RDBMS

#itsem4 #informationtechnology #rdbmsnotes #semester4 #completenotes

MADE BY

KASHIF SAYYAD

Module I

Introduction

Concept of RDBMS

RDBMS (Relational Database Management System) is a type of database management system that stores data in a structured format using tables (relations) consisting of rows and columns. The relational model, proposed by E.F. Codd in 1970, forms the foundation of RDBMS, organizing data into relations where each table represents an entity and relationships between entities are established through common attributes. RDBMS stores data in a tabular format where each row (tuple) represents a record and each column represents an attribute. The system manages data through a set of formal operations and ensures data integrity through constraints. RDBMS uses Structured Query Language (SQL) as the standard interface for defining, manipulating, and querying data. The relational approach provides data independence, meaning changes to the physical storage don't affect applications. RDBMS systems like Oracle, MySQL, PostgreSQL, SQL Server, and DB2 are widely used in enterprise applications, web applications, and data warehousing. The relational model's mathematical foundation ensures consistency, reduces redundancy through normalization, and provides powerful query capabilities through relational algebra and calculus.

Features of RDBMS

RDBMS offers numerous features that make it the preferred choice for data management in modern applications. **Data Storage in Tables** organizes information in two-dimensional tables with rows and columns, providing intuitive and structured data representation. **Data Integrity** is maintained through constraints like primary keys, foreign keys, unique constraints, check constraints, and not null constraints, ensuring accuracy and consistency. **ACID Properties** (Atomicity, Consistency, Isolation, Durability) guarantee reliable transaction processing even in case of failures. **Data Security** provides authentication, authorization, and access control mechanisms to protect sensitive information from unauthorized access. **Data Independence** separates logical data structure from physical storage, allowing changes to storage without affecting applications. **Concurrent Access** enables multiple users to access and modify data simultaneously while maintaining consistency through locking and transaction management. **Backup and Recovery** mechanisms protect against data loss through regular backups and

recovery procedures. **Query Optimization** automatically determines the most efficient way to execute queries. **Normalization** reduces data redundancy and improves data integrity by organizing data into related tables. **Scalability** allows databases to handle growing amounts of data and users through vertical and horizontal scaling approaches.

Difference between DBMS and RDBMS

Aspect	DBMS	RDBMS
Data Storage	Stores data as files in hierarchical or navigational form	Stores data in tabular form (tables with rows and columns)
Data Relationship	No or limited relationship between data elements	Establishes relationships between tables using foreign keys
Normalization	Generally not supported or limited	Supports normalization to reduce redundancy
Distributed Database	Does not support distributed databases	Supports distributed databases
Data Integrity	Limited integrity constraints	Strong integrity constraints (primary key, foreign key, etc.)
ACID Properties	May not fully support ACID properties	Fully supports ACID properties for transactions
Users	Supports single user or limited concurrent users	Supports multiple concurrent users efficiently
Data Volume	Suitable for small amounts of data	Handles large volumes of data efficiently
Security	Basic security features	Advanced security features with multiple levels
Examples	File systems, XML databases	Oracle, MySQL, PostgreSQL, SQL Server, DB2
Query Language	Each DBMS may have different query methods	Uses standardized SQL for querying
Hardware Requirements	Lower hardware and software requirements	Requires more hardware resources and sophisticated software

The fundamental difference is that RDBMS is built on the relational model with tables and relationships, while traditional DBMS uses file-based or hierarchical structures without formal relationships.

Terminologies

Relation

A relation in RDBMS is a table consisting of rows and columns, representing an entity or relationship in the database. Formally, a relation is a subset of the Cartesian product of domains, meaning it's a set of tuples (rows) where each tuple contains values from specified domains. Each relation has a unique name and is defined by its schema (relation name and attribute names with their domains). Relations have several important properties: **order of tuples is insignificant** (rows can appear in any order), **order of attributes is insignificant** (columns can be rearranged), **all tuples are unique** (no duplicate rows), **atomic values only** (each cell contains a single, indivisible value from the attribute's domain), and **attribute names must be unique** within a relation. For example, a STUDENT relation might have attributes like StudentID, Name, Age, and Department. Relations are the fundamental structure in RDBMS, and all data operations are performed on relations. The degree of a relation is the number of attributes (columns), while the cardinality is the number of tuples (rows).

Attribute

An attribute is a named column in a relation that represents a characteristic or property of the entity described by the table. Each attribute has a domain, which is the set of allowable values that the attribute can take. For example, in a STUDENT table, attributes might include StudentID (integer domain), Name (string domain), DateOfBirth (date domain), and GPA (decimal domain). Attributes are atomic, meaning they contain single, indivisible values rather than composite or multi-valued data. **Attribute types** include simple attributes (cannot be divided further), composite attributes (can be divided into sub-parts like Address into Street, City, State), single-valued attributes (one value per entity), multi-valued attributes (multiple values like phone numbers, normalized into separate tables), derived attributes (calculated from other attributes like Age from DateOfBirth), and key attributes (uniquely identify tuples). Proper attribute definition is crucial for database design, ensuring data integrity and efficient querying.

Domain

A domain is the set of all possible values that an attribute can contain, defining the data type, range, and constraints for that attribute. Domains ensure data consistency and integrity by restricting values to valid entries. For example, the domain for an "Age" attribute might be integers between 0 and 150, while the domain for "Gender" might be {'M', 'F', 'Other'}. **Domain characteristics** include the data type (integer, string, date, boolean, etc.), the format (like date format MM/DD/YYYY), the range of values (minimum and maximum), and constraints (like positive numbers only). Domains can be simple (atomic values) or structured (composite types). In SQL, domains are implemented through data types (INT, VARCHAR, DATE) and constraints (CHECK constraints). Defining proper domains prevents invalid data entry, reduces errors, and ensures semantic correctness. For instance, a domain for email addresses would include format validation to ensure values match email patterns. Domains provide a semantic layer that goes beyond simple data types, capturing business rules and validation logic.

Tuple

A tuple is a single row in a relation, representing a single instance or record of the entity described by the table. Each tuple contains a value for every attribute in the relation's schema, and all values must come from the corresponding attribute's domain. For example, in a STUDENT table, one tuple might be (101, 'John Smith', 20, 'Computer Science'), where each value corresponds to an attribute (StudentID, Name, Age, Department). Tuples must be unique within a relation—no two tuples can have identical values for all attributes (this is enforced by having at least one key attribute). The number of tuples in a relation is called its **cardinality**, which changes as records are inserted or deleted. Tuples are unordered, meaning their position in the table is not significant. Operations on relations are tuple-oriented: selection identifies specific tuples, projection extracts certain attributes from tuples, and joins combine tuples from multiple relations. Understanding tuples is essential for data manipulation and query operations.

Entities

An entity is a real-world object or concept that has independent existence and can be distinctly identified, representing something about which data is stored in the database. Entities can be tangible (like Student, Employee, Product) or intangible (like Course, Department, Transaction). Each entity is described by a set of attributes that capture its properties. For example, a STUDENT entity might have attributes like StudentID, Name, DateOfBirth, and Major. **Entity types** include strong entities (exist independently with their own primary key, like EMPLOYEE) and weak entities (depend on other entities for identification, like DEPENDENT which depends on EMPLOYEE). In database design, entities become tables, entity instances become tuples, and entity attributes become table columns. The Entity-Relationship (ER) model uses entities as building blocks for database design, showing entities as rectangles in ER diagrams. Proper entity identification is crucial for effective database design, ensuring all necessary data is captured while avoiding redundancy. Relationships between entities (one-to-one, one-to-many, many-to-many) determine how tables are connected through foreign keys.

Degree

The degree of a relation is the number of attributes (columns) in that relation, indicating the complexity and structure of the table. For example, a STUDENT table with attributes StudentID, Name, Age, and Department has a degree of 4. Degree is also called "arity" of the relation. Relations can have different degrees: **unary (degree 1)** contains one attribute, **binary (degree 2)** contains two attributes, **ternary (degree 3)** contains three attributes, and **n-ary (degree n)** contains n attributes. The degree is determined during database design and remains constant unless the schema is altered. Higher degree relations can store more information per tuple but may be more complex to manage and query. The degree affects query performance, indexing strategies, and normalization decisions. For instance, a relation with too many attributes might violate normalization rules and should be decomposed into multiple relations. Understanding degree helps in schema design, determining whether a relation should be split or combined with others for optimal structure.

Codd's Rules

Codd's 12 Rules (actually 13, numbered 0-12) were proposed by E.F. Codd to define what constitutes a truly relational database management system. These rules serve as benchmarks for evaluating RDBMS products. **Rule 0 (Foundation Rule)** states the system must use relational capabilities exclusively for database management. **Rule 1 (Information Rule)** requires all information to be represented as values in tables. **Rule 2 (Guaranteed Access Rule)** mandates every data value must be accessible through table name, primary key, and column name. **Rule 3 (Systematic Treatment of Null Values)** requires uniform representation of missing information through nulls. **Rule 4 (Dynamic Online Catalog)** specifies the database structure must be stored as tables accessible through SQL. **Rule 5 (Comprehensive Data Sublanguage Rule)** requires at least one supported language with linear syntax for data definition, manipulation, integrity, authorization, and transactions. **Rule 6 (View Updating Rule)** states all theoretically updatable views should be updatable by the system. **Rule 7 (High-level Insert, Update, Delete)** requires set-level operations for insert, update, and delete. **Rule 8 (Physical Data Independence)** means application programs remain unaffected by changes to physical storage. **Rule 9 (Logical Data Independence)** requires application independence from changes to table structures. **Rule 10 (Integrity Independence)** mandates integrity constraints must be stored in the catalog, not in applications. **Rule 11 (Distribution Independence)** requires that distributed database capabilities don't affect applications. **Rule 12 (Non-subversion Rule)** prevents low-level access from bypassing integrity rules. Few RDBMS systems satisfy all rules completely, but they provide a theoretical framework for relational database design.

Relational Model

Structure of Relational Database

The relational database structure is built on mathematical set theory and predicate logic, organizing data into collections of tables (relations) with defined relationships between them. The structure consists of three main components: **schema** (defines the structure, including table names, attribute names, data types, and constraints), **instance** (the actual data stored at a particular moment), and **metadata** (data about data, stored in system catalog tables). Each table represents an entity or relationship, with rows representing individual instances and columns representing attributes. **Key structural elements** include tables/relations (fundamental storage units), attributes/columns (properties of entities), tuples/rows (individual records), domains (allowable values for attributes), keys (uniquely identify tuples), and constraints (enforce data integrity). Tables are related through foreign keys, which reference primary keys in other tables, creating a network of interconnected data. The structure supports three levels of abstraction: **external level** (user views), **conceptual level** (logical structure of entire database), and **internal level** (physical storage). This three-schema architecture provides data independence, allowing changes at one level without affecting others. The relational structure enables powerful query capabilities through SQL, supports data integrity through constraints, and facilitates normalization to reduce redundancy.

Concept of Relational Algebra

Relational Algebra is a procedural query language that provides a formal foundation for relational database operations, defining how to manipulate relations to retrieve desired data. It consists of a set of operations that take one or two relations as input and produce a new relation as output, allowing operations to be nested. Relational algebra is theoretical but forms the basis for SQL query optimization. **Basic operations** include Selection (σ), Projection (π), Union (∪), Set Difference (−), Cartesian Product (×), and Rename (ρ). **Derived operations** include Intersection (∩), Join (⋈), Division (÷), and various join types. Each operation is formally defined with mathematical notation and precise semantics. Relational algebra is **procedural** because it specifies the sequence of operations to obtain results, unlike declarative SQL where you specify what you want, not how to get it. The algebra is **closed**, meaning operations on relations produce relations, enabling composition of operations. Query optimization in RDBMS involves converting SQL queries to relational algebra expressions and then applying transformation rules to find efficient execution plans. Understanding relational algebra helps database developers write better queries and understand query execution.

Role and Responsibilities of DBA

The Database Administrator (DBA) is responsible for the design, implementation, maintenance, and security of an organization's databases, ensuring data availability, integrity, and performance. **Key responsibilities** include database design and planning (schema design, capacity planning, selecting DBMS), installation and configuration (installing database software, configuring servers, setting parameters), security management (user authentication, authorization, implementing security policies, auditing access), backup and recovery (scheduling backups, testing recovery procedures, disaster recovery planning), performance monitoring and tuning (identifying bottlenecks, optimizing queries, index management, resource allocation), database maintenance (applying patches and updates, reorganizing tables, managing storage), user support (assisting users, troubleshooting issues, providing training), data integrity (enforcing constraints, monitoring data quality, implementing validation rules), capacity planning (monitoring growth, planning expansions), documentation (maintaining database documentation, change logs, procedures), and compliance (ensuring regulatory compliance like GDPR, HIPAA). **DBA types** include system DBA (focuses on technical aspects like installation and performance), application DBA (focuses on database design and development support), and development DBA (works with developers on schema design and optimization). The DBA must balance conflicting requirements like performance vs. security, availability vs. consistency, and cost vs. functionality. Effective DBAs possess technical skills (SQL, database architecture, operating systems), analytical skills (problem-solving, performance analysis), and communication skills (working with developers, management, and users).

Integrity Constraints

Integrity constraints are rules enforced by the RDBMS to ensure data accuracy, consistency, and validity, preventing invalid data entry and maintaining database quality. **Types of integrity constraints** include **Domain Constraints** (restrict values to valid domains, like age must be positive integer), **Entity Integrity** (requires primary key values to be unique and not null, ensuring each tuple is identifiable), **Referential Integrity** (enforces valid relationships through foreign keys, ensuring referenced records exist), **Key Constraints** (enforce uniqueness through primary and candidate keys), and **User-Defined Constraints** (custom business rules like CHECK constraints). **Primary Key Constraint** ensures each table has a unique identifier with no null values. **Foreign Key Constraint** maintains relationships between tables, with options for cascade operations (ON DELETE CASCADE, ON UPDATE CASCADE) or restrict/set null actions. **Unique Constraint** prevents duplicate values in specified columns while allowing nulls. **Not Null Constraint** requires values for specific attributes. **Check Constraint** validates data against specific conditions (like salary > 0). Integrity constraints can be defined during table creation or added later through ALTER TABLE statements. They are enforced automatically by the RDBMS during INSERT, UPDATE, and DELETE operations, rejecting operations that violate constraints. Proper constraint definition is crucial for maintaining data quality, supporting business rules, and preventing logical inconsistencies. Constraints also improve query optimization as the query optimizer can use constraint information to simplify execution plans.

Relational Algebra and Calculus

Selection

Selection (σ) is a unary relational algebra operation that retrieves tuples (rows) from a relation that satisfy a specified condition (predicate), effectively filtering data horizontally. The selection operation is denoted as $\sigma_{condition}(Relation)$, where the condition involves comparison operators (=, ≠, <, >, ≤, ≥) and logical operators (AND, OR, NOT) on attributes. For example, $\sigma_{Age>20}(STUDENT)$ selects all student tuples where age is greater than 20. **Properties of Selection**: it produces a relation with the same schema (same attributes) as the input but potentially fewer tuples; it is commutative ($\sigma_{c1}(\sigma_{c2}(R)) = \sigma_{c2}(\sigma_{c1}(R))$); cascading selections can be combined into a single selection with AND ($\sigma_{c1}(\sigma_{c2}(R)) = \sigma_{c1\ AND\ c2}(R)$). The selection operation corresponds to the WHERE clause in SQL queries. Examples include $\sigma_{Dept='CS'}(STUDENT)$ to find computer science students, $\sigma_{Salary>50000\ AND\ Age<30}(EMPLOYEE)$ to find young high earners. Selection is fundamental for query processing and is typically the first operation in query execution to reduce data volume before other operations.

Projection

Projection (π) is a unary relational algebra operation that retrieves specified attributes (columns) from a relation while eliminating duplicates, effectively filtering data vertically. The projection operation is denoted as $\pi_{attribute\text{-}list}(Relation)$, where attribute-list specifies which

columns to include. For example, $\pi_{Name,Age}(STUDENT)$ returns only the Name and Age columns from the STUDENT table. **Properties of Projection**: it produces a relation with fewer or equal attributes than the input; the resulting relation may have fewer tuples due to duplicate elimination (unlike selection which preserves tuple count); it is NOT commutative (order of attributes matters); cascading projections can be simplified ($\pi_{A1}(\pi_{A2}(R)) = \pi_{A1}(R)$ if $A1 \subseteq A2$). Projection corresponds to the SELECT clause in SQL (though SQL SELECT doesn't automatically eliminate duplicates unless DISTINCT is specified). The operation is useful for: reducing data volume by excluding unnecessary attributes, focusing on specific information, preparing data for joins, and hiding sensitive columns. Examples include $\pi_{StudentID,Name}(STUDENT)$ to get student identifiers and names, $\pi_{Department}(EMPLOYEE)$ to get unique department names.

Union

Union ($\cup$) is a binary relational algebra operation that combines tuples from two relations, producing a result containing all tuples from both relations while eliminating duplicates. The union operation is denoted as $R \cup S$, where R and S must be **union-compatible** (same degree and corresponding attributes from compatible domains). For example, if STUDENT1 and STUDENT2 are two student tables with identical schemas, $STUDENT1 \cup STUDENT2$ produces a combined relation with all students from both tables. **Properties of Union**: it is commutative ($R \cup S = S \cup R$); it is associative ($R \cup (S \cup T) = (R \cup S) \cup T$); the result has the same schema as input relations; duplicates are automatically eliminated. Union corresponds to the SQL UNION operator (UNION removes duplicates, UNION ALL keeps them). Common uses include combining data from multiple sources, merging partitioned tables, finding complete sets across divisions. Example: $\pi_{Name}(GRADUATE\_STUDENTS) \cup \pi_{Name}(UNDERGRADUATE\_STUDENTS)$ gives all student names regardless of program level. The cardinality of the union is $\leq$ cardinality(R) + cardinality(S), with equality only when relations have no common tuples.

Joins

Join operations combine tuples from two or more relations based on related attributes, fundamental for querying data across multiple tables. **Theta Join ($\bowtie_\theta$)** combines relations based on a condition θ involving comparison operators, producing all tuple combinations that satisfy the condition. **Equi Join** is a theta join where the condition uses only equality (=), the most common join type. **Natural Join ($\bowtie$)** is an equi join on all common attributes, automatically eliminating duplicate columns from the result. **Outer Joins** preserve tuples that don't match: **Left Outer Join** ($\bowtie$) includes all tuples from the left relation, **Right Outer Join** ($\bowtie$) includes all tuples from the right relation, and **Full Outer Join** includes tuples from both relations, filling missing values with nulls. **Semi Join ($\ltimes$)** returns tuples from the left relation that have matches in the right relation, essentially a projection after a join. **Cross Join (Cartesian Product $\times$)** produces all possible tuple combinations from two relations without any condition, rarely used directly but fundamental to join theory. Joins correspond to SQL JOIN clauses with various types

(INNER JOIN, LEFT JOIN, RIGHT JOIN, FULL JOIN, CROSS JOIN). Performance considerations include using indexes on join attributes, choosing appropriate join algorithms (nested loop, hash join, merge join), and minimizing the number of tuples before joining through selection and projection.

Aggregate Functions

Aggregate functions perform calculations on sets of values and return a single result, essential for data analysis and reporting in relational databases. **Common aggregate functions** include COUNT (counts number of tuples or non-null values), SUM (calculates total of numeric values), AVG (computes average of numeric values), MAX (finds maximum value), and MIN (finds minimum value). These functions are typically used with GROUP BY to partition data into groups and compute aggregates for each group. In relational algebra, aggregation is denoted as $_{G}\gamma_{F}(R)$, where G is the grouping attributes and F is the aggregate function list. For example, $_{Department}\gamma_{COUNT(), AVG(Salary)}(EMPLOYEE)$ *groups employees by department and computes count and average salary for each.* ***HAVING clause*** *filters groups based on aggregate conditions, applied after GROUP BY. Aggregate functions ignore null values (except COUNT()*). Common patterns include finding totals (SUM(Sales) for total sales), averages (AVG(Grade) for class average), frequencies (COUNT(*) GROUP BY Category for counts per category), extremes (MAX(Salary) for highest paid employee), and statistical summaries. Aggregation is crucial for business intelligence, reporting, data analysis, and answering questions that require summary information rather than individual records. Understanding aggregation helps in writing effective analytical queries and designing appropriate database structures for reporting requirements.

Module II

Transaction Management

ACID Properties

ACID properties are the fundamental characteristics that guarantee reliable transaction processing in database systems, ensuring data integrity even in the face of errors, power failures, or concurrent access. **Atomicity** ensures that a transaction is treated as a single, indivisible unit of work—either all operations within the transaction are completed successfully, or none are applied. If any operation fails, the entire transaction is rolled back to its initial state. For example, in a bank transfer, debiting one account and crediting another must both succeed or both fail. **Consistency** guarantees that a transaction brings the database from one valid state to another valid state, preserving all defined integrity constraints, rules, and relationships. The database must satisfy all constraints before and after the transaction. **Isolation** ensures that concurrent transactions execute independently without interference, as if they were executed serially. Intermediate states of a transaction are invisible to other transactions,

preventing dirty reads, non-repeatable reads, and phantom reads depending on the isolation level. **Durability** guarantees that once a transaction is committed, its effects are permanent and survive system failures. Committed changes are written to non-volatile storage and can be recovered after crashes. These properties work together to ensure database reliability: atomicity and durability are maintained through transaction logs and recovery mechanisms, consistency through constraint enforcement, and isolation through concurrency control protocols. ACID compliance is critical for applications requiring high data integrity like banking, e-commerce, healthcare, and financial systems.

Transaction Concept

A transaction is a logical unit of work consisting of one or more database operations (read, write, insert, update, delete) that are executed as a single atomic unit. Transactions represent business operations and must maintain database consistency. A transaction begins with a **BEGIN TRANSACTION** statement and ends with either **COMMIT** (making changes permanent) or **ROLLBACK** (undoing all changes). **Transaction states** form a lifecycle: **Active** (initial state during execution), **Partially Committed** (after final operation executes but before commit), **Committed** (after successful completion and COMMIT), **Failed** (when transaction cannot proceed or is aborted), and **Aborted** (after rollback, database restored to pre-transaction state). Transactions can abort due to system failures, transaction errors, or explicit rollback. After abort, the transaction can be restarted or killed depending on the error type. **Transaction operations** include Read(X) reading data item X, Write(X) writing data item X, Commit saving changes permanently, and Abort canceling changes. The transaction manager ensures ACID properties through logging (write-ahead logging records all changes before applying them), recovery mechanisms (using logs to restore consistency after failures), and concurrency control (coordinating concurrent transactions). Understanding transactions is essential for application developers to ensure data integrity and handle failures gracefully.

Transaction State

Transaction states represent the different stages a transaction goes through from initiation to completion or termination. The **Active state** is the initial state where the transaction is executing its operations (reads and writes). The transaction remains active as long as operations are executing normally. From active state, the transaction can move to partially committed or failed state. **Partially Committed state** occurs when the final operation of the transaction has executed successfully but changes haven't been permanently written to disk yet. This is a transient state before commit, where the transaction is waiting for confirmation that all changes can be safely committed. **Committed state** is reached when the transaction has completed successfully and all changes are permanently saved to the database. Once committed, changes are durable and survive system failures. **Failed state** indicates that the transaction cannot proceed further or has encountered an error. This can occur from active or partially committed state due to hardware failures, software errors, transaction logic errors, or violation of integrity constraints. **Aborted state** is reached after rollback operations complete, undoing all changes made by the transaction and restoring the database to its pre-transaction state. After abort, the

transaction can either be restarted (if the error was temporary like a deadlock) or killed (if the error was logical or permanent). The state transition diagram shows: Active → Partially Committed → Committed (success path) or Active → Failed → Aborted (failure path). Understanding transaction states helps in implementing proper error handling, recovery mechanisms, and transaction management in applications.

Implementation of Atomicity and Durability

Atomicity and durability are implemented through sophisticated logging and recovery mechanisms that track and persist transaction changes. **Write-Ahead Logging (WAL)** is the primary technique where all changes are recorded in a log file before being applied to the actual database. Log records include transaction ID, data item identifier, old value (before image), and new value (after image). The WAL protocol ensures that log records are written to stable storage before the corresponding database modifications are written. **Shadow Paging** is an alternative approach maintaining two page tables: current and shadow. During transaction execution, updates modify the current page table while shadow remains unchanged. Commit simply switches pointers, making changes atomic. **Atomicity implementation** uses the undo log containing before-images of modified data. If a transaction aborts or system crashes before commit, the recovery manager reads the undo log backward and restores old values, effectively rolling back the transaction. **Durability implementation** uses the redo log containing after-images. After system recovery, the recovery manager reads the redo log forward and reapplies committed transaction changes that may not have been written to disk. **Checkpointing** periodically writes all in-memory changes to disk and records a checkpoint in the log, limiting the amount of log that must be processed during recovery. The recovery process includes undo phase (rollback incomplete transactions) and redo phase (reapply committed transactions). Modern databases use ARIES (Algorithm for Recovery and Isolation Exploiting Semantics) protocol combining WAL, checkpointing, and sophisticated recovery algorithms. Buffer management coordinates with logging to ensure correct order of writes.

Concurrent Executions

Concurrent execution allows multiple transactions to execute simultaneously, improving system throughput, resource utilization, and response time. In a multi-user database environment, interleaving transaction operations maximizes CPU and I/O device utilization while one transaction waits for I/O, others can use the CPU. However, uncontrolled concurrent execution can lead to data inconsistencies. **Benefits of concurrency** include increased throughput (more transactions completed per unit time), reduced waiting time (shorter average response time), improved resource utilization (CPU working while I/O operations execute), and better user experience (multiple users served simultaneously). **Concurrency control** mechanisms ensure that concurrent executions produce results equivalent to some serial execution, maintaining database consistency. A **schedule** is a sequence of operations from multiple transactions, showing the chronological order of execution. **Serial schedules** execute transactions completely one after another without interleaving, always producing consistent results but offering no concurrency benefits. **Concurrent schedules** interleave operations from multiple

transactions, potentially improving performance but requiring careful control. **Serializable schedules** are concurrent schedules whose effect is equivalent to some serial schedule, guaranteeing correctness. Not all concurrent schedules are serializable—only those that maintain data consistency. The scheduler component of the DBMS determines the order of operation execution, using concurrency control protocols to ensure serializability while maximizing concurrency.

Anomalies Due to Interleaved Execution of Transactions

Uncontrolled concurrent execution can cause several types of anomalies that compromise data integrity and consistency. **Lost Update Problem** occurs when two transactions read the same data and both update it, with the second update overwriting the first, causing the first update to be lost. For example, two users simultaneously updating the same account balance will lose one person's update. **Dirty Read (Temporary Update Problem)** happens when a transaction reads data written by another uncommitted transaction. If the writing transaction aborts and rolls back, the reading transaction has used invalid data, leading to inconsistent results. **Unrepeatable Read (Inconsistent Retrieval Problem)** occurs when a transaction reads the same data twice and gets different values because another transaction modified the data between reads. This violates the isolation property, as the transaction doesn't see a consistent database snapshot. **Phantom Read** happens when a transaction re-executes a query and finds new rows inserted by another committed transaction, seeing different result sets for the same query. **Write-Write Conflict** occurs when two transactions simultaneously try to write to the same data item, potentially causing inconsistent database states. These anomalies can be prevented through proper concurrency control mechanisms like locking protocols, timestamp ordering, or multiversion concurrency control. Different isolation levels (Read Uncommitted, Read Committed, Repeatable Read, Serializable) offer varying degrees of protection against these anomalies, with trade-offs between consistency guarantees and concurrency performance.

Serializability

Serializability is a correctness criterion for concurrent transaction execution, defining when a concurrent schedule is correct by ensuring it produces the same result as some serial execution of the same transactions. A schedule is **serializable** if it is equivalent to a serial schedule, guaranteeing database consistency despite concurrent execution. **Conflict Serializability** examines conflicting operations (two operations from different transactions accessing the same data item, with at least one being a write). Two operations conflict if they cannot be swapped without changing the result. 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): create nodes for each transaction, draw an edge from Ti to Tj if Ti's operation conflicts with and precedes Tj's operation. The schedule is conflict serializable if and only if the precedence graph is acyclic. **View Serializability** is a broader concept: two schedules are view equivalent if they have the same initial reads (each transaction reads the same initial values), the same dependent reads (if transaction Ti reads a value written by Tj in

one schedule, it does so in the other), and the same final writes (the same transactions perform final writes to each data item). A schedule is view serializable if it is view equivalent to some serial schedule. All conflict serializable schedules are view serializable, but not vice versa. Testing view serializability is computationally harder than conflict serializability. Serializability theory provides the foundation for designing concurrency control protocols in database systems.

Recoverability

Recoverability ensures that committed transactions are not lost and that aborted transactions can be properly rolled back without affecting committed work. A schedule is **recoverable** if no transaction commits until all transactions whose changes it has read have committed. This prevents cascading rollbacks where aborting one transaction forces aborting others. **Cascading Rollback** occurs when a single transaction failure forces the rollback of multiple other transactions that read uncommitted data from the failed transaction. This is inefficient and complicates recovery. **Cascadeless Schedules** (Avoid Cascading Abort, ACA) prevent cascading rollback by ensuring transactions only read committed data—no transaction reads data written by uncommitted transactions. All cascadeless schedules are recoverable, but not vice versa. **Strict Schedules** are even more restrictive: transactions can neither read nor write data written by uncommitted transactions. Both read and write operations wait for the writing transaction to commit or abort. All strict schedules are cascadeless (and therefore recoverable). Strict schedules simplify recovery because aborting a transaction only requires restoring before-images without complex dependency tracking. Most commercial DBMSs implement strict schedules through strict two-phase locking. **Recovery guarantees** include committed transactions never need to be undone, aborted transactions can be completely rolled back, and no transaction sees intermediate states of other transactions. The recoverability hierarchy is: Strict $\subset$ Cascadeless $\subset$ Recoverable $\subset$ All Schedules, with strictness providing strongest guarantees but potentially limiting concurrency.

Implementation of Isolation

Isolation is implemented through concurrency control mechanisms that coordinate concurrent transaction execution while preventing anomalies and ensuring serializability. The primary approaches are locking protocols, timestamp ordering, and multiversion concurrency control. **Isolation levels** defined by SQL standard provide varying degrees of isolation with different performance trade-offs: **Read Uncommitted** (lowest isolation) allows dirty reads, non-repeatable reads, and phantom reads; **Read Committed** prevents dirty reads but allows non-repeatable and phantom reads; **Repeatable Read** prevents dirty and non-repeatable reads but allows phantom reads; **Serializable** (highest isolation) prevents all anomalies by ensuring truly serializable execution. Higher isolation levels provide stronger consistency guarantees but reduce concurrency and may impact performance. Implementation techniques include **lock-based isolation** using shared and exclusive locks with different protocols, **timestamp-based isolation** assigning timestamps to transactions and ordering conflicting operations, and **multiversion concurrency control (MVCC)** maintaining multiple versions of data items to allow readers to access older consistent versions while writers create new versions. MVCC is

widely used in modern databases (PostgreSQL, Oracle, MySQL InnoDB) because it allows readers and writers to proceed concurrently without blocking each other. The choice of isolation level and implementation depends on application requirements: financial transactions typically need Serializable isolation, while reporting queries might use Read Committed for better performance.

Concurrency Control

Lock-based Concurrency Control

Lock-based concurrency control uses locks to control concurrent access to data items, preventing conflicting operations from executing simultaneously. Transactions must acquire appropriate locks before accessing data and release locks when done. **Lock types** include **Shared Lock (S-lock/Read Lock)** allowing multiple transactions to read a data item concurrently but preventing writes, and **Exclusive Lock (X-lock/Write Lock)** allowing a transaction to both read and write while preventing all other accesses. Lock compatibility: multiple S-locks can coexist, but X-locks are incompatible with any other locks. **Lock operations** include lock-S(X) for shared lock request, lock-X(X) for exclusive lock request, and unlock(X) for lock release. The **lock manager** maintains a lock table tracking which transactions hold which locks on which data items, granting lock requests if compatible with existing locks or making transactions wait if conflicts exist. **Lock granularity** can vary: database-level (entire database), table-level (entire tables), page-level (disk pages), or row-level (individual tuples). Finer granularity allows more concurrency but increases overhead from managing many locks. **Deadlock** is a major issue where two or more transactions wait for each other's locks, creating a circular wait condition. Deadlock handling uses either prevention (forcing order on lock requests), avoidance (wait-die or wound-wait schemes), or detection (periodically checking for cycles in wait-for graph and aborting transactions to break deadlocks). Lock-based protocols are widely implemented but can reduce concurrency and cause deadlocks.

Two-Phase Locking (2PL)

Two-Phase Locking is a concurrency control protocol guaranteeing conflict serializability by restricting when transactions can acquire and release locks. 2PL divides transaction execution into two phases: **Growing Phase** where the transaction may acquire locks but cannot release any locks, and **Shrinking Phase** where the transaction may release locks but cannot acquire new locks. The point where the transaction acquires its last lock (transitioning from growing to shrinking) is called the **lock point**. **Basic 2PL** guarantees conflict serializability by ensuring all schedules follow the two-phase rule, but doesn't guarantee recoverability—cascading aborts can occur if transactions release locks before committing. **Conservative 2PL (Static 2PL)** requires transactions to acquire all locks at once before execution begins, preventing deadlocks but reducing concurrency and requiring knowledge of all needed resources upfront. **Strict 2PL** holds all exclusive locks until commit or abort, ensuring cascadeless schedules and simplifying recovery. **Rigorous 2PL** holds all locks (both shared and exclusive) until commit or

abort, providing the strictest guarantees and simplifying reasoning about concurrency. Most commercial databases use Strict 2PL as it balances serializability, recoverability, and practicality. **Limitations** include potential for deadlocks (requiring detection/prevention), reduced concurrency (transactions may wait unnecessarily), and lock overhead. However, 2PL's simplicity and effectiveness make it the most widely implemented concurrency control protocol in practice.

Deadlocks

Deadlock is a situation where two or more transactions are permanently blocked, each waiting for locks held by others in a circular chain. For example, Transaction T1 holds lock on A and waits for B, while T2 holds lock on B and waits for A, creating deadlock. **Deadlock conditions** (all must hold simultaneously): Mutual Exclusion (resources cannot be shared), Hold and Wait (transactions hold resources while requesting others), No Preemption (resources cannot be forcibly removed), and Circular Wait (circular chain of transactions waiting for each other). **Deadlock prevention** eliminates one of the necessary conditions: requiring all locks be acquired atomically (eliminates hold and wait), imposing total ordering on resources (eliminates circular wait), or using timeout-based preemption. **Deadlock avoidance** uses transaction timestamps: **Wait-Die** scheme allows older transactions to wait for younger but younger must abort if waiting for older (non-preemptive); **Wound-Wait** allows younger to wait for older but older preempts younger (preemptive). **Deadlock detection** periodically constructs a **wait-for graph** with transactions as nodes and edges representing wait relationships; cycles indicate deadlocks. When detected, the system selects a victim transaction to abort based on factors like age, number of locks held, or work completed. The aborted transaction is restarted. Detection frequency involves trade-offs: frequent checking catches deadlocks quickly but adds overhead, while infrequent checking reduces overhead but allows deadlocks to persist longer. Most databases use detection approaches as they allow maximum concurrency until deadlock actually occurs.

Time Stamping Methods

Timestamp-based concurrency control assigns unique timestamps to transactions and uses these to order conflicting operations, avoiding locks entirely. Each transaction T receives a timestamp TS(T) when it begins, typically using system clock or logical counter. Each data item X maintains read timestamp R-timestamp(X) (timestamp of youngest transaction that read X) and write timestamp W-timestamp(X) (timestamp of youngest transaction that wrote X). **Timestamp Ordering Protocol** rules: For Read(X) by transaction T, if TS(T) < W-timestamp(X), the read is rejected and T is aborted (trying to read future data); otherwise, the read executes and R-timestamp(X) = max(R-timestamp(X), TS(T)). For Write(X) by transaction T, if TS(T) < R-timestamp(X), the write is rejected and T aborted (trying to overwrite already-read data); if TS(T) < W-timestamp(X), the write might be ignored (Thomas Write Rule) or T aborted; otherwise, write executes and W-timestamp(X) = TS(T). **Advantages** include no deadlocks (no waiting, only aborts), scheduling is deadlock-free, and good for read-dominated workloads. **Disadvantages** include potential starvation (long transactions may repeatedly

abort), cascading rollbacks (unless using strict timestamp ordering), and overhead of maintaining timestamps. **Multiversion Timestamp Ordering** maintains multiple versions of each data item, allowing reads to access appropriate historical versions, improving concurrency significantly. Timestamp methods are less common in practice than locking but are used in some distributed and real-time database systems.

Module III

PL/SQL

Introduction to PL/SQL

PL/SQL (Procedural Language/Structured Query Language) is Oracle Corporation's procedural extension to SQL, combining the data manipulation power of SQL with the processing capabilities of procedural programming languages. While SQL is a non-procedural language that specifies what to do without defining how to do it, PL/SQL adds procedural constructs like variables, conditions, loops, and exception handling, enabling complex business logic implementation within the database. PL/SQL was introduced in Oracle version 6 and has evolved significantly, becoming integral to Oracle database development. The language allows developers to write blocks of code that can be stored in the database as stored procedures, functions, packages, and triggers, executing on the database server for improved performance by reducing network traffic. **Key advantages** include tight integration with SQL (seamless SQL statement embedding), improved performance (reduced client-server communication, compiled and cached execution plans), portability (runs on any platform supporting Oracle), security (code stored in database with access controls), and reusability (procedures and functions can be called from multiple applications). PL/SQL supports object-oriented programming features, collections, dynamic SQL, and extensive built-in packages. It's essential for database-centric application development, data validation, complex calculations, and automating database administration tasks. Understanding PL/SQL enables developers to build robust, efficient, and maintainable database applications.

Architecture of PL/SQL

PL/SQL architecture consists of two main components: the **PL/SQL Engine** and the **Database Server**. The PL/SQL engine is responsible for processing PL/SQL code and can reside in either the Oracle database server or client-side tools like Oracle Forms and Reports. When a PL/SQL block executes, the engine separates procedural statements from SQL statements. **Procedural statements** are processed by the procedural statement executor within the PL/SQL engine, which handles control structures, variable assignments, and procedural logic. **SQL statements** are sent to the SQL statement executor in the Oracle database server, which parses, optimizes, and executes them, returning results to the PL/SQL engine. This architecture enables efficient execution: all SQL statements in a PL/SQL block are sent to the database server in a single call, dramatically reducing network traffic compared to executing individual

SQL statements from a client application. The **PL/SQL compiler** translates source code into m-code (machine-readable code), which is stored in the database and executed by the PL/SQL virtual machine. **Compilation phases** include syntax checking (verifying language rules), semantic checking (validating database objects referenced), and code generation (creating executable m-code). The architecture supports **shared pool caching** where compiled PL/SQL code is cached in memory for reuse, avoiding recompilation. This design provides performance benefits, tight SQL integration, and seamless execution in both server and client environments.

Data Types

PL/SQL supports a rich set of data types for variable declaration and data manipulation, categorized into scalar types, composite types, reference types, and LOB types. **Scalar types** hold single values: **CHAR(size)** for fixed-length character strings up to 32,767 bytes, **VARCHAR2(size)** for variable-length strings up to 32,767 bytes (database limit is 4000), **NUMBER(p,s)** for numeric values with precision p and scale s, **BINARY_INTEGER** for signed integers, **PLS_INTEGER** for signed integers (faster than NUMBER), **BOOLEAN** storing TRUE, FALSE, or NULL (available only in PL/SQL, not in SQL), **DATE** for dates and times, **TIMESTAMP** for date/time with fractional seconds. **Composite types** group multiple values: **RECORD** is a user-defined type grouping related fields (like a row structure), **TABLE** (associative array/index-by table) is a one-dimensional collection, **VARRAY** (variable-size array) is a bounded ordered collection, **NESTED TABLE** is an unbounded ordered collection. **Reference types** hold pointers: **REF CURSOR** points to cursor result sets, enabling dynamic queries. **LOB types** handle large objects: **CLOB** (Character LOB) up to 128TB of character data, **BLOB** (Binary LOB) for binary data, **NCLOB** for national character set data, **BFILE** for external file references. **%TYPE** attribute declares a variable with the same type as a database column, ensuring consistency and avoiding maintenance when column types change. **%ROWTYPE** declares a record type representing an entire database row. Understanding data types is fundamental for proper variable declaration, memory efficiency, and data integrity in PL/SQL programs.

Operators

PL/SQL provides various operators for performing operations on variables and values, categorized by functionality. **Arithmetic operators** perform mathematical calculations: + (addition), - (subtraction), *(multiplication), / (division),* ***(exponentiation), and unary + and - for sign indication. Division by zero raises ZERO_DIVIDE exception.***
Relational/Comparison operators ***compare values and return TRUE, FALSE, or NULL: = (equal to), != or <> (not equal), < (less than), > (greater than), <= (less than or equal), >= (greater than or equal).*** *Logical operators* ***combine Boolean conditions: AND (true if both operands true), OR (true if either operand true), NOT (reverses Boolean value). These operators use three-valued logic (TRUE, FALSE, NULL), where NULL represents unknown.*** *String operators* ***manipulate text: || (concatenation operator) combines strings.*** *Set operators* ***work with collections and queries: UNION, UNION ALL, INTERSECT, MINUS.*** *Operator precedence* ***determines evaluation order: highest to lowest is*** *, (unary*

+/-), (/), (+/-), || , relational operators, NOT, AND, OR. Parentheses override default precedence. **NULL handling**: any arithmetic operation with NULL yields NULL, comparison with NULL yields NULL (use IS NULL or IS NOT NULL for null checks), logical operations follow three-valued logic rules. Understanding operator precedence and NULL behavior is critical for writing correct PL/SQL expressions and avoiding logical errors.

Decision Making and Looping Statement

PL/SQL provides control structures for conditional execution and iteration, essential for implementing complex business logic. **Decision-making statements** alter execution flow based on conditions. **IF-THEN** executes statements only if condition is true:

```
IF condition THEN statements; END IF;
```

IF-THEN-ELSE provides alternative execution path:

```
IF condition THEN statements1; ELSE statements2; END IF;
```

IF-THEN-ELSIF handles multiple conditions (note ELSIF spelling):

```
IF condition1 THEN statements1; ELSIF condition2 THEN statements2; ELSE
statements3; END IF;
```

CASE statement provides cleaner multi-way branching:

```
CASE variable WHEN value1 THEN statements1; WHEN value2 THEN statements2; ELSE
statements3; END CASE;
```

Looping statements enable repetitive execution. **Basic LOOP** creates indefinite loop requiring explicit exit:

```
LOOP statements; EXIT WHEN condition; END LOOP;
```

WHILE LOOP tests condition before each iteration:

```
WHILE condition LOOP statements; END LOOP;
```

FOR LOOP iterates through specified range:

```
FOR counter IN [REVERSE] lower..upper LOOP statements; END LOOP;
```

CONTINUE statement skips current iteration and proceeds to next. **EXIT statement** terminates loop execution. Loop labels allow nested loop control. Proper use of control structures creates maintainable, efficient code with clear logic flow.

Simple PL/SQL Programmes

Simple PL/SQL programs demonstrate fundamental concepts through practical examples. A **basic PL/SQL block structure** includes three sections:

```
DECLARE
   -- Variable declarations (optional)
   v_name VARCHAR2(50);
   v_salary NUMBER;
BEGIN
   -- Executable statements (mandatory)
   SELECT name, salary INTO v_name, v_salary
   FROM employees WHERE emp_id = 101;
   DBMS_OUTPUT.PUT_LINE('Name: ' || v_name);
EXCEPTION
   -- Exception handlers (optional)
   WHEN NO_DATA_FOUND THEN
      DBMS_OUTPUT.PUT_LINE('Employee not found');
END;
```

Example programs: 1) **Hello World** demonstrates output:

```
BEGIN DBMS_OUTPUT.PUT_LINE('Hello World'); END;
```

2. **Variable operations** show arithmetic:

```
DECLARE v_num NUMBER := 10; BEGIN v_num := v_num * 2; END;
```

3. **IF condition** illustrates decision-making:

```
DECLARE v_age NUMBER := 25; BEGIN IF v_age >= 18 THEN
DBMS_OUTPUT.PUT_LINE('Adult'); END IF; END;
```

4. **Loop example** demonstrates iteration:

```
BEGIN FOR i IN 1..5 LOOP DBMS_OUTPUT.PUT_LINE('Count: ' || i); END LOOP; END;
```

5. **Cursor usage** shows data retrieval:

```
DECLARE CURSOR c1 IS SELECT * FROM employees; v_emp c1%ROWTYPE; BEGIN OPEN c1;
FETCH c1 INTO v_emp; CLOSE c1; END;
```

These examples build foundation for complex PL/SQL development.

Triggers

Triggers are named PL/SQL blocks automatically executed (fired) in response to specific database events (DML operations or DDL statements), used for enforcing complex business rules, auditing, maintaining derived data, and implementing security. **Trigger components** include trigger name, triggering event (INSERT, UPDATE, DELETE, or DDL operations), trigger timing (BEFORE or AFTER), trigger level (row-level or statement-level), and trigger body (PL/SQL code to execute). **Trigger types**: **DML triggers** fire on INSERT, UPDATE, or DELETE operations on tables; **INSTEAD OF triggers** fire on views to handle operations not directly supported; **DDL triggers** fire on CREATE, ALTER, DROP statements; **Database triggers** fire on database-level events like STARTUP, SHUTDOWN, LOGON, LOGOFF. **Row-level triggers** (FOR EACH ROW) fire once per affected row, accessing old and new values through :OLD and :NEW pseudo-records. **Statement-level triggers** fire once per triggering statement regardless of rows affected. **Trigger timing**: BEFORE triggers execute before the triggering operation (can modify values), AFTER triggers execute after (cannot modify values). **Syntax**:

```
CREATE OR REPLACE TRIGGER trigger_name
{BEFORE | AFTER} {INSERT | UPDATE | DELETE} ON table_name
[FOR EACH ROW]
BEGIN
   -- trigger body
END;
```

Common uses include enforcing referential integrity, maintaining audit trails, deriving column values, preventing invalid operations, and replicating data. **Limitations** include mutating table errors (query or modify table being changed), performance impact (excessive triggers slow operations), and maintenance complexity (hidden business logic). Proper trigger design requires careful consideration of necessity, performance, and maintainability.

Cursors

Cursors are database objects that point to result sets of SQL queries, allowing row-by-row processing of query results in PL/SQL. While SQL operates on entire result sets, cursors enable procedural processing of individual rows, essential for complex data manipulation. **Implicit cursors** are automatically created by Oracle for single-row SELECT INTO, INSERT, UPDATE, and DELETE statements. Attributes include SQL%FOUND (true if operation affected rows), SQL%NOTFOUND (opposite of FOUND), SQL%ROWCOUNT (number of rows affected), SQL%ISOPEN (always false for implicit cursors as Oracle closes them automatically). **Explicit**

cursors are programmer-defined for queries returning multiple rows, providing greater control. **Cursor operations** follow a lifecycle: 1) **DECLARE** cursor with SELECT statement in DECLARE section: `CURSOR cursor_name IS SELECT statement;` 2) **OPEN** cursor to execute query and allocate memory: `OPEN cursor_name;` 3) **FETCH** retrieves rows one at a time: `FETCH cursor_name INTO variables;` 4) **CLOSE** cursor to release resources: `CLOSE cursor_name;` **Cursor attributes** for explicit cursors: cursor_name%FOUND, cursor_name%NOTFOUND, cursor_name%ROWCOUNT, cursor_name%ISOPEN. **Cursor FOR LOOP** simplifies processing by implicitly opening, fetching, and closing:

```
FOR record IN cursor_name LOOP
   -- process record
END LOOP;
```

Parameterized cursors accept parameters for flexible queries:

```
CURSOR c1(p_dept NUMBER) IS SELECT * FROM employees WHERE dept_id = p_dept;
```

Cursor variables (REF CURSOR) enable dynamic cursor handling. Understanding cursors is crucial for efficient data processing when row-by-row operations are necessary.

Handling Errors and Expectations

Exception handling in PL/SQL provides a structured way to handle runtime errors, preventing abrupt program termination and enabling graceful error recovery and reporting. **Exceptions** are runtime errors that disrupt normal execution flow. The EXCEPTION section catches and handles these errors. **Exception types**: **Predefined exceptions** are standard Oracle errors with names like NO_DATA_FOUND (SELECT INTO returns no rows), TOO_MANY_ROWS (SELECT INTO returns multiple rows), ZERO_DIVIDE (division by zero), VALUE_ERROR (arithmetic or conversion error), INVALID_CURSOR (illegal cursor operation), DUP_VAL_ON_INDEX (duplicate value in unique index). **User-defined exceptions** are declared in DECLARE section and raised explicitly using RAISE statement when business logic violations occur. **Exception handling syntax**:

```
BEGIN
   -- executable statements
EXCEPTION
   WHEN exception_name1 THEN
      -- handler statements
   WHEN exception_name2 OR exception_name3 THEN
      -- handler for multiple exceptions
   WHEN OTHERS THEN
      -- generic handler for all other exceptions
END;
```

RAISE statement explicitly raises exceptions: `RAISE exception_name;` or `RAISE_APPLICATION_ERROR(error_number, error_message);` for custom errors with error codes -20000 to -20999. **Exception propagation**: unhandled exceptions propagate to calling environment. **SQLCODE** and **SQLERRM** functions return error code and message for current exception. **Best practices** include always including exception handlers, using specific exception names before WHEN OTHERS, logging errors for debugging, avoiding overusing WHEN OTHERS (may mask problems), and providing meaningful error messages. Proper exception handling makes applications robust, maintainable, and user-friendly by gracefully managing error conditions.
