing robust, production-ready PL/SQL programs.

```
BEGIN
    SELECT salary INTO v_sal FROM employees WHERE employee_id = 999;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Employee not found!');
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('Error: ' || SQLERRM);
END;
```