DI03000031: Relational Database Management System

GTU Course Presentation

July 25, 2026

Course Contents I

Agenda I

- 1. Data and Information: Definitions and Differences
- 2. Data Elements: Fields, Records, Files, and Metadata
- 3. Data Dictionary: Components and Significance
- 4. Database Concepts: Characteristics and Importance
- 5. Database Systems and Environment: Architecture and Functions
- 6. Schemas and Instances: Concepts, Sub-schemas, and States

Data vs. Information I

- Data: Raw, unorganized facts and figures that need to be processed. It lacks context and meaning on its own. For example, '98', 'John', '1995-10-12'.
- Information: Data that has been processed, organized, structured, or presented in a given context so as to make it useful. For example, 'John was born on 1995-10-12 and scored 98 in the exam'.
- Key Differences:
- - Meaning: Data is meaningless alone; Information is meaningful.
- - Form: Data is in raw unorganized form; Information is organized.
- - Dependence: Information depends on Data; Data does not depend on Information.

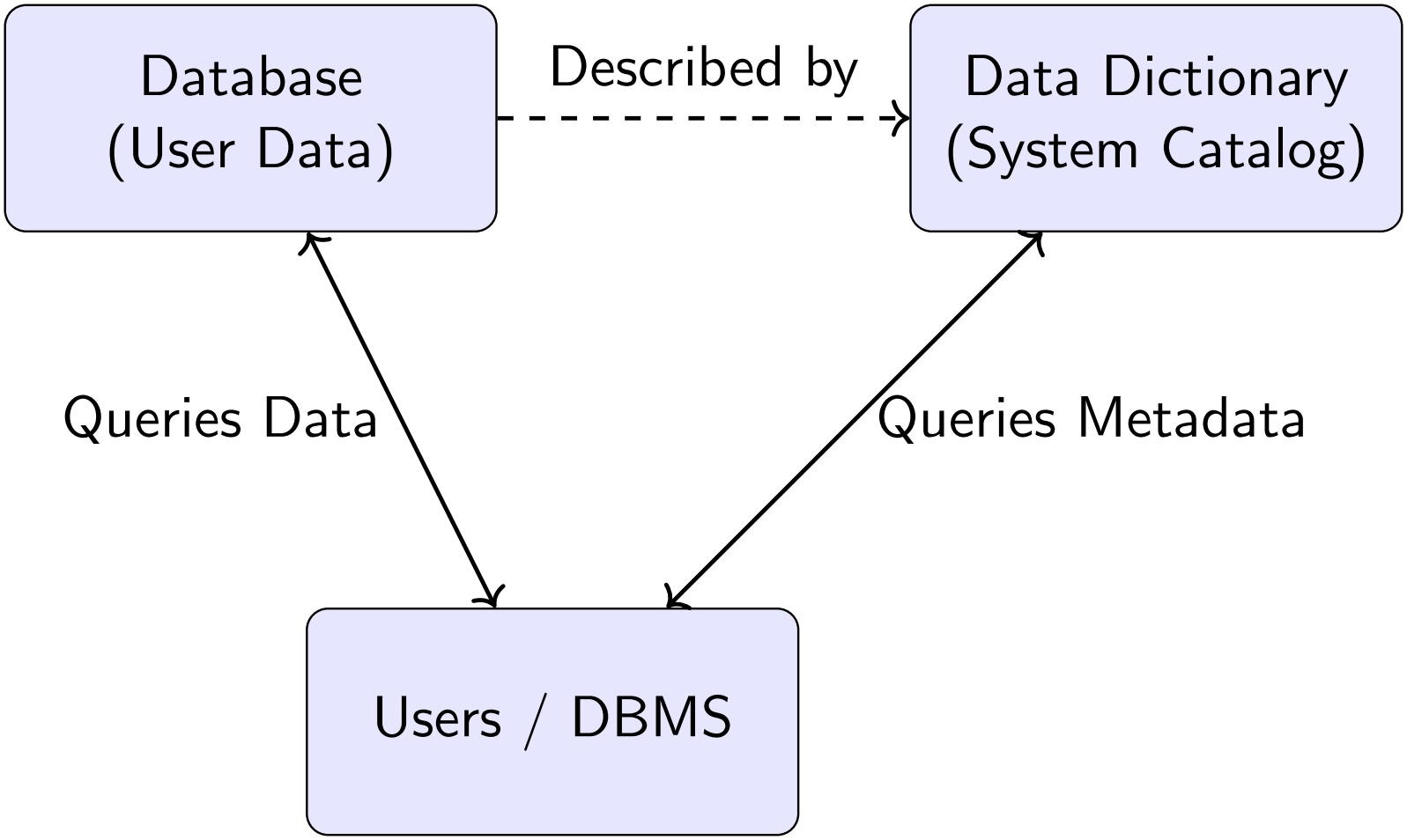
Data vs. Information II

- - Example in RDBMS: A table cell contains data, whereas a report generated from an SQL query like \{}texttt{SELECT AVG(salary) FROM employees WHERE dept='IT'} provides actionable information.

Data Elements I

- Data Item (Field/Attribute): The smallest unit of named application data recognized by system software, e.g., \{}texttt{Employee\{}_ID}. In an RDBMS, this corresponds to a column.
- Record (Tuple): A collection of related data items treated as a single unit, representing an entity. For example, an entire row representing one specific employee.
- File (Table/Relation): A collection of related records of a single type. In relational terms, a table containing all employee records.
- Metadata: Data about data. It provides context, such as data types (e.g., \{}texttt{VARCHAR2(50)}, \{}texttt{INTEGER}), constraints (e.g., \{}texttt{PRIMARY KEY}, \{}texttt{NOT NULL}), and structure definition of the database objects.

Data Dictionary: Structure Diagram I



Data Dictionary: Definition & Significance I

- Definition: A centralized repository of information about data such as meaning, relationships to other data, origin, usage, and format.
- Components of Data Dictionary:
- 1. Table and Column definitions (Names, Data Types, Sizes).
- 2. Integrity Constraints (Primary Keys, Foreign Keys, CHECK constraints).
- 3. Programmatic Metadata: Stores compiled objects like triggers, functions, and PL/SQL blocks (e.g., \{}texttt{CREATE PROCEDURE GetEmp(p\{}_id NUMBER) IS BEGIN ... END;}).
- 4. Security information (User accounts, roles, privileges).
- Significance:
- - Enforces data integrity and consistency.

Data Dictionary: Definition & Significance II

- - Acts as a primary reference for the DBMS to parse and optimize queries (e.g., checking if a table exists during a \{}texttt{SELECT} statement).
- - Crucial for database administrators for auditing, performance tuning, and capacity planning.

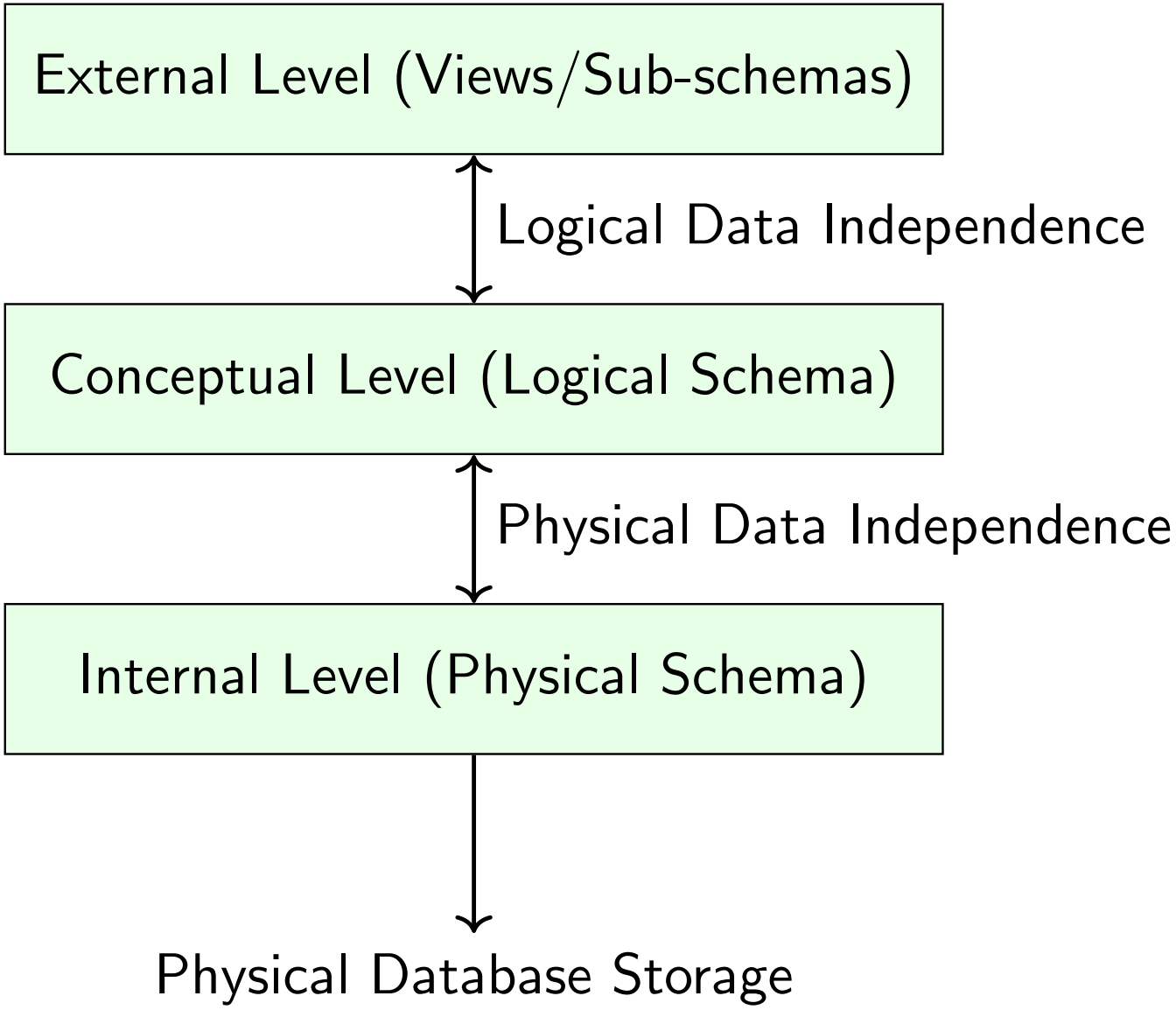
Database Concepts & Characteristics I

- Database: A shared, integrated computer structure that stores a collection of end-user data and metadata.
- Key Characteristics:
- 1. Self-describing nature: A database system contains not only the database itself but also a complete definition (metadata) of the database structure.
- 2. Insulation between programs and data: Program-Data Independence allows changing data structures without changing application programs.
- 3. Data Abstraction: Hides the complex physical storage details from users.
- 4. Multi-user Transaction Processing: Concurrency control mechanisms (like locking) ensure data consistency during simultaneous access.

Database Concepts & Characteristics II

- Importance: Eliminates data redundancy, ensures data consistency, provides data sharing, enforces security restrictions, and supports ACID properties (Atomicity, Consistency, Isolation, Durability).

Three-Schema Architecture I



Database Systems and Environment I

- DBMS Environment Components: Hardware, Software (DBMS, OS, Network software), People (DBA, Database Designers, Application Programmers, End Users), Procedures, and Data.
- Architecture Models: Centralized, Client-Server (2-tier, 3-tier, n-tier), and Distributed architectures.
- Key DBMS Functions:
- - Data Storage, Retrieval, and Update (Core CRUD operations).
- - Transaction Support (ACID properties).
- - Concurrency Control Services (Pessimistic vs Optimistic Locking, Timestamping).
- - Recovery Services (Write-Ahead Logging, Rollbacks).
- - Authorization Services (\{}texttt{GRANT} / \{}texttt{REVOKE} Data Control Language commands).

Schemas, Sub-schemas, and Instances I

- Database Schema: The logical design and overall description of the database. It is specified during database design and is expected to change infrequently. Examples: \{}texttt{CREATE TABLE} statements define the schema.
- Sub-schema (External Schema): A subset of the schema tailored to the needs of a particular user group or application. In RDBMS, this is typically implemented using Views (e.g., \{}texttt{CREATE VIEW Employee\{}_Public AS SELECT Name, Department FROM Employees}).
- Database State/Instance: The actual data stored in the database at a particular moment in time. Also called a snapshot. Every time a record is \{}texttt{INSERT}ed, \{}texttt{UPDATE}d, or \{}texttt{DELETE}d, the instance changes.

Schemas, Sub-schemas, and Instances II

- Summary: The schema is the structural blueprint (intension), while the instance is the actual stored data at a specific point in time (extension).

Agenda I

- Data vs. Information: Definitions and Differences
- Data Elements: Fields, Records, Files, and Metadata
- Data Dictionary: Components and Significance
- Database Concepts: Characteristics and Importance
- Database Systems Environment: Architecture and Components
- Schemas, Sub-schemas, and Instances: ANSI/SPARC Architecture

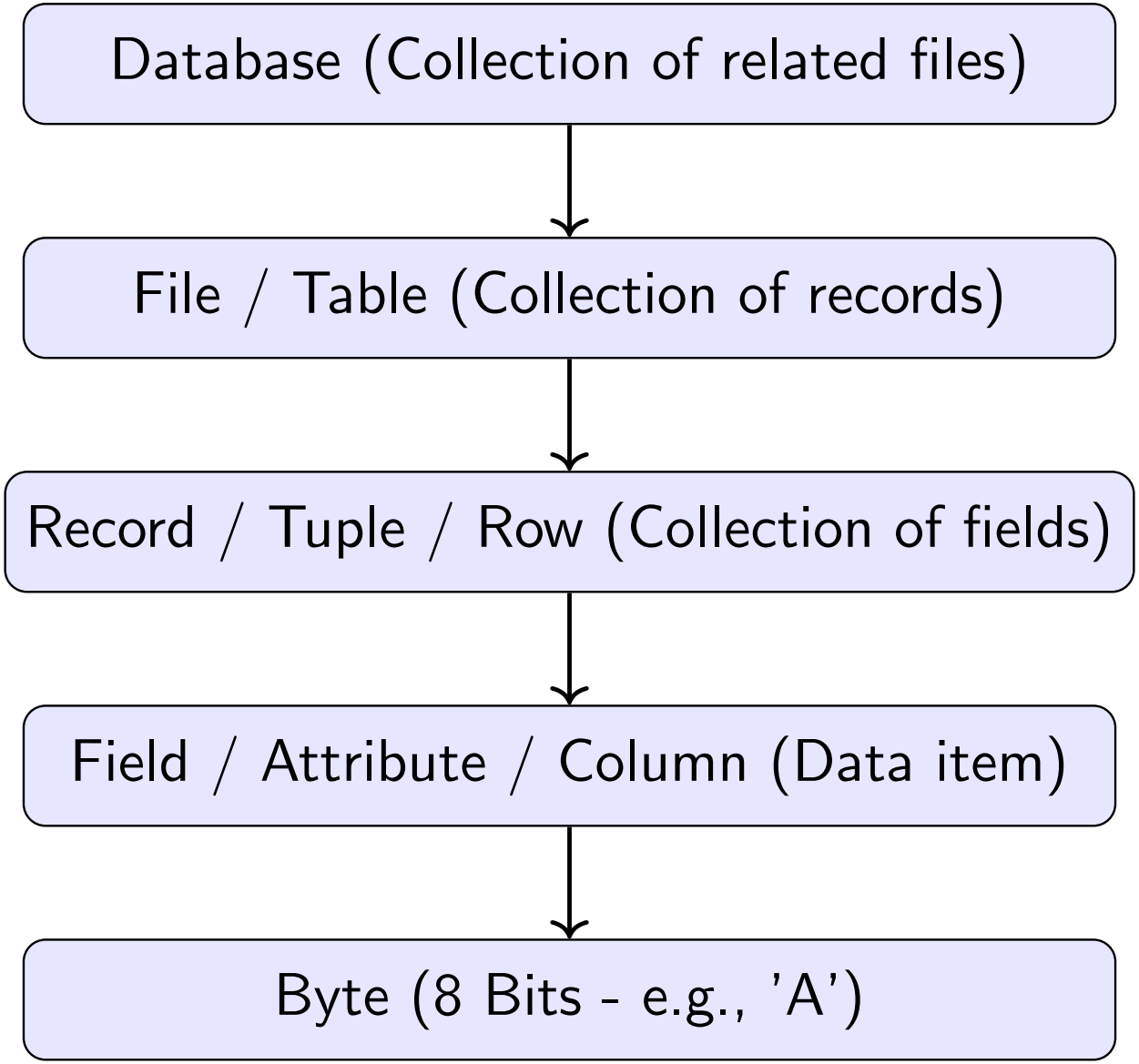
Data and Information I

- **Data**: Raw, unorganized facts, figures, or details that need to be processed. It lacks context and meaning on its own. Example: The integer '85' or the string 'Alice'.
- **Information**: Data that has been processed, organized, structured, or presented in a given context to make it useful. Example: 'Alice scored 85 on her RDBMS exam.'
- **Differences**:
- 1. **Context**: Data has no context; Information is data with context.
- 2. **Dependence**: Information depends on data; Data does not depend on information.
- 3. **Decision Making**: Information drives business decisions and insights; Data is simply the raw material.

Data and Information II

- In an RDBMS, data is stored persistently in relational tables, and information is derived using SQL queries (e.g., SELECT statements using aggregations and filters).

Hierarchy of Data Elements I



Data Dictionary I

- **Definition**: A centralized repository of information about data (metadata) such as its meaning, relationships to other data, origin, usage, and format.
- **Components**:
- - **Data Elements**: Names, descriptions, types, and lengths of attributes.
- - **Relationships**: Integrity constraints, foreign keys, and cardinalities.
- - **Users & Access Rights**: Privileges and roles (e.g., ‘GRANT SELECT ON Employee TO UserA‘).
- **Significance**: Essential for DBMS operation. It ensures data consistency, provides a semantic reference for application developers, and allows the RDBMS query optimizer to generate efficient execution plans.

Data Dictionary II

- **Example**: In Oracle, the Data Dictionary is exposed through system views like ‘USER_TABLES‘, ‘ALL_TAB_COLUMNS‘, and ‘DBA_CONSTRAINTS‘.

Database Concepts I

- **Definition**: A shared, integrated computer structure that houses a collection of logically related data. It consists of end-user data (raw facts) and metadata.
- **Characteristics**:
- - **Self-describing nature**: Contains not only the data itself but also a complete description of the database structure.
- - **Program-Data Independence**: Insulation between applications and the underlying data structures, allowing schema changes without application rewrites.
- - **Multi-user transaction processing**: Concurrency control mechanisms (like locking) ensure safe multi-user access while maintaining ACID (Atomicity, Consistency, Isolation, Durability) properties.

Database Concepts II

- **Importance**: Eliminates uncontrolled data redundancy, ensures data integrity and validation, improves data security, and facilitates unified data sharing across an organization.

Database Systems and Environment I

- **Database System**: A comprehensive organization of components that define and regulate the collection, storage, management, and use of data.
- **Components of the Environment**:
- 1. **Hardware**: Servers, secondary storage devices, networking equipment.
- 2. **Software**: The DBMS (e.g., PostgreSQL, Oracle, MySQL), Operating System, and application programs connecting via drivers like JDBC/ODBC.
- 3. **People**: Database Administrators (DBAs), Application Developers, and End Users.
- 4. **Procedures**: Formal instructions and rules that govern the design, use, and maintenance of the database system (e.g., backup and recovery protocols).

Database Systems and Environment II

- 5. **Data**: The core component acting as the crucial bridge between the machine components (hardware/software) and human components.

Schemas, Sub-schemas, and Instances I

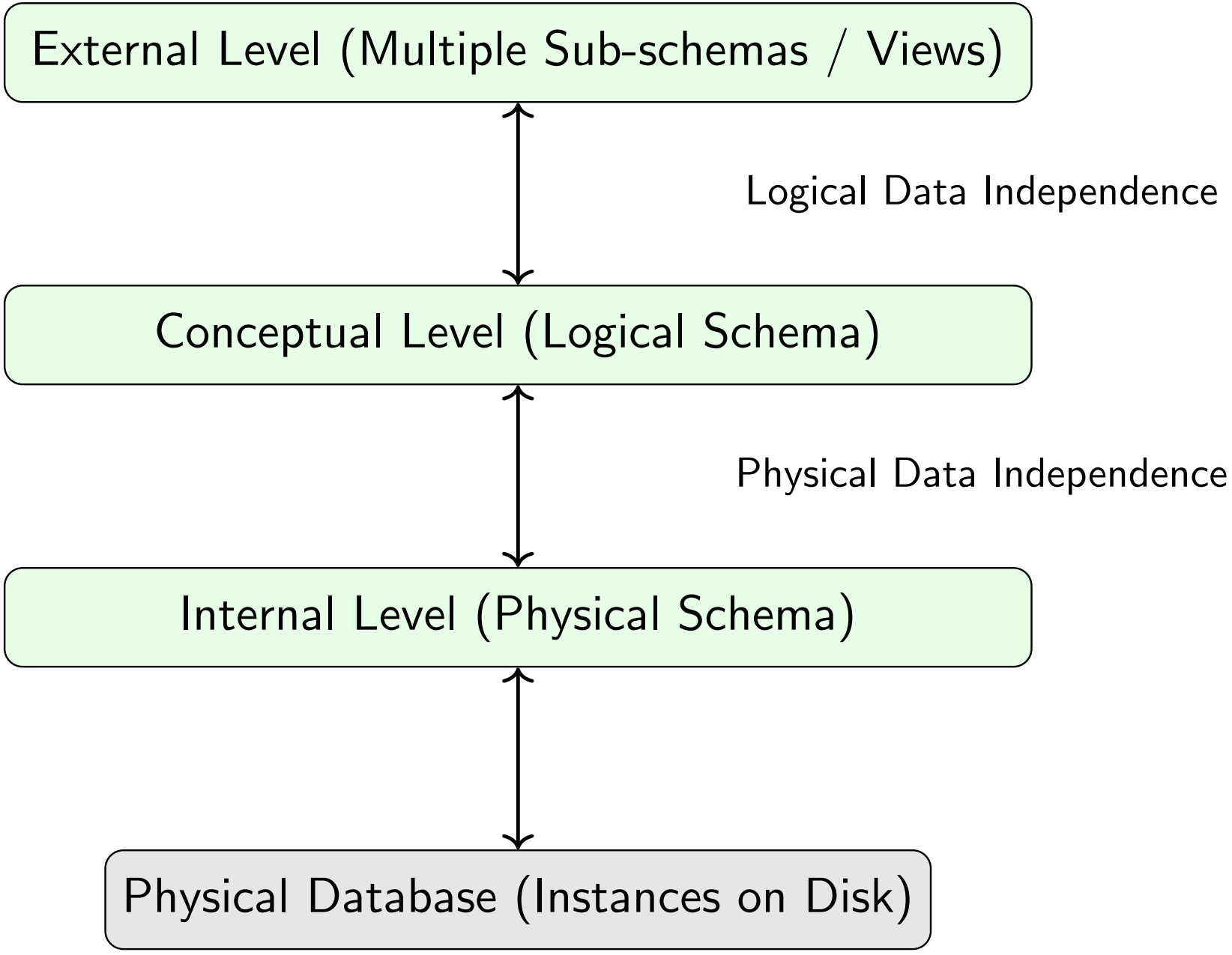
- **Schema**: The overall logical design and description of the database. It is highly static and rarely changes. It is typically specified during the initial database design phase using DDL (Data Definition Language).
- **Sub-schema (External Schema)**: A localized subset of the main schema that is tailored to one or more specific applications or user groups. It hides irrelevant data and provides customized perspectives (often implemented using ‘CREATE VIEW‘).
- **Instance (State)**: The actual data stored in the database at a specific moment in time. Unlike the schema, the instance changes frequently with every ‘INSERT‘, ‘UPDATE‘, or ‘DELETE‘ (DML) operation.

Schemas, Sub-schemas, and Instances II

- **Data Independence**: The ability to modify a schema definition in one level of the database architecture without affecting a schema definition in the next higher level.

ANSI/SPARC 3-Schema Architecture I

ANSI/SPARC 3-Schema Architecture II



Summary & Practical Metadata Query I

- **Summary**: We explored the transformation of Data into Information, the structural hierarchy of Data Elements, the necessity of a Data Dictionary, the components of a DBMS Environment, and the 3-Schema Architecture.
- **Practical Application (SQL)**:
- To explore the instance of metadata (Data Dictionary) in an Oracle database, a DBA or Developer can execute:
- “‘sql
- SELECT table_name, num_rows, last_analyzed
- FROM all_tables
- WHERE owner = ’HR’;
- “‘
- This query dynamically retrieves the names of tables (schema), the number of records (instances), and optimization statistics directly from the RDBMS Data Dictionary.

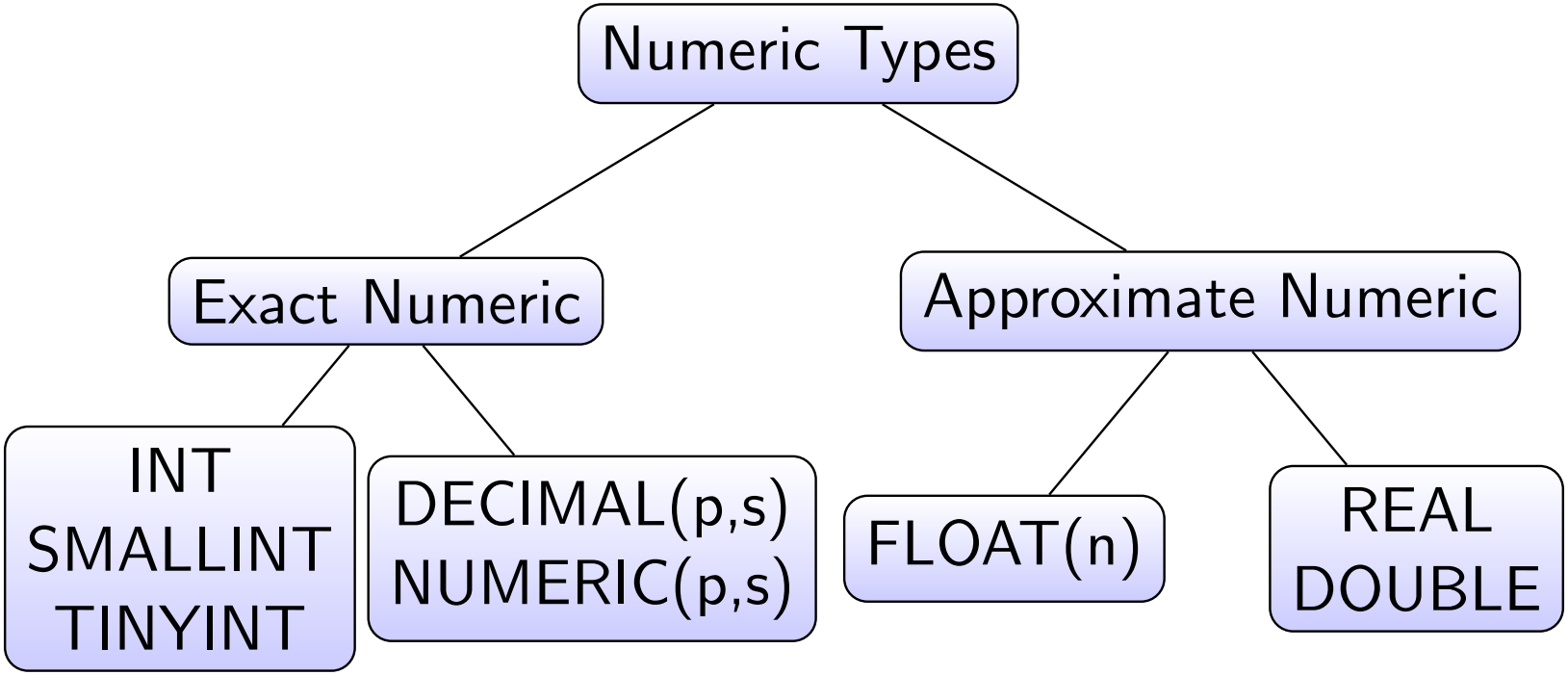
Agenda I

- Overview of SQL Data Types
- Numeric Data Types
- Character String Data Types
- Date and Time Data Types
- Best Practices in Selecting Data Types

Overview of SQL Data Types I

- A data type in SQL defines the kind of value that a column can hold: integer data, character data, monetary data, date and time data, binary strings, etc.
- Every column, local variable, expression, and parameter has a related data type.
- SQL standardizes many data types, but specific RDBMS vendors (Oracle, MySQL, PostgreSQL, SQL Server) often include their own extensions.
- Broad categories include:
- - Numeric types (Exact and Approximate)
- - Character and String types (Fixed-length, Variable-length)
- - Date and Time types
- - Binary data types (e.g., BLOBs)

Hierarchy of Numeric Data Types I



Numeric Data Types Details I

- Exact Numeric Types store values exactly as specified. Examples:
- - INT: Standard integer, typically 4 bytes (e.g., -2,147,483,648 to 2,147,483,647).
- - DECIMAL(p, s) or NUMERIC(p, s): Fixed precision and scale. 'p' is total digits, 's' is digits after decimal. E.g., DECIMAL(5,2) can store 123.45.
- Approximate Numeric Types store floating-point data, useful for scientific calculations where exact precision isn't critical.
- - FLOAT and REAL: Adhere to IEEE 754 standard.
- Important: Never use approximate types for financial data! Always use DECIMAL or NUMERIC to prevent rounding errors.

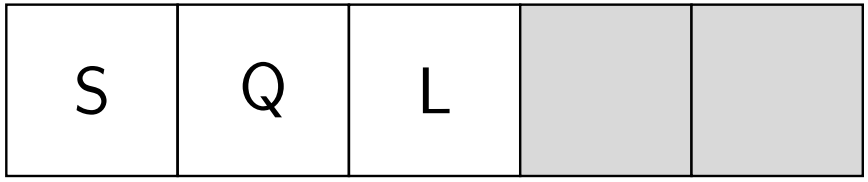
Character String Data Types I

- Used to store text. Choosing between fixed and variable length is crucial for performance and storage.
- CHAR(n): Fixed-length character string. Padded with spaces if the input is shorter than 'n'. Good for fixed-length data like MD5 hashes, ISO country codes (e.g., 'US', 'IN').
- VARCHAR(n) / VARCHAR2(n): Variable-length character string. Only takes up as much space as the actual string plus 1 or 2 bytes for length metadata.
- TEXT / CLOB (Character Large Object): Used for storing massive amounts of text (e.g., full articles). Usually not stored directly in the row data pages.

Storage Allocation: CHAR vs VARCHAR I

Input: 'SQL'

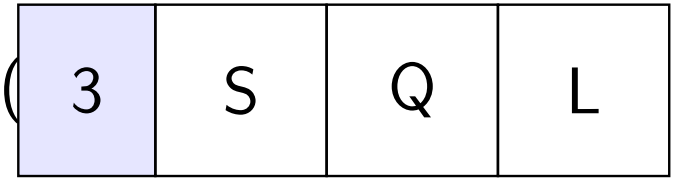
CHAR(5):



(Padded with spaces)

Length

VARCHAR(



(No padding, saves space)

Date and Time Data Types I

- Handle temporal data. Standard formats follow ISO 8601 (YYYY-MM-DD).
- DATE: Stores date only (Year, Month, Day). Example: '2023-10-27'.
- TIME: Stores time of day only (Hour, Minute, Second, Fractions). Example: '14:30:00.000'.
- DATETIME / TIMESTAMP: Stores both date and time.
- TIMESTAMP WITH TIME ZONE: Crucial for global applications. Stores the UTC offset along with the timestamp.
- SQL functions like CURRENT_DATE, CURRENT_TIMESTAMP, and EXTRACT() are used to manipulate these types.

Specialized Data Types I

- BOOLEAN: Represents truth values (TRUE, FALSE, UNKNOWN). Note: Oracle traditionally lacks a native BOOLEAN type in SQL, often using NUMBER(1) or CHAR(1) with 'Y'/'N'.
- BINARY/VARBINARY: Stores binary strings (e.g., images, compiled code).
- JSON/XML: Modern RDBMS (PostgreSQL, MySQL, SQL Server) support native JSON and XML data types.
- Benefits of JSON/XML types:
- - Validation of document structure upon insertion.
- - Optimized binary storage format.
- - Built-in indexing (e.g., GIN indexes in PostgreSQL for JSONB) for fast querying of document attributes.

Best Practices in Selecting Data Types I

- 1. Use the smallest data type that can reliably contain all possible values (e.g., TINYINT instead of INT if values are 0-255). Reduces memory and disk I/O.
- 2. Avoid using strings for dates or numbers. It prevents utilizing native date/math functions and breaks natural sorting (e.g., '10' sorts before '2' as a string).
- 3. Use VARCHAR over CHAR unless strings are almost always the exact same length.
- 4. Be consistent across tables (e.g., if customer_id is INT in the customers table, it must be INT in the orders table for optimal JOIN performance).

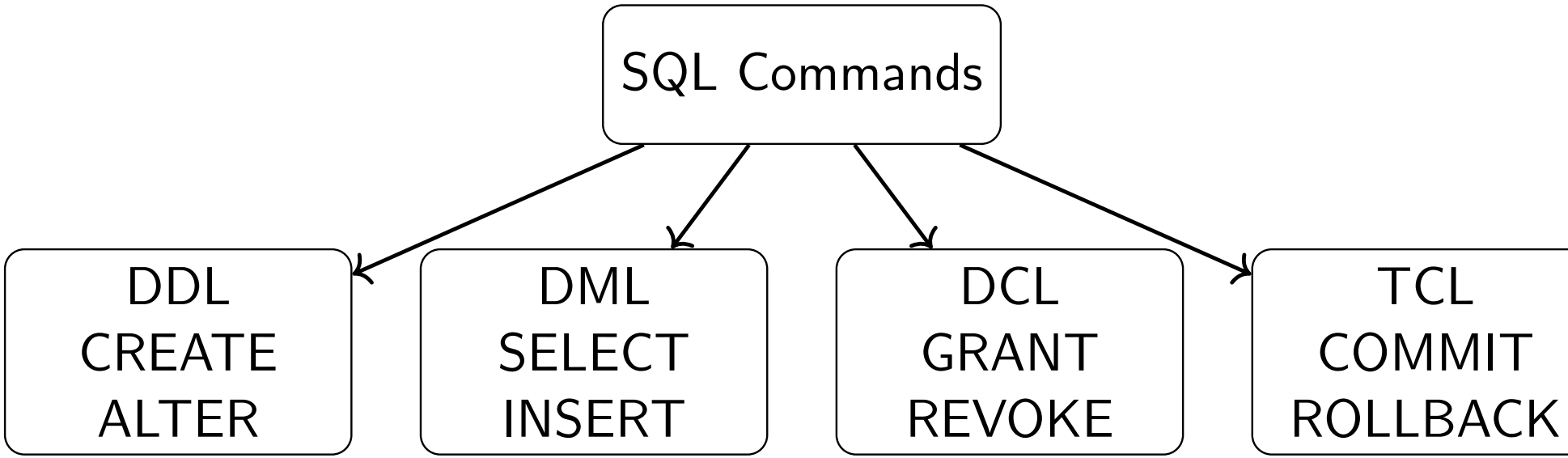
Agenda I

- Introduction to SQL Commands
- Categories of SQL Commands
- DDL: CREATE Database and Table
- DDL: ALTER Table
- DDL: DROP Table
- DDL: TRUNCATE Table
- DELETE vs DROP vs TRUNCATE

Introduction to SQL and DDL I

- Structured Query Language (SQL) is the standard language for relational database management systems.
- Data Definition Language (DDL) consists of SQL commands used to define and manage the database schema.
- DDL commands are typically auto-committed, meaning they permanently save all structural changes in the database.
- Key DDL commands include CREATE, ALTER, DROP, and TRUNCATE.
- These commands operate on database objects like databases, tables, indexes, views, and schemas rather than the actual data rows.

Categories of SQL Commands I



DDL: CREATE Command I

- The CREATE command is used to instantiate new databases, tables, views, or other database objects.
- Syntax to create a database: \{}texttt{CREATE DATABASE database\{}_name;}
- Syntax to create a table: \{}texttt{CREATE TABLE table\{}_name (column1 datatype constraints, ...);}
- Example:\{}\{} \{}texttt{CREATE TABLE Employees (}\{}\{} \{}texttt{ EmpID INT PRIMARY KEY,}\{}\{} \{}texttt{ FirstName VARCHAR(50) NOT NULL,}\{}\{} \{}texttt{ HireDate DATE}\{}\{} \{}texttt{);}
- This defines a new table structure along with domain constraints.

Relational Table Structure I

EmpID (PK)	FirstName	HireDate
101	John	2023-01-15
102	Jane	2023-02-20

DDL: ALTER TABLE Command I

- The ALTER TABLE command modifies the schema of an existing table without dropping it.
- It supports adding, modifying, or dropping columns and constraints.
- Add a column: \{}texttt{ALTER TABLE Employees ADD Email VARCHAR(100);}
- Modify a column datatype (syntax varies by RDBMS): \{}texttt{ALTER TABLE Employees MODIFY Email VARCHAR(255);}
- Drop a column: \{}texttt{ALTER TABLE Employees DROP COLUMN Email;}
- Add a constraint: \{}texttt{ALTER TABLE Employees ADD CONSTRAINT fk\{}_dept FOREIGN KEY (DeptID) REFERENCES Departments(DeptID);}

DDL: DROP Command I

- The DROP command permanently removes objects from the database.
- Dropping a table deletes its structure, data, indexes, permissions, and triggers.
- Syntax: \{}texttt{DROP TABLE table\{}_name;}
- Example: \{}texttt{DROP TABLE Employees;}
- To drop an entire database: \{}texttt{DROP DATABASE database\{}_name;}
- Caution: Unlike DML operations, a DROP command cannot be rolled back in most RDBMS unless executed within specific transactional contexts.

DDL: TRUNCATE Command I

- The TRUNCATE TABLE command deletes all rows from a table rapidly while keeping the schema intact.
- Syntax: \{}texttt{TRUNCATE TABLE table\{}_name;}
- TRUNCATE is a DDL command because it operates by deallocating the data pages used to store the table data, rather than logging individual row deletions.
- TRUNCATE resets any identity (auto-increment) columns back to their initial seed value.
- It is generally faster than a DELETE without a WHERE clause but cannot be used on tables referenced by foreign keys.

DELETE vs DROP vs TRUNCATE I

- DELETE is a DML command; it selectively removes rows based on a WHERE clause, logs each deletion, and fires triggers. It can be rolled back.
- DROP is a DDL command; it completely destroys the table structure, data, and associated objects.
- TRUNCATE is a DDL command; it removes all data by deallocating pages, preserves the structure, and cannot be rolled back in some RDBMS.
- Best Practice: Use TRUNCATE to clear an entire table efficiently. Use DELETE for conditional data removal. Use DROP to clean up obsolete schemas.

Agenda I

- Introduction to Data Definition Language (DDL)
- CREATE DATABASE: Initializing logical storage
- CREATE TABLE: Defining schemas, data types, and constraints
- ALTER TABLE: Modifying existing structures dynamically
- DROP TABLE: Removing structures and data completely
- TRUNCATE TABLE: Fast data deletion mechanism
- Comparative Analysis of DDL Commands

Introduction to DDL I

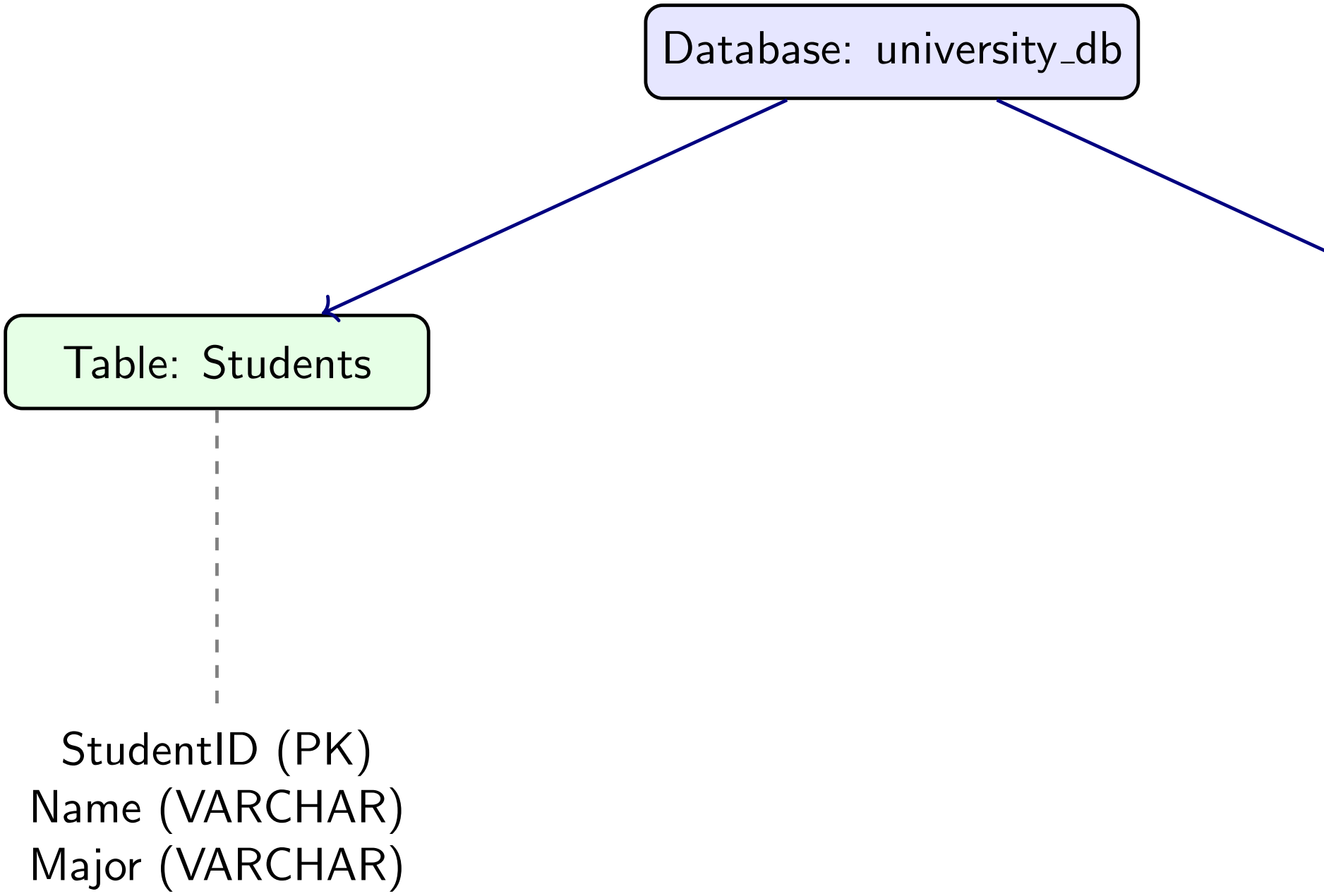
- Data Definition Language (DDL) is a core subset of SQL used to define and manage all logical structures in a database schema.
- DDL commands are typically auto-committed in most RDBMS (like Oracle, MySQL). This means they permanently modify the database catalog (Data Dictionary) without requiring an explicit COMMIT transaction.
- Key objects managed by DDL include databases, tables, indexes, views, sequences, and synonyms.
- Unlike DML (Data Manipulation Language) which handles data row mutations, DDL strictly focuses on the metadata and container schema.
- A typical PL/SQL block executing DDL dynamically requires the EXECUTE IMMEDIATE statement: \{}texttt{EXECUTE IMMEDIATE 'TRUNCATE TABLE employees';}

CREATE DATABASE and CREATE TABLE I

- The \{}texttt{CREATE DATABASE} statement initializes a new database, allocating system files (data files and log files).
- Syntax: \{}texttt{CREATE DATABASE university\{}_db;}
- The \{}texttt{CREATE TABLE} statement defines a new table, its columns, data types, and physical attributes (e.g., TABLESPACE in Oracle).
- Constraints like PRIMARY KEY, FOREIGN KEY, UNIQUE, and CHECK enforce Entity Integrity and Referential Integrity.
- Example:\{}\{} \{}texttt{CREATE TABLE Students (\{}\{} \{}phantom{xx}StudentID INT PRIMARY KEY,\{}\{} \{}phantom{xx}FirstName VARCHAR(50) NOT NULL,\{}\{} \{}phantom{xx}Email VARCHAR(100) UNIQUE,\{}\{} \{}phantom{xx}EnrollmentDate DATE DEFAULT CURRENT\{}_DATE\{}\{});}

Database Schema Diagram (TikZ) I

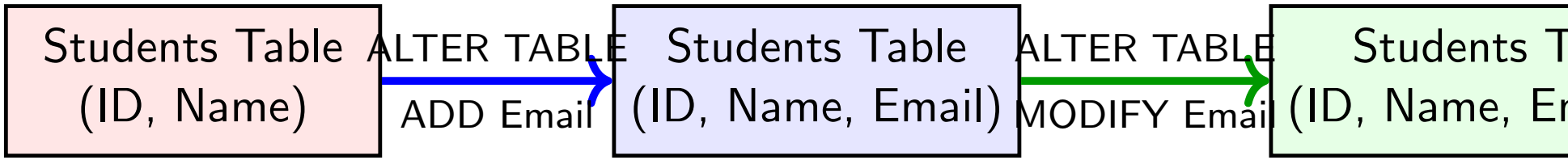
Database Schema Diagram (TikZ) II



ALTER TABLE Command I

- The \{}texttt{ALTER TABLE} statement allows database engineers to add, delete, or modify columns in an existing table without dropping it.
- It is strictly an online schema change (in modern RDBMS), allowing tables to be altered while mitigating table locking duration.
- Add Column: \{}texttt{ALTER TABLE Students ADD DateOfBirth DATE;}
- Modify Column Type (MySQL/Oracle): \{}texttt{ALTER TABLE Students MODIFY Email VARCHAR(255);}
- Alter Column Type (PostgreSQL/SQL Server): \{}texttt{ALTER TABLE Students ALTER COLUMN Email TYPE VARCHAR(255);}
- Add Constraint: \{}texttt{ALTER TABLE Students ADD CONSTRAINT chk\{}_credits CHECK (Credits ¿ 0);}

ALTER TABLE Transformations (TikZ) I



DROP TABLE Command I

- The \{}texttt{DROP TABLE} command is a destructive DDL operation that removes a table definition and all of its associated data, indexes, triggers, constraints, and permission specifications.
- Syntax: \{}texttt{DROP TABLE Students;}
- Storage deallocation: Space used by the table in the tablespace is freed immediately.
- Referential Integrity Blocks: If the table is referenced by a FOREIGN KEY constraint in a child table, the DROP statement will fail with an integrity constraint violation.
- To force drop including dependencies: \{}texttt{DROP TABLE Students CASCADE CONSTRAINTS;} (Oracle) or \{}texttt{DROP TABLE Students CASCADE;} (PostgreSQL).

TRUNCATE TABLE Command I

- The \{}texttt{TRUNCATE TABLE} command physically removes all rows from a table, but the table schema, columns, constraints, and indexes remain intact.
- Syntax: \{}texttt{TRUNCATE TABLE Students;}
- Performance: It is significantly faster than \{}texttt{DELETE} because it operates by deallocating the database data pages used to store the table data, rather than logging individual row deletions in the transaction log.
- Constraints: TRUNCATE cannot be executed on a table referenced by an active FOREIGN KEY constraint.
- Triggers: TRUNCATE bypasses row-level \{}texttt{DELETE} triggers.

TRUNCATE TABLE Command II

- Sequences: Identity columns (auto-increment) are typically reset to their initial seed value when truncated (e.g., \{}texttt{TRUNCATE TABLE t RESTART IDENTITY} in Postgres).

Summary & Best Practices I

- DDL commands (CREATE, ALTER, DROP, TRUNCATE) strictly define and manage the metadata and schemas of the RDBMS.
- DDL operations are usually auto-committed, rendering them irreversible without point-in-time recovery (PITR) or flashback technologies.
- For bulk data clearing, prefer \{}texttt{TRUNCATE} over \{}texttt{DELETE} to prevent massive transaction log bloat and performance degradation.
- When designing schemas via \{}texttt{CREATE TABLE}, proactively define foreign keys and constraints to ensure ACID properties and relational integrity.
- Always execute \{}texttt{DROP} commands with caution; use \{}texttt{DROP IF EXISTS} to prevent script execution errors in CI/CD database migration pipelines.

Agenda I

1. DML Concepts & ACID Transaction Context
2. The INSERT Statement (Single, Multi-row, Subqueries)
3. The SELECT Statement (Projection & Selection)
4. Advanced SELECT (Sorting, Filtering, Aggregation)
5. The UPDATE Statement & Best Practices
6. The DELETE Statement (vs TRUNCATE)
7. Architectural Flow of DML Execution
8. SQL Command Categorization

DML Concepts & Execution Context I

- Data Manipulation Language (DML) commands form the core operational vocabulary of SQL, enabling users to interact directly with the tuple data persisted inside tables.
- Unlike Data Definition Language (DDL) which auto-commits in many RDBMS, DML operations are deeply intertwined with the ACID properties (Atomicity, Consistency, Isolation, Durability).
- DML commands operate within the boundaries of Database Transactions. They place locks on rows or pages to maintain isolation levels.
- Modifications made by INSERT, UPDATE, and DELETE are staged in transaction logs (like WAL in PostgreSQL or Redo logs in Oracle) and are not globally visible until a COMMIT is issued.

DML Concepts & Execution Context II

- If a constraint violation occurs or a ROLLBACK is triggered, the RDBMS engine reverses the DML effects, ensuring Atomicity.

The INSERT Statement in Depth I

- The INSERT statement appends new records to a table's data pages.
- Basic standard syntax: INSERT INTO table_name (col1, col2) VALUES (val1, val2);
- When dealing with identity columns or AUTO_INCREMENT primary keys, omit the key column from the insert list; the database engine will auto-generate the sequence.
- Bulk Inserts: To minimize network latency and transaction log overhead, group inserts: INSERT INTO users (id, role) VALUES (1, 'admin'), (2, 'user'), (3, 'user');
- Data Replication: The INSERT ... SELECT paradigm allows moving massive datasets entirely server-side: INSERT INTO archive_log (user_id) SELECT user_id FROM active_log WHERE created_at ¡ '2022-01-01';

The SELECT Statement: Projection & Selection I

- SELECT is the declarative means of retrieving data, representing the Relational Algebra operations of Projection (choosing columns) and Selection (filtering rows).
- Projection: While 'SELECT *' is valid, engineering best practice requires explicit column enumeration (SELECT id, username FROM users) to prevent application breakage if schemas evolve.
- Selection (Filtering): The WHERE clause applies boolean predicates. Engine optimizers use this to perform Index Seeks rather than Full Table Scans.
- Predicate Evaluation: Conditions like WHERE status = 'ACTIVE' AND login_count ¿ 10 are evaluated. The optimizer determines the most efficient access path, utilizing B-Tree or Hash indexes.

The SELECT Statement: Projection & Selection II

- NULL Handling: NULL represents unknown. Standard comparisons (=, !=) fail with NULL. Use IS NULL or IS NOT NULL instead.

Advanced SELECT: Aggregation & Sorting I

- Relational databases return unsorted sets by default. To enforce determinism, ORDER BY is mandatory.
- Syntax: ORDER BY column1 ASC, column2 DESC. Sorting is expensive; if sorting by unindexed columns, the DB engine performs an in-memory or on-disk 'filesort'.
- Aggregation allows summarizing large datasets: COUNT(), SUM(), AVG(), MIN(), MAX().
- Grouping context is provided by the GROUP BY clause, which partitions the table data into subsets before applying aggregates.
- HAVING Clause: While WHERE filters individual rows before aggregation, HAVING filters the grouped results after aggregate calculation. Example: SELECT dept_id, COUNT(*) FROM employees GROUP BY dept_id HAVING COUNT(*) ¿ 50;

The UPDATE Statement & Constraints I

- The UPDATE statement modifies existing data in place. Syntax: UPDATE table_name SET col1 = val1, col2 = val2 WHERE condition;
- Safety Warning: Executing an UPDATE without a WHERE clause results in a catastrophic bulk modification of every row in the table. Always run a SELECT with the intended WHERE clause first.
- Under the hood, an UPDATE is often implemented logically as a DELETE followed by an INSERT in the transaction log, maintaining undo/redo capabilities.
- Concurrency Control: When a row is updated, the RDBMS places an exclusive row-level lock (e.g., in InnoDB). Concurrent transactions attempting to modify the same row will block until the first transaction commits.

The UPDATE Statement & Constraints II

- Performance constraint: Updating indexed columns forces the database engine to rebuild or modify the associated index trees (B-Trees), adding I/O cost.

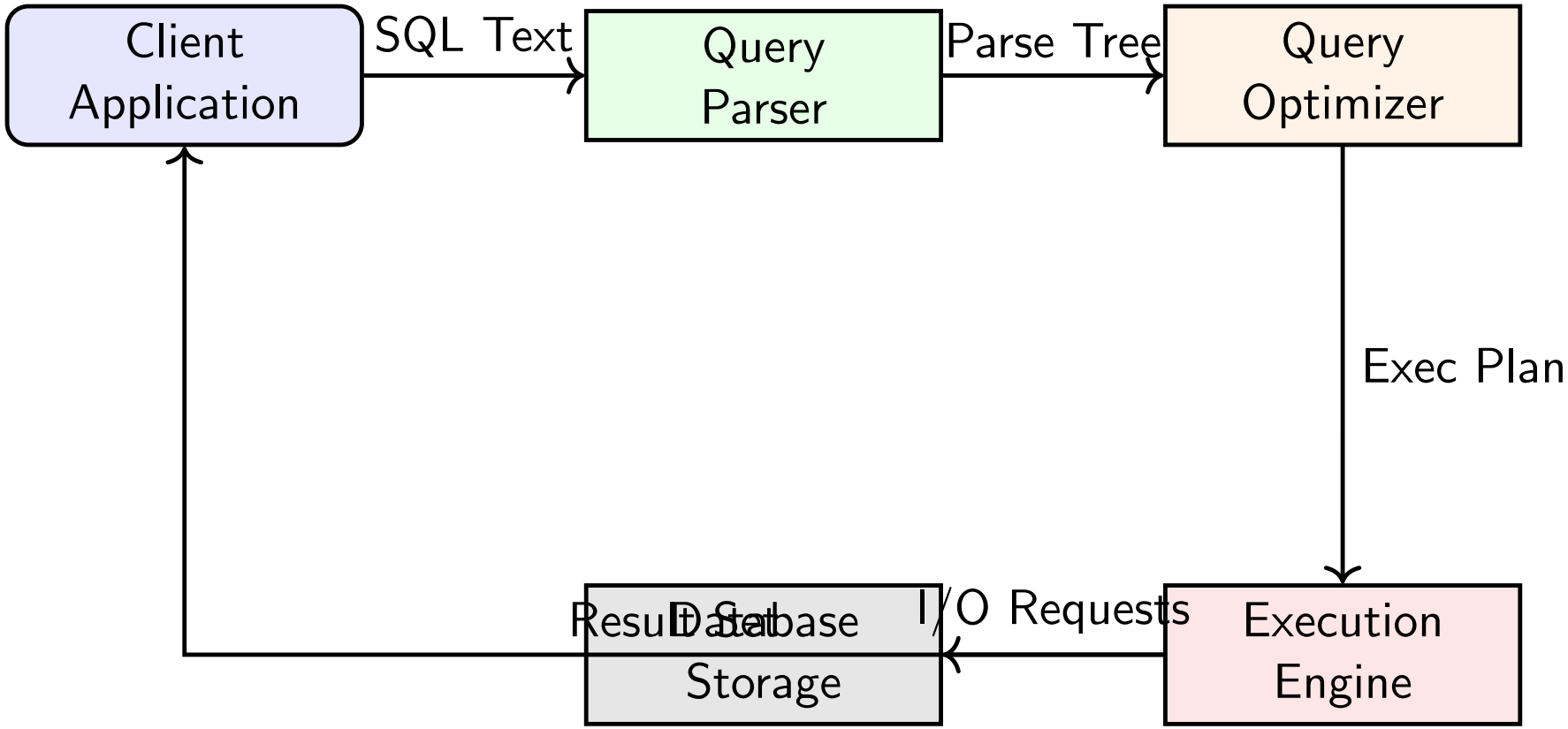
The DELETE Statement & Referential Integrity I

- The DELETE statement removes tuples from a relation. Syntax: DELETE FROM table_name WHERE condition;
- Like UPDATE, omitting the WHERE clause causes a full table purge. However, DELETE evaluates row by row, firing triggers and checking constraints, which is resource-intensive.
- Referential Integrity: Foreign Key (FK) constraints safeguard relational links. Attempting to DELETE a primary key referenced elsewhere will raise a constraint violation unless ON DELETE CASCADE is configured.
- Soft Deletes: Modern engineering often avoids physical DELETEs. Instead, a boolean column 'is_deleted' or timestamp 'deleted_at' is UPDATEd. This preserves historical data and audit trails.

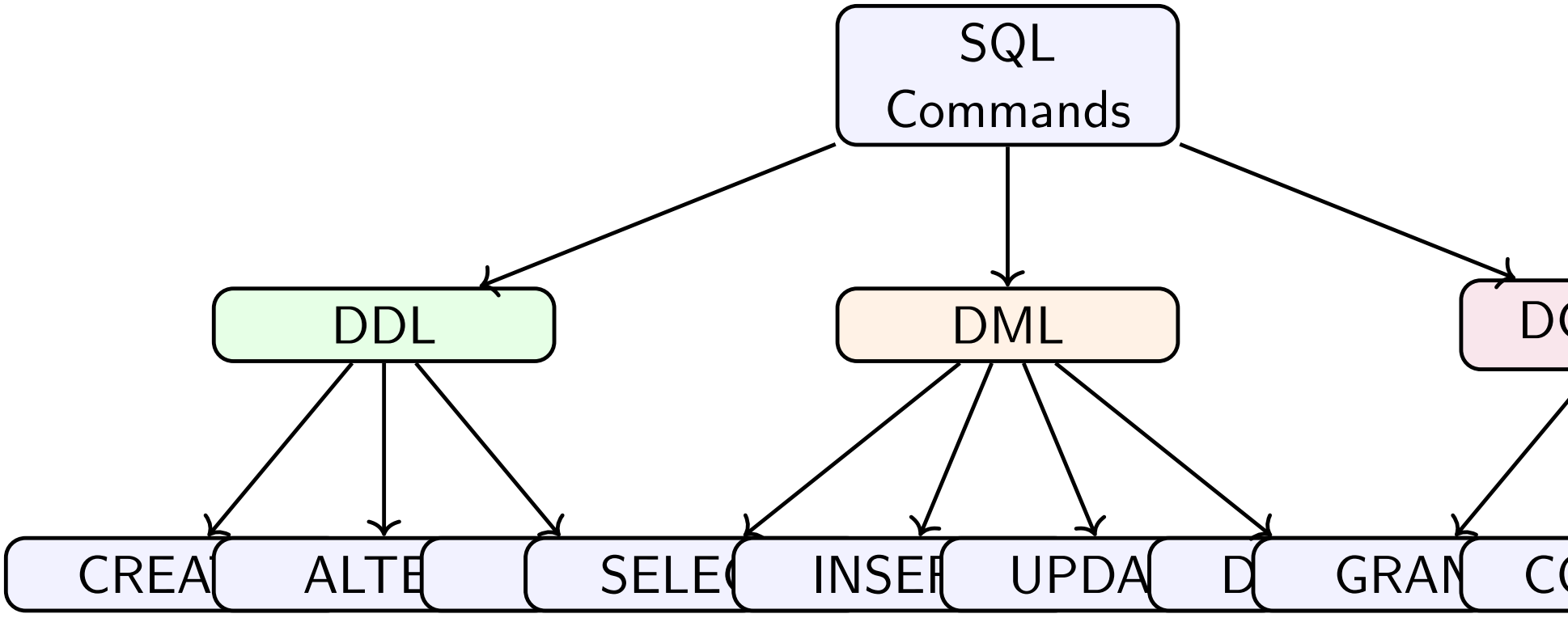
The DELETE Statement & Referential Integrity II

- TRUNCATE vs DELETE: TRUNCATE is DDL, simply resetting the high-water mark of table storage and deallocating pages. It is infinitely faster than DELETE but cannot be filtered via WHERE.

Architectural Flow of DML Execution I



SQL Command Categorization I



Agenda I

- Overview of Data Query Language (DQL)
- The Anatomy of a SELECT Statement
- Filtering Data with the WHERE Clause
- Operators used in the WHERE Clause
- Eliminating Duplicates with SELECT DISTINCT

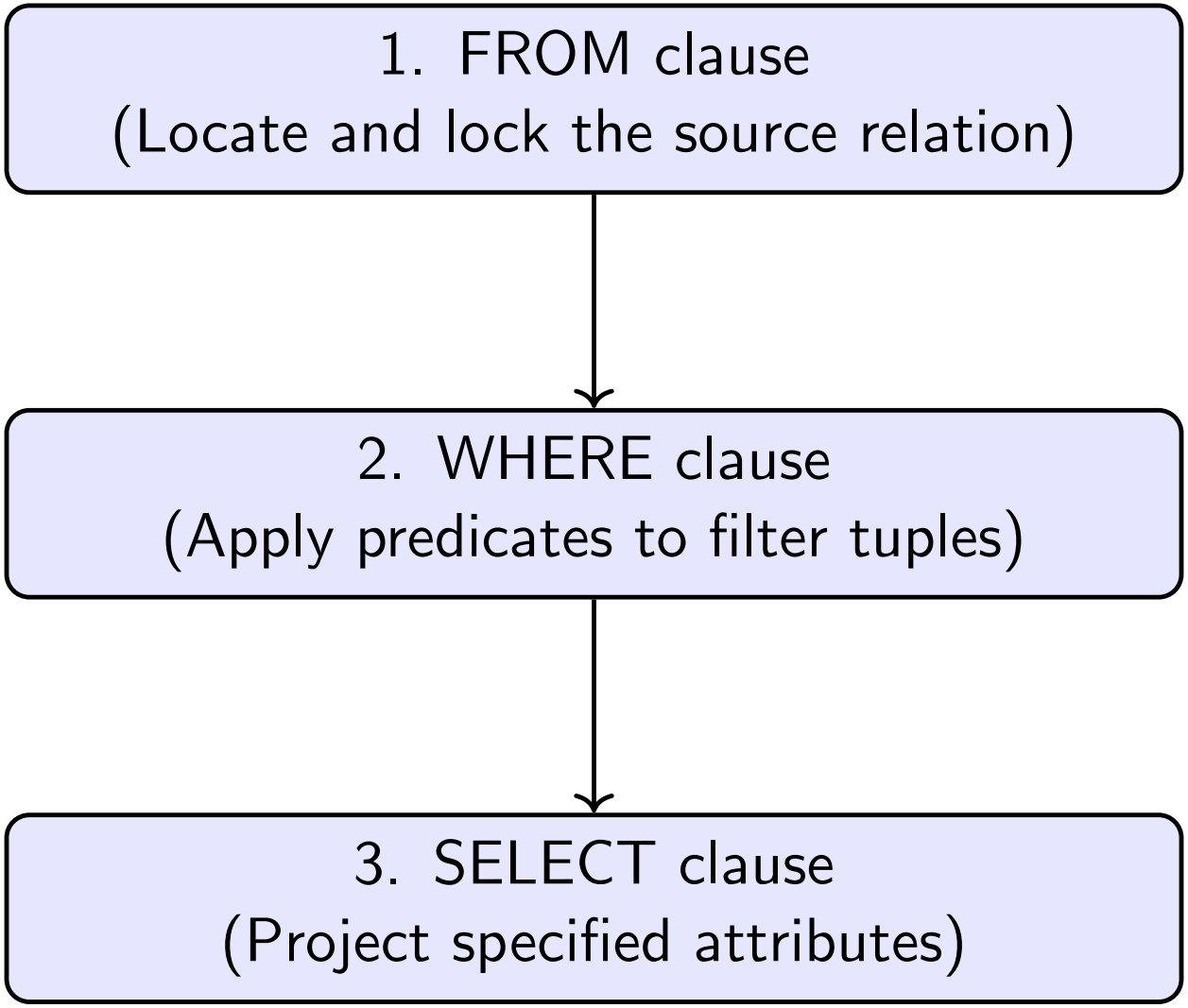
Introduction to DQL I

- Data Query Language (DQL) is a subset of SQL primarily focused on retrieving data from a relational database.
- The fundamental, and often considered the only, command in DQL is the SELECT statement.
- DQL does not modify the underlying data or schema; it only queries the existing state and projects it into a result set.
- The result set of a DQL query is essentially a derived table, demonstrating the principle of relational closure: operations on relations produce relations.
- A typical query engine parses the DQL statement, optimizes it using cost-based optimizers (CBO), and executes the query plan to fetch requested data blocks from storage.

The SELECT Statement I

- The SELECT statement is used to project specific attributes (columns) from one or more relations (tables).
- Basic Syntax: SELECT column1, column2 FROM table_name;
- To select all attributes from a relation, use the asterisk (*) wildcard: SELECT * FROM Employees;
- Column Aliases: You can rename columns in the result set using the AS keyword: SELECT emp_id AS EmployeeID, salary * 12 AS AnnualSalary FROM Employees;
- Expressions: The SELECT list can contain not only column names but also literal values, arithmetic expressions, and built-in scalar functions.
- Dual Table: In systems like Oracle, if you need to evaluate an expression without querying an actual table, you can select from the system-provided DUAL table.

Logical Execution Order of a Basic SELECT Query I



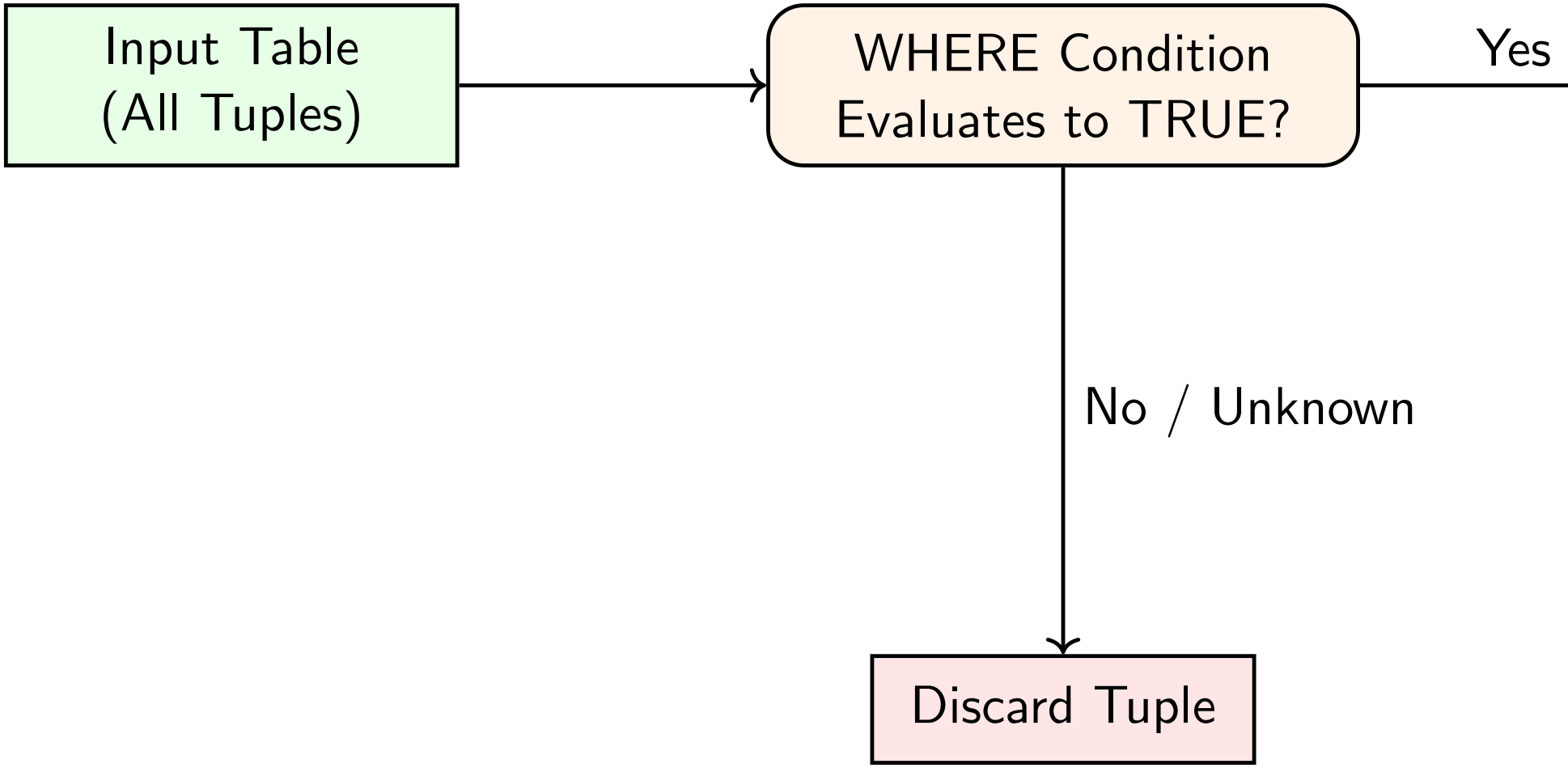
The WHERE Clause I

- The WHERE clause is used to filter records. It extracts only those records that fulfill a specified condition (predicate).
- Syntax: SELECT column1, column2 FROM table_name WHERE condition;
- The condition evaluates to a boolean state: TRUE, FALSE, or UNKNOWN (in the case of NULL values, demonstrating SQL's three-valued logic).
- Only tuples for which the WHERE condition evaluates to TRUE are included in the subsequent phases of query processing.
- The WHERE clause can significantly reduce the amount of data transferred over the network and processed by the database engine.

The WHERE Clause II

- Performance Tip: Conditions in the WHERE clause are heavily utilized by the query optimizer to choose appropriate access paths (e.g., Index Scan vs. Full Table Scan).

Filtering Process using WHERE I



Filtering Operators in WHERE Clause I

- Comparison Operators: =, !=, ¡¿, ¡, ¿, ¡=, ¿= (e.g., SELECT * FROM Employees WHERE salary ¿ 50000;)
- Logical Operators: AND, OR, NOT allow combining multiple predicates. Operator precedence applies (NOT ¿ AND ¿ OR).
- BETWEEN Operator: Selects values within a given inclusive range. Example: WHERE age BETWEEN 25 AND 35;
- IN Operator: Allows specifying multiple values in a WHERE clause, acting as a shorthand for multiple OR conditions. Example: WHERE department_id IN (10, 20, 30);
- LIKE Operator: Used for pattern matching with wildcards '%' (zero, one, or multiple characters) and '_' (single character). Example: WHERE last_name LIKE 'S%';

The SELECT DISTINCT Statement I

- In relational algebra, relations are sets, meaning they naturally contain no duplicate tuples. However, standard SQL query results are multisets (bags) by default.
- To force relational (set-like) behavior and eliminate duplicate rows from the result set, use the DISTINCT keyword.
- Syntax: SELECT DISTINCT column1, column2 FROM table_name;
- The DISTINCT keyword operates on the entire tuple defined by the SELECT list, not just individual columns. A row is considered a duplicate only if all selected columns have matching values.
- Under the hood, implementing DISTINCT often requires an expensive sorting or hashing operation by the execution engine to identify and remove duplicates.

The SELECT DISTINCT Statement II

- Example: SELECT DISTINCT department_id FROM Employees; (Returns a unique list of department IDs where employees are currently assigned).

Summary I

- DQL consists primarily of the SELECT statement, used to retrieve and project data from relational tables without mutating the data.
- The SELECT clause defines the projection (columns), while the FROM clause defines the source relation.
- The WHERE clause defines the selection (rows) using boolean predicates consisting of comparison, logical, and specialized operators.
- SQL operates using three-valued logic (TRUE, FALSE, UNKNOWN) due to the presence of NULLs, heavily impacting WHERE clause evaluations.
- The DISTINCT keyword alters the default multiset semantics of SQL to set semantics, ensuring the result set contains only unique tuples.

Agenda I

- 1. Introduction to Data Query Language (DQL)
- 2. The SELECT Statement Anatomy
- 3. Relational Projection vs Selection
- 4. Filtering Data with the WHERE Clause
- 5. Complex Filtering (AND, OR, NOT, IN, BETWEEN)
- 6. Eliminating Duplicates with SELECT DISTINCT

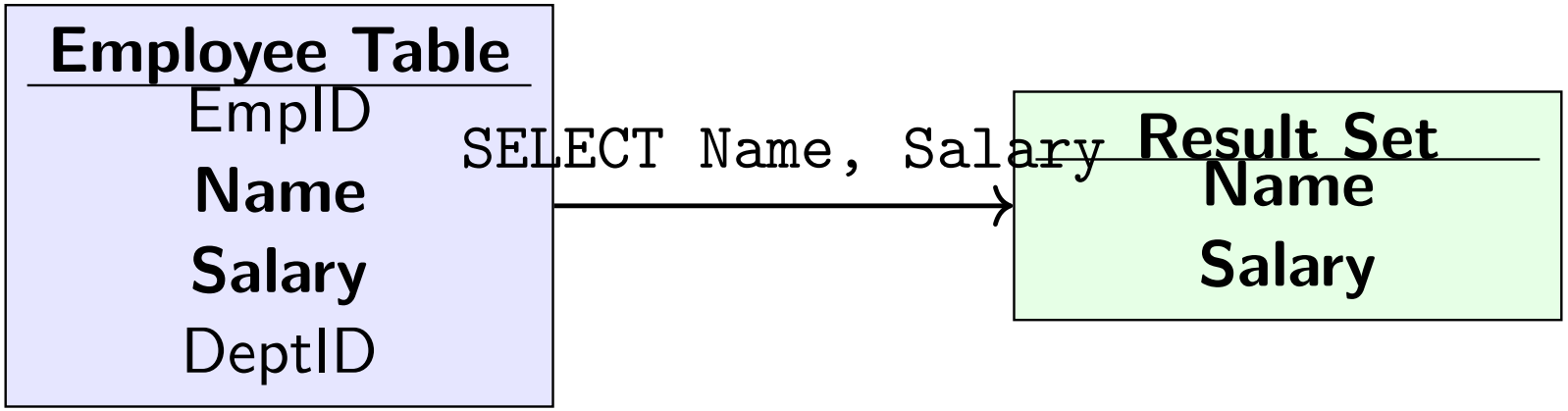
Introduction to Data Query Language (DQL) I

- DQL is a subset of SQL focused exclusively on data retrieval.
- It allows users to query databases to extract information without modifying the underlying structures or data (unlike DDL and DML).
- The primary command in DQL is the SELECT statement, which forms the basis for all data retrieval operations.
- DQL operations logically map to Relational Algebra operations, primarily Projection (pi) and Selection (sigma).
- Result sets are always returned in the form of a logical table or relation, which can be further queried, filtered, or joined.

The Core: SELECT Statement Anatomy I

- The SELECT statement specifies the columns (attributes) to retrieve.
- Basic Syntax: SELECT column1, column2 FROM table_name;
- To retrieve all columns, use the asterisk wildcard (*): SELECT * FROM table_name;
- The FROM clause specifies the source relation (table or view) from which to retrieve data.
- Example: SELECT first_name, last_name, hire_date FROM employees;
- Column aliases can be used for readability using the AS keyword (e.g., SELECT first_name AS 'First Name').

Concept: Relational Projection I



Filtering Data: The WHERE Clause I

- The WHERE clause implements the Selection (sigma) operation from relational algebra.
- It filters rows based on specified conditions, ensuring only tuples that evaluate to TRUE are included in the result set.
- Syntax: SELECT column_list FROM table_name WHERE condition;
- Logical execution order: FROM clause executes first (loads table), then WHERE filters rows, and finally SELECT projects columns.
- Example: SELECT first_name, salary FROM employees WHERE department_id = 50;

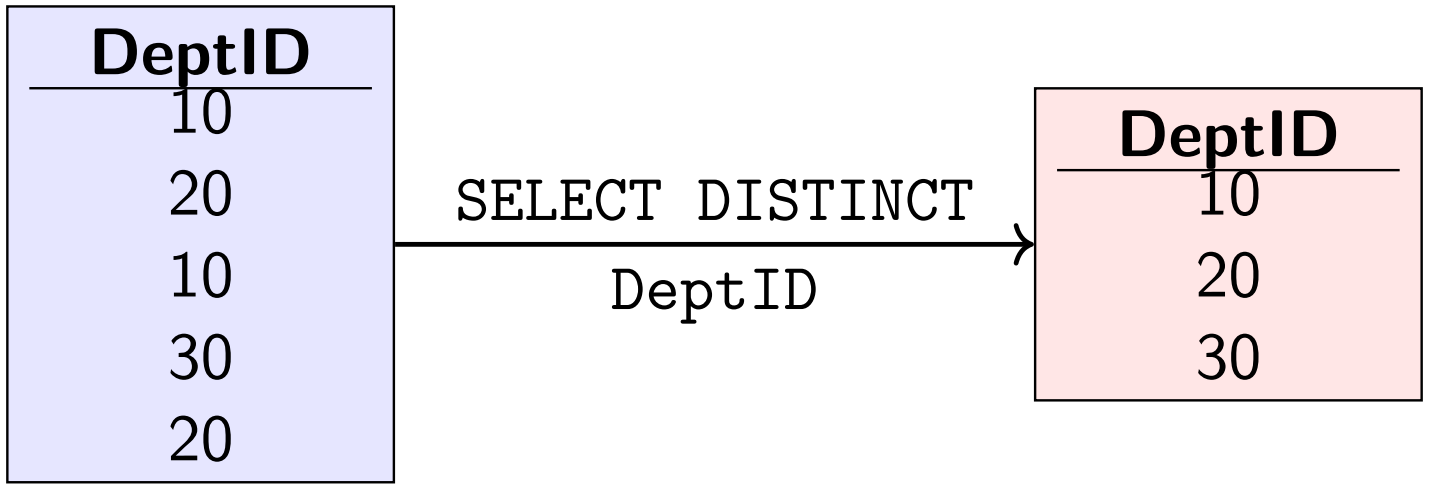
Operators in the WHERE Clause I

- Relational Operators: = (Equal), ¡¿ or != (Not Equal), ¿, ¡, ¿=, ¡=.
- Example: SELECT * FROM products WHERE price ¿= 100.00;
- 1. BETWEEN ... AND ... : Inclusive range check (e.g., WHERE salary BETWEEN 50000 AND 80000).
- 2. IN (list): Checks if a value matches any value in a discrete list (e.g., WHERE job_id IN ('IT_PROG', 'SA_REP')).
- 3. LIKE: Pattern matching using wildcards % (zero or more chars) and _ (exactly one char). (e.g., WHERE last_name LIKE 'S%').
- 4. IS NULL / IS NOT NULL: Checks for missing or unknown data. Avoid using = NULL as it evaluates to UNKNOWN, not TRUE.

Complex Filtering: Logical Operators I

- Conditions can be combined using logical operators: AND, OR, and NOT.
- AND requires all combined conditions to be TRUE.
- OR requires at least one condition to be TRUE.
- NOT reverses the boolean outcome of a condition.
- Precedence Rules: NOT evaluates first, followed by AND, then OR. Use parentheses to enforce desired evaluation order.
- Complex Example: SELECT emp_id FROM employees WHERE (department_id = 90 OR department_id = 100) AND salary ¿ 10000 AND commission_pct IS NOT NULL;

Eliminating Duplicates: SELECT DISTINCT I



Deep Dive: SELECT DISTINCT Mechanics I

- The DISTINCT keyword removes duplicate rows from the final result set.
- Syntax: SELECT DISTINCT column1, column2 FROM table_name;
- If multiple columns are specified, DISTINCT evaluates the unique combination of all specified columns, not individual ones.
- Performance Impact: DISTINCT often requires the RDBMS engine to sort or hash the result set to identify duplicates, which can be computationally expensive on large datasets.
- NULL Handling: In SELECT DISTINCT, all NULL values are considered equal and are collapsed into a single NULL row in the output.

Deep Dive: SELECT DISTINCT Mechanics II

- Best Practice: Use DISTINCT only when necessary. If querying a Primary Key column, DISTINCT is redundant and wastes execution time.

Agenda I

- Introduction to Data Control Language (DCL)
- Database Security: Authentication vs Authorization
- Creating and Managing Users
- Role-Based Access Control (RBAC)
- The GRANT Command: Assigning Privileges
- The REVOKE Command: Removing Privileges
- Cascading Effects of Revocation

Introduction to DCL I

- Data Control Language (DCL) is a subset of SQL used to control access to data within a database.
- It allows database administrators (DBAs) to enforce security by granting or revoking privileges.
- Two primary commands in DCL: GRANT and REVOKE.
- Privileges can be granted on various database objects: Tables, Views, Procedures, Sequences, and more.
- System Privileges: Rights to perform a particular action in the database (e.g., CREATE TABLE, CREATE USER).
- Object Privileges: Rights to perform a particular operation on a specific object (e.g., SELECT ON employees, UPDATE ON payroll).

Creating Users I

- Before assigning privileges, a user account must exist. Authentication verifies identity, while authorization (via DCL) determines access.
- Syntax: CREATE USER username IDENTIFIED BY password;
- Example: CREATE USER db_analyst IDENTIFIED BY 'SecurePass123!';
- Additional parameters can specify default tablespaces, temporary tablespaces, and account locking status.
- Example: CREATE USER app_user IDENTIFIED BY 'AppPass!' DEFAULT TABLESPACE users TEMPORARY TABLESPACE temp QUOTA 50M ON users PASSWORD EXPIRE;

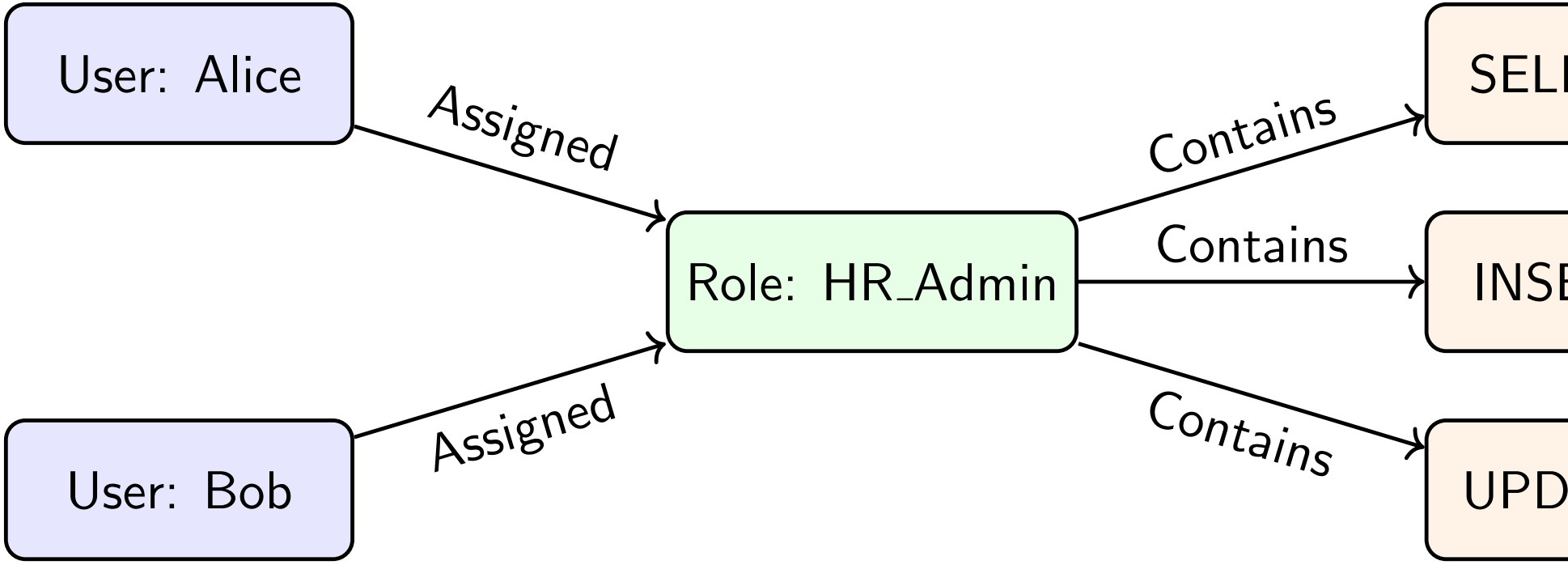
Creating and Managing Roles I

- A role is a named group of related privileges that can be granted to users or other roles.
- Roles simplify privilege management. Instead of granting 10 privileges to 100 users, grant them to 1 role, and assign the role to 100 users.
- Syntax: CREATE ROLE role_name;
- Example: CREATE ROLE hr_manager_role;
- Granting privileges to a role: GRANT SELECT, INSERT, UPDATE ON employees TO hr_manager_role;
- Assigning a role to a user: GRANT hr_manager_role TO alice, bob;

The GRANT Command I

- The GRANT command gives users or roles specific privileges.
- Syntax for Object Privileges: GRANT privilege_list ON object_name TO user_or_role [WITH GRANT OPTION];
- WITH GRANT OPTION allows the recipient to further grant the privilege to other users.
- Example 1 (System Privilege): GRANT CREATE SESSION, CREATE TABLE TO data_engineer;
- Example 2 (Object Privilege): GRANT SELECT, DELETE ON departments TO hr_admin;
- Example 3 (Column-level Privilege): GRANT UPDATE (salary, commission) ON employees TO payroll_manager;

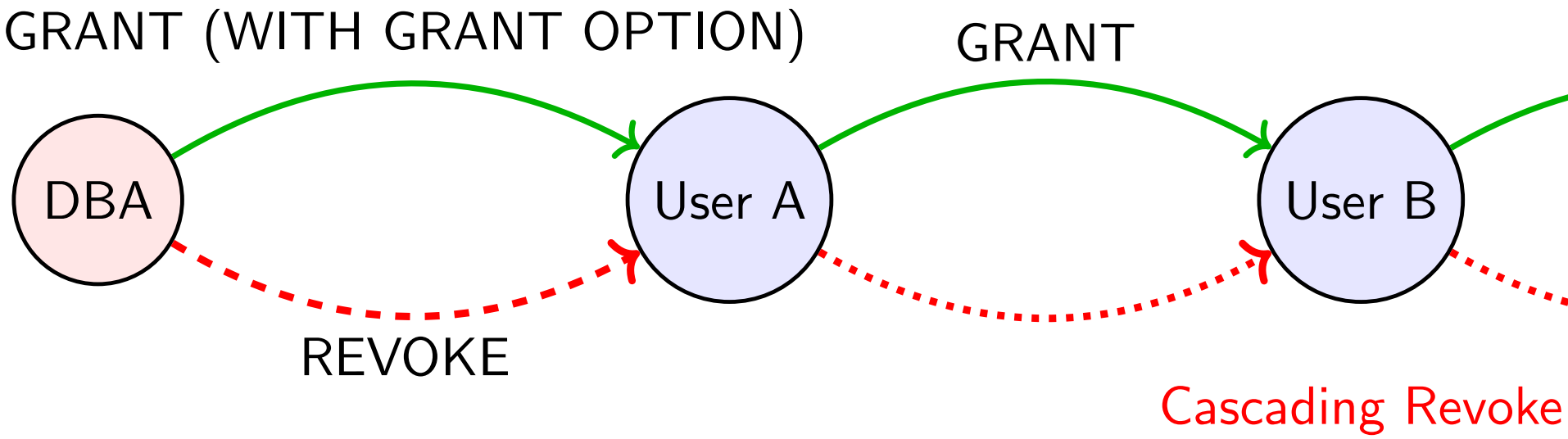
Role-Based Access Control Architecture I



The REVOKE Command I

- The REVOKE command removes privileges previously granted to a user or role.
- Syntax: REVOKE privilege_list ON object_name FROM user_or_role [CASCADE CONSTRAINTS];
- Example 1: REVOKE DELETE ON departments FROM hr_admin;
- Example 2: REVOKE hr_manager_role FROM alice;
- Note on System Privileges: If a user was granted a privilege WITH ADMIN OPTION, revoking it does NOT cascade to users they granted it to.
- Note on Object Privileges: If a user was granted a privilege WITH GRANT OPTION, revoking it DOES cascade to users they granted it to.

Cascading Revocation Flow I



Best Practices in DCL I

- Principle of Least Privilege (PoLP): Users should be granted only the absolute minimum privileges necessary to perform their job functions.
- Avoid granting the DBA role or broad privileges (e.g., SELECT ANY TABLE) unless explicitly required for administrative accounts.
- Use Roles for scaling: Never grant privileges directly to users in a large enterprise system. Always group them into roles.
- Regular Audits: Periodically review granted privileges and roles using system views (e.g., DBA_SYS_PRIVS, DBA_TAB_PRIVS).
- Revoke dormant accounts and clean up unused roles to minimize the potential attack surface.

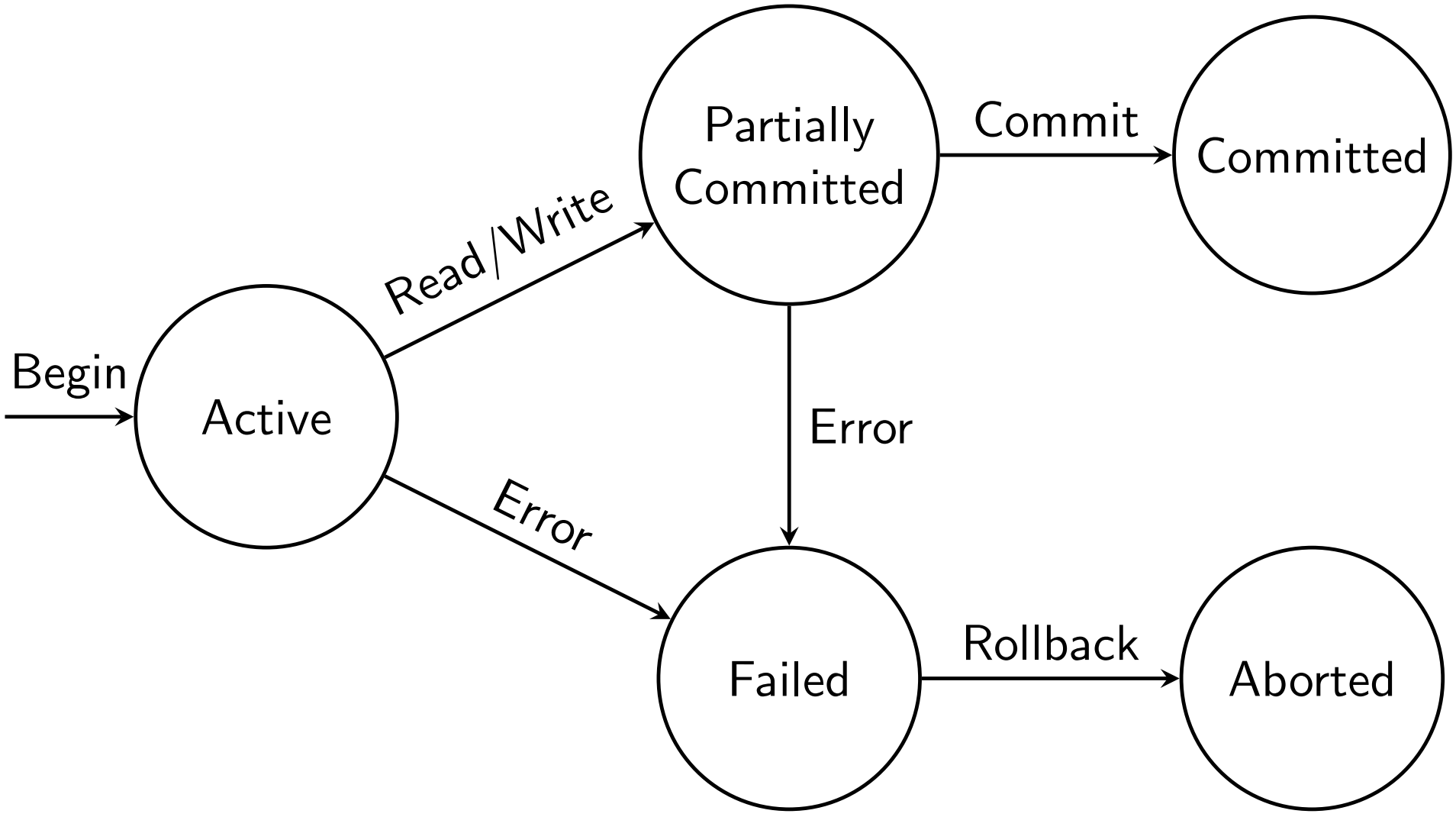
Agenda I

- 1. Introduction to Database Transactions
- 2. Transaction States and Lifecycle
- 3. The ACID Properties
- 4. Transaction Control Language (TCL) Commands
- 5. COMMIT and Data Persistence
- 6. ROLLBACK for Error Recovery
- 7. SAVEPOINT and Partial Rollbacks
- 8. Summary and Best Practices

Introduction to Transactions I

- Definition: A transaction is a logical unit of work that contains one or more SQL statements.
- Transactions are highly critical in RDBMS to ensure data integrity and consistency, particularly in multi-user environments.
- A transaction begins with the first executable SQL statement (like INSERT, UPDATE, DELETE).
- It ends when a COMMIT or ROLLBACK statement is explicitly issued, or when a DDL statement (like CREATE) is executed which implicitly commits the current transaction.

Transaction State Lifecycle I



ACID Properties I

- Atomicity: 'All or nothing'. The transaction completes fully, or it doesn't execute at all. If one part fails, the entire transaction fails and the database state is left unchanged.
- Consistency: The database must remain in a consistent state before and after the transaction. All integrity constraints and rules must be obeyed.
- Isolation: Concurrent transactions execute completely isolated from one another. The intermediate states of a transaction are invisible to other transactions.
- Durability: Once a transaction is committed, its changes are permanent and survive any subsequent system failures, crashes, or power losses.

The COMMIT Command I

- Purpose: Permanently saves all changes made during the current transaction to the database.
- Once committed, the previous state of the data is lost, and the changes become visible to all other users and transactions.
- Releases any locks held on rows or tables by the transaction.
- Example SQL: BEGIN; UPDATE accounts SET balance = balance - 500 WHERE account_id = 101; UPDATE accounts SET balance = balance + 500 WHERE account_id = 102; COMMIT;

The ROLLBACK Command I

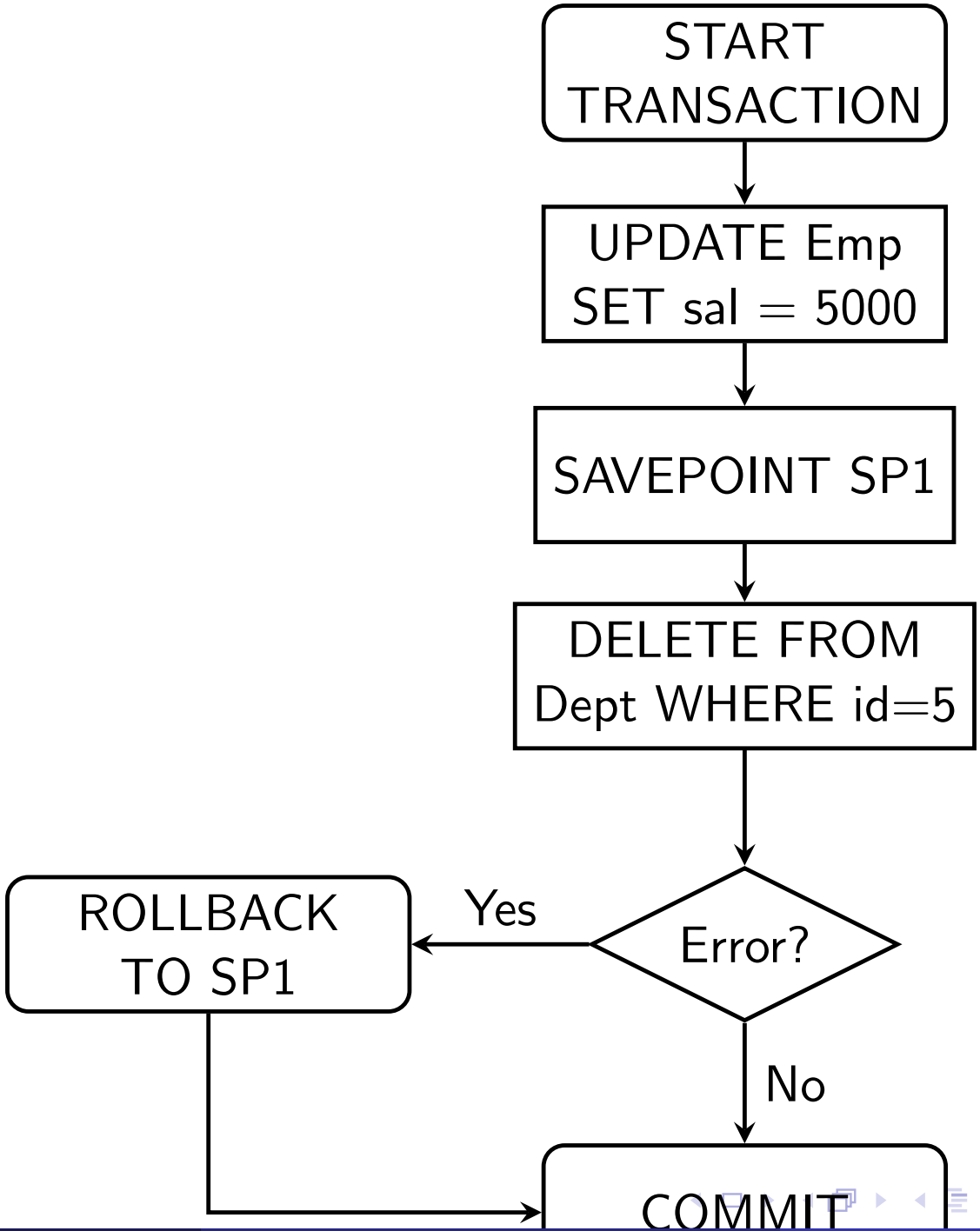
- Purpose: Undoes all changes made during the current transaction, reverting the database to its state prior to the transaction's start.
- Crucial for error recovery, application failures, or when business logic validation fails mid-transaction.
- Also releases any locks held by the transaction.
- Example SQL: BEGIN; DELETE FROM active_sessions; – Business validation failed! ROLLBACK;

The SAVEPOINT Command I

- Purpose: Creates a defined marker within a transaction, allowing a partial rollback to that specific point rather than rolling back the entire transaction.
- Useful in complex, long-running transactions where you might want to retry a specific sub-task without losing the progress of previous steps.
- Syntax: SAVEPOINT savepoint_name; ROLLBACK TO savepoint_name;
- Note: A COMMIT or a full ROLLBACK invalidates all existing savepoints in the transaction.

Visualizing SAVEPOINT and ROLLBACK I

Visualizing SAVEPOINT and ROLLBACK II



Summary & Best Practices I

- Keep transactions as short as possible to minimize lock contention and improve concurrency.
- Always handle errors and exceptions in your application logic to ensure transactions are either committed or rolled back explicitly.
- Avoid user interaction (like prompts or network waits) inside an active transaction to prevent locking database resources.
- Remember ACID: Atomicity, Consistency, Isolation, and Durability form the bedrock of reliable relational databases.
- TCL commands (COMMIT, ROLLBACK, SAVEPOINT) are your tools for managing these transactions effectively.

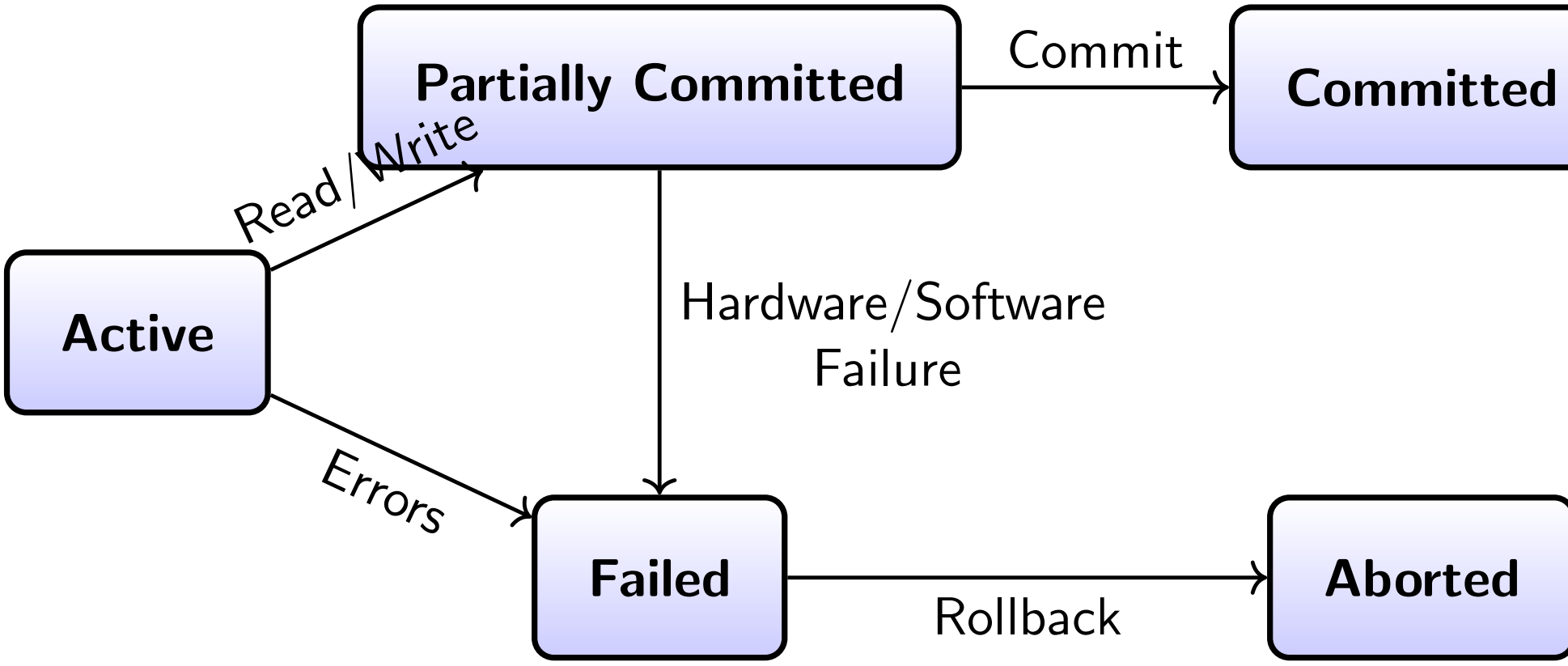
Agenda I

- 1. Introduction to Database Transactions
- 2. Transaction States and Lifecycle
- 3. ACID Properties of a Transaction
- 4. COMMIT: Persisting Changes
- 5. ROLLBACK: Reverting Changes
- 6. SAVEPOINT: Partial Rollbacks

Introduction to Transactions I

- A transaction is a logical unit of work that contains one or more SQL statements.
- It is an atomic operation; either all the statements execute successfully, or none do.
- Transactions are essential for maintaining database integrity in a multi-user, concurrent environment.
- SQL uses Transaction Control Language (TCL) commands to manage these transactions.
- By default, many DBMS (like MySQL) operate in 'autocommit' mode, meaning every single SQL statement is treated as a complete transaction.
- To group multiple statements, autocommit must be disabled (e.g., START TRANSACTION in MySQL, or implicit start in Oracle).

State Diagram of a Transaction I



ACID Properties: Overview I

- To ensure data integrity during transaction execution, the database enforces four vital properties, known as ACID.
- A - Atomicity: 'All or nothing.' A transaction is treated as a single, indivisible logical unit of work.
- C - Consistency: A transaction must transform the database from one consistent state to another consistent state, respecting all constraints and rules.
- I - Isolation: The execution of concurrent transactions leaves the database in the same state as if they were executed sequentially.
- D - Durability: Once a transaction has been committed, its effects are permanently recorded in the database and will survive subsequent failures (e.g., power loss).

ACID Properties: Deep Dive I

- Atomicity is usually managed by the Transaction Management component, which initiates rollbacks if a step fails.
- Consistency is enforced by the Application and Database Constraints (e.g., CHECK, UNIQUE, FOREIGN KEY).
- Isolation is handled by the Concurrency Control system using locks or multiversion concurrency control (MVCC). Isolation levels include: READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, SERIALIZABLE.
- Durability is ensured by the Recovery Management component, typically using Write-Ahead Logging (WAL) or Redo Logs to rebuild committed transactions if the system crashes before writing to disk.

TCL Command: COMMIT I

- The COMMIT command permanently saves all changes made during the current transaction.
- Once a transaction is committed, the changes are visible to other users and transactions, and the locks held by the transaction are released.
- Syntax: COMMIT [WORK];
- Example Scenario:
- BEGIN TRANSACTION;
- UPDATE Accounts SET Balance = Balance - 100 WHERE AccountID = 1;
- UPDATE Accounts SET Balance = Balance + 100 WHERE AccountID = 2;
- COMMIT; – Funds are permanently transferred.

TCL Command: ROLLBACK I

- The ROLLBACK command undoes all changes made during the current transaction, reverting the database to its previous consistent state.
- It is used when an error occurs or a business condition is not met.
- Syntax: ROLLBACK [WORK];
- Example Scenario:
- BEGIN TRANSACTION;
- DELETE FROM Employees WHERE DepartmentID = 5;
- – Realize mistake! We should not delete all these employees.
- ROLLBACK; – The deleted rows are restored and locks are released.

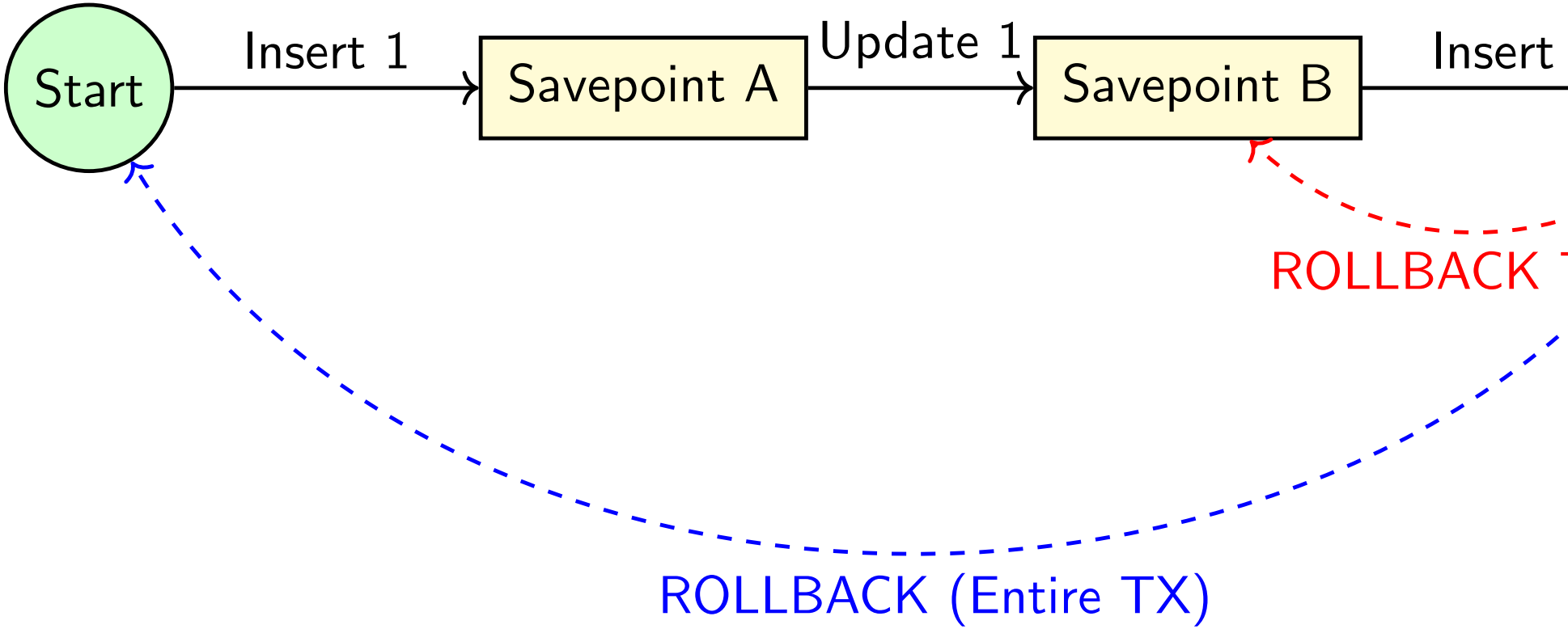
TCL Command: SAVEPOINT I

- A SAVEPOINT marks a specific point within a transaction to which you can later roll back, without rolling back the entire transaction.
- Useful for long, complex transactions where partial failures can be recovered gracefully.
- Syntax to create: SAVEPOINT savepoint_name;
- Syntax to revert: ROLLBACK TO savepoint_name;
- Example:
- INSERT INTO Orders (OrderID) VALUES (1);
- SAVEPOINT sp1;
- INSERT INTO OrderDetails (OrderID, Item) VALUES (1, 'Laptop');
- – Error happens here, rollback only the item insertion
- ROLLBACK TO sp1;

TCL Command: SAVEPOINT II

- COMMIT; – Order 1 is committed, but without the Laptop detail.

Transaction Flow & Savepoints I



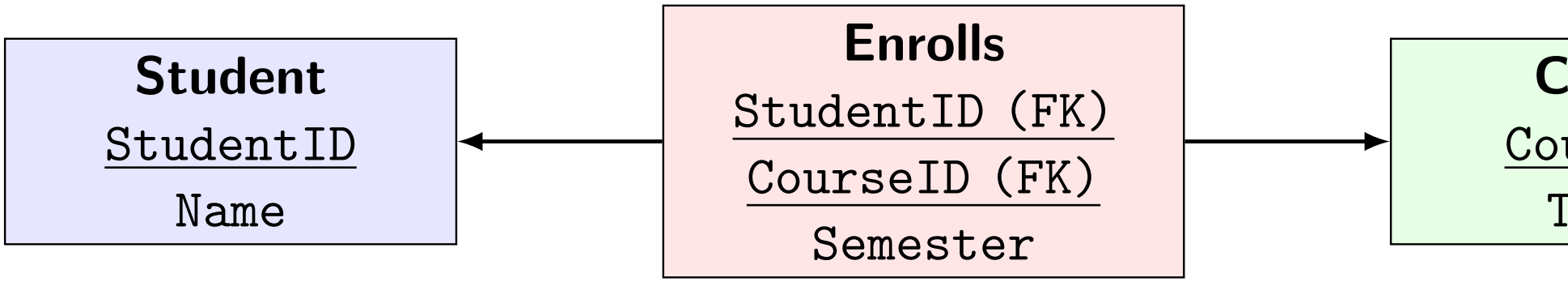
Agenda I

- 1. Entity-Relationship to Relational Schema Mapping
- 2. Normalization Scenarios (1NF to BCNF)
- 3. Advanced SQL Query Resolution
- 4. Correlated Subqueries and Joins
- 5. Concurrency and Transaction Schedules

ER to Relational Mapping: Scenario I

- Problem: Map a many-to-many relationship between STUDENT and COURSE.
- Rule 1: Create a table for each strong entity. STUDENT(StudentID, Name), COURSE(CourseID, Title).
- Rule 2: For an M:N relationship (ENROLLS), create a new junction table.
- Junction Table: ENROLLS(StudentID, CourseID, Semester).
- Primary Key of ENROLLS is the composite key (StudentID, CourseID). Both are Foreign Keys referencing STUDENT and COURSE respectively.

Visualizing ER to Relational Mapping I



Normalization Practice: Unnormalized Form (UNF) to 1NF I

- Problem: A table contains multi-valued attributes: INVOICE(InvID, CustName, {Item, Qty, Price}).
- Solution (1NF): Remove repeating groups by flattening the table or creating a separate table.
- Approach A: Flattening into INVOICE(InvID, CustName, Item, Qty, Price). Primary Key becomes (InvID, Item).
- Warning: Approach A introduces partial dependencies (CustName depends only on InvID), violating 2NF.
- Best Practice: Jump to 2NF directly. INVOICE(InvID, CustName) and INVOICE_LINE(InvID, Item, Qty, Price).

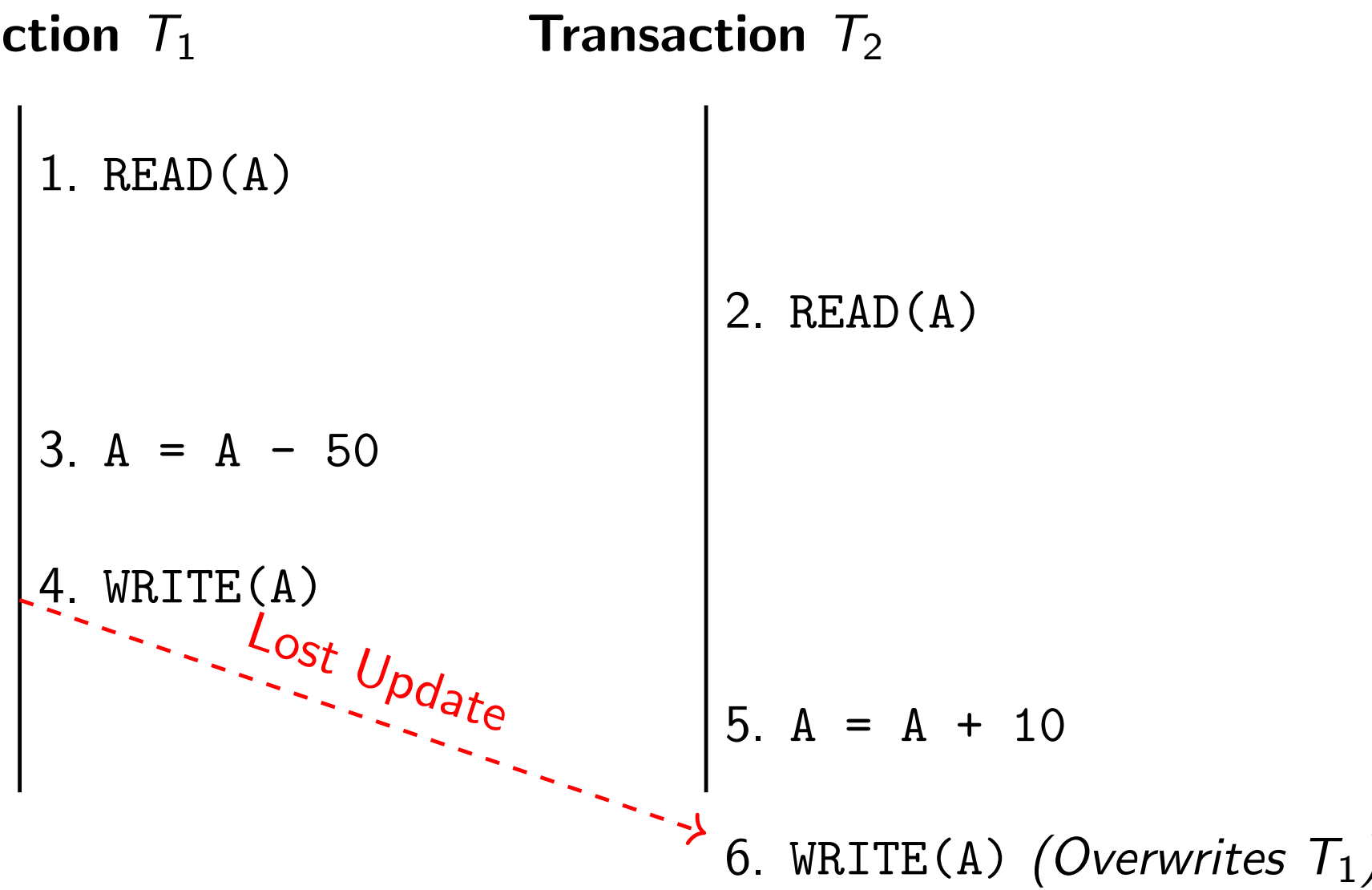
SQL Problem Solving: Correlated Subqueries I

- Problem: Find employees whose salary is strictly greater than the average salary of their specific department.
- Query:
- SELECT e1.EmpName, e1.Salary, e1.DeptID
- FROM Employee e1
- WHERE e1.Salary ¿ (
- SELECT AVG(e2.Salary)
- FROM Employee e2
- WHERE e1.DeptID = e2.DeptID
-);
- Execution logic: The inner query re-evaluates for every row processed by the outer query, matching DeptID.

SQL Problem Solving: Recursive CTEs I

- Problem: Retrieve the organizational hierarchy (Manager-Subordinate chain) starting from the CEO.
- Query:
- WITH RECURSIVE OrgChart AS (
- SELECT EmpID, Name, ManagerID, 1 AS Level
- FROM Employee WHERE ManagerID IS NULL
- UNION ALL
- SELECT e.EmpID, e.Name, e.ManagerID, o.Level + 1
- FROM Employee e
- INNER JOIN OrgChart o ON e.ManagerID = o.EmpID
-)
- SELECT * FROM OrgChart ORDER BY Level;

Transaction Schedules: Lost Update Problem I



Concurrency Control Mechanisms I

- Problem: How to prevent the Lost Update anomaly shown in the previous schedule?
- Solution 1: Strict Two-Phase Locking (2PL). T1 acquires an exclusive lock (X-Lock) on A before reading/writing.
- T2's attempt to READ(A) will be blocked until T1 commits and releases the X-Lock.
- Solution 2: Timestamp Ordering. Each transaction gets a timestamp TS(T). Reads and writes are checked against Read-TS(A) and Write-TS(A).
- Solution 3: Multi-Version Concurrency Control (MVCC). Reads do not block writes, writes create new versions. T2 would read an older version or abort on write.

Summary and Next Steps I

- Key Takeaway 1: Proper mapping from ER to relational schemas avoids data anomalies from the start.
- Key Takeaway 2: Correlated subqueries and CTEs are powerful tools for complex hierarchical or comparative analytics.
- Key Takeaway 3: Transaction isolation levels and locking protocols are essential for data consistency in concurrent environments.
- Action Item: Complete Assignment 4 on PL/SQL trigger creation and concurrency schedules.

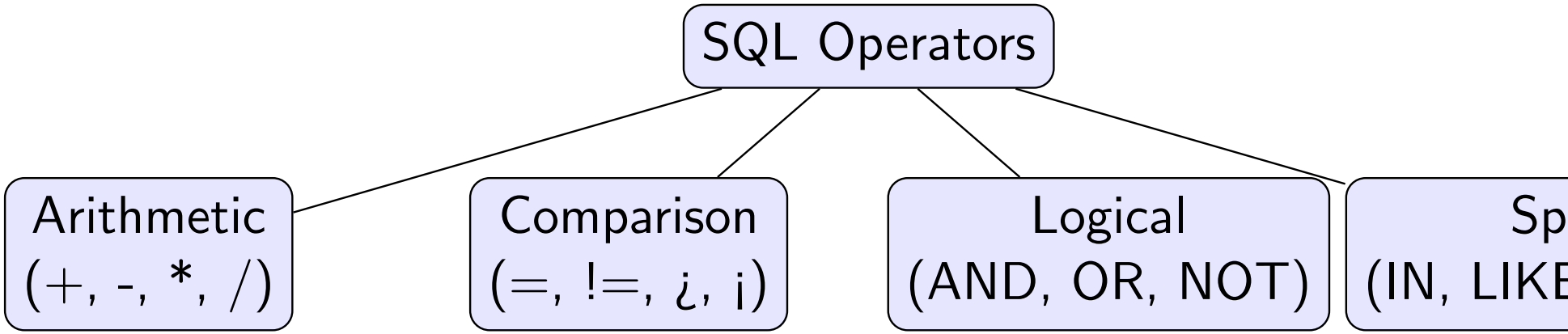
Agenda I

- Arithmetic Operators
- Comparison & Logical Operators
- Special Operators: IN, ANY, ALL
- Special Operators: BETWEEN, EXISTS, LIKE
- Use of System Table (DUAL table)

Arithmetic Operators I

- Used to perform mathematical calculations on numeric columns or literals.
- Addition (+): Adds two operands. Example: SELECT salary + 500 FROM employees;
- Subtraction (-): Subtracts right operand from left. Example: SELECT price - discount FROM products;
- Multiplication (*): Multiplies two operands. Example: SELECT qty * price FROM order_items;
- Division (/): Divides left operand by right. Example: SELECT total_marks / 5 FROM students;
- Modulo (% or MOD): Returns the remainder of division. Example: SELECT MOD(emp_id, 2) FROM employees;
- Note on NULLs: Any arithmetic operation with a NULL value evaluates to NULL. e.g., 100 + NULL = NULL.

SQL Operator Categories I



Comparison & Logical Operators I

- Comparison Operators test conditions and return TRUE, FALSE, or UNKNOWN (due to NULLs).
- Equality (=) and Inequality (!= or ¡¿): e.g., SELECT * FROM depts WHERE dept_name ¡¿ 'Sales';
- Relational (¿, ¡, ¿=, ¡=): Used for numerical, date, and lexicographical string comparisons.
- Logical Operators combine multiple boolean expressions:
- AND: Returns TRUE only if both conditions are TRUE. e.g., WHERE age ¿ 25 AND dept = 'IT';
- OR: Returns TRUE if at least one condition is TRUE. e.g., WHERE status = 'Active' OR role = 'Admin';
- NOT: Reverses the boolean value of an expression. e.g., WHERE NOT (salary ¡ 30000);
- Operator Precedence: Arithmetic ¿ Concatenation ¿ Comparison ¿ NOT ¿ AND ¿ OR.

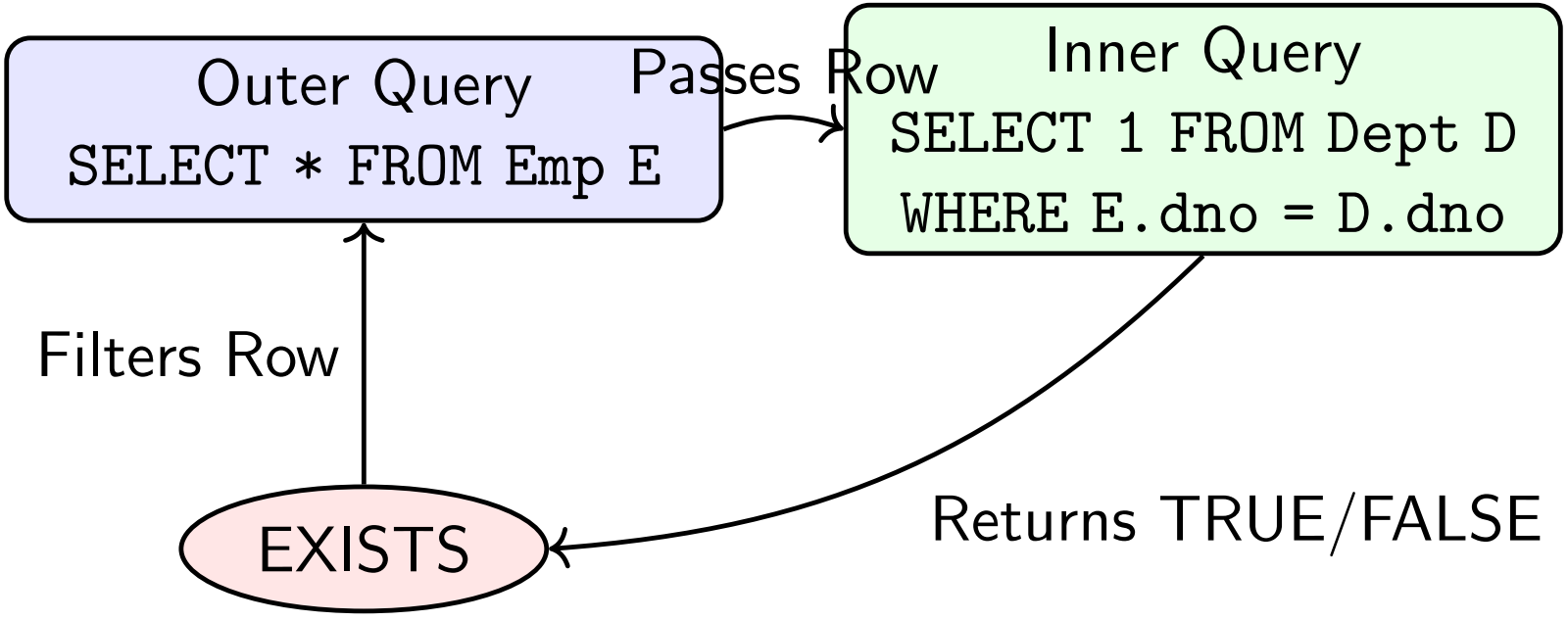
Special Operators: IN, ANY, ALL I

- IN: Checks if a value matches any value in a list or subquery. Equivalent to multiple ORs.
- Example: SELECT emp_name FROM emp WHERE job_id IN ('IT_PROG', 'SA_REP', 'FI_ACCOUNT');
- ANY / SOME: Compares a scalar value with a single-column set of values returned by a subquery.
- Condition evaluates to TRUE if it holds for *at least one* value in the set.
- Example: SELECT * FROM emp WHERE salary ¿ ANY (SELECT salary FROM emp WHERE dept_id = 10); (Returns employees earning more than the lowest paid employee in dept 10).
- ALL: Compares a scalar value with *every* value returned by a subquery.

Special Operators: IN, ANY, ALL II

- Condition is TRUE only if it holds for *all* values in the set.
- Example: SELECT * FROM emp WHERE salary ¿ ALL (SELECT salary FROM emp WHERE dept_id = 10); (Returns employees earning more than the highest paid employee in dept 10).

Special Operator: EXISTS (Correlated Subqueries) I



Special Operators: BETWEEN and LIKE I

- BETWEEN: Checks if a value falls within an inclusive range. Useful for numbers and dates.
- Example: SELECT * FROM orders WHERE order_date BETWEEN '2023-01-01' AND '2023-12-31';
- Equivalent to: order_date ¿= '2023-01-01' AND order_date ¡= '2023-12-31'.
- LIKE: Used for pattern matching in strings using wildcard characters.
- Percent (%) Wildcard: Matches zero or more characters. e.g., LIKE 'A%' matches 'Apple', 'Adam'.
- Underscore (_) Wildcard: Matches exactly one character. e.g., LIKE 'Sm_th' matches 'Smith', 'Smyth'.
- ESCAPE clause: Used to search for actual literal '%' or '_' characters. e.g., LIKE '%20\{}%%' ESCAPE '\{}\{}'.

Use of System Table / DUAL Table I

- DUAL is a special one-row, one-column table present by default in Oracle and some other RDBMS.
- It is used for selecting pseudo-columns, evaluating expressions, or calling functions when a table isn't necessary.
- Structure: Contains a single column named 'DUMMY' with a single record containing the value 'X'.
- Why use it? The SELECT syntax requires a FROM clause in standard SQL (Oracle strictly enforces this).
- Example 1 (Math): SELECT 5 * 10 AS result FROM DUAL;
- Example 2 (System Date): SELECT SYSDATE FROM DUAL;
- Example 3 (Sequence): SELECT employees_seq.NEXTVAL FROM DUAL;

Summary I

- Arithmetic, Comparison, and Logical operators form the foundational logic of SQL data filtering and calculation.
- Special operators (IN, ANY, ALL, BETWEEN, LIKE) provide powerful, shorthand ways to express complex conditions.
- EXISTS is a highly efficient operator for correlated subqueries, returning boolean values without manifesting result sets.
- The DUAL table acts as a dummy table for evaluating standalone expressions and functions in RDBMS environments that require a FROM clause.
- Mastering operator precedence and NULL handling is critical for writing robust and bug-free SQL queries.

Agenda I

- 1. Arithmetic and Comparison Operators
- 2. Logical Operators (AND, OR, NOT)
- 3. Special Operators: IN, BETWEEN, LIKE
- 4. Quantified Comparison: ANY, ALL
- 5. Existence Testing: EXISTS
- 6. The DUAL Table

Arithmetic and Comparison Operators I

- Arithmetic operators (+, -, *, /) perform mathematical operations on numeric data types. Modulo is typically implemented via the MOD() function in SQL standard.
- Comparison operators (=, !=, ¡¿, ¡, ¿, ¡=, ¿=) test conditions and return boolean values (TRUE, FALSE, UNKNOWN).
- When comparing with NULL, standard equality operators return UNKNOWN. Use IS NULL or IS NOT NULL instead of = NULL or != NULL.
- Example: SELECT employee_id, salary + (salary * 0.10) AS new_salary FROM employees WHERE department_id ¡¿ 20;

Logical Operators I

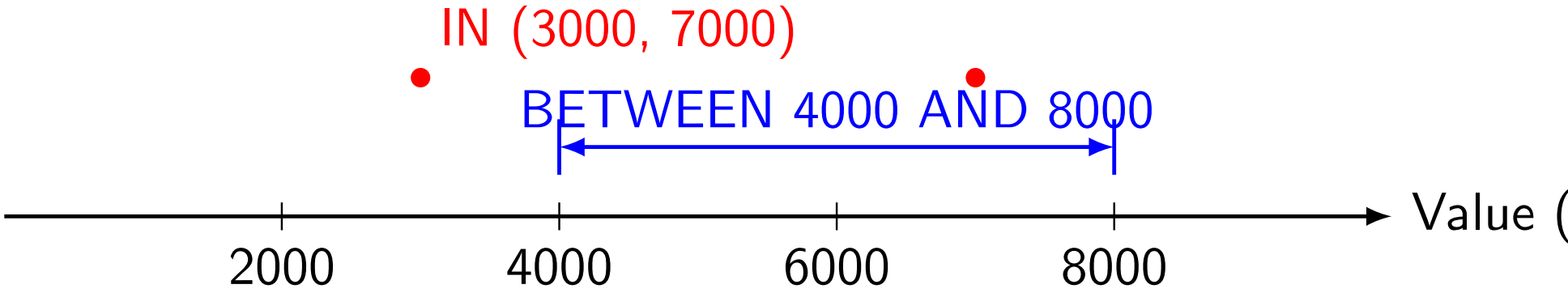
- Logical operators combine multiple conditions in a WHERE or HAVING clause. The primary logical operators are AND, OR, and NOT.
- AND requires both conditions to evaluate to TRUE. OR requires at least one condition to evaluate to TRUE. NOT reverses the boolean value.
- Operator Precedence (highest to lowest): NOT, AND, OR. Always use parentheses to explicitly define evaluation order and avoid logical errors.
- Three-valued logic applies when NULLs are involved: TRUE AND UNKNOWN evaluates to UNKNOWN, whereas TRUE OR UNKNOWN evaluates to TRUE.

Special Operators: IN, BETWEEN, LIKE I

- The IN operator tests for membership in a specified set of literal values or a subquery result set. E.g., WHERE department_id IN (10, 20, 30).
- The BETWEEN operator tests if a value falls within a specified inclusive range. E.g., WHERE salary BETWEEN 5000 AND 10000 is semantically identical to WHERE salary ¿= 5000 AND salary ¡= 10000.
- The LIKE operator performs string pattern matching using wildcard characters: '%' (matches zero or more characters) and '_' (matches exactly one character).
- Example: SELECT first_name FROM employees WHERE last_name LIKE '_a%e'; (Finds names where the second letter is 'a' and it contains 'e' somewhere after).

Visualizing Range and Pattern Matching I

LIKE 'J%': Matches 'John', 'Jane', 'Joe'



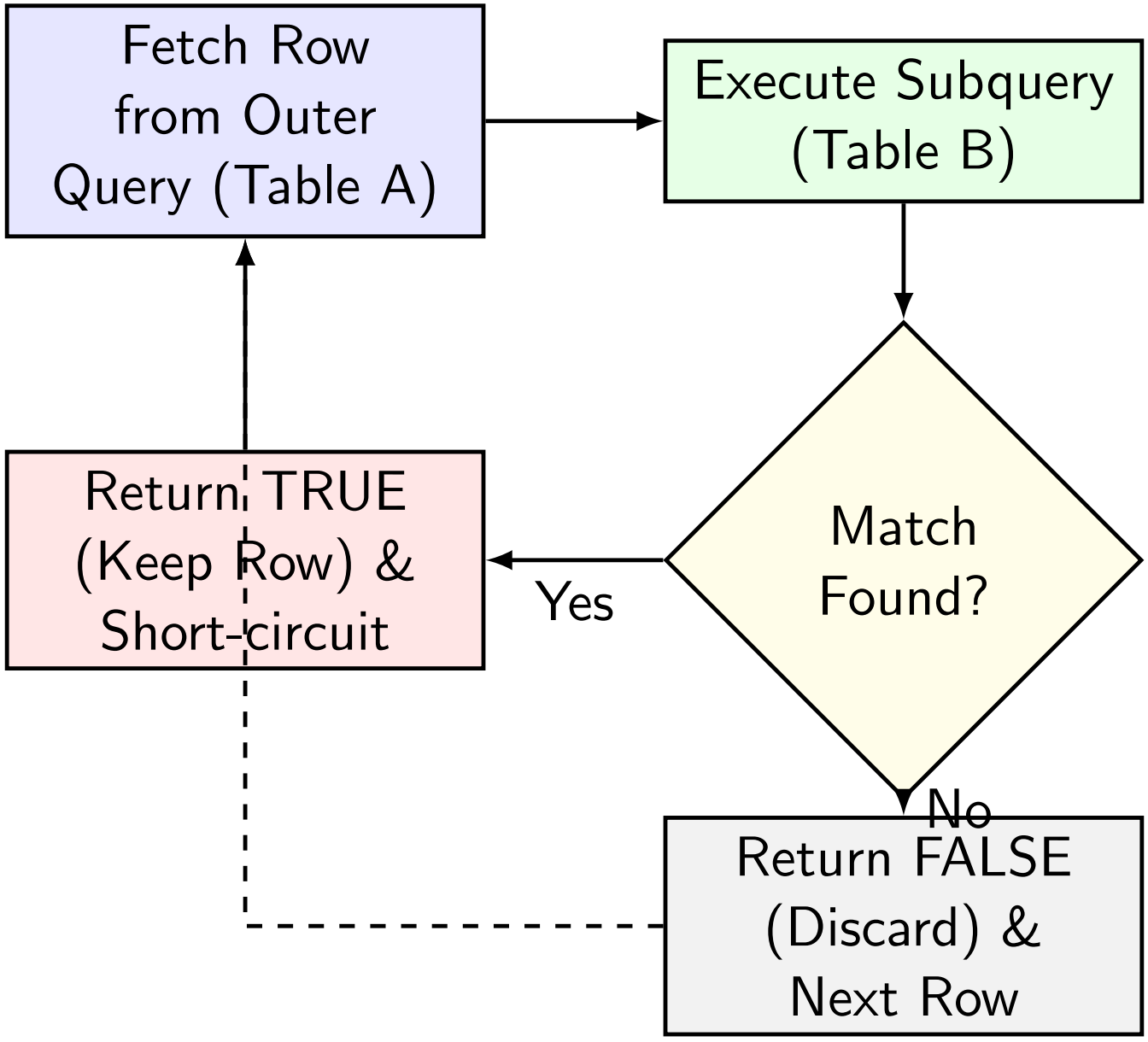
Quantified Comparison: ANY, ALL I

- ANY (or SOME) compares a scalar value with a single-column set of values returned by a subquery. The condition evaluates to TRUE if it holds for *at least one* value in the set.
- Example: WHERE salary ¿ ANY (SELECT salary FROM employees WHERE job_id = 'IT_PROG'); (Finds employees earning more than the lowest-paid IT programmer).
- ALL compares a scalar value with every value returned by a subquery. The condition evaluates to TRUE only if it holds for *every* value in the set.
- Example: WHERE salary ¿ ALL (SELECT salary FROM employees WHERE department_id = 10); (The salary must be strictly greater than the maximum salary in department 10).

Subquery Operators: EXISTS vs IN I

- The EXISTS operator tests for the existence of rows returned by a subquery. It returns TRUE if the subquery returns at least one row, and FALSE otherwise.
- EXISTS is typically used with correlated subqueries. The RDBMS optimizer can employ 'short-circuiting', stopping the subquery execution as soon as the first matching row is found, which is highly efficient.
- NOT IN with a subquery returning any NULL values evaluates to UNKNOWN for all rows, resulting in zero rows returned. NOT EXISTS handles NULLs logically by checking for row existence regardless of column nullability.
- Example: SELECT d.department_name FROM departments d WHERE EXISTS (SELECT 1 FROM employees e WHERE e.department_id = d.department_id);

Execution Flow of EXISTS in Correlated Subqueries I



Use of System table / DUAL table I

- The DUAL table is a special one-row, one-column system table present by default in Oracle database and implemented in other RDBMS (like MySQL) for syntax compatibility.
- It is used for selecting pseudo-columns, evaluating mathematical expressions, or calling scalar functions where the SELECT statement syntactically requires a FROM clause but no actual user table data is needed.
- Structure: Contains a single column named 'DUMMY' of type VARCHAR2(1) with a single row containing the value 'X'.
- Example 1: SELECT SYSDATE FROM DUAL; (Returns the current system date and time). Example 2: SELECT 3.14 * POWER(5, 2) FROM DUAL; (Evaluates and returns the result of the mathematical expression).

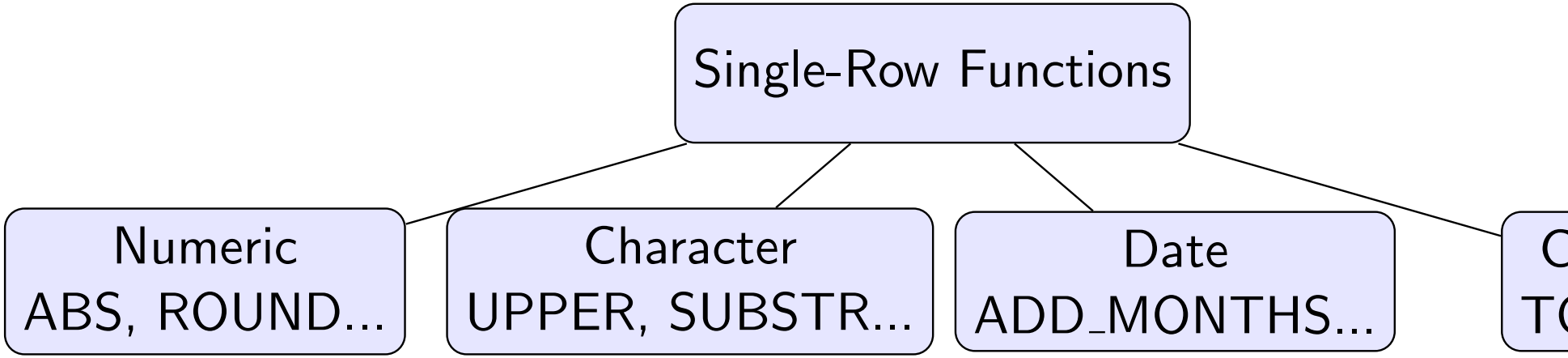
Agenda I

- Introduction to Single Row Functions
- Categorization of SQL Functions
- Numeric Functions (ABS, POWER, MOD, ROUND, TRUNC, SQRT)
- Character Functions (INITCAP, LOWER, UPPER, LTRIM, RTRIM, REPLACE, SUBSTR, INSTR)
- Date Functions (ADD_MONTHS, MONTHS_BETWEEN, ROUND, TRUNC)
- Conversion Functions (TO_CHAR, TO_DATE, TO_NUMBER)

Introduction to SQL Functions I

- Functions in SQL provide dynamic data processing, transforming input values into desired output directly within queries.
- SQL functions are primarily categorized into Single-Row Functions and Multiple-Row (Aggregate) Functions.
- **Key Properties of Single-Row Functions:**
- 1. Operate on a single row and return exactly one result per row evaluated.
- 2. Can be utilized in ‘SELECT‘, ‘WHERE‘, and ‘ORDER BY‘ clauses.
- 3. Support argument acceptance of varying data types and can be deeply nested.
- 4. Facilitate complex mathematical transformations, string manipulations, and formatting tasks.

Classification of Single-Row Functions I



Numeric Functions I

- Numeric functions manipulate mathematical numbers and return exact or approximate numeric values.
- 1. **ABS(n)**: Returns the absolute (positive) magnitude of n. ‘SELECT ABS(-15) FROM DUAL; – Returns 15‘
- 2. **POWER(m, n)**: Returns m raised to the nth exponential power. ‘SELECT POWER(2, 3) FROM DUAL; – Returns 8‘
- 3. **MOD(m, n)**: Returns the arithmetic remainder of m divided by n. ‘SELECT MOD(10, 3) FROM DUAL; – Returns 1‘
- 4. **ROUND(n, [m])**: Rounds n to m decimal places (m defaults to 0). ‘SELECT ROUND(45.926, 2) FROM DUAL; – Returns 45.93‘

Numeric Functions II

- 5. **TRUNC(n, [m])**: Truncates n to m decimal places without rounding up. ‘SELECT TRUNC(45.926, 2) FROM DUAL; – Returns 45.92‘
- 6. **SQRT(n)**: Returns the mathematical square root of n. ‘SELECT SQRT(25) FROM DUAL; – Returns 5‘

Character Functions: Case Manipulation & Padding I

- These functions accept character inputs and structurally modify string casing or spacing.
- 1. **INITCAP(char)**: Converts the first letter of each alphabetic word to uppercase. ‘SELECT INITCAP('hello sql') FROM DUAL; – Returns 'Hello Sql'‘
- 2. **LOWER(char)**: Converts all characters to lowercase. Highly useful for standardizing data for case-insensitive searching.
- 3. **UPPER(char)**: Converts all characters to uppercase. Often used during data normalization pipelines.
- 4. **LTRIM(char, [set])**: Trims specified characters (or spaces by default) from the left boundary. ‘SELECT LTRIM(' SQL') FROM DUAL; – Returns 'SQL'‘

Character Functions: Case Manipulation & Padding II

- 5. **RTRIM(char, [set])**: Trims specified characters from the right boundary. ‘SELECT RTRIM('SQL ') FROM DUAL; – Returns 'SQL'‘

Character Functions: String Manipulation I

- These functions perform complex positional extractions and pattern evaluations on string data.
- 1. **REPLACE(char, search_string, [replacement_string])**: Replaces a specific character sequence. ‘SELECT REPLACE('JACK and JUE', 'J', 'BL') FROM DUAL; – Returns 'BLACK and BLUE'‘
- 2. **SUBSTR(char, position, [substring_length])**: Extracts a string portion. Note: SQL relies on 1-based indexing. ‘SELECT SUBSTR('HelloWorld', 1, 5) FROM DUAL; – Returns 'Hello'‘
- 3. **INSTR(char, search_string, [position, [occurrence]])**: Returns the numeric starting index of a string pattern. ‘SELECT INSTR('HelloWorld', 'W') FROM DUAL; – Returns 6‘

Date Functions I

- Databases store dates internally as numerical offsets, permitting arithmetic. Date functions provide specialized calendrical features.
- 1. **ADD_MONTHS(date, n)**: Shifts a date by n calendar months handling leap years automatically. ‘SELECT ADD_MONTHS(SYSDATE, 1) FROM DUAL;‘
- 2. **MONTHS_BETWEEN(date1, date2)**: Computes the precise number of months between date1 and date2. Results can be fractional.
- 3. **ROUND(date, [fmt])**: Rounds the date up or down to the nearest unit specified by the format model (e.g., 'MONTH', 'YEAR').

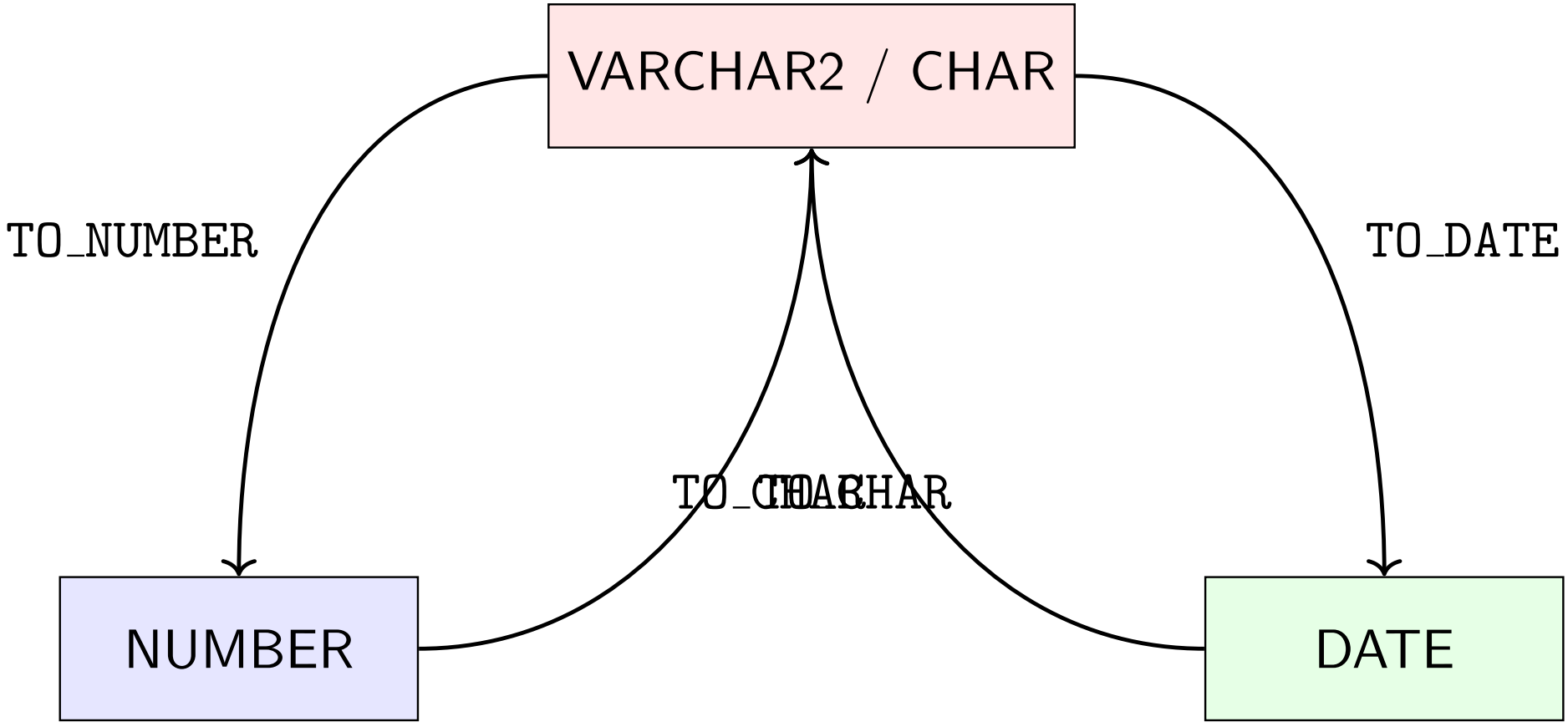
Date Functions II

- 4. **TRUNC(date, [fmt])**: Truncates the date back to the exact beginning of the specified temporal unit. ‘SELECT TRUNC(SYSDATE, ’YEAR’) FROM DUAL; – Returns January 1st at 00:00:00‘

Conversion Functions I

- Conversion functions explicitly cast values across domains, mitigating the severe performance overhead and unreliability of implicit casting.
- 1. **TO_CHAR(value, [fmt])**: Casts a numeric or temporal value into a structured character string using a format model. ‘SELECT TO_CHAR(SYSDATE, ’DD-MON-YYYY HH24:MI:SS’) FROM DUAL;‘
- 2. **TO_DATE(char, [fmt])**: Converts a string representing a calendar date into a native DATE data type. ‘SELECT TO_DATE(’01-JAN-2026’, ’DD-MON-YYYY’) FROM DUAL;‘
- 3. **TO_NUMBER(char, [fmt])**: Extracts a structural NUMBER from a string containing digits and symbols. ‘SELECT TO_NUMBER(’$1,000.50’, ’$9,999.99’) FROM DUAL;‘

Explicit Data Type Conversion Architecture I



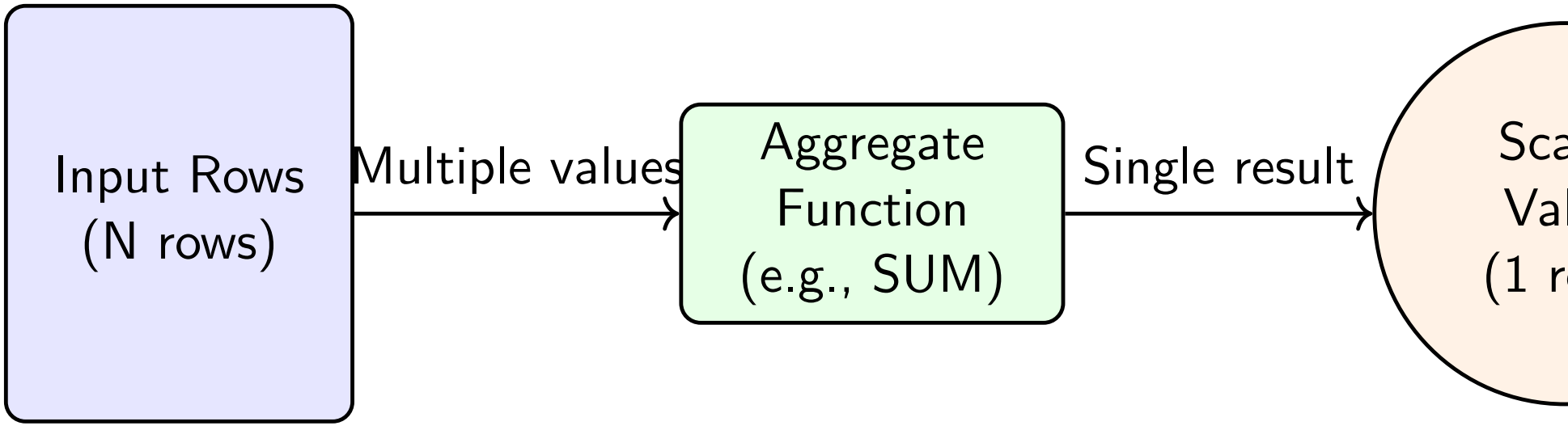
Agenda I

- 1. Introduction to Aggregate Functions
- 2. The COUNT Function
- 3. The SUM and AVG Functions
- 4. The MIN and MAX Functions
- 5. Integration with GROUP BY and HAVING Clauses

Introduction to Aggregate Functions I

- Aggregate functions perform a calculation on a set of values and return a single, scalar value.
- They are deterministic, meaning they always return the same result when called with the same set of input rows.
- Most aggregate functions ignore NULL values during their calculations (the exception is COUNT(*)).
- These functions are commonly used in the SELECT list of a query and in the HAVING clause.
- When used without a GROUP BY clause, an aggregate function operates on all rows returned by the query.

Conceptual Model of Aggregation I



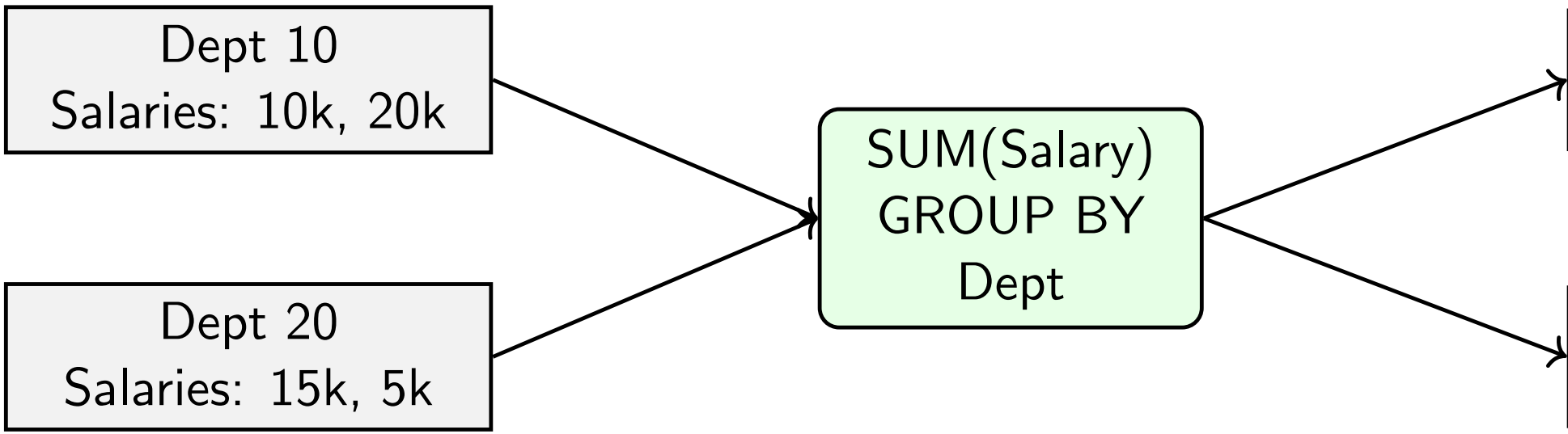
The COUNT Function I

- The COUNT function returns the number of rows or the number of non-NULL values.
- Syntax 1: \{}texttt{COUNT(*)} - Returns the total number of rows in a table or group, including rows containing NULLs and duplicates.
- Syntax 2: \{}texttt{COUNT(column\{}_name)} - Returns the number of non-NULL values present in the specified column.
- Syntax 3: \{}texttt{COUNT(DISTINCT column\{}_name)} - Returns the number of unique, non-NULL values in the specified column.
- Example: \{}texttt{SELECT COUNT(*), COUNT(manager\{}_id), COUNT(DISTINCT manager\{}_id) FROM employees;}

The SUM and AVG Functions I

- \{}texttt{SUM()} returns the total arithmetic sum of a numeric column, ignoring NULL values.
- \{}texttt{AVG()} returns the arithmetic average (mean) of a numeric column. It also strictly ignores NULL values in both the numerator (sum) and denominator (count).
- Crucial Note: \{}texttt{AVG(salary)} is equivalent to \{}texttt{SUM(salary) / COUNT(salary)}, which is NOT the same as \{}texttt{SUM(salary) / COUNT(*)}.
- To treat NULLs as zero in an average calculation, use a null-handling function like COALESCE: \{}texttt{AVG(COALESCE(salary, 0))}.
- Example: \{}texttt{SELECT SUM(salary) AS total\{}_payroll, AVG(salary) AS avg\{}_salary FROM employees WHERE department\{}_id = 10;}

Aggregation with GROUP BY I



The MAX and MIN Functions I

- \{}texttt{MAX()} returns the maximum (highest) value in an expression, and \{}texttt{MIN()} returns the minimum (lowest) value.
- Unlike SUM and AVG, which only operate on numeric data, MIN and MAX can be applied to numeric, character, string, and temporal (date/time) data types.
- For character string columns, MIN and MAX use the database's configured collation rules (usually alphabetical sorting) to determine the highest and lowest values.
- For temporal data, \{}texttt{MAX()} evaluates to the most recent chronological date, while \{}texttt{MIN()} evaluates to the oldest chronological date.
- Example: \{}texttt{SELECT MIN(hire\{}_date) AS earliest\{}_hire, MAX(salary) AS top\{}_salary FROM employees;}

Filtering Aggregates with HAVING I

- The \{}texttt{WHERE} clause filters individual rows BEFORE any grouping or aggregate functions are applied.
- The \{}texttt{HAVING} clause acts as a filter that operates strictly AFTER groups have been formed and aggregate functions have been computed.
- You cannot use aggregate functions directly in the \{}texttt{WHERE} clause (e.g., \{}texttt{WHERE AVG(salary) ¿ 5000} is invalid SQL).
- Example SQL: \{}texttt{SELECT department\{}_id, AVG(salary) FROM employees WHERE status = 'ACTIVE' GROUP BY department\{}_id HAVING AVG(salary) ¿ 50000;}
- Logical Evaluation Order: FROM $\{}rightarrow$ WHERE $\{}rightarrow$ GROUP BY $\{}rightarrow$ HAVING $\{}rightarrow$ SELECT $\{}rightarrow$ ORDER BY.

Summary and Best Practices I

- Aggregate functions summarize granular data into meaningful metrics (SUM, AVG, MIN, MAX, COUNT).
- Always account for NULL propagation: \{}texttt{COUNT(*)} is the only aggregate that counts NULL rows; all others discard them.
- A strict SQL rule: Any column in the SELECT list that is not encapsulated within an aggregate function MUST explicitly appear in the GROUP BY clause.
- Use analytical (window) functions (e.g., \{}texttt{SUM(salary) OVER(PARTITION BY department\{}_id)}) when you need aggregated metrics but also wish to retain individual row-level detail.
- Performance optimization: Aggregating large tables requires significant compute; ensure covering indexes exist on columns used in WHERE and GROUP BY clauses.

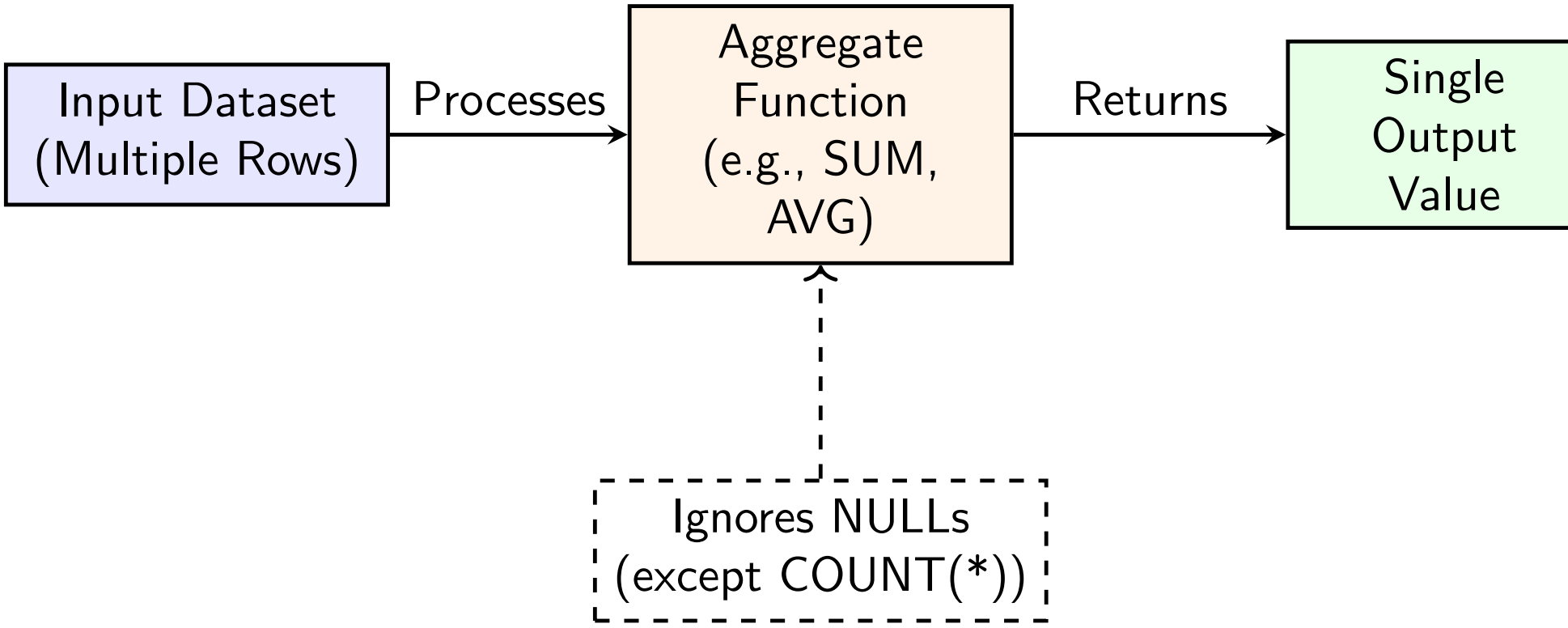
Agenda I

- Introduction to Aggregate (Group) Functions
- COUNT: Tallying Rows and Non-Null Values
- SUM & AVG: Mathematical Aggregations over Numeric Domains
- MAX & MIN: Identifying Extremes in Datasets
- Combining Aggregates with GROUP BY
- Filtering Aggregated Results using the HAVING Clause

Introduction to Aggregate Functions I

- Aggregate functions operate on a collection of values (a set or a multiset) and return a single summarizing value.
- Unlike scalar functions which operate on a per-row basis (e.g., UPPER, ROUND), aggregate functions collapse multiple rows into a single output row.
- By default, aggregate functions ignore NULL values (with the sole exception of COUNT(*)).
- They are deterministic: applying them to the same static dataset will always yield the identical result.
- Most aggregate functions can be used with the DISTINCT keyword to consider only unique values (e.g., COUNT(DISTINCT department_id)).
- Standard SQL Aggregate Functions defined in ISO/IEC 9075: COUNT, SUM, AVG, MAX, MIN.

Data Flow in Aggregate Functions I



The COUNT() Function I

- COUNT() returns the total number of rows that match a specified criterion.
- COUNT(*) returns the total number of rows in the table or group, including rows with NULL values and duplicate rows.
- Example: SELECT COUNT(*) FROM employees; – Returns total employee row count.
- COUNT(column_name) returns the number of non-NULL values in that specific column.
- Example: SELECT COUNT(commission_pct) FROM employees; – Returns only the count of employees who actually have a commission.
- COUNT(DISTINCT column_name) evaluates uniqueness, returning the count of distinct, non-NULL values.

The COUNT() Function II

- Performance Note: Most RDBMS optimizers can resolve COUNT(*) on an indexed table by reading the index tree without accessing underlying table data blocks.

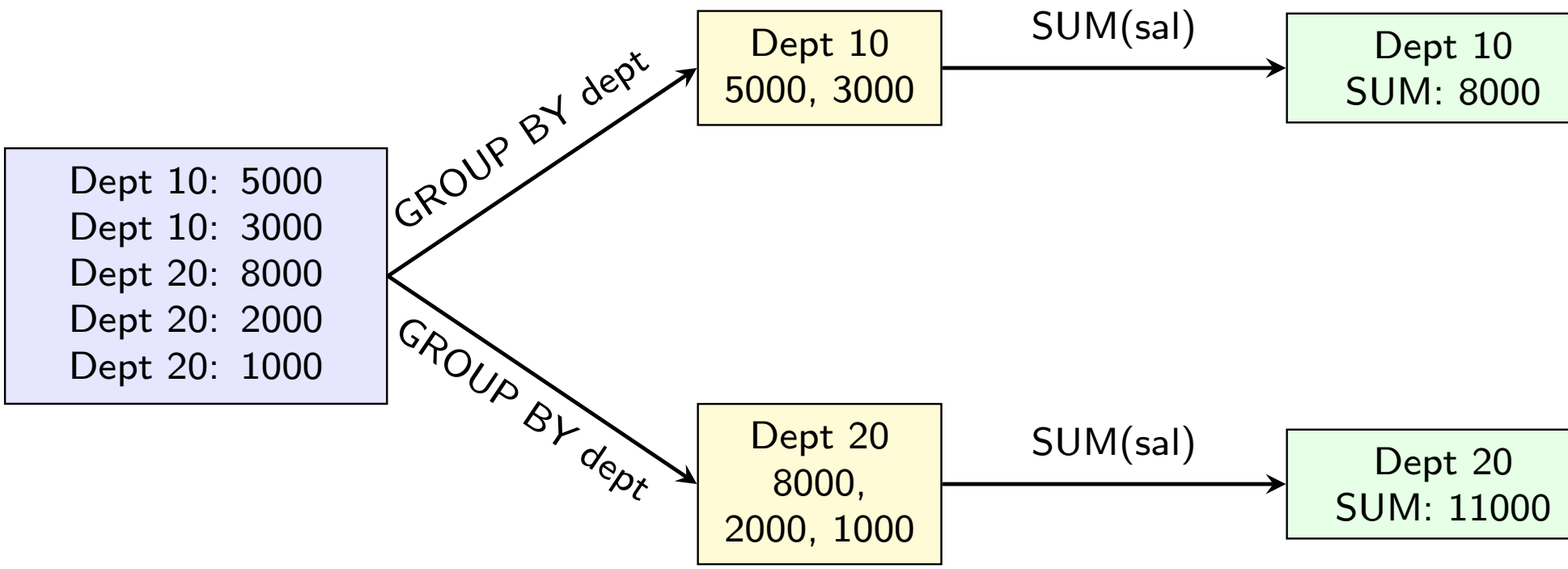
The SUM() and AVG() Functions I

- SUM() calculates the total sum of a numeric column. It completely ignores NULL values during summation.
- Example: SELECT SUM(salary) FROM employees WHERE department_id = 50;
- AVG() calculates the arithmetic mean of a numeric column. It also strictly ignores NULL values.
- Critical conceptual point: AVG(commission_pct) divides the sum of commissions by the number of non-NULL commission entries, NOT by the total number of rows.
- To calculate an average across ALL rows (treating NULLs as zero), you must use COALESCE or NVL: SELECT AVG(COALESCE(commission_pct, 0)) FROM employees;
- Both functions accept the DISTINCT keyword (e.g., SUM(DISTINCT salary)), though this is rarely used in typical business logic.

The MAX() and MIN() Functions I

- MAX() and MIN() return the highest and lowest values in a dataset, respectively.
- Unlike SUM and AVG, which operate only on numeric domains, MAX and MIN operate on numeric, character/string, and temporal (date/time) data types.
- For dates: MAX(hire_date) returns the most recent hire date; MIN(hire_date) returns the oldest date in the dataset.
- For strings: MAX(last_name) returns the last name alphabetically (based on the database character set collation); MIN returns the first.
- Both functions ignore NULL values. If a column contains entirely NULL values for the given set, MAX and MIN will both return NULL.
- Example Query: SELECT MIN(salary) AS lowest_paid, MAX(salary) AS highest_paid FROM employees;

Aggregations with GROUP BY I



Filtering Aggregated Results with HAVING I

- The WHERE clause filters rows BEFORE aggregate functions are applied. You CANNOT use aggregate functions in a WHERE clause.
- The HAVING clause filters groups AFTER aggregate functions are applied, evaluating the condition against the group's aggregate result.
- SQL Engine Execution Order: FROM -¿ WHERE -¿ GROUP BY -¿ HAVING -¿ SELECT -¿ ORDER BY.
- Example Problem: Find departments with an average salary greater than 8000.
- Query: SELECT department_id, AVG(salary) FROM employees GROUP BY department_id HAVING AVG(salary) ¿ 8000;
- Best Practice Tip: Use WHERE to filter base rows to reduce the dataset size before grouping, and strictly use HAVING for conditions involving aggregated values.

Common Pitfalls and Best Practices I

- Pitfall 1: Selecting unaggregated columns without placing them in the GROUP BY clause. This results in the classic 'not a single-group group function' (ORA-00937) database error.
- Pitfall 2: Forgetting that aggregate functions silently drop NULLs. This can lead to mathematically skewed results, especially when computing averages or counts over sparse data.
- Best Practice: Explicitly handle NULLs if they represent a valid numeric state (like a zero commission) using COALESCE(col, 0).
- Best Practice: Understand the performance implications. Aggregations on large datasets require memory-intensive sorting or hashing; ensure proper indexing on GROUP BY columns.

Common Pitfalls and Best Practices II

- Advanced Usage: Modern SQL provides Window Functions (e.g., SUM(salary) OVER (PARTITION BY department_id)) which compute aggregates without collapsing rows, preserving the original data grain.

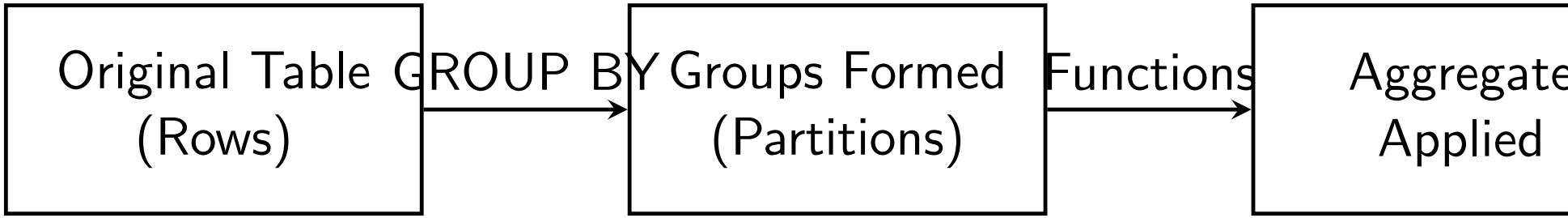
Agenda I

- 1. The GROUP BY Clause
- 2. Aggregate Functions with Grouping
- 3. The HAVING Clause
- 4. WHERE vs. HAVING Differences
- 5. The ORDER BY Clause
- 6. Logical Query Processing Order

The GROUP BY Clause I

- The GROUP BY clause groups rows that have the same values into summary rows.
- It is often used with aggregate functions (COUNT, MAX, MIN, SUM, AVG) to group the result-set by one or more columns.
- Syntax: SELECT column1, COUNT(column2) FROM table_name GROUP BY column1;
- Every non-aggregated column in the SELECT list MUST be present in the GROUP BY clause.
- Example: SELECT department_id, AVG(salary) FROM employees GROUP BY department_id;

GROUP BY Execution Flow I



Aggregate Functions & Multi-Level Grouping I

- Aggregate functions perform a calculation on a set of values and return a single value.
- When GROUP BY is used, the RDBMS partitions the rows and applies the aggregate function to each partition independently.
- Multi-level Grouping: You can group by multiple columns. The grouping happens left-to-right.
- Example: SELECT department_id, job_id, SUM(salary) FROM employees GROUP BY department_id, job_id;
- If a column has NULL values, all NULLs are grouped into a single partition (NULL is considered equal to another NULL for grouping purposes).

The HAVING Clause I

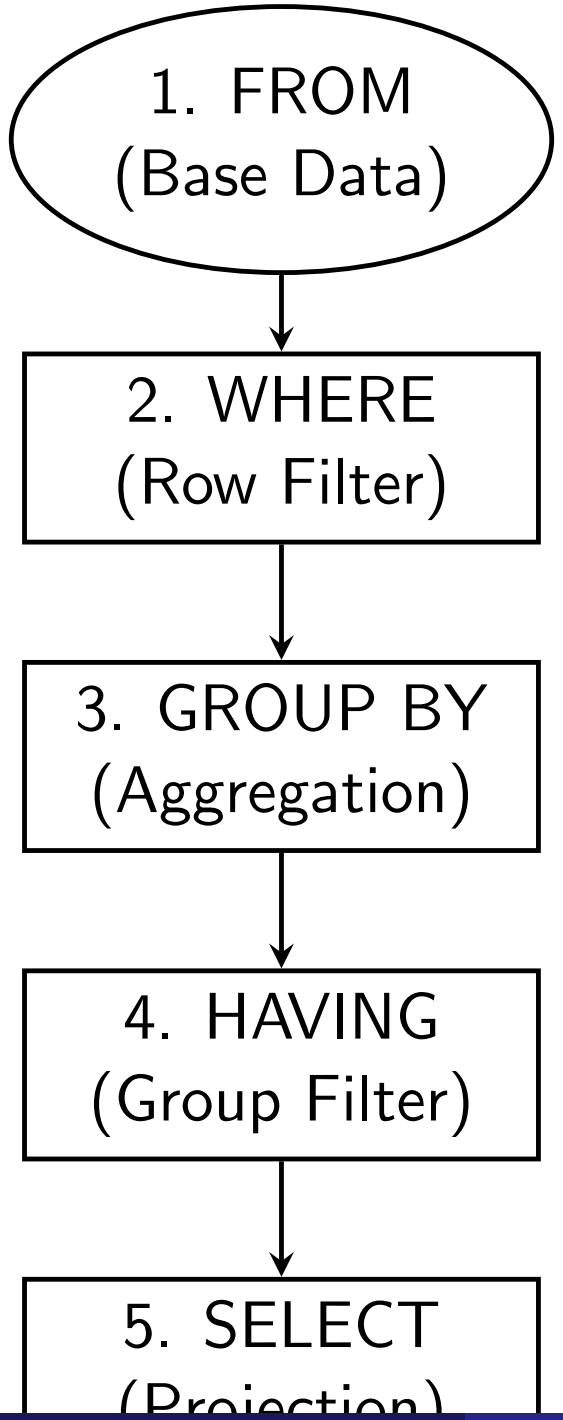
- The HAVING clause was added to SQL because the WHERE keyword cannot be used with aggregate functions.
- HAVING is used to filter groups created by the GROUP BY clause based on a specified condition.
- It acts like the WHERE clause but operates on aggregated data (groups) rather than individual rows.
- Syntax: SELECT column_name, COUNT(column_name) FROM table_name GROUP BY column_name HAVING condition;
- Example: SELECT department_id, COUNT(employee_id) FROM employees GROUP BY department_id HAVING COUNT(employee_id) ¿ 5;

WHERE vs. HAVING I

- WHERE filters rows BEFORE grouping. HAVING filters groups AFTER grouping.
- WHERE can be used with SELECT, UPDATE, and DELETE statements. HAVING is strictly used with SELECT statements containing GROUP BY.
- Performance tip: Filter as much data as possible using WHERE before grouping to reduce the number of rows the RDBMS must aggregate.
- You can use both in the same query: SELECT department_id, SUM(salary) FROM employees WHERE hire_date ¿ '2020-01-01' GROUP BY department_id HAVING SUM(salary) ¿ 100000;

Logical Query Processing Diagram I

Logical Query Processing Diagram II



Rows filtered before grouping.

Groups filtered after aggregation.

The ORDER BY Clause I

- The ORDER BY clause is used to sort the result-set in ascending (ASC) or descending (DESC) order.
- Default sorting is ascending. To sort in descending order, use the DESC keyword.
- Syntax: SELECT column1, column2 FROM table_name ORDER BY column1 ASC, column2 DESC;
- You can sort by column aliases, column positions, or columns not present in the SELECT list (in most RDBMS).
- Execution: It is always evaluated LAST in the logical query processing order.

Summary of Logical Query Processing I

- The RDBMS engine does not execute clauses in the order they are written (lexical order).
- Logical Evaluation Order:
- 1. FROM (with JOINs): Determines the source data.
- 2. WHERE: Filters the source rows.
- 3. GROUP BY: Partitions rows into groups.
- 4. HAVING: Filters the grouped data.
- 5. SELECT: Evaluates expressions, creates aliases, and determines columns.
- 6. DISTINCT: Removes duplicate rows.
- 7. ORDER BY: Sorts the final output.

Agenda I

- 1. Introduction to Joins and Relational Algebra
- 2. CROSS JOIN (Cartesian Product)
- 3. INNER JOIN
- 4. LEFT and RIGHT OUTER JOIN
- 5. FULL OUTER JOIN
- 6. SELF JOIN
- 7. Visualizing Joins (Diagrams)

Introduction to Joins I

- A JOIN clause is used to combine rows from two or more tables, based on a related column between them.
- In Relational Algebra, a Join is a derived operator, essentially a Cartesian Product followed by a Selection process.
- Foreign Keys play a critical role in Joins, as they maintain the referential integrity across the relations.
- Primary operations involved: Filtering tuples that satisfy the join condition (predicate) and projecting the requested attributes.
- Joins reduce data redundancy by allowing database normalization, fetching correlated data on-the-fly rather than storing wide, denormalized tables.

CROSS JOIN (Cartesian Product) I

- A CROSS JOIN returns the Cartesian product of rows from the tables in the join.
- It combines each row of the first table with each row of the second table indiscriminately.
- If Table A has N rows and Table B has M rows, the result set will have N * M rows.
- Syntax: SELECT column_list FROM table1 CROSS JOIN table2;
- Example Query: SELECT Employees.Name, Departments.DeptName FROM Employees CROSS JOIN Departments;
- Use cases: Generating all possible combinations for matrices, test data generation, or comprehensive pairings.

INNER JOIN I

- The INNER JOIN keyword selects records that have matching values in both tables.
- If there are records in the 'Employees' table that do not have matches in 'Departments', these employees will not be shown.
- Syntax: SELECT column_list FROM table1 INNER JOIN table2 ON table1.column_name = table2.column_name;
- Equi-Join: A specific type of INNER JOIN that uses only the equality operator (=) in the ON clause.
- Theta-Join: An INNER JOIN that uses comparison operators other than equality (e.g., ¡, ¿, ¡=).
- Example Query: SELECT e.EmpName, d.DeptName FROM Employee e INNER JOIN Department d ON e.DeptID = d.DeptID;

LEFT OUTER JOIN I

- The LEFT JOIN returns all records from the left table (table1), and the matched records from the right table (table2).
- The result is NULL from the right side, if there is no match for the left table's record.
- Crucial for identifying missing relationships (e.g., finding all employees who have not been assigned a department).
- Syntax: SELECT column_list FROM table1 LEFT JOIN table2 ON table1.column_name = table2.column_name;
- Example Query: SELECT Customers.CustomerName, Orders.OrderID FROM Customers LEFT JOIN Orders ON Customers.CustomerID = Orders.CustomerID;
- In this query, customers without any orders will still appear in the result set, with a NULL OrderID.

RIGHT OUTER JOIN I

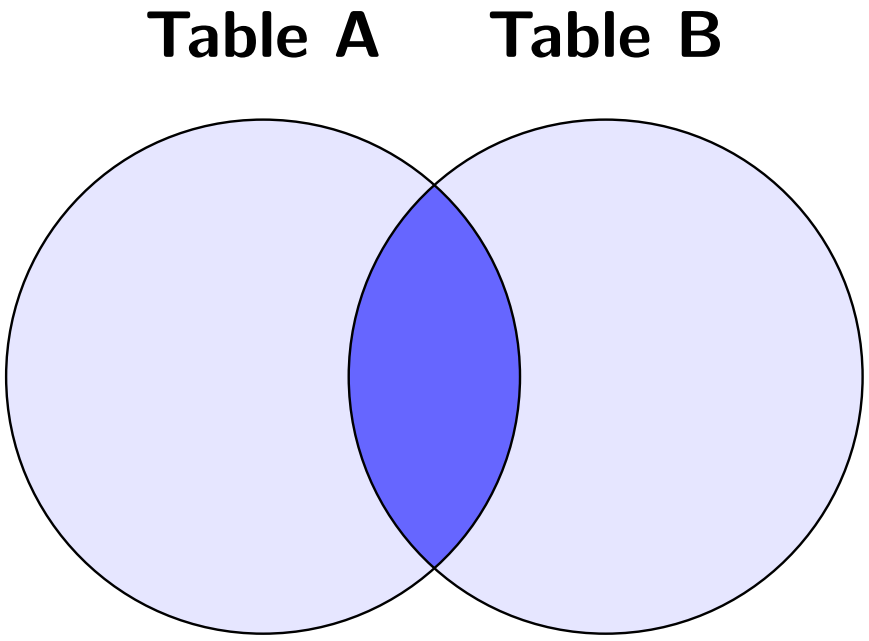
- The RIGHT JOIN returns all records from the right table (table2), and the matched records from the left table (table1).
- The result is NULL from the left side, when there is no match for the right table's record.
- Functionally symmetrical to the LEFT JOIN; A LEFT JOIN B is functionally equivalent to B RIGHT JOIN A.
- Syntax: SELECT column_list FROM table1 RIGHT JOIN table2 ON table1.column_name = table2.column_name;
- Example Query: SELECT Employees.LastName, Orders.OrderID FROM Orders RIGHT JOIN Employees ON Orders.EmployeeID = Employees.EmployeeID;
- Use case: Guaranteeing the preservation of the target entity's records while looking up optional descriptive attributes.

FULL OUTER JOIN & SELF JOIN I

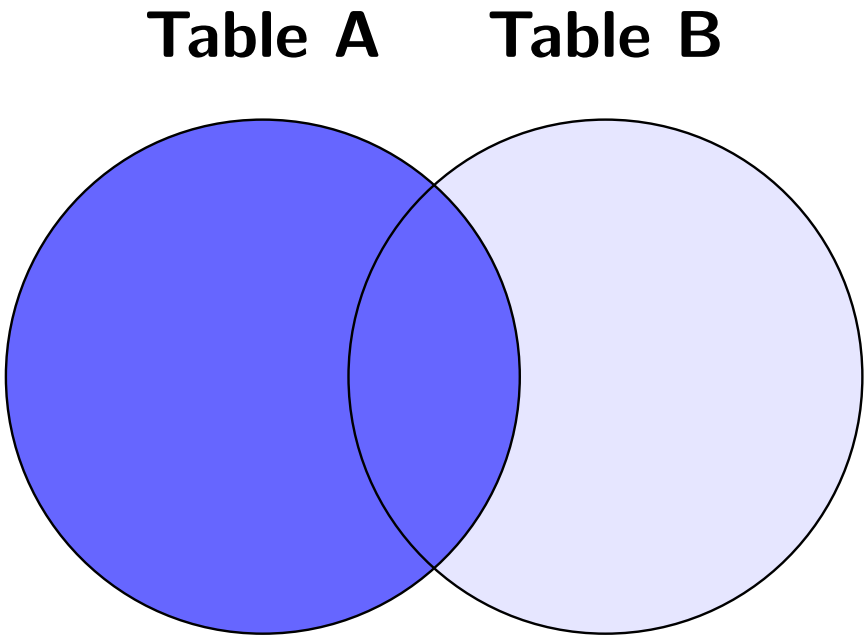
- FULL OUTER JOIN returns all records when there is a match in either left or right table records.
- It essentially combines the result sets of both LEFT and RIGHT outer joins.
- Syntax: SELECT column_list FROM table1 FULL OUTER JOIN table2 ON table1.column_name = table2.column_name;
- SELF JOIN is a regular join, but the table is joined with itself.
- Requires the use of table aliases to distinguish the two instances of the same table during execution.
- Example Query (Manager-Employee hierarchy): SELECT e1.EmpName AS Employee, e2.EmpName AS Manager FROM Employees e1 INNER JOIN Employees e2 ON e1.ManagerID = e2.EmpID;

Visualizing Joins with Venn Diagrams I

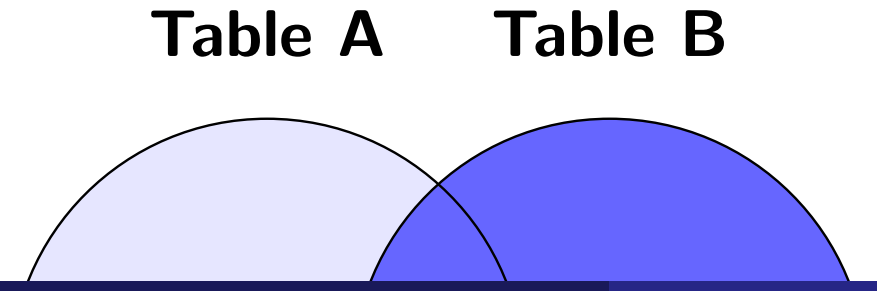
Visualizing Joins with Venn Diagrams II

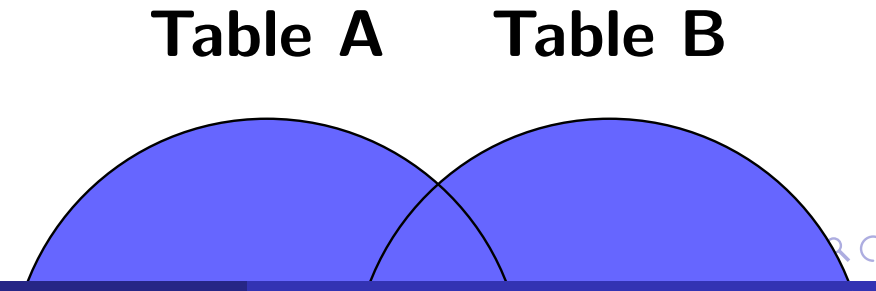


INNER JOIN



LEFT JOIN





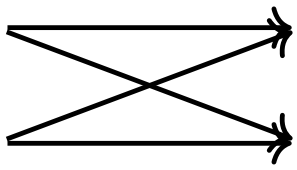
CROSS JOIN Mechanics Visualization I

Table A (Colors)

ID — Color
1 — Red
2 — Blue

Table B (Sizes)

ID — Size
1 — S
2 — M



CROSS JOIN Result (4 rows

A.Color — B.Size
Red — S
Red — M
Blue — S
Blue — M

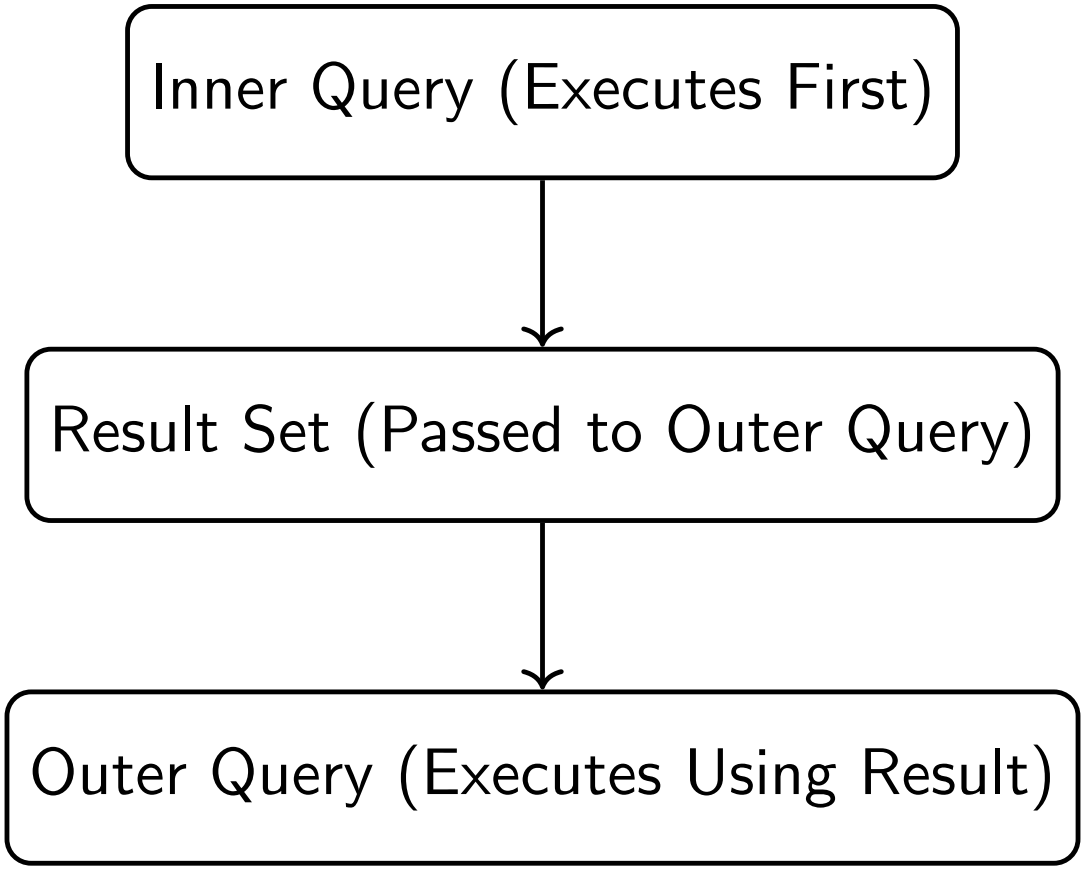
Agenda I

- Introduction to Subqueries
- Nested Queries: Execution and Syntax
- Correlated Subqueries
- EXISTS and NOT EXISTS Operators
- Performance Considerations and Best Practices

Introduction to Subqueries I

- A subquery is a SQL query nested inside a larger query, also known as an inner query or nested query.
- It can be placed in a SELECT, FROM (inline view), or WHERE clause.
- Subqueries can be nested inside another subquery, up to multiple levels depending on the RDBMS.
- Example: SELECT employee_id FROM employees WHERE department_id IN (SELECT department_id FROM departments WHERE location_id = 1700);

Nested Query Execution Flow I



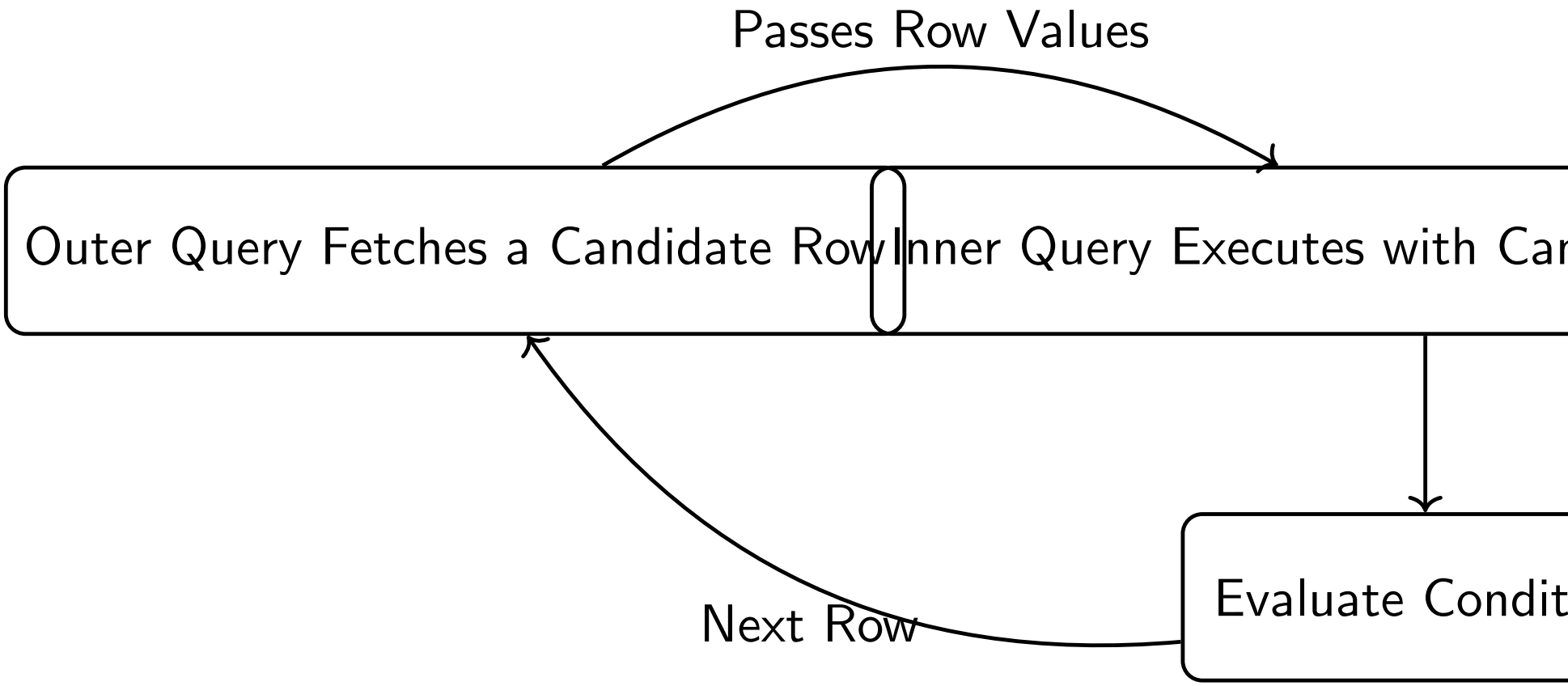
Types of Subqueries I

- Single-Row Subquery: Returns exactly zero or one row. Used with single-row comparison operators (=, ¿, ¡, ¿=, ¡=, ¡¿).
- Example: SELECT first_name FROM employees WHERE salary ¿ (SELECT AVG(salary) FROM employees);
- Multiple-Row Subquery: Returns one or more rows. Must be used with multiple-row operators (IN, ANY, ALL).
- Multiple-Column Subquery: Returns more than one column. Often used to compare multiple columns simultaneously against a tuple.

Multiple-Row Operators: IN, ANY, ALL I

- IN: Matches any value in the list or returned by the subquery.
- ANY: Compares a value to each value returned by a subquery. For example, ¿ ANY means greater than the minimum value.
- ALL: Compares a value to every value returned by a subquery. For example, ¿ ALL means greater than the maximum value.
- Example: SELECT employee_id FROM employees WHERE salary ¡ ALL (SELECT salary FROM employees WHERE department_id = 50);

Correlated Subquery Execution Flow I



Correlated Subqueries vs Nested Queries I

- Uncorrelated Subquery: The inner query executes exactly once before the outer query, completely independent of it.
- Correlated Subquery: The inner query references a column from a table in the outer query. It is evaluated once for each row processed by the outer query.
- Example: SELECT e1.first_name, e1.salary FROM employees e1 WHERE e1.salary ¿ (SELECT AVG(salary) FROM employees e2 WHERE e1.department_id = e2.department_id);
- Performance Impact: Correlated subqueries can be significantly slower on large datasets due to repeated execution.

EXISTS and NOT EXISTS Operators I

- EXISTS evaluates to TRUE if the subquery returns at least one row, otherwise FALSE. It does not evaluate the actual data returned.
- NOT EXISTS evaluates to TRUE if the subquery returns no rows.
- Highly efficient for correlated subqueries because the engine stops searching (short-circuits) as soon as it finds the first matching row.
- Example: SELECT department_name FROM departments d WHERE EXISTS (SELECT 1 FROM employees e WHERE e.department_id = d.department_id);

Summary and Best Practices I

- Use uncorrelated nested queries for one-time computation of aggregates or sets.
- Use correlated subqueries when you need row-by-row comparative logic based on outer query values.
- Beware of performance bottlenecks with correlated subqueries; consider using JOINs or Window Functions as an alternative.
- Prefer EXISTS over IN for correlated subqueries as it provides better performance via short-circuit evaluation.

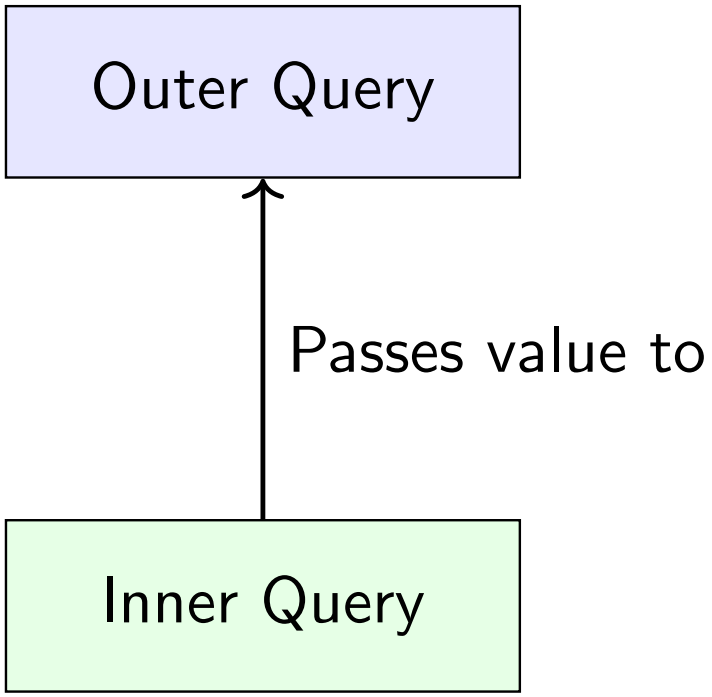
Agenda I

- 1. Introduction to Subqueries
- 2. Subquery Execution Flow
- 3. Nested Queries
- 4. Correlated Subqueries
- 5. Correlated vs. Non-Correlated
- 6. Summary and Best Practices

Introduction to Subqueries I

- A subquery is a query nested inside another query such as SELECT, INSERT, UPDATE, or DELETE.
- It is also known as an Inner Query or Nested Query, while the statement containing it is the Outer Query.
- Subqueries can return individual values (scalar), a single list of values (row/column), or a virtual table.
- They must be enclosed in parentheses () and placed on the right side of the comparison operator.
- Example syntax: SELECT column_name FROM table_name WHERE column_name = (SELECT column_name FROM table_name WHERE condition);

Subquery Execution Flow (Diagram) I



Types of Subqueries I

- 1. Single-row subquery: Returns zero or one row. Uses single-row operators like =, ¿, ¡, ¿=, ¡=, ¡¿.
- 2. Multiple-row subquery: Returns one or more rows. Uses multiple-row operators like IN, ANY, ALL.
- 3. Multiple-column subquery: Returns more than one column. Often used with the IN operator.
- Example of Multiple-row: SELECT ename, sal, deptno FROM emp WHERE sal IN (SELECT MIN(sal) FROM emp GROUP BY deptno);

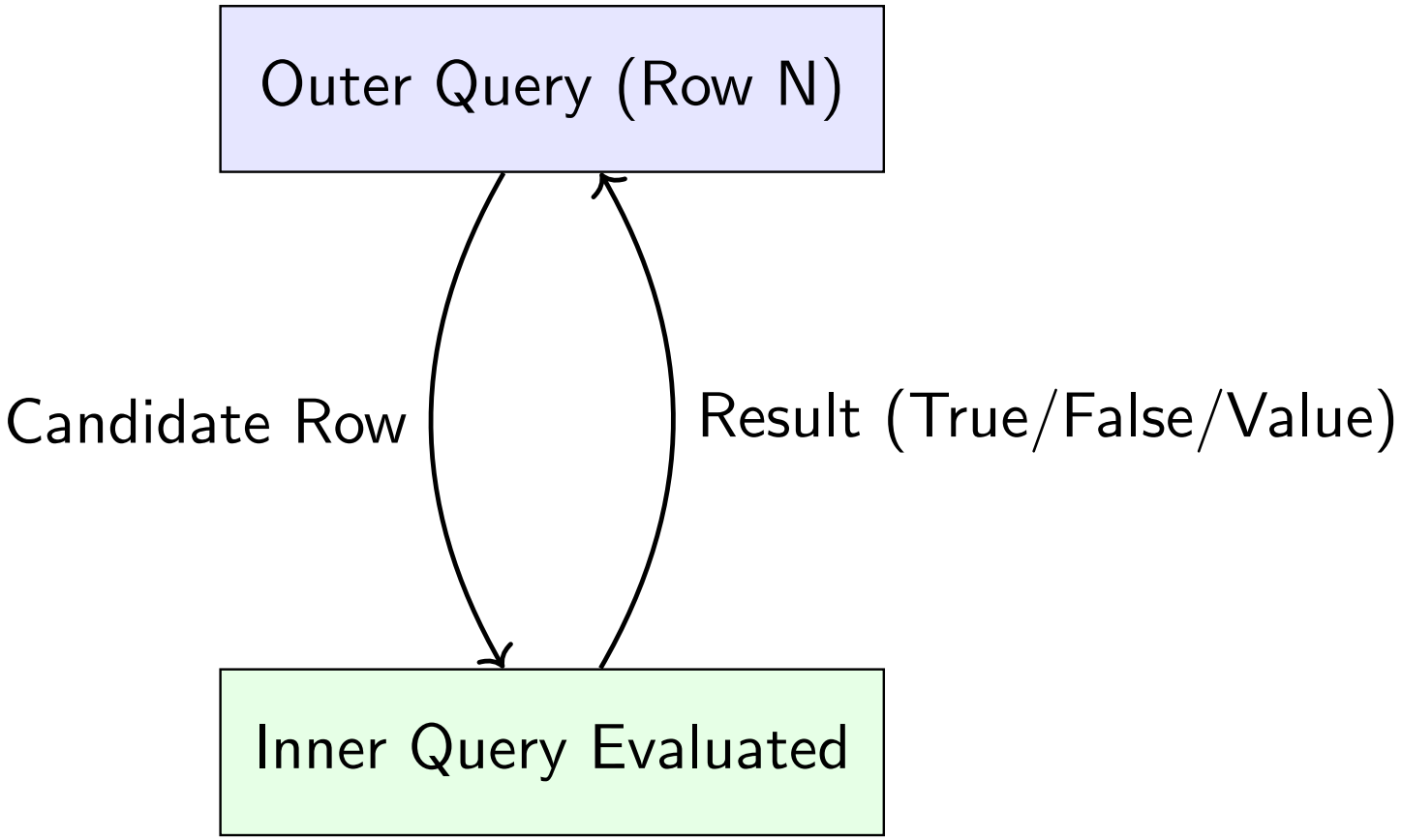
Nested Queries: Execution Details I

- In a standard (non-correlated) nested query, the Inner Query executes EXACTLY ONCE before the Outer Query.
- The result of the Inner Query is held in memory and substituted into the Outer Query.
- Example: Find employees earning more than 'JONES'.
- SELECT ename, sal FROM emp WHERE sal ¿ (SELECT sal FROM emp WHERE ename = 'JONES');
- Step 1: Inner query executes, fetching JONES's salary (e.g., 2975).
- Step 2: Outer query becomes: SELECT ename, sal FROM emp WHERE sal ¿ 2975;

Correlated Subqueries: Concept I

- A Correlated Subquery is a subquery that uses values from the Outer Query.
- Unlike a standard nested query, it CANNOT be evaluated independently of the Outer Query.
- Execution flow: The Outer Query fetches a row, passes the required column value to the Inner Query, the Inner Query executes, and returns the result to the Outer Query condition.
- This process repeats for EVERY ROW considered by the Outer Query (row-by-row execution).
- Often uses the EXISTS operator or references a table alias defined in the Outer Query.

Correlated Subquery Execution (Diagram) I



Correlated Subqueries: Example I

- Example: Find employees whose salary is above the average salary of their own department.
- SELECT ename, sal, deptno FROM emp e1 WHERE sal ¿ (SELECT AVG(sal) FROM emp e2 WHERE e1.deptno = e2.deptno);
- For each row in 'e1', the subquery computes the average salary for that specific 'deptno'.
- Performance consideration: Because it executes per row, correlated subqueries can be computationally expensive ($O(N^2)$ *complexityifunindexed*)*comparedtostandardjoinsorwindowf*

Summary & Best Practices I

- Subqueries simplify complex queries by breaking them into logical steps.
- Non-correlated subqueries execute bottom-up (inner first), whereas correlated subqueries execute top-down (outer passes row to inner).
- Avoid using subqueries when a simple JOIN would suffice, as standard JOINs are typically better optimized by the RDBMS engine.
- Use EXISTS instead of IN for correlated subqueries checking for existence, as EXISTS short-circuits upon finding the first match.
- Always enclose subqueries in parentheses and format them with proper indentation for readability.

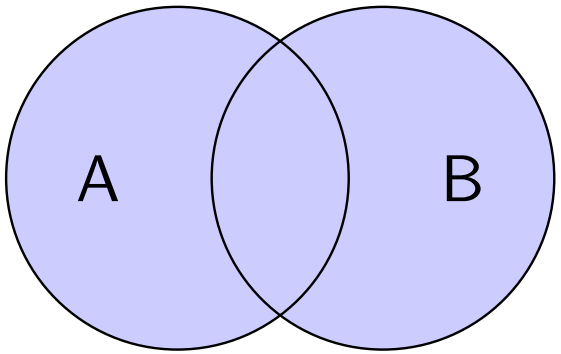
Agenda I

- Introduction to Set Operators
- Visualizing Set Operations (Venn Diagrams)
- The UNION Operator
- The UNION ALL Operator
- The INTERSECT Operator
- The MINUS Operator
- Differences between Set Operators
- Guidelines and ORDER BY Restrictions

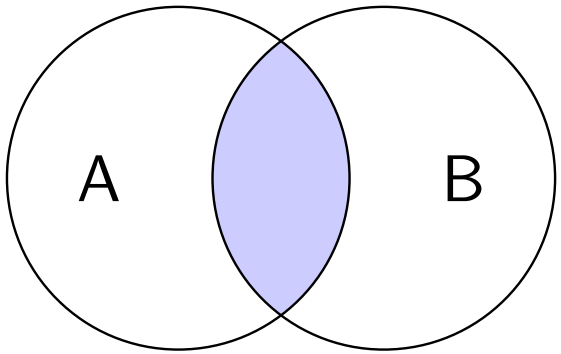
Introduction to Set Operators I

- Set operators combine the results of two or more component queries into a single result set.
- Queries containing set operators are called compound queries.
- Fundamental requirements for set operations:
- 1. The number of expressions in the SELECT lists must be the same in all queries.
- 2. The data types of the corresponding columns in the SELECT lists must match or be implicitly convertible.
- 3. The column names of the final result set are determined by the column names in the first SELECT statement.
- Available Set Operators in SQL: UNION, UNION ALL, INTERSECT, and MINUS.

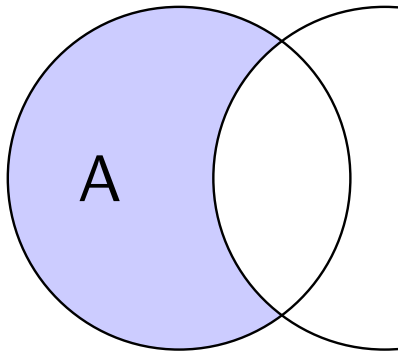
Visualizing Set Operations I



UNION



INTERSECT

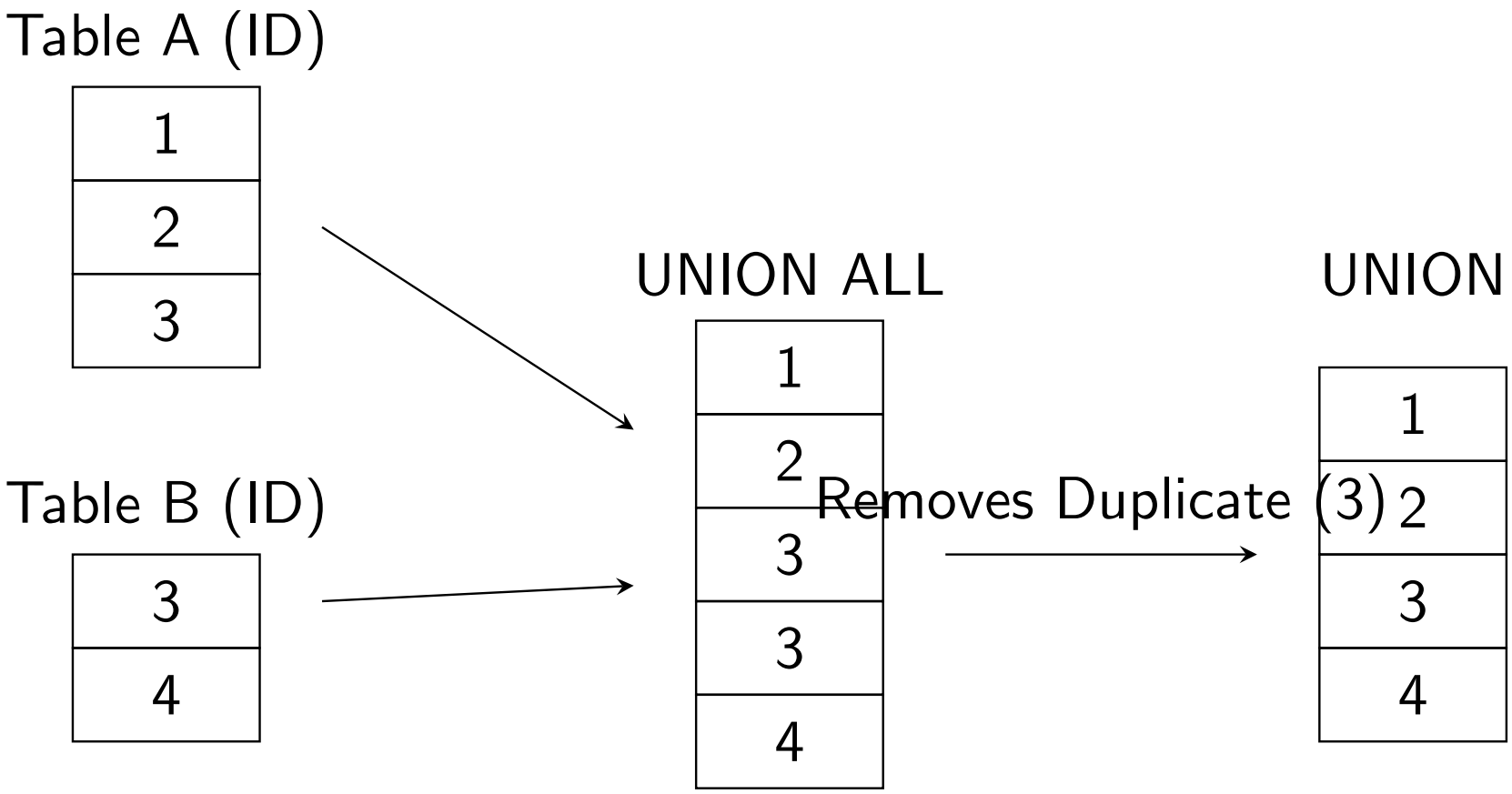


MINUS (A -

The UNION Operator I

- The UNION operator returns all distinct rows selected by either query.
- It automatically eliminates duplicate rows from the result set.
- The UNION operator performs a sort operation on the result set to remove duplicates, which can impact performance for large datasets.
- Syntax:
- SELECT column1, column2 FROM table1
- UNION
- SELECT column1, column2 FROM table2;
- Example:
- SELECT employee_id FROM employees
- UNION
- SELECT employee_id FROM retired_employees;

UNION vs UNION ALL: Data Flow I



The UNION ALL Operator I

- The UNION ALL operator returns all rows selected by either query, including all duplicates.
- Unlike UNION, it does not sort the data or remove duplicates.
- Because it skips the sorting and deduplication step, UNION ALL is significantly faster than UNION.
- Best Practice: Use UNION ALL by default unless you specifically need to eliminate duplicate records.
- Example: Combining monthly sales data tables into a single report.
- SELECT product_id, amount FROM sales_jan
- UNION ALL
- SELECT product_id, amount FROM sales_feb;

The INTERSECT Operator I

- The INTERSECT operator returns only the distinct rows that are present in BOTH queries' result sets.
- It identifies the common elements between two sets of data.
- Like UNION, the INTERSECT operator also sorts the result set and removes duplicate rows.
- Example: Finding employees who are both managers and project leads.
- SELECT employee_id FROM managers
- INTERSECT
- SELECT employee_id FROM project_leads;
- If an employee is in the managers table but not in the project_leads table, they will not be included in the result.

The MINUS Operator I

- The MINUS operator (known as EXCEPT in standard SQL and some other RDBMS like SQL Server/PostgreSQL) returns all distinct rows selected by the first query but not the second.
- The order of queries is crucial for the MINUS operator (A MINUS B is not the same as B MINUS A).
- It also removes duplicates and sorts the final result set.
- Example: Finding products that have never been sold.
- SELECT product_id FROM products
- MINUS
- SELECT product_id FROM order_details;
- This query effectively subtracts the set of ordered products from the set of all inventory products.

Restrictions and Guidelines I

- 1. ORDER BY Clause: The ORDER BY clause can only appear once, at the very end of the compound query.
- It applies to the final merged result set, not individual SELECT statements.
- You must sort by columns from the first SELECT statement (by name, alias, or positional notation).
- 2. Data Types: BLOB, CLOB, BFILE, and VARRAY data types cannot be used with set operators.
- 3. NULL Values: Set operators treat NULLs as equal to other NULLs when identifying duplicates.
- Example of correct ORDER BY usage:
- SELECT dept_id, dept_name FROM old_departments
- UNION ALL
- SELECT dept_id, dept_name FROM new_departments
- ORDER BY dept_id ASC;

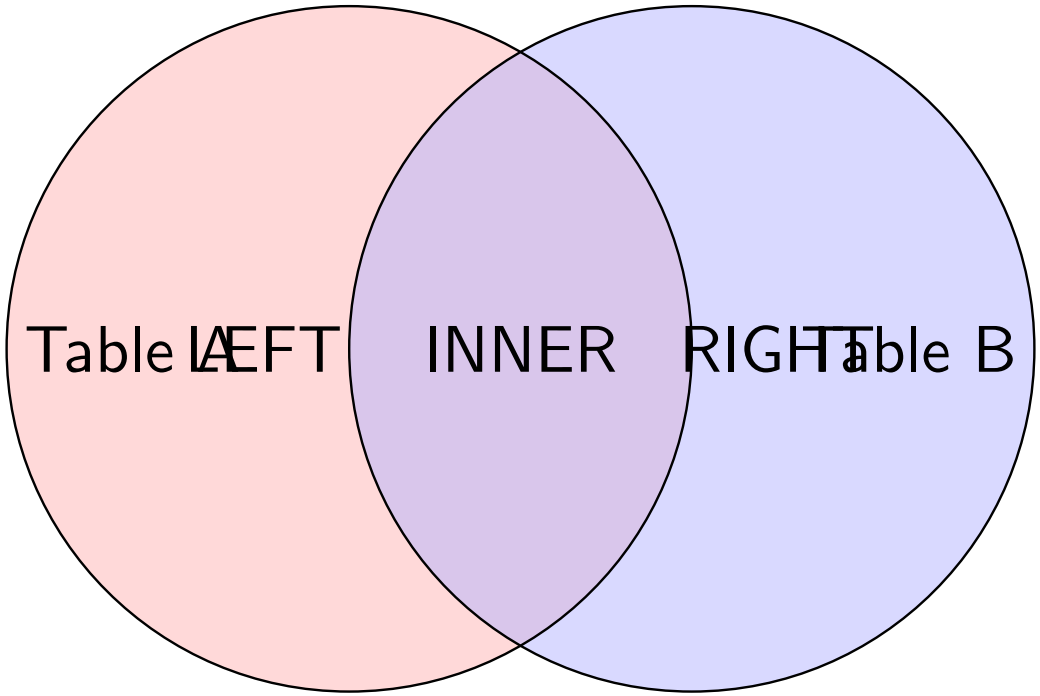
Agenda I

- 1. Recap of SQL Joins and Aggregations
- 2. Visualizing Joins with Venn Diagrams
- 3. Problem 1: Correlated Subqueries for Max Values
- 4. Schema Visualization using ER Diagrams
- 5. Problem 2: Implementing Anti-Joins for Missing Records
- 6. Query Optimization Best Practices

Recap: Joins and Aggregation I

- Joins allow combining columns from one or more tables in a relational database based on a related column between them.
- INNER JOIN returns records that have matching values in both tables, forming the logical intersection.
- LEFT (OUTER) JOIN returns all records from the left table, and the matched records from the right table. Unmatched records result in NULLs on the right side.
- Aggregate functions (COUNT, SUM, AVG, MAX, MIN) operate on a set of rows and return a single summarizing value.
- The GROUP BY statement groups rows that have the same values into summary rows, enabling partition-level aggregation.

Visualizing Table Joins I



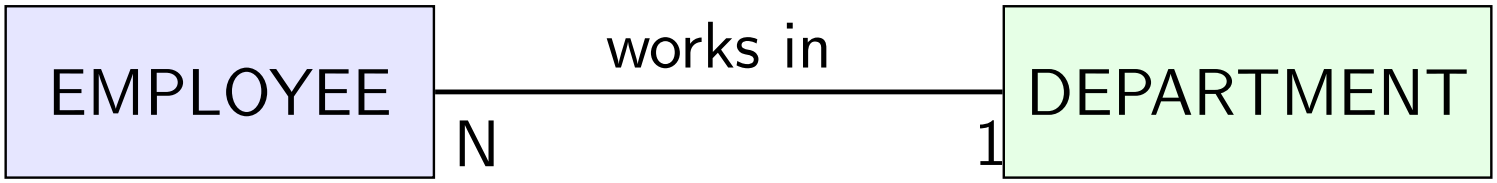
Problem 1: Top Earners per Department I

- Scenario: We need to find the employee who earns the highest salary in each department.
- Common mistake: Using MAX(salary) with GROUP BY department_id gives the max salary but loses the employee's name because the SELECT list must match the GROUP BY clause or be aggregated.
- Approach 1: Correlated Subquery. For each row evaluated in the outer query, the inner query computes the maximum salary for that specific employee's department.
- Approach 2: Using window functions like RANK() or DENSE_RANK() with PARTITION BY department_id ORDER BY salary DESC.

Solution 1: Correlated Subquery Example I

- SQL Query Implementation:
- SELECT e.employee_name, e.department_id, e.salary
- FROM employees e
- WHERE e.salary = (
- SELECT MAX(salary)
- FROM employees m
- WHERE m.department_id = e.department_id
-);
- Explanation: The inner subquery references the outer query's 'e.department_id' and computes the max salary exclusively for that department. The outer WHERE clause checks if the current employee's salary matches this maximum.

Entity-Relationship Visualization I



Problem 2: Finding Orphan Records I

- Scenario: Find all departments that currently have zero employees associated with them.
- This is a classic 'Anti-Join' problem, asking for records in one set that have no correspondence in another.
- Approach 1: LEFT JOIN with IS NULL. We join Departments to Employees and filter for rows where the Employee ID IS NULL.
- Approach 2: NOT EXISTS. We use a correlated subquery to check the negation of existence for any employee mapped to the department.
- Approach 3: NOT IN (Beware: NOT IN can return unexpected empty sets if the subquery returns any NULL values).

Solution 2: Implementing Anti-Joins I

- Using LEFT JOIN:
- SELECT d.department_name
- FROM departments d
- LEFT JOIN employees e ON d.department_id = e.department_id
- WHERE e.employee_id IS NULL;
- Using NOT EXISTS (Often evaluated more efficiently by RDBMS optimizers):
- SELECT d.department_name
- FROM departments d
- WHERE NOT EXISTS (
- SELECT 1 FROM employees e WHERE e.department_id = d.department_id
-);

Query Optimization Best Practices I

- Always filter data as early as possible using the WHERE clause rather than HAVING, since HAVING applies after the expensive aggregation step.
- Avoid using SELECT * in production code; only fetch the columns you actually need to reduce disk I/O and network overhead.
- Prefer EXISTS or NOT EXISTS over IN or NOT IN when working with subqueries, particularly if the subquery result set is large or susceptible to NULLs.
- Ensure foreign keys and frequently joined columns possess appropriate B-tree indexes to accelerate JOIN and lookup operations.

Agenda I

- Definition of Normalization
- Data Anomalies (Insertion, Deletion, Update)
- Importance of Normalization
- Overview of Normal Forms

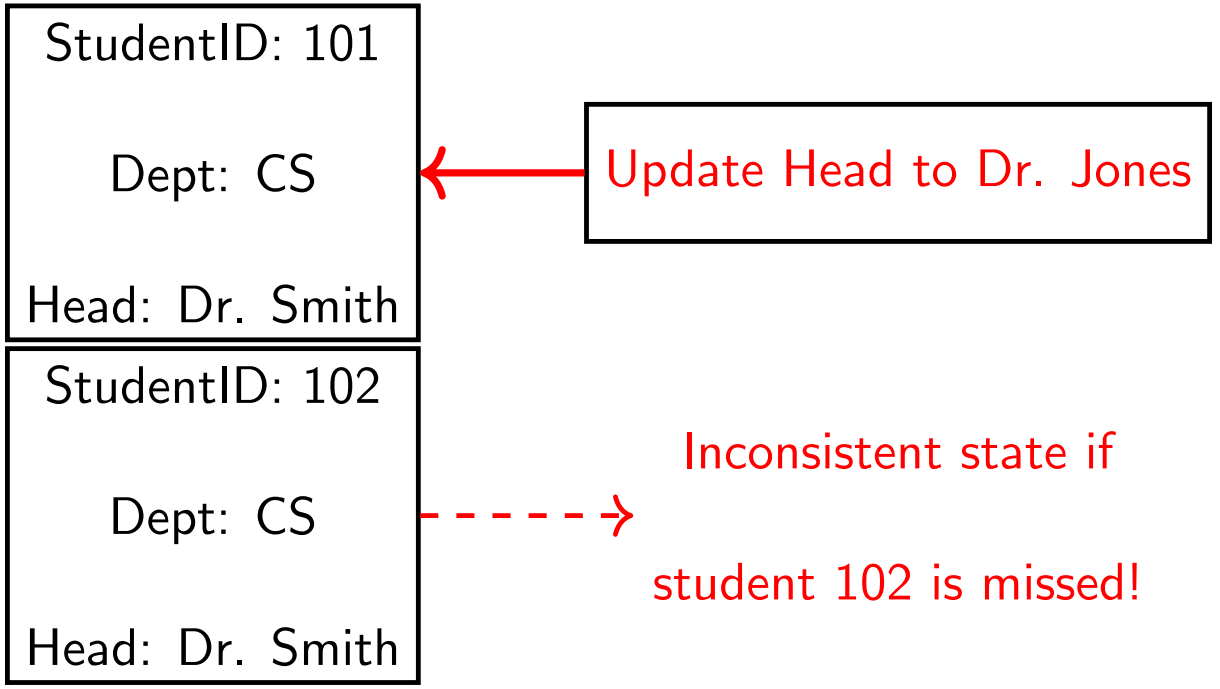
What is Normalization? I

- Normalization is a formal process in relational database design that organizes data to minimize redundancy.
- Proposed by Edgar F. Codd in 1970 as part of the relational model.
- It involves dividing a database into two or more tables and defining relationships between them.
- The main goal is to ensure data dependencies make sense and to isolate data so that additions, deletions, and modifications can be made in just one table and then propagated through the rest of the database via defined relationships.

Data Anomalies: The Need for Normalization I

- Unnormalized databases suffer from data anomalies when inserting, updating, or deleting data.
- Insertion Anomaly: Inability to add data to the database due to absence of other data. E.g., cannot add a new course without enrolling a student if course and student data are in the same table.
- Deletion Anomaly: Unintended loss of data due to deletion of other data. E.g., deleting a student's enrollment record might also delete the only record of the course they were enrolled in.
- Update Anomaly: Data inconsistency that results from data redundancy and a partial update. E.g., changing a department's location requires updating multiple student records; if one is missed, the database becomes inconsistent.

Visualizing Update Anomalies I



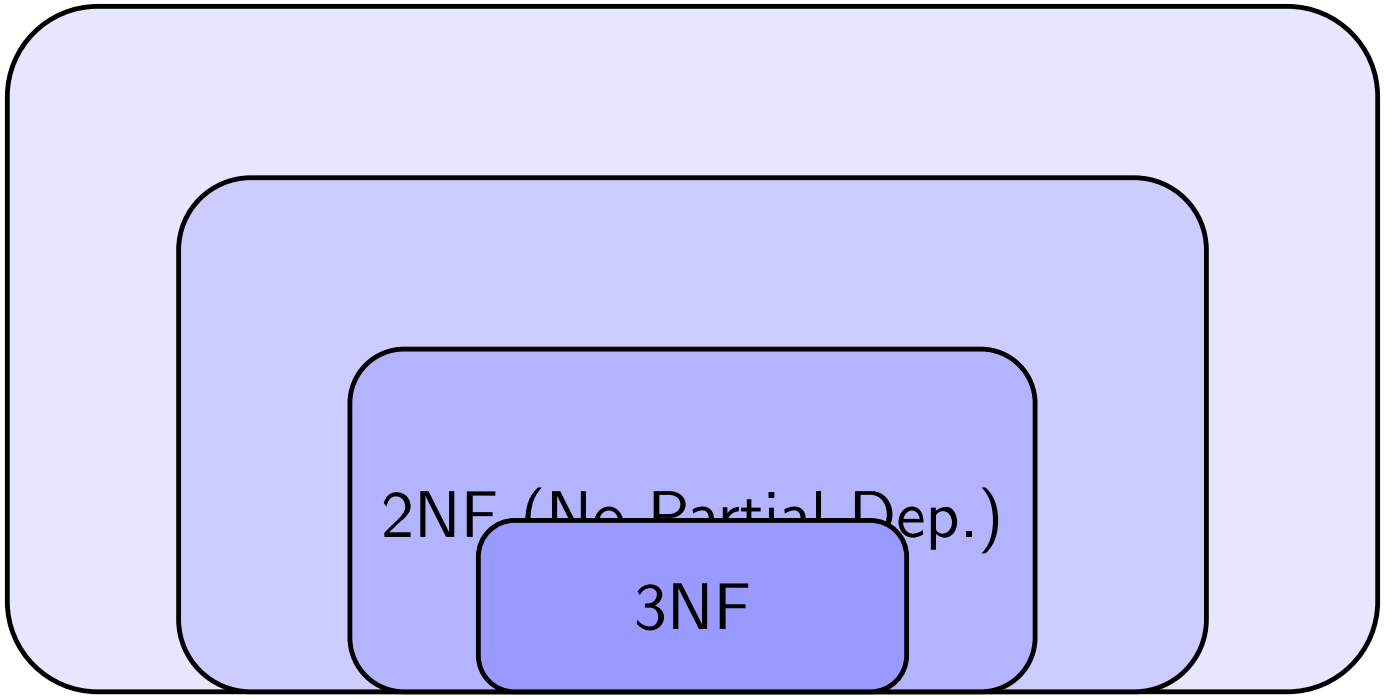
Importance of Normalization I

- Minimizes Data Redundancy: Reduces storage costs and improves buffer cache efficiency since more distinct rows fit into memory.
- Eliminates Data Modification Anomalies: Ensures that DML operations (INSERT, UPDATE, DELETE) leave the database in a consistent state.
- Simplifies Queries: Properly normalized tables often have cleaner schemas, although complex queries might require multiple JOIN operations.
- Improves Data Integrity: Enforcement of functional dependencies and referential integrity (foreign keys) becomes straightforward and reliable.
- Schema Evolution: Adding new attributes or relationships often requires fewer structural changes in a normalized database.

Functional Dependencies (The Foundation) I

- A functional dependency X -¿ Y implies that if two tuples agree on the attributes X, they must also agree on the attributes Y.
- It is a many-to-one relationship from the domain of X to the domain of Y.
- Determinant: The attribute or set of attributes on the left side (X).
- Dependent: The attribute or set of attributes on the right side (Y).
- Example: \{}texttt{StudentID -¿ StudentName, DateOfBirth}. Knowing the StudentID uniquely identifies the Name and DOB.
- Normalization uses functional dependencies to decompose relations, ensuring that each non-key attribute is fully functionally dependent on the primary key.

Hierarchy of Normal Forms I



The Trade-off: Normalization vs. Performance I

- While normalization ensures data integrity, higher normal forms require dividing data across many tables.
- This division means that retrieving a complete business entity often requires complex multi-table SQL \{}texttt{JOIN} operations.
- Extensive JOINs consume significant CPU and memory resources, potentially slowing down read-heavy queries in OLAP (Online Analytical Processing) systems.
- Denormalization: The deliberate introduction of redundancy to improve read performance. It is a controlled violation of normalization rules.
- Rule of thumb: Normalize until it hurts performance, then denormalize until it works, using mechanisms like materialized views or caching.

Summary & Next Steps I

- Normalization is a systematic approach to decomposing tables to eliminate data redundancy and anomalies.
- Core benefits include data integrity, simplified constraints, and efficient storage.
- The process is driven by the concept of Functional Dependencies.
- Next Lecture: Deep dive into First Normal Form (1NF), Second Normal Form (2NF), and the concept of partial dependencies.
- Reading Assignment: Chapter 14 (Database Design) and Chapter 15 (Normalization) in standard database textbooks.

Agenda I

- 1. What is Normalization?
- 2. The Problem: Data Redundancy
- 3. Understanding Data Anomalies
- 4. Types of Anomalies in Detail
- 5. Introduction to Functional Dependencies
- 6. The Normalization Process
- 7. Goals and Advantages of Normalization
- 8. Trade-offs and Summary

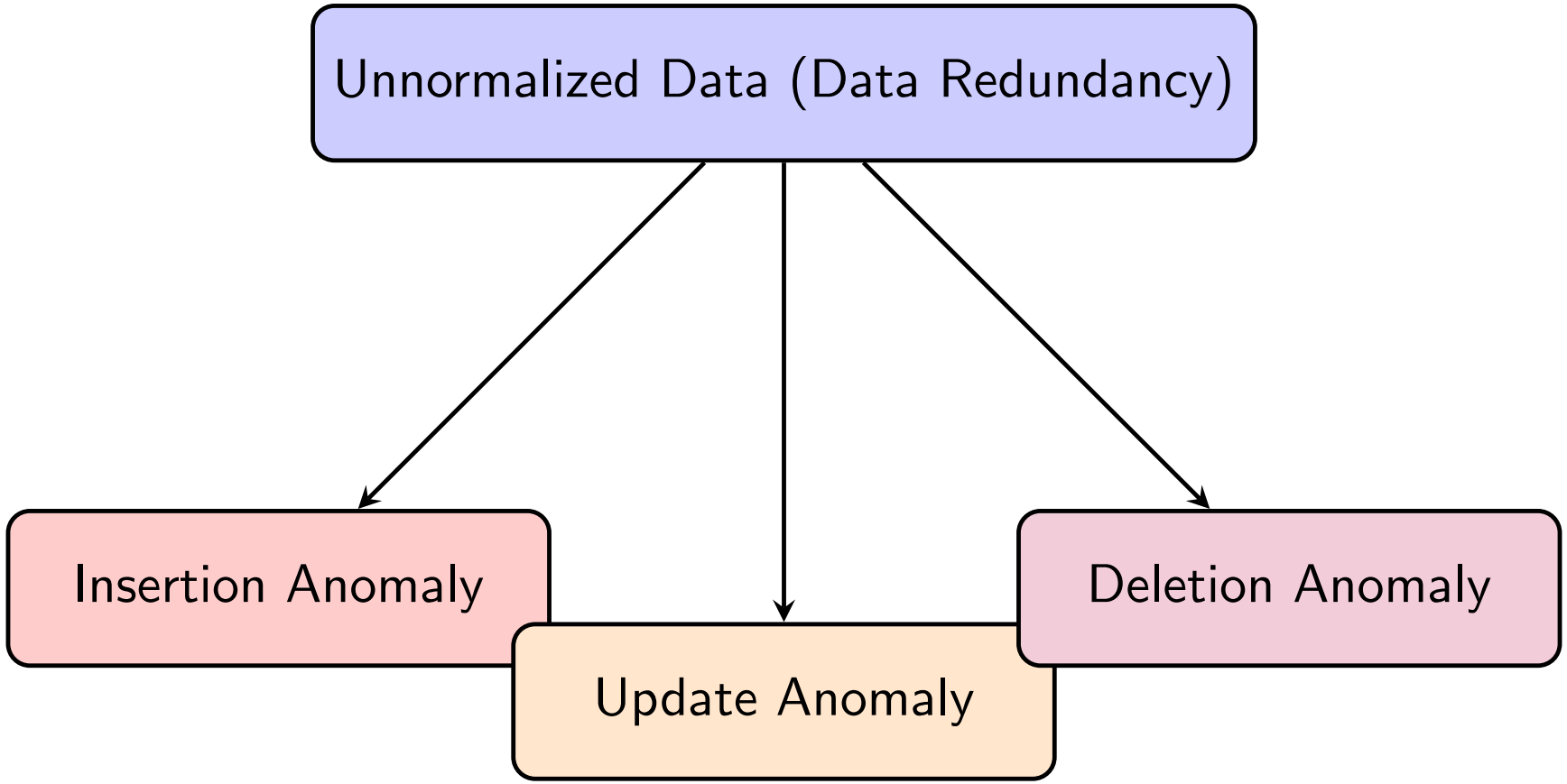
What is Normalization? I

- Normalization is a systematic approach of decomposing tables to eliminate data redundancy (repetition) and undesirable characteristics like Insertion, Update, and Deletion Anomalies.
- It is a multi-step process that puts data into tabular form, removing duplicated data from the relational tables.
- Proposed by Edgar F. Codd, the inventor of the relational model, as part of his 1970 paper.
- The process involves dividing large tables into smaller, more manageable ones and linking them using relationships (Primary Keys and Foreign Keys).

The Problem: Data Redundancy I

- Redundancy occurs when the same piece of data is stored in multiple places within the database.
- Consider the following unnormalized table structure:
- CREATE TABLE Student_Courses (StudentID INT, StudentName VARCHAR(100), CourseID INT, CourseName VARCHAR(100), ProfessorName VARCHAR(100));
- Notice the redundancy: if multiple students take the same course, CourseName and ProfessorName are repeated for every student.
- Problems caused by redundancy include wasted storage space, data inconsistency when updates occur, and overall database performance degradation.

Understanding Data Anomalies I



Types of Anomalies in Detail I

- **Insertion Anomaly**: Occurs when certain attributes cannot be inserted into the database without the presence of other attributes. ‘INSERT INTO Student_Courses (CourseID, CourseName, ProfessorName) VALUES (202, ’Databases’, ’Dr. Codd’);‘ – Fails if StudentID is part of a composite Primary Key.
- **Update Anomaly**: Occurs when redundant data is partially updated, leading to an inconsistent state. ‘UPDATE Student_Courses SET ProfessorName = ’Dr. Smith’ WHERE CourseID = 101;‘ – If missed in some rows, different students see different professors for the exact same course.

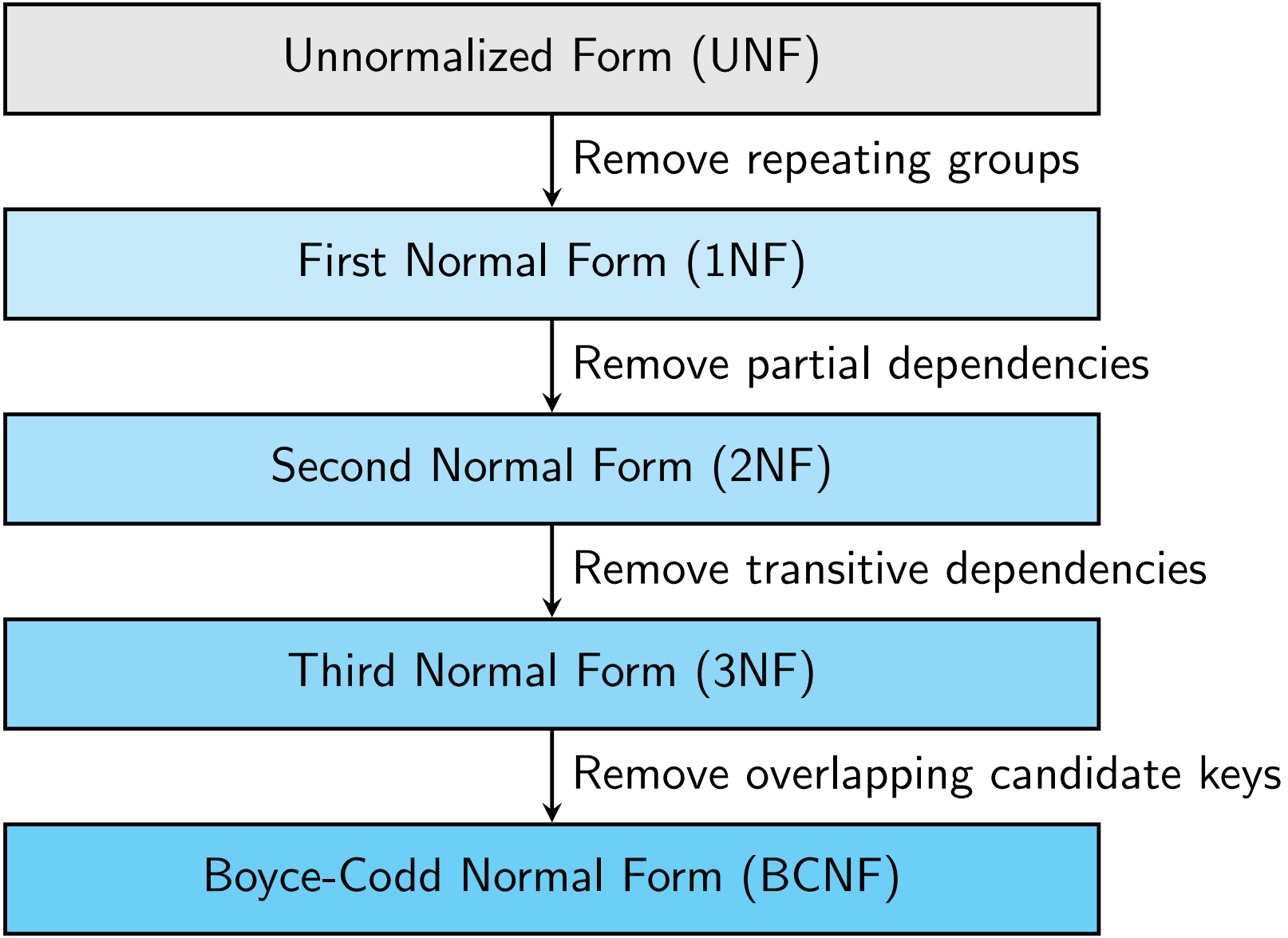
Types of Anomalies in Detail II

- **Deletion Anomaly**: Occurs when deleting a row inadvertently causes the loss of other unrelated data. ‘DELETE FROM Student_Courses WHERE StudentID = 543;‘ – If student 543 was the only one enrolled in Course 101, all information about Course 101 is permanently lost!

Introduction to Functional Dependencies I

- Functional Dependency (FD) is the absolute foundation of the normalization process.
- It describes the relationship between attributes in a relation. Notation: X -¿ Y (X functionally determines Y).
- Meaning: If two tuples have the same value for attribute X, they MUST have the same value for attribute Y.
- Here, X is known as the determinant set, and Y is the dependent attribute.
- Example: ‘StudentID -¿ StudentName‘. Knowing the unique ID strictly identifies the Name.
- A relational schema is essentially a set of attributes and a set of defining functional dependencies.

The Normalization Process I



Goals and Advantages of Normalization I

- **Primary Goals:**
- 1. Eliminate redundant data (ensure data is stored only once).
- 2. Ensure data dependencies make sense (store logically related data in the same table).
- **Advantages:**
- - Minimizes data redundancy, saving significant storage space.
- - Prevents destructive insertion, update, and deletion anomalies.
- - Improves data integrity and overall consistency of the Relational schema.
- - Simplifies complex queries as relations are tightly structured and predictable.
- - Makes the database structure more scalable and flexible for future modifications.

Trade-offs and Summary I

- While normalization guarantees structural integrity, it introduces performance trade-offs.
- **Disadvantages:**
- - Decomposing tables means retrieving complete business objects often requires complex ‘JOIN‘ operations.
- - Heavy read-operations may suffer from CPU/memory degradation due to multiple high-cost joins.
- **Denormalization:** In data warehousing or read-heavy (OLAP) applications, engineers deliberately introduce redundancy (denormalization) to avoid costly joins.
- **Summary:** Normalization is essential for OLTP (Online Transaction Processing) systems to maintain strict data integrity, but structural balance is key to optimizing real-world workloads.

Agenda I

- 1. Database Anomalies and the Need for Normalization
- 2. First Normal Form (1NF): Eliminating Repeating Groups
- 3. Second Normal Form (2NF): Eliminating Partial Dependencies
- 4. Third Normal Form (3NF): Eliminating Transitive Dependencies
- 5. Normalization Examples and PL/SQL Best Practices

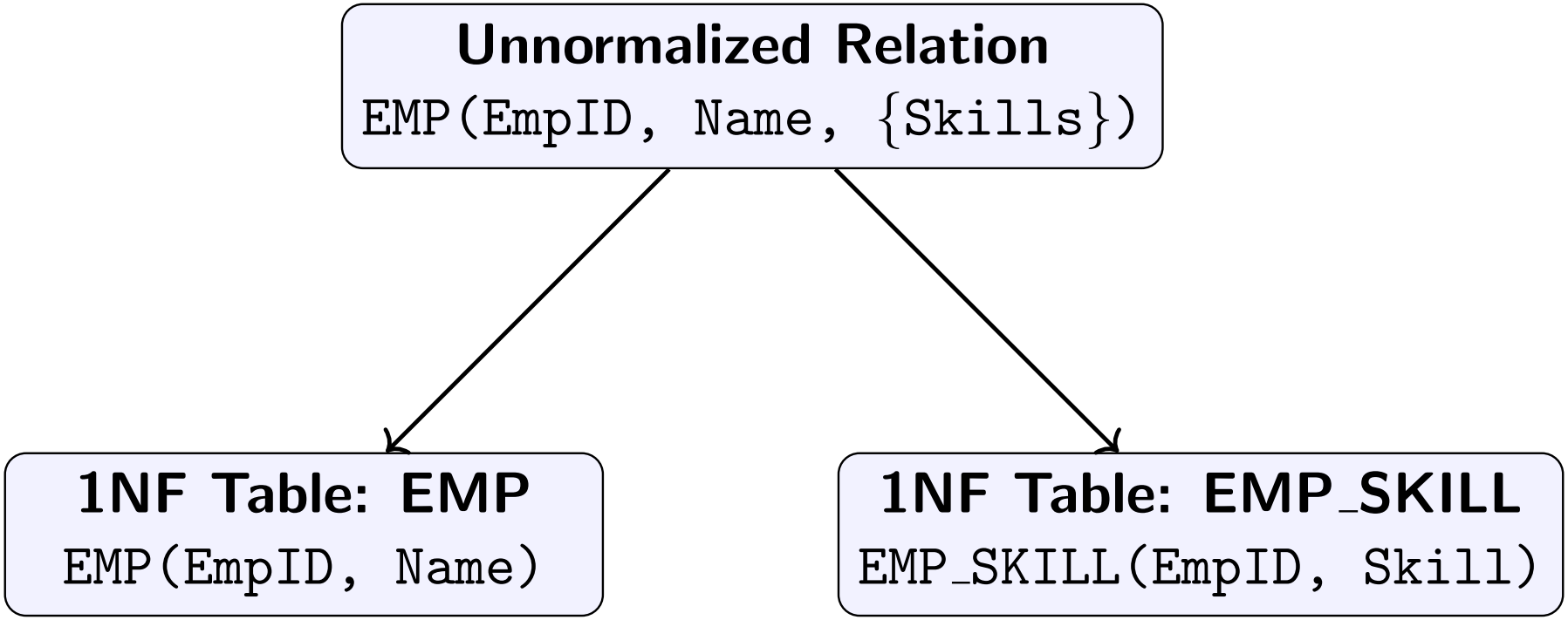
Introduction to Normal Forms & Anomalies I

- Normalization is a formal process of organizing the columns and tables of a relational database to minimize data redundancy.
- Without normalization, databases suffer from three main anomalies:
- • Insertion Anomaly: Inability to add data to the database due to absence of other data (e.g., cannot add a new department until an employee is hired).
- • Deletion Anomaly: Unintended loss of data due to deletion of other data (e.g., deleting the last employee in a department also deletes the department's information).
- • Update Anomaly: Data inconsistency resulting from data redundancy and a partial update.
- Normal Forms (NF) define rules for relation schemas to avoid these anomalies.

First Normal Form (1NF): Theory I

- A relation is in First Normal Form (1NF) if and only if the domain of each attribute contains only atomic (indivisible) values.
- Rules for 1NF:
- 1. Single Valued Attributes: No repeating groups or arrays as column values.
- 2. Attribute Domain: Values must be of the same type.
- 3. Unique Column Names: Each column in a table must have a unique name.
- SQL Example (Violation of 1NF - conceptual):
- CREATE TABLE Emp (EmpID INT, EmpName VARCHAR(50), Skills VARCHAR(255));
- – 'Skills' storing 'Java, Python, C++' as a single string violates the spirit of 1NF.
- To normalize, decompose the table to ensure atomicity.

1NF Decomposition Strategy I



Second Normal Form (2NF): Theory I

- A relation is in 2NF if it is in 1NF and every non-prime attribute is fully functionally dependent on the candidate key.
- Partial Dependency: Occurs when a non-prime attribute is dependent on only a part of a composite candidate key.
- If a table has a single-attribute primary key and is in 1NF, it is automatically in 2NF.
- Consider schema: COURSE_REG(StudentID, CourseID, CourseName, Instructor)
- Primary Key: (StudentID, CourseID)
- Dependency: CourseID -¿ CourseName. Here, CourseName is a non-prime attribute dependent on CourseID (a proper subset of the candidate key).
- This partial dependency violates 2NF and causes update anomalies.

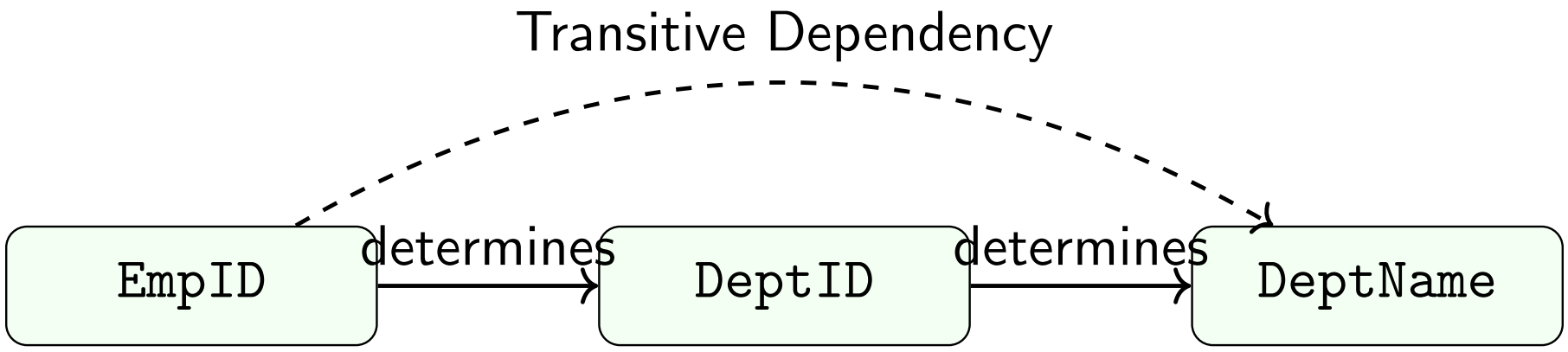
Second Normal Form (2NF): Example I

- To convert to 2NF, we decompose COURSE_REG(StudentID, CourseID, CourseName, Instructor) into two tables:
- 1. REGISTRATION(StudentID, CourseID)
- 2. COURSE(CourseID, CourseName, Instructor)
- SQL DDL Implementation:
- CREATE TABLE Course (CourseID INT PRIMARY KEY, CourseName VARCHAR(100), Instructor VARCHAR(50));
- CREATE TABLE Registration (
- StudentID INT,
- CourseID INT,
- PRIMARY KEY (StudentID, CourseID),
- FOREIGN KEY (CourseID) REFERENCES Course(CourseID)
-);
- Now, the non-prime attributes in 'Course' depend entirely on 'CourseID'.

Third Normal Form (3NF): Theory I

- A relation is in 3NF if it is in 2NF and there are no transitive dependencies of non-prime attributes on the candidate key.
- Transitive Dependency: If X -¿ Y and Y -¿ Z (where Y is not a candidate key), then X -¿ Z is a transitive dependency.
- Rule of Thumb for 3NF: 'Every non-key attribute must provide a fact about the key, the whole key, and nothing but the key.'
- Consider schema: EMP(EmpID, Name, DeptID, DeptName)
- Primary Key: EmpID
- Dependencies: EmpID -¿ DeptID, DeptID -¿ DeptName.
- Therefore, EmpID -¿ DeptName is a transitive dependency, violating 3NF.

3NF Transitive Dependency Diagram I



3NF Decomposition:

`EMP(EmpID, Name, DeptID)`

`DEPT(DeptID, DeptName)`

Summary & PL/SQL Validation I

- Summary:
- 1NF: Atomic values. 2NF: No partial dependencies. 3NF: No transitive dependencies.
- Normalization aligns database schemas to ensure structural integrity and reduce data anomalies.
- PL/SQL block to enforce a business rule conceptually aligned with referential integrity after normalization:
- DECLARE
- v_dept_count NUMBER;
- BEGIN
- SELECT COUNT(*) INTO v_dept_count FROM DEPT WHERE DeptID = 10;
- IF v_dept_count = 0 THEN

Summary & PL/SQL Validation II

- RAISE_APPLICATION_ERROR(-20001, 'Parent key not found in DEPT table!');
- ELSE
- INSERT INTO EMP (EmpID, DeptID) VALUES (1001, 10);
- COMMIT;
- END IF;
- END;

Agenda I

- Recap: What is Normalization?
- Advantages of Normalization
- Disadvantages of Normalization
- The Performance Trade-off (Normalization vs. Denormalization)
- Summary & Conclusion

Recap: What is Normalization? I

- Normalization is a systematic approach of decomposing tables to eliminate data redundancy and undesirable characteristics like Insertion, Update and Deletion Anomalies.
- It is a multi-step process that puts data into tabular form by removing duplicated data from the relation tables.
- The normal forms most commonly used are 1NF, 2NF, 3NF, and BCNF.
- The core principle is to ensure that non-key attributes depend on the primary key, the whole primary key, and nothing but the primary key.

Advantage 1: Reduced Data Redundancy I

- In an unnormalized database, the same data might be stored in multiple locations. For example, a customer's address might be stored in every order record they place.
- Normalization ensures that each piece of non-key data is stored exactly once, relying on foreign keys to link related data.
- Storage Efficiency: By eliminating duplicate data, the overall storage requirement of the database is significantly reduced.
- This reduction in storage footprint can lead to cost savings in large-scale systems and reduces the I/O bottleneck during sequential scans.

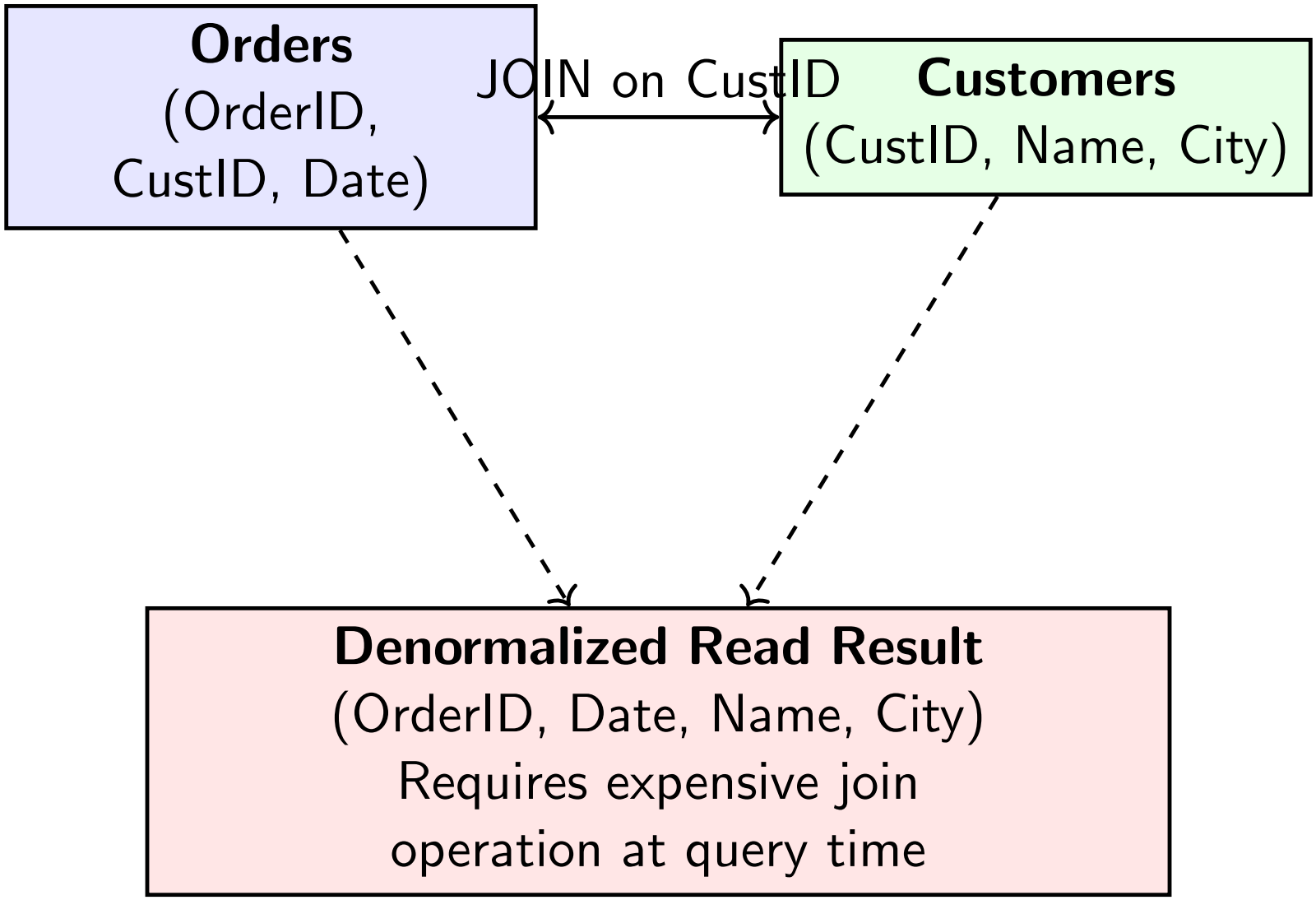
Advantage 2: Prevention of Anomalies I

- Insertion Anomaly: Cannot insert data without the presence of other data (e.g., adding a new course without assigning a student). Normalization resolves this by separating entities.
- Update Anomaly: Updating a duplicated value requires multiple row updates. Normalization ensures the value is in one place, requiring only a single scalar update.
- Deletion Anomaly: Deleting a record might inadvertently delete other necessary information (e.g., deleting the last student in a course deletes the course details). Decomposing tables preserves independent entities.
- By achieving 3NF or BCNF, the database schema is robust against these data manipulation anomalies.

Advantage 3: Improved Data Integrity and Consistency I

- With redundancy minimized, the chance of inconsistent data (the 'multiple truths' problem) is drastically reduced.
- Normalization inherently enforces referential integrity through the strict use of Primary Keys (PKs) and Foreign Keys (FKs).
- It simplifies the enforcement of business rules and domain constraints at the database engine level.
- Database triggers and stored procedures become simpler because data mutations happen in predictable, isolated locations.

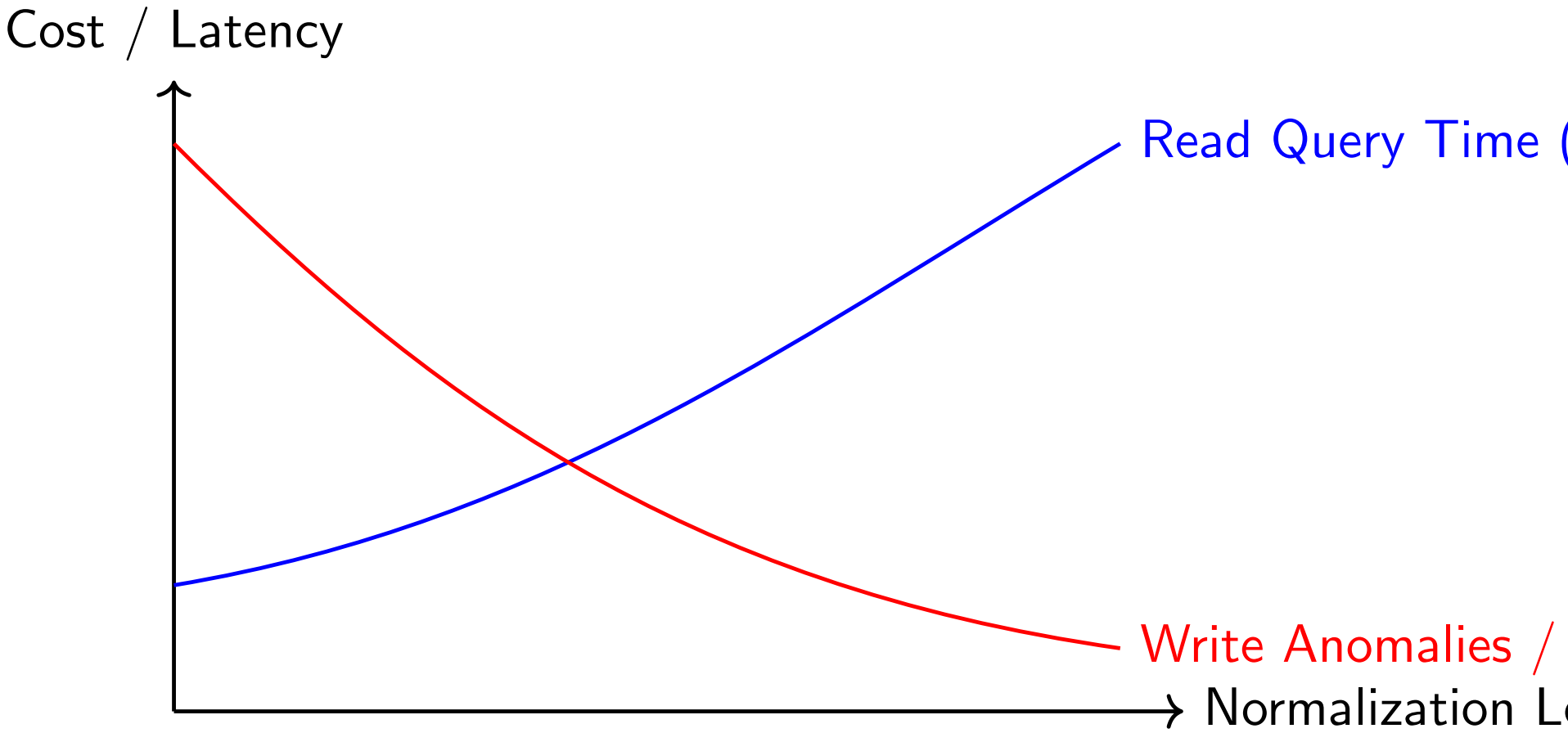
Disadvantage 1: Performance Overhead due to Joins (Visualized) I



Disadvantage 2: Complexity in Application Design I

- Highly normalized databases often contain a vast number of tables, obscuring the macroscopic business entities.
- Developers must write complex SQL statements with multiple JOIN clauses to extract meaningful, consolidated information.
- Example: SELECT o.OrderID, c.Name, p.ProductName FROM Orders o JOIN Customers c ON o.CustID = c.CustID JOIN OrderDetails od ON o.OrderID = od.OrderID JOIN Products p ON od.ProductID = p.ProductID;
- This fragmentation requires Object-Relational Mapping (ORM) tools to do heavy lifting, sometimes resulting in the 'N+1 selects problem'.

Disadvantage 3: The Read vs. Write Trade-off I



Summary & Conclusion I

- Normalization is essential for OLTP (Online Transaction Processing) systems where data integrity, ACID compliance, and fast writes are paramount.
- However, strict normalization is often detrimental to Data Warehouses or OLAP (Online Analytical Processing) systems.
- Denormalization (e.g., Star Schema or Snowflake Schema) is consciously applied in read-heavy analytical environments to pre-join data and optimize query performance.
- Ultimately, database design is an engineering trade-off: architects must balance the strictness of normal forms against the specific performance requirements of the application workload.

Agenda I

- 1. Recap: The Purpose of Normalization
- 2. Advantages: Data Integrity & Consistency
- 3. Advantages: Efficient DML Operations
- 4. Disadvantages: Read Query Complexity
- 5. Disadvantages: Performance Overhead
- 6. The Normalization Trade-off
- 7. Mitigating Disadvantages: Denormalization & Views
- 8. Summary & Golden Rules

Recap: The Purpose of Normalization I

- Normalization is a systematic approach to decomposing tables to eliminate data redundancy and undesirable characteristics like Insertion, Update, and Deletion anomalies.
- By dividing larger tables into smaller, logically connected tables (using Foreign Keys), we adhere to the 'single source of truth' principle.
- The process involves sequential tests (Normal Forms: 1NF, 2NF, 3NF, BCNF) to ensure dependencies are mathematically sound.
- Fact: A fully normalized database ensures that every non-key attribute is fully functionally dependent on the primary key, the whole primary key, and nothing but the primary key.

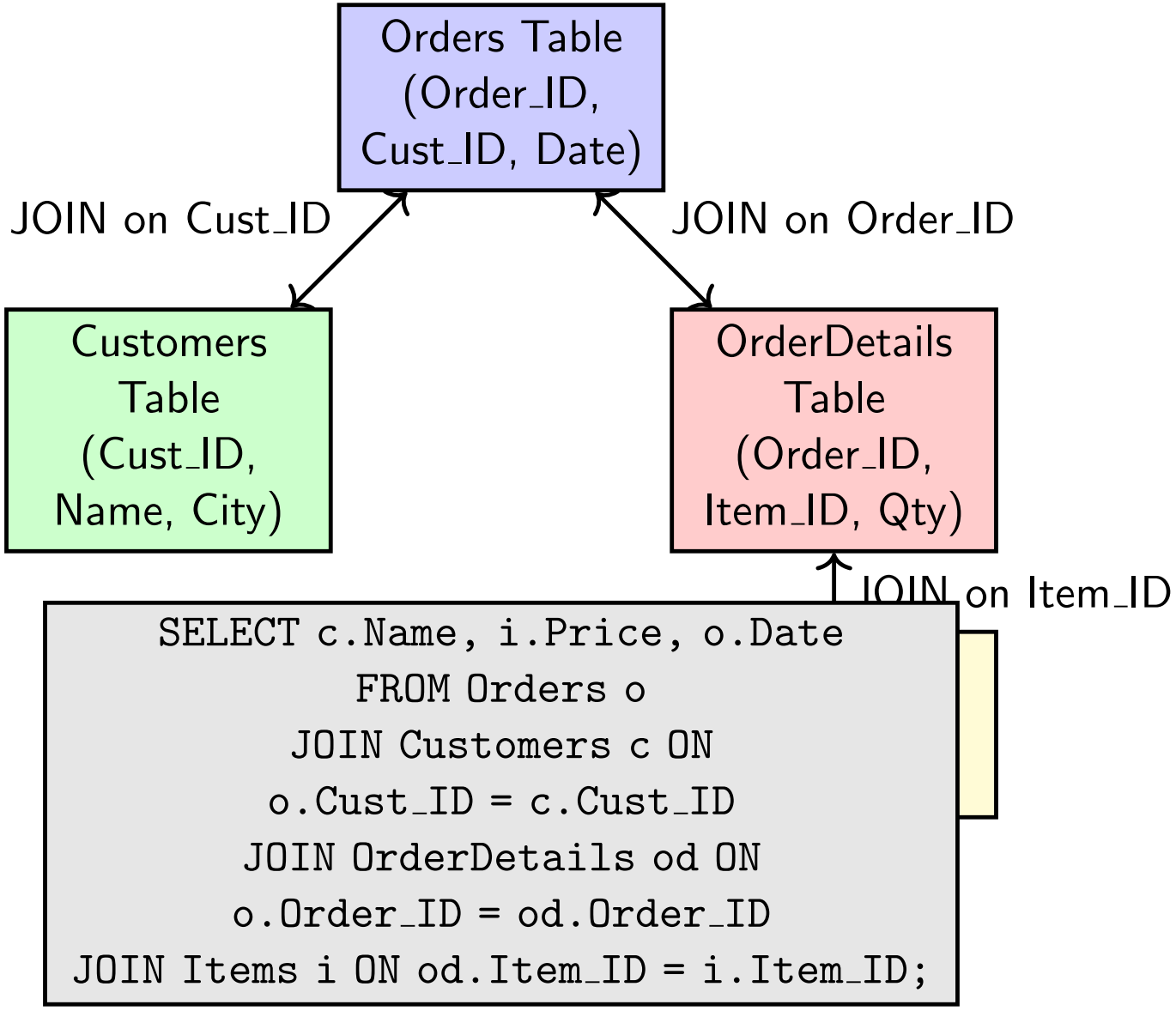
Advantages: Data Integrity & Consistency I

- 1. Elimination of Redundancy: Reduces duplicate data storage, directly lowering the chances of inconsistent data states.
- 2. Update Anomalies Prevented: When a fact changes (e.g., a customer's address), it only needs to be updated in one place.
- Example: UPDATE Customers SET City = 'Seattle' WHERE Cust_ID = 101;
- In a denormalized structure, this would require updating millions of order records, risking an Update Anomaly if a single row is missed.
- 3. Enforcement of Referential Integrity: Strict foreign key constraints ensure that orphaned records cannot exist (e.g., an order cannot belong to a non-existent customer).

Advantages: Efficient DML Operations (Writes) I

- 1. Faster INSERTs and UPDATEs: Because tables are narrower (fewer columns) and data is stored exactly once, write operations modify fewer data blocks on disk.
- 2. Reduced Lock Contention: Updates target highly specific, narrow rows. This reduces the footprint of row-level locks in engines like InnoDB, improving transaction concurrency.
- 3. Deletion Anomalies Prevented: Deleting a record from one table doesn't accidentally wipe out unrelated facts.
- For instance, deleting the last order for a product does not delete the product's metadata (Name, Price) if the database is properly normalized to 3NF.

Disadvantages: Read Query Complexity I

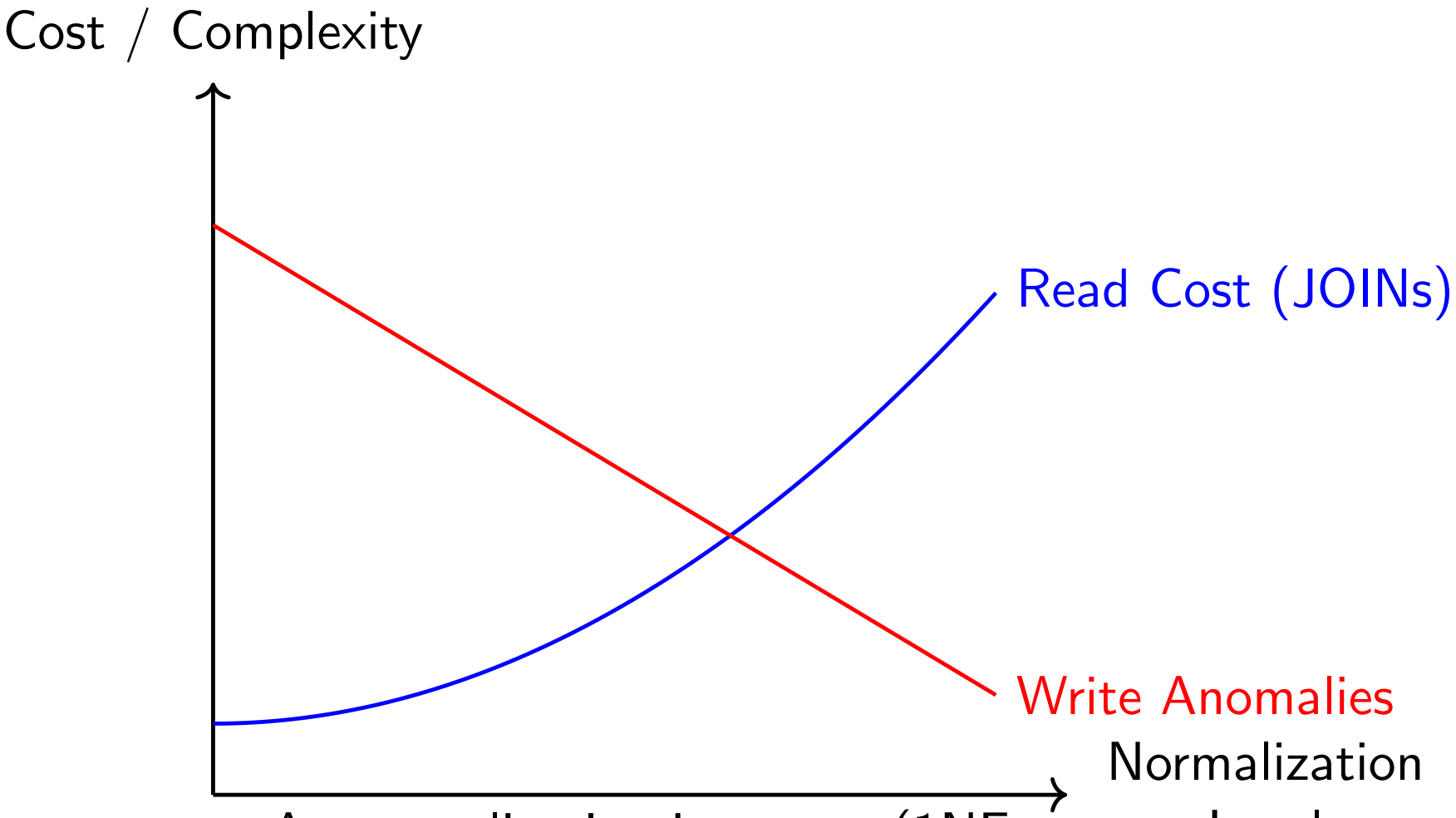


Disadvantages: Performance Overhead for Reads I

- 1. Excessive JOIN Operations: To reconstruct a complete business entity (e.g., an Invoice), the RDBMS must perform multiple JOINs across disparate tables.
- 2. Increased CPU and Memory Usage: Hash JOINs and Sort Merge JOINs consume significant temporary tablespace (TEMP) and CPU cycles when datasets are massive.
- 3. Index Maintenance: Highly normalized databases require more foreign keys and associated indexes, which adds storage overhead and slows down bulk inserts.
- 4. Decreased Read Performance: In OLAP (Online Analytical Processing) systems, normalization causes unacceptable query latency. Denormalization is preferred for Data Warehouses.

The Normalization Trade-off I

The Normalization Trade-off II



As normalization increases (1NF $\rightarrow$ 3NF), write anomalies decrease but read complexity increases due to table fragmentation.

Mitigating Disadvantages: Denormalization & Views I

- To offset the read performance penalty in a normalized schema, engineers employ specific database objects:
- 1. Materialized Views: Pre-compute and physically store the results of complex JOINs. \{}texttt{CREATE MATERIALIZED VIEW Sales\{}_Summary AS SELECT c.Name, SUM(i.Price * od.Qty) as Total FROM Customers c JOIN Orders o ON ... GROUP BY c.Name;}
- 2. Controlled Denormalization: Deliberately violating 3NF by adding redundant columns to eliminate a JOIN (e.g., storing 'Total_Amount' in the Orders table, maintained by Triggers).
- 3. Caching Layers: Using external tools like Redis or Memcached to store heavily read, joined datasets directly in memory.

Summary I

- 1. Normalization is essential for OLTP databases where data integrity, consistency, and fast atomic writes are paramount.
- 2. Advantages: Eliminates redundancy, prevents insertion/update/deletion anomalies, and heavily reduces transaction lock contention.
- 3. Disadvantages: Table fragmentation necessitates costly JOIN operations, heavily slowing down read-heavy reporting and analytics queries.
- 4. The Golden Rule: Normalize to 3NF or BCNF by default during initial logical design, and selectively denormalize only when performance bottlenecks are proven via EXPLAIN plans.

Agenda I

- 1. Brief Recap of Normal Forms (1NF, 2NF, 3NF, BCNF)
- 2. Problem 1: Closure of a Set of Attributes
- 3. Problem 2: Identifying Candidate Keys
- 4. Problem 3: Determining the Highest Normal Form
- 5. Problem 4: Lossless Join and Dependency Preserving Decomposition

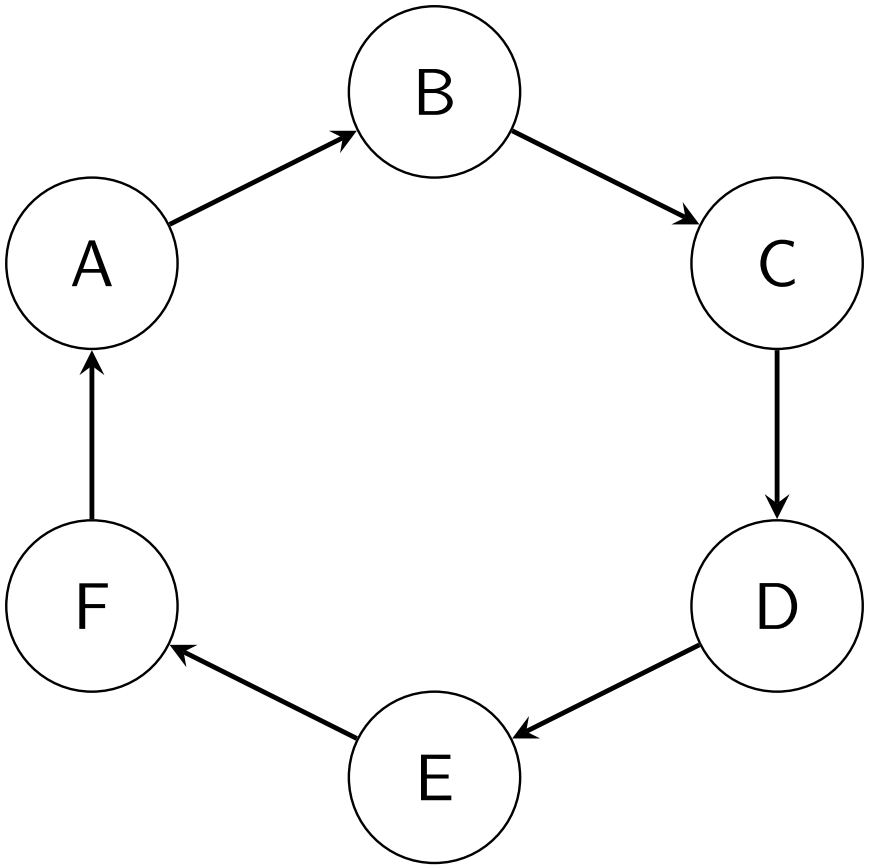
Recap: Functional Dependencies & Keys I

- **Functional Dependency (FD):** X -¿ Y implies that if two tuples agree on attributes X, they must also agree on attributes Y.
- **Closure of Attributes (X+):** The set of all attributes that can be functionally determined from X using Armstrong's axioms (Reflexivity, Augmentation, Transitivity).
- **Candidate Key:** A minimal superkey. A set of attributes K is a candidate key for relation R if K -¿ R and no proper subset of K functionally determines R.
- **Prime vs. Non-Prime Attributes:** Attributes belonging to any candidate key are prime attributes; all other attributes are non-prime.

Problem 1: Attribute Closure I

- **Given:** Relation R(A, B, C, D, E, F) and a set of FDs F = {A -¿ B, B -¿ C, C -¿ D, D -¿ E, E -¿ F, F -¿ A}.
- **Find:** The closure of attribute A, denoted as (A)+.
- **Solution Steps:**
- 1. Initialize closure to the attribute itself: (A)+ = {A}
- 2. Apply A -¿ B: (A)+ = {A, B}
- 3. Apply B -¿ C: (A)+ = {A, B, C}
- 4. Continue transitive application: C -¿ D adds D, D -¿ E adds E, E -¿ F adds F.
- **Result:** (A)+ = {A, B, C, D, E, F}. Since (A)+ contains all attributes in the relation, A is a superkey (and in this case, a candidate key).

Problem 1: Functional Dependency Graph I



Problem 2: Finding Candidate Keys I

- **Given:** Relation R(A, B, C, D, E) and FDs F = {AB -¿ C, C -¿ D, D -¿ B, D -¿ E}.
- **Objective:** Find all candidate keys for R.
- **Step 1:** Identify attributes that are never on the right-hand side (RHS) of any FD.
- Attribute 'A' is never on the RHS. Therefore, 'A' MUST be part of every candidate key.
- **Step 2:** Compute closure of A alone. $(A)+ = \{A\}$. This is not a key.
- **Step 3:** Test 'A' combined with other attributes.
- - $(AB)+ = \{A, B, C, D, E\}$ -¿ AB is a key.
- - $(AC)+ = \{A, C, D, B, E\}$ -¿ AC is a key.
- - $(AD)+ = \{A, D, B, C, E\}$ -¿ AD is a key.
- **Result:** The Candidate Keys are {AB, AC, AD}.

Problem 3: Determining Normal Form I

- **Given:** R(A, B, C, D, E) with Candidate Keys: AB, AC, AD. FDs: {AB -¿ C, C -¿ D, D -¿ B, D -¿ E}.
- **Question:** What is the highest normal form of R?
- **Analysis:**
- 1. **Prime Attributes:** A, B, C, D. **Non-Prime Attribute:** E.
- 2. Check **2NF**: Are there partial dependencies? (A proper subset of a key determining a non-prime attribute).
- - The only non-prime attribute is E. E is determined by D (via D -¿ E). Since D is prime but D alone is NOT a proper subset of any candidate key (the keys are AB, AC, AD), there is no partial dependency on a candidate key. R is in 2NF.
- 3. Check **3NF**: Are there transitive dependencies?

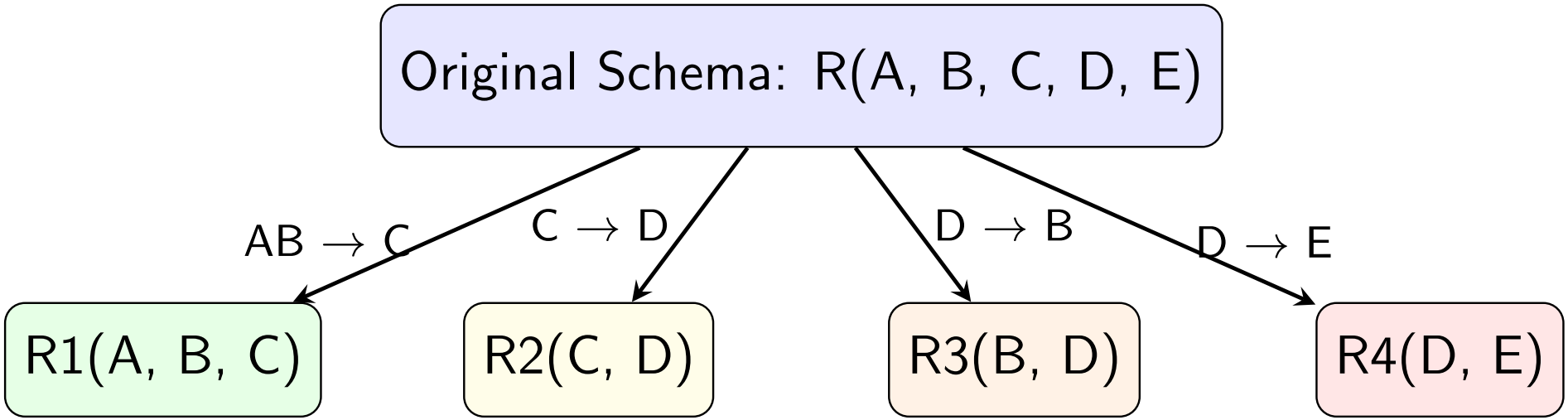
Problem 3: Determining Normal Form II

- - Consider D -¿ E. 'D' is not a superkey, and 'E' is not a prime attribute. This violates the 3NF condition.
- **Conclusion:** The relation R is in 2NF, but not in 3NF.

Problem 4: 3NF Decomposition I

- **Objective:** Decompose R(A, B, C, D, E) into 3NF without losing data or dependencies.
- **Algorithm (Dependency Preserving & Lossless 3NF Decomposition):**
- 1. Find the Minimal Cover of F. In this case, F = {AB -¿ C, C -¿ D, D -¿ B, D -¿ E} is already minimal.
- 2. Create a relation for each FD in the minimal cover.
- - R1(A, B, C) from AB -¿ C
- - R2(C, D) from C -¿ D
- - R3(B, D) from D -¿ B
- - R4(D, E) from D -¿ E
- 3. **Check Candidate Key:** Does any resulting relation contain a candidate key of the original R?
- - Yes, R1 contains AB, which is a candidate key of R.
- **Result:** The final 3NF decomposition is {R1, R2, R3, R4}.

Problem 4: Decomposition Diagram I



Summary & Key Takeaways I

- 1. **Attribute Closure** is the foundational algorithm for identifying keys and testing functional dependencies.
- 2. **Candidate Key** identification involves finding minimal sets of attributes whose closure contains all attributes of the relation.
- 3. **Normal Form Determination** must be done systematically: check 1NF, then 2NF (no partial dependencies), then 3NF (no transitive dependencies), and finally BCNF (LHS of all FDs are superkeys).
- 4. **Decomposition** to 3NF can always be achieved such that it is both lossless and dependency-preserving, whereas decomposition to BCNF might lose some functional dependencies.

Summary & Key Takeaways II

- 5. Mastery of these concepts requires practicing with various sets of functional dependencies and evaluating all conditions meticulously.

Agenda I

- Introduction to Integrity Constraints
- Domain Integrity: NOT NULL and CHECK
- Entity Integrity: UNIQUE and PRIMARY KEY
- Entity Integrity: Diagrammatic Representation
- Referential Integrity: FOREIGN KEY and REFERENCES
- Referential Integrity: Diagrammatic Representation
- Referential Actions: ON DELETE CASCADE

Introduction to Integrity Constraints I

- Integrity constraints are a set of mathematical and logical rules used to maintain the quality, accuracy, and consistency of data in a relational database.
- They ensure that DML operations (INSERT, UPDATE, DELETE) do not lead to invalid states, guarding against accidental corruption.
- These constraints are enforced at the schema level by the RDBMS engine, providing a centralized data validation layer.
- The three primary classifications in the Relational Model are Domain Constraints, Entity Constraints, and Referential Constraints.
- By enforcing these constraints during table definition (DDL), developers ensure that bad data never enters the storage engine.

Domain Integrity Constraints: NOT NULL and CHECK I

- Domain constraints limit the valid set of values for a specific attribute (column) based on data types, lengths, and logical conditions.
- NOT NULL: Prevents missing values (NULLs) in mandatory columns, ensuring the attribute always holds a defined value.
- CHECK: Evaluates a predicate logic expression. The DBMS rejects any transaction where the CHECK condition evaluates to FALSE.
- SQL Example: CREATE TABLE Employee (EmpID INT NOT NULL, Age INT CHECK (Age ¿= 18));

Domain Integrity Constraints: NOT NULL and CHECK II

- PL/SQL Example for complex domain rules: CREATE OR REPLACE TRIGGER trg_check_age BEFORE INSERT ON Employee FOR EACH ROW BEGIN IF :NEW.Age ¡ 18 THEN RAISE_APPLICATION_ERROR(-20001, 'Age must be ¿= 18'); END IF; END;

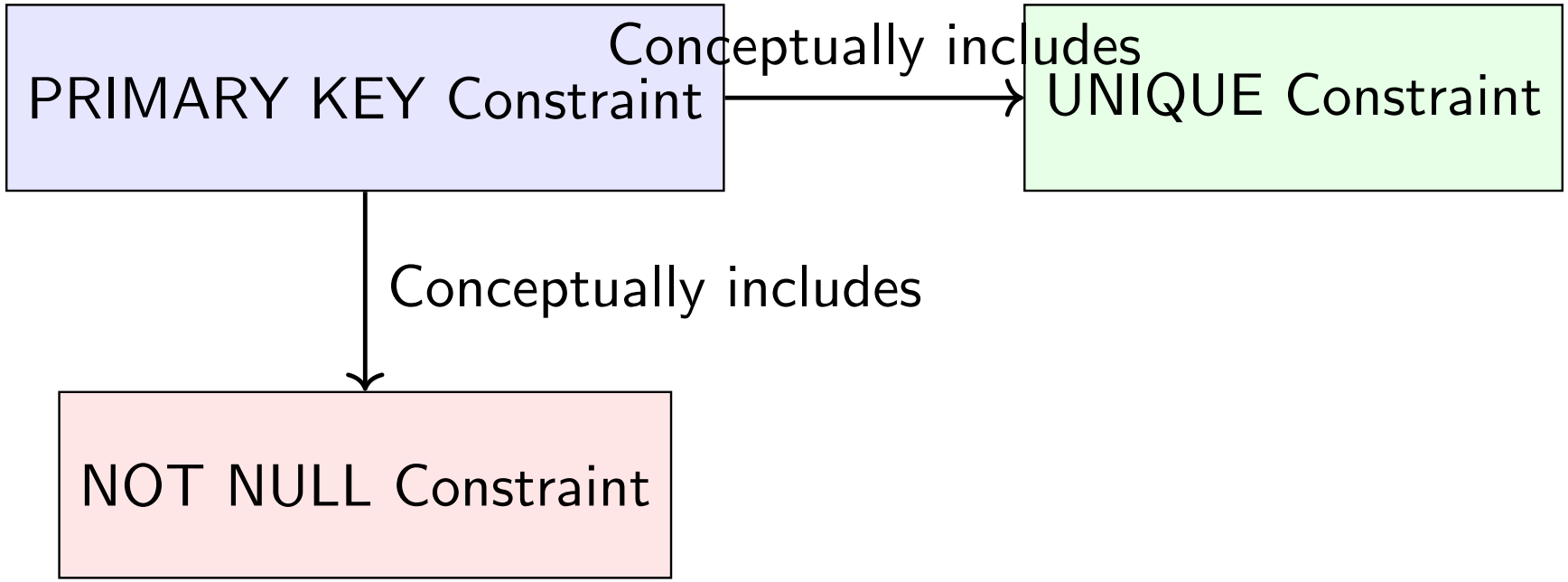
Entity Integrity Constraints: UNIQUE and PRIMARY KEY I

- Entity integrity ensures that each row (tuple) in a table can be uniquely identified, typically reflecting real-world entities.
- UNIQUE Constraint: Ensures all values in an indexed column or composite column set are distinct. Multiple UNIQUE constraints can exist per table.
- PRIMARY KEY (PK): The definitive unique identifier for a row. A table can have at most ONE primary key constraint.
- A PRIMARY KEY implicitly enforces both NOT NULL and UNIQUE constraints.
- SQL Example: CREATE TABLE Departments (DeptID INT PRIMARY KEY, DeptName VARCHAR(100) UNIQUE NOT NULL);

Entity Integrity Constraints: UNIQUE and PRIMARY KEY II

- Under the hood, the RDBMS automatically creates a B-Tree unique index on the PRIMARY KEY columns to optimize lookups.

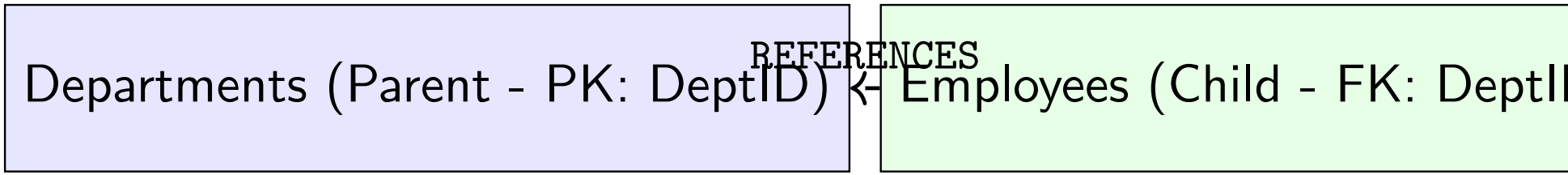
Entity Integrity Visualized I



Referential Integrity Constraints: FOREIGN KEY I

- Referential integrity governs the semantic relationships between multiple tables, typically a Parent-Child (1:N) relationship.
- FOREIGN KEY (FK): A column or set of columns in the child table that precisely matches the candidate key (usually PK) of the parent table.
- The REFERENCES clause explicitly links the FK to the target table and target column, building the constraint dependency.
- This prevents the creation of 'orphan records'—child rows that point to non-existent parent rows.
- SQL Example: CREATE TABLE Employees (EmpID INT PRIMARY KEY, DeptID INT, FOREIGN KEY (DeptID) REFERENCES Departments(DeptID));

Referential Integrity Relationship I



Referential Actions: ON DELETE CASCADE I

- Referential actions dictate the automatic behavior of the DBMS when a parent record referenced by foreign keys is updated or deleted.
- ON DELETE CASCADE: If a parent row is deleted, the DBMS will automatically delete all child rows that reference that parent.
- Other actions include ON DELETE SET NULL (sets FKs to NULL), and ON DELETE RESTRICT (aborts the parent deletion if children exist).
- SQL Example: CREATE TABLE Orders (OrderID INT PRIMARY KEY, CustomerID INT, FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID) ON DELETE CASCADE);

Referential Actions: ON DELETE CASCADE II

- Engineers must be cautious with cascading deletes, as a single DELETE on a root entity could wipe out millions of dependent records globally.

Summary & Best Practices I

- Integrity constraints are foundational to relational database design, moving validation directly to the storage tier.
- Domain Constraints (NOT NULL, CHECK): Ensure attributes hold legal and logical values within their scope.
- Entity Constraints (PRIMARY KEY, UNIQUE): Guarantee record uniqueness and proper row identification.
- Referential Constraints (FOREIGN KEY): Enforce structural relationships and prevent orphaned records in normalized schemas.
- Best Practice: Always name constraints explicitly using the CONSTRAINT keyword (e.g., CONSTRAINT fk_dept FOREIGN KEY) to make debugging and dropping constraints easier.

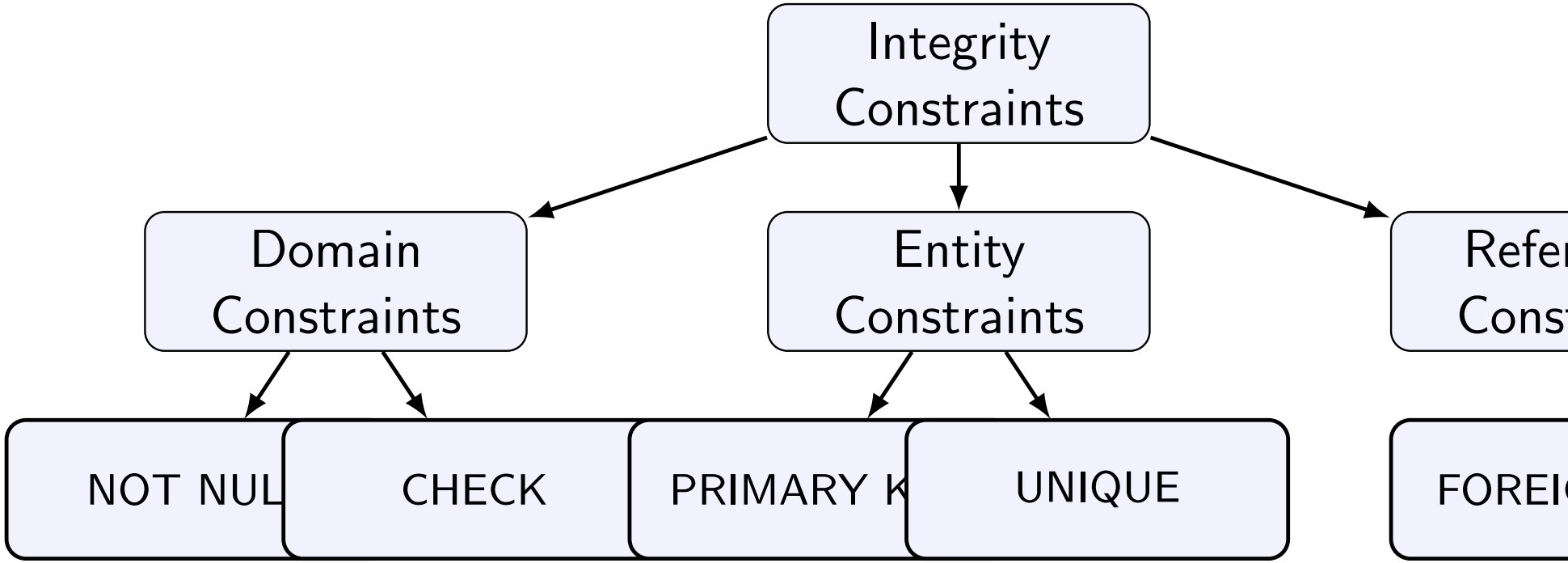
Agenda I

1. Introduction to Integrity Constraints
2. Domain Integrity: NOT NULL & CHECK
3. Entity Integrity: UNIQUE & PRIMARY KEY
4. Referential Integrity: FOREIGN KEY & REFERENCES
5. Referential Actions: ON DELETE CASCADE

Introduction to Integrity Constraints I

- Definition: Integrity constraints are rules applied to columns or tables to ensure the accuracy, consistency, and reliability of data in a relational database.
- Purpose: They prevent accidental damage to the database by restricting the type of data that can be inserted or updated.
- Enforcement: Constraints are actively enforced by the RDBMS engine during Data Manipulation Language (DML) operations.
- Types:
- - Domain Integrity: Restricts the valid range of values for a column.
- - Entity Integrity: Ensures each row in a table is uniquely identifiable.
- - Referential Integrity: Maintains the relationship between multiple tables.

Classification of Integrity Constraints I



Domain Integrity Constraints: NOT NULL & CHECK I

- Domain integrity ensures that a specific column contains valid data, according to its data type, format, and range.
- 1. NOT NULL Constraint: Ensures that a column cannot have a NULL value. By default, a column can hold NULL values.
- Example: CREATE TABLE employees (emp_id INT NOT NULL, name VARCHAR(50) NOT NULL);
- 2. CHECK Constraint: Ensures that all values in a column satisfy specific logical conditions.
- Example SQL:
- CREATE TABLE students (
- student_id INT PRIMARY KEY,
- age INT CHECK (age ¿= 18),
- gender CHAR(1) CHECK (gender IN ('M', 'F', 'O'))
-);

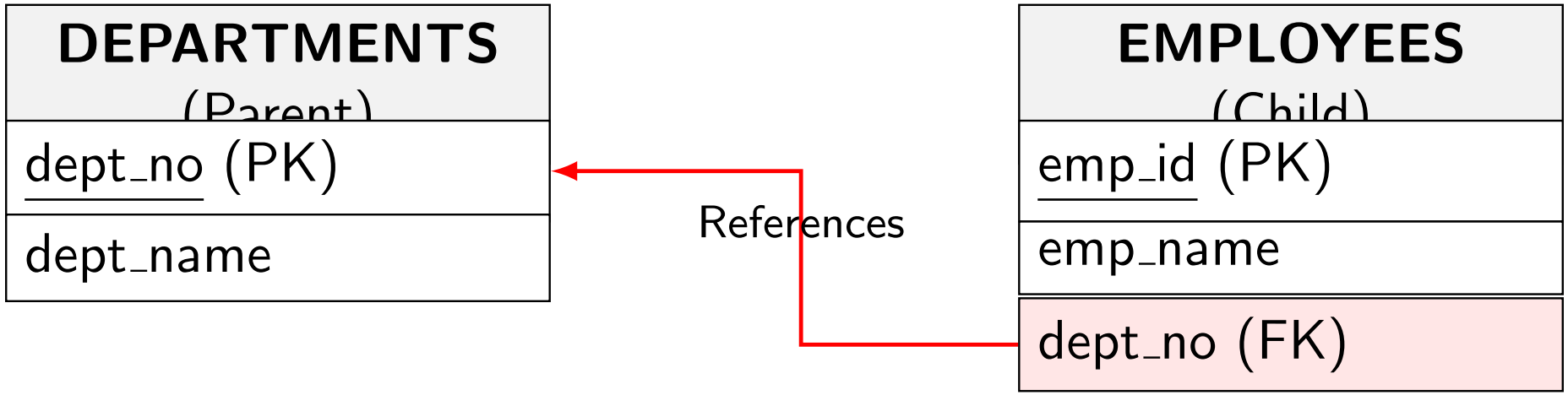
Entity Integrity Constraints: UNIQUE I

- Entity integrity guarantees that each row in a table is unique and not duplicated.
- UNIQUE Constraint: Ensures that all values in a column or a combination of columns are unique across the table.
- - Allows multiple NULL values depending on the specific RDBMS implementation (e.g., PostgreSQL allows multiple NULLs, SQL Server allows only one).
- Syntax & Example:
- CREATE TABLE users (
- user_id INT,
- email VARCHAR(255) UNIQUE,
- username VARCHAR(50),
- CONSTRAINT uk_username UNIQUE (username)
-);

Entity Integrity Constraints: PRIMARY KEY I

- PRIMARY KEY Constraint: Uniquely identifies each record in a database table.
- - A Primary Key must contain UNIQUE values and cannot contain NULL values.
- - A table can have only one Primary Key, which may consist of single or multiple columns (Composite Key).
- - The RDBMS automatically creates a unique B-Tree index on the primary key columns to enforce uniqueness and speed up retrieval.
- Example:
- CREATE TABLE orders (
- order_id INT PRIMARY KEY,
- product_id INT,
- quantity INT
-);

Referential Integrity & Foreign Keys I



Referential Integrity Constraints: FOREIGN KEY I

- Referential Integrity ensures that relationships between tables remain logically consistent.
- FOREIGN KEY Constraint: A column or group of columns in a relational database table that provides a link between data in two tables.
- The FOREIGN KEY constraint prevents invalid data from being inserted into the foreign key column, as it must point to an existing record in the parent table.
- Example Implementation:
- CREATE TABLE employee (
- emp_id INT PRIMARY KEY,
- dept_id INT,
- FOREIGN KEY (dept_id) REFERENCES department(dept_id)
-);

Referential Actions: ON DELETE CASCADE I

- When an UPDATE or DELETE operation affects a key value in the parent table that has matching rows in the child table, Referential Actions dictate the engine's behavior.
- ON DELETE CASCADE: If a record in the parent table is deleted, all corresponding records in the child table are automatically deleted.
- Alternative actions: ON DELETE SET NULL, ON DELETE RESTRICT / NO ACTION.
- Example with CASCADE:
- CREATE TABLE order_items (
- item_id INT PRIMARY KEY,
- order_id INT,
- CONSTRAINT fk_order FOREIGN KEY (order_id)
- REFERENCES orders(order_id) ON DELETE CASCADE

Referential Actions: ON DELETE CASCADE II

-);
- Deleting an order from 'orders' automatically removes its items from 'order_items'.

Agenda I

- 1. Overview of Integrity Constraints
- 2. Domain Integrity Constraints: NOT NULL, CHECK
- 3. Entity Integrity Constraints: UNIQUE, PRIMARY KEY
- 4. Referential Integrity Constraints: FOREIGN KEY, REFERENCES
- 5. Cascading Actions: ON DELETE CASCADE
- 6. Summary and Architectural Best Practices

Overview of Integrity Constraints I

- Integrity constraints are strict rules enforced by the RDBMS to guarantee data accuracy, consistency, and reliability across the database.
- They prevent the entry of invalid data and govern the behavior of DML operations (INSERT, UPDATE, DELETE).
- Constraints can be defined at the column level (applying to a single attribute) or at the table level (applying to combinations of multiple attributes).
- Modern RDBMS engines evaluate these constraints synchronously before committing transactions to ensure ACID properties, particularly consistency, are upheld.

Domain Integrity Constraints: NOT NULL & CHECK I

- Domain integrity dictates that all entries in a specific column must fall within a predefined set of valid values and conditions.
- NOT NULL: Ensures a column cannot hold a NULL marker. It mandates that an explicit value must be supplied during row insertion.
- Example: CREATE TABLE Employees (EmpID INT NOT NULL, Name VARCHAR(50));
- CHECK: Defines a boolean predicate that must evaluate to TRUE or UNKNOWN for every row.
- Example: CREATE TABLE Accounts (Balance DECIMAL(10,2), CONSTRAINT chk_balance CHECK (Balance ¿= 0));

Domain Integrity Constraints: NOT NULL & CHECK II

- Multiple conditions can be combined logically in a CHECK constraint: CHECK (Age ¿= 18 AND Status IN ('Active', 'Suspended')).

Entity Integrity Constraints: UNIQUE & PRIMARY KEY I

- Entity integrity ensures that each row in a table is uniquely identifiable and prevents duplicated records.
- UNIQUE constraint: Guarantees that all values in a column or set of columns are distinct from one another. Depending on the RDBMS standard (e.g., PostgreSQL vs SQL Server), it may allow single or multiple NULL values.
- PRIMARY KEY: A semantic combination of a NOT NULL and a UNIQUE constraint. It acts as the definitive identifier for a table record.
- A relational table can have only ONE Primary Key, which may be a single attribute or a Composite Primary Key spanning multiple columns.

Entity Integrity Constraints: UNIQUE & PRIMARY KEY II

- Internally, databases typically construct a unique B-Tree index on Primary Key columns to optimize query execution plans and index lookups.

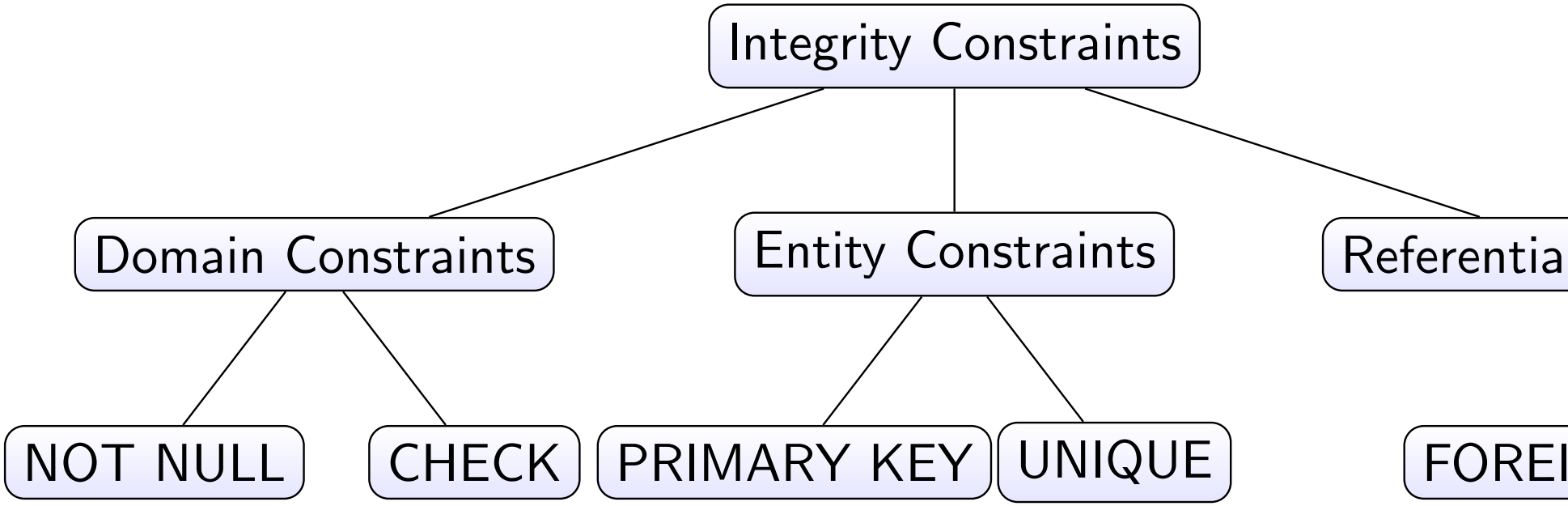
Referential Integrity Constraints: FOREIGN KEY & REFERENCES I

- Referential integrity establishes and maintains relationships between tables. It ensures that references from one table to another are always valid.
- FOREIGN KEY: An attribute or set of attributes in a referencing (child) table that structurally links to the PRIMARY KEY or UNIQUE constraint of a referenced (parent) table.
- The REFERENCES clause explicitly maps the child table's column to the parent table and its target column.
- Example: CREATE TABLE Orders (OrderID INT PRIMARY KEY, EmpID INT, FOREIGN KEY (EmpID) REFERENCES Employees(EmpID));

Referential Integrity Constraints: FOREIGN KEY & REFERENCES II

- If a DML operation (INSERT/UPDATE) attempts to introduce a foreign key value not present in the parent table, the engine rejects the transaction and raises a constraint violation.

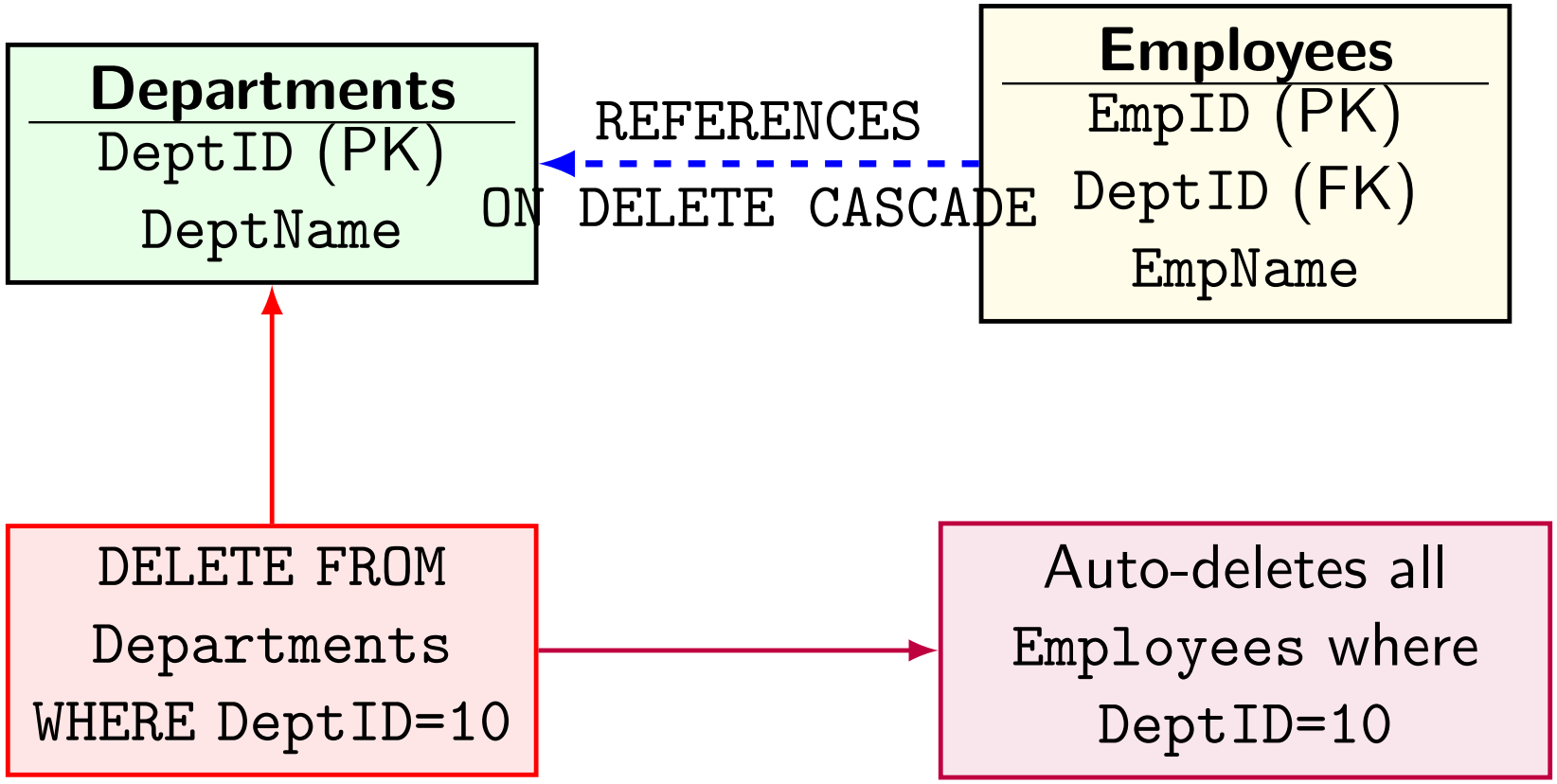
Taxonomy of Database Integrity Constraints I



Cascading Actions: ON DELETE CASCADE I

- By default, if a parent record is referenced by child records, the database blocks deletion of the parent (ON DELETE RESTRICT or NO ACTION) to prevent orphaned rows.
- ON DELETE CASCADE: Automates referential cleanup. When a parent record is deleted, all corresponding referencing rows in the child table are implicitly deleted by the database engine.
- This maintains structural integrity seamlessly, avoiding complex multi-step application logic.
- Example syntax: FOREIGN KEY (DeptID) REFERENCES Departments(DeptID) ON DELETE CASCADE
- Alternatives include ON DELETE SET NULL (replaces child foreign keys with NULL) and ON DELETE SET DEFAULT (replaces them with the column's default scalar value).

Visualizing Referential Integrity & Cascade Actions I



Best Practices for Database Constraints I

- 1. Define constraints during the initial schema design (CREATE TABLE) rather than appending them later (ALTER TABLE) to avoid retroactive data cleansing operations.
- 2. Explicitly name constraints. Avoid system-generated names (e.g., SYS_C0014) in favor of readable identifiers like CONSTRAINT fk_emp_dept FOREIGN KEY.
- 3. Avoid circular foreign key dependencies (Table A referencing Table B, and B referencing A), which vastly complicate row insertion and cascading deletions.
- 4. Exercise immense caution with ON DELETE CASCADE in production systems; a single accidental parent deletion can wipe out millions of critical child records.

Best Practices for Database Constraints II

- 5. Leverage DEFERRABLE constraints in advanced RDBMS instances (Oracle, PostgreSQL) for complex bulk inserts, delaying the integrity validation check until the transaction COMMIT phase.

Agenda I

- Database Objects Overview
- Views: CREATE VIEW, ALTER VIEW, DROP VIEW
- Synonyms: CREATE SYNONYM, DROP SYNONYM
- Sequences: CREATE SEQUENCE, ALTER SEQUENCE, DROP SEQUENCE
- Indexes: CREATE INDEX (UNIQUE, COMPOSITE), DROP INDEX

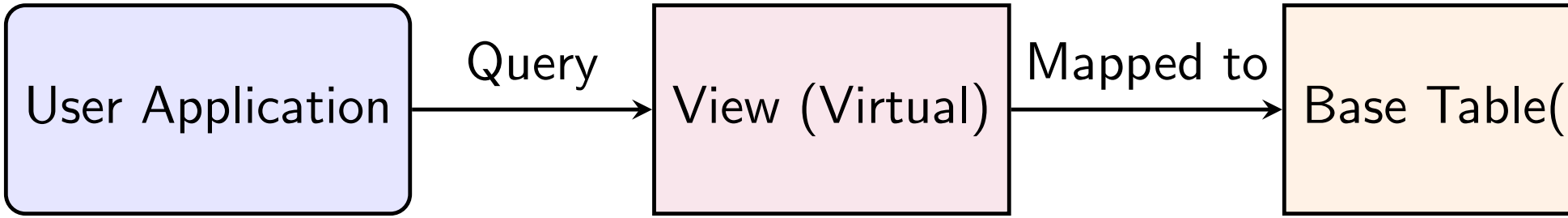
Database Objects: Views I

- A View is a logical or virtual table based on a query of one or more base tables.
- Views contain no data themselves; they act merely as a window into the underlying tables.
- Syntax: \{}texttt{CREATE [OR REPLACE] VIEW view\{}_name AS subquery;}
- Example: \{}texttt{CREATE VIEW emp\{}_vu AS SELECT emp\{}_id, name FROM employees WHERE dept = 10;}
- Benefits: Restricts data access, hides data complexity, and provides data independence.

Views: ALTER and DROP I

- ALTER VIEW is typically used to explicitly recompile a view that has become invalid (e.g., when base tables change).
- Syntax: \{}texttt{ALTER VIEW view\{}_name COMPILE;}
- To fundamentally modify a view's underlying query, use \{}texttt{CREATE OR REPLACE VIEW}.
- DROP VIEW removes the view definition from the database schema.
- Syntax: \{}texttt{DROP VIEW view\{}_name;}
- Dropping a view does not affect the data in the underlying base tables.

View Architecture Diagram I



Database Objects: Synonyms I

- A Synonym is an alias or alternative name for a database object such as a table, view, or sequence.
- Synonyms simplify SQL statements by hiding the object's schema and real name, providing location transparency.
- Syntax: \{}texttt{CREATE [PUBLIC] SYNONYM synonym\{}_name FOR object\{}_name;}
- Example: \{}texttt{CREATE PUBLIC SYNONYM emp FOR hr.employees;}
- DROP SYNONYM removes the alias from the database.
- Syntax: \{}texttt{DROP [PUBLIC] SYNONYM synonym\{}_name;}

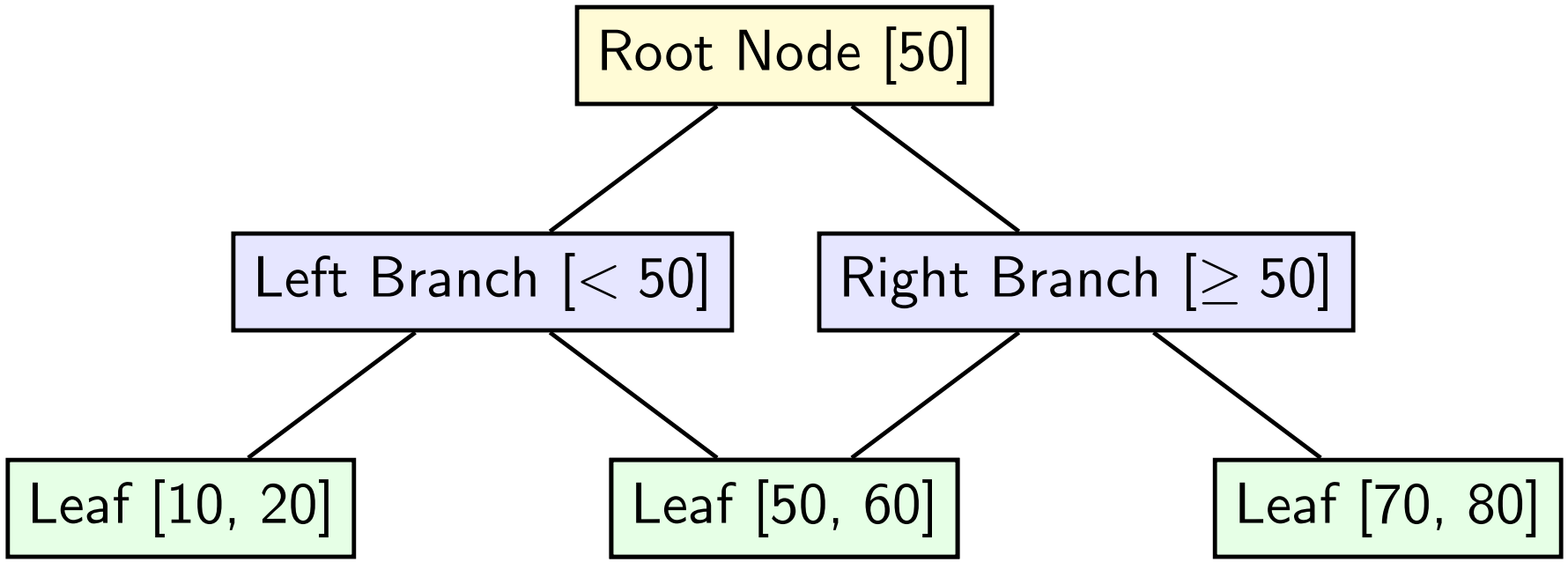
Database Objects: Sequences (Creation) I

- A Sequence is a database object used to automatically generate unique, sequential integers (often for Primary Keys).
- Syntax: \{}texttt{CREATE SEQUENCE seq\{}_name [START WITH n] [INCREMENT BY n] [MAXVALUE n] [CYCLE — NOCYCLE];}
- Example: \{}texttt{CREATE SEQUENCE dept\{}_seq START WITH 10 INCREMENT BY 10 MAXVALUE 100 NOCYCLE;}
- Sequence values are accessed via pseudocolumns in SQL queries:
- 1. \{}texttt{NEXTVAL}: Increments the sequence and returns the next value.
- 2. \{}texttt{CURRVAL}: Returns the current value (must call NEXTVAL first).

Sequences: ALTER and DROP I

- ALTER SEQUENCE modifies sequence attributes like increment, maxvalue, minvalue, cycle, or cache options.
- Important Note: You cannot change the \{}texttt{START WITH} value using \{}texttt{ALTER SEQUENCE}.
- Syntax: \{}texttt{ALTER SEQUENCE seq\{}_name INCREMENT BY 5 MAXVALUE 999;}
- DROP SEQUENCE removes the sequence object from the database.
- Syntax: \{}texttt{DROP SEQUENCE seq\{}_name;}
- Example: \{}texttt{DROP SEQUENCE dept\{}_seq;}

Indexes: B-Tree Architecture Diagram I



Database Objects: Indexes I

- An Index is an optional structure associated with a table to speed up data retrieval, similar to a book index.
- It creates a rapid path access method (usually B-Tree or Bitmap) to locate data quickly, reducing disk I/O.
- UNIQUE INDEX: Ensures uniqueness in the indexed columns. Example: \{}texttt{CREATE UNIQUE INDEX emp\{}_id\{}_idx ON emp(id);}
- COMPOSITE INDEX: An index on multiple columns. Example: \{}texttt{CREATE INDEX emp\{}_name\{}_idx ON emp(last\{}_name, first\{}_name);}
- DROP INDEX: Removes the index from the database. Syntax: \{}texttt{DROP INDEX index\{}_name;}

Agenda I

- Views: CREATE VIEW, ALTER VIEW, DROP VIEW
- Synonyms: CREATE SYNONYM, DROP SYNONYM
- Sequences: CREATE SEQUENCE, ALTER SEQUENCE, DROP SEQUENCE
- Indexes: CREATE INDEX (UNIQUE, COMPOSITE), DROP INDEX

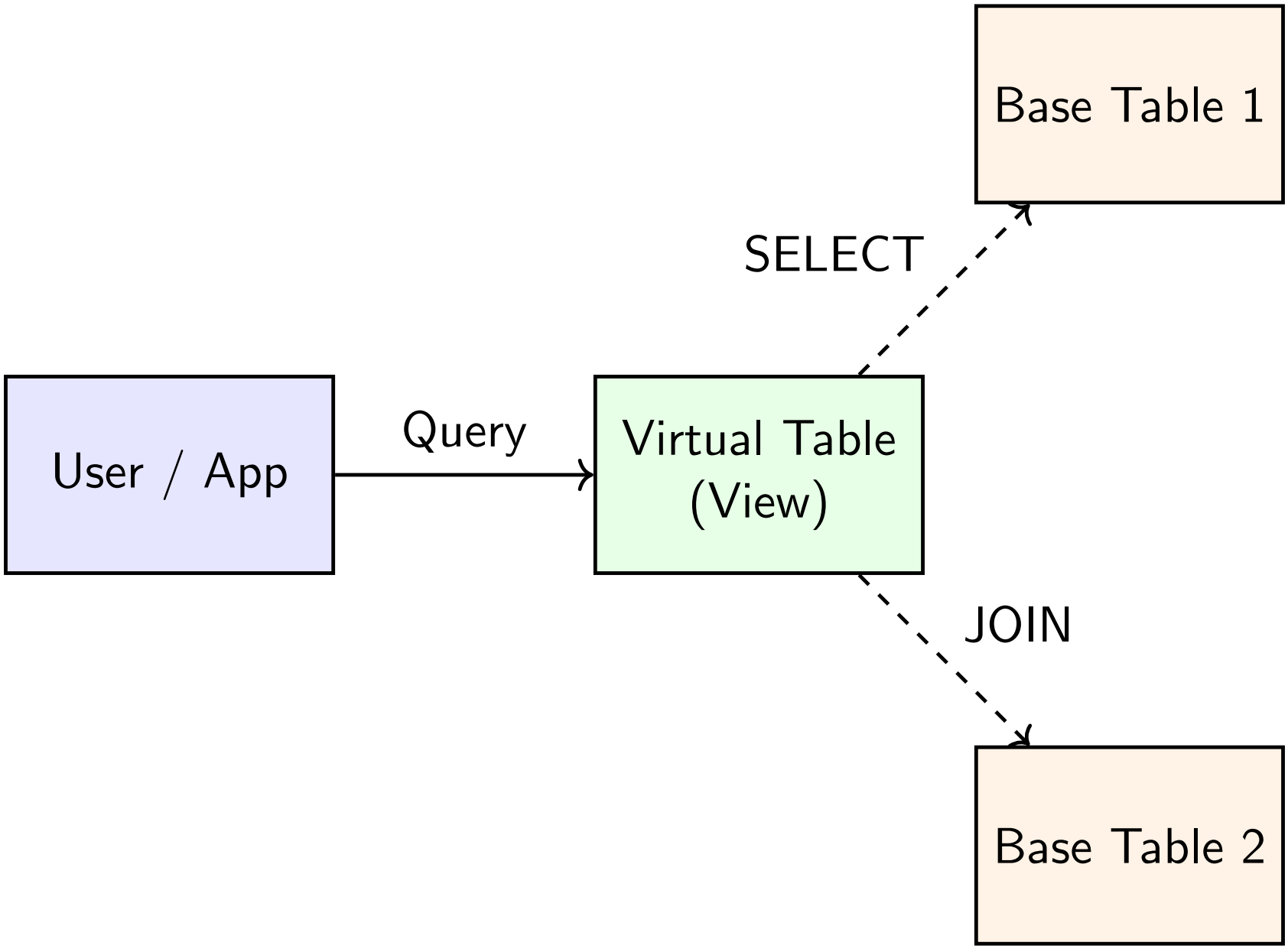
Database Views: CREATE VIEW I

- A View is a logical, virtual table created by executing a query. It does not store data itself (except for materialized views).
- Provides logical data independence and enhances security by restricting access to specific rows and columns.
- Syntax: CREATE [OR REPLACE] VIEW view_name AS SELECT columns FROM table_name WHERE condition;
- Example: CREATE VIEW active_employees AS SELECT emp_id, name, department FROM employees WHERE status = 'ACTIVE';
- The WITH CHECK OPTION constraint ensures that any INSERT or UPDATE through the view must satisfy the view's defining condition.

Database Views: ALTER and DROP VIEW I

- ALTER VIEW: Used to change the definition of an existing view without dropping it. Note: Many RDBMS, like Oracle and MySQL, prefer using CREATE OR REPLACE VIEW instead.
- Syntax (SQL Server/PostgreSQL): ALTER VIEW view_name AS SELECT new_columns FROM table_name;
- DROP VIEW: Removes the view definition from the database dictionary.
- Syntax: DROP VIEW view_name;
- Example: DROP VIEW active_employees;
- Important Note: Dropping a view does NOT affect the underlying base tables or their data.

Architecture of a Database View I



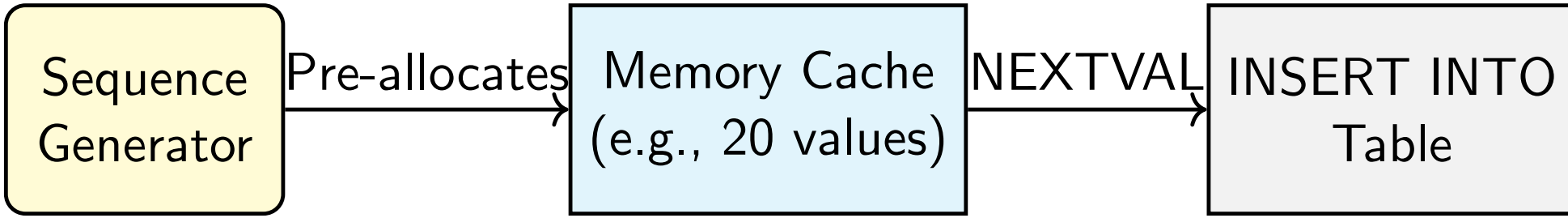
Synonyms: Alternative Naming I

- A Synonym provides an alternative alias for a database object, simplifying queries, enhancing security, and masking object details.
- It provides location transparency in distributed databases.
- Syntax: CREATE [PUBLIC] SYNONYM synonym_name FOR schema.object_name;
- Example: CREATE PUBLIC SYNONYM hr_emp FOR human_resources.employees_table_v2;
- Using a synonym: SELECT * FROM hr_emp; (Simplifies the query compared to the fully qualified name).
- DROP SYNONYM: Removes the alias. Syntax: DROP [PUBLIC] SYNONYM synonym_name;

Sequences: Auto-generating Values I

- A Sequence is a database object that generates a series of unique integers, primarily used for populating surrogate primary keys.
- CREATE SEQUENCE: Syntax: CREATE SEQUENCE seq_name [START WITH n] [INCREMENT BY n] [MAXVALUE n] [CACHE n] [NOCYCLE];
- Example: CREATE SEQUENCE seq_emp_id START WITH 100 INCREMENT BY 1 NOCACHE NOCYCLE;
- ALTER SEQUENCE: Modifies attributes like increment value or max limit. (Note: You cannot change START WITH).
- Syntax: ALTER SEQUENCE seq_emp_id INCREMENT BY 5 CACHE 50;
- DROP SEQUENCE: Removes the sequence. Syntax: DROP SEQUENCE seq_emp_id;

Sequence Generation Flow I



Indexes: CREATE and DROP I

- An Index accelerates data retrieval by pointing directly to data rows using structures like B-Trees. Trade-off: Slows down DML (INSERT, UPDATE, DELETE).
- CREATE INDEX: Creates a standard index. Example: CREATE INDEX idx_emp_dept ON employees(dept_id);
- CREATE UNIQUE INDEX: Enforces column uniqueness while indexing. Example: CREATE UNIQUE INDEX idx_emp_email ON employees(email);
- COMPOSITE INDEX: Indexes multiple columns together to satisfy multi-column queries. Example: CREATE INDEX idx_emp_name ON employees(last_name, first_name);
- DROP INDEX: Removes the index from the database. Syntax: DROP INDEX index_name; (In some RDBMS: DROP INDEX table_name.index_name);

Summary of Database Objects I

- Views act as virtual tables to abstract complex queries, provide data independence, and enhance row/column level security.
- Synonyms provide location transparency and simplified, secure aliases for underlying database objects.
- Sequences offer a highly concurrent, lock-free method to generate unique numeric identifiers for primary keys.
- Indexes critically improve SELECT query performance using specialized data structures, supporting UNIQUE and COMPOSITE configurations.
- All mentioned database objects are managed using standard Data Definition Language (DDL) statements: CREATE, ALTER, and DROP.

Agenda I

1. Complex Table Creation with Constraints
2. Sequence and Default value usage
3. View creation for complex queries
4. Indexing scenarios and query tuning
5. Trigger implementation for business rules

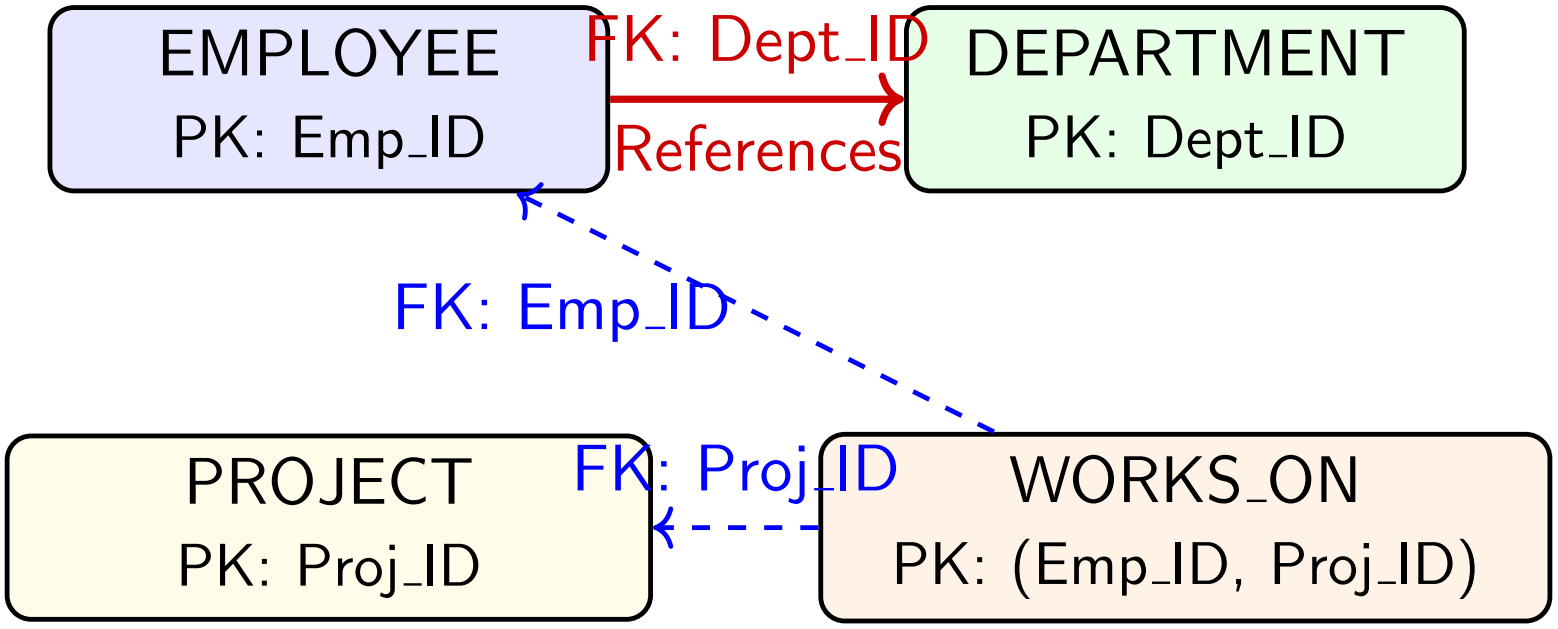
Problem 1: Complex Integrity Constraints I

- **Scenario:** Create a table \{}texttt{EMPLOYEE} and \{}texttt{DEPARTMENT}. An employee must belong to a valid department. Employee salaries must be strictly greater than 30000. Employee email must be unique.
- **SQL Solution:**
- \{}texttt{CREATE TABLE DEPARTMENT (Dept\{}_ID INT PRIMARY KEY, Dept\{}_Name VARCHAR(50) NOT NULL);}
- \{}texttt{CREATE TABLE EMPLOYEE (}
- \{}texttt{ Emp\{}_ID INT PRIMARY KEY,}
- \{}texttt{ Emp\{}_Name VARCHAR(100) NOT NULL,}
- \{}texttt{ Email VARCHAR(100) UNIQUE,}
- \{}texttt{ Salary DECIMAL(10,2) CHECK (Salary ¿ 30000),}
- \{}texttt{ Dept\{}_ID INT,}

Problem 1: Complex Integrity Constraints II

- \{}texttt{ CONSTRAINT fk\{}_dept FOREIGN KEY (Dept\{}_ID) REFERENCES DEPARTMENT(Dept\{}_ID)}
- \{}texttt{);}
- **Concepts applied:** PRIMARY KEY, FOREIGN KEY, UNIQUE constraint, and CHECK constraint.

Diagram: ER to Relational Mapping I



Problem 2: Sequences and Defaults I

- **Scenario:** You need to automatically generate the \{}texttt{Emp\{}_ID} for new employees starting from 1000, incrementing by 1.
- **Oracle SQL Solution using Sequences:**
- \{}texttt{CREATE SEQUENCE emp\{}_seq START WITH 1000 INCREMENT BY 1 NOCACHE;}
- **Insertion:**
- \{}texttt{INSERT INTO EMPLOYEE (Emp\{}_ID, Emp\{}_Name, Email, Salary, Dept\{}_ID) }
- \{}texttt{VALUES (emp\{}_seq.NEXTVAL, 'Alice Smith', 'alice@corp.com', 45000, 10);}
- **PostgreSQL Alternative (SERIAL/IDENTITY):**
- \{}texttt{Emp\{}_ID INT GENERATED ALWAYS AS IDENTITY (START WITH 1000 INCREMENT BY 1)}

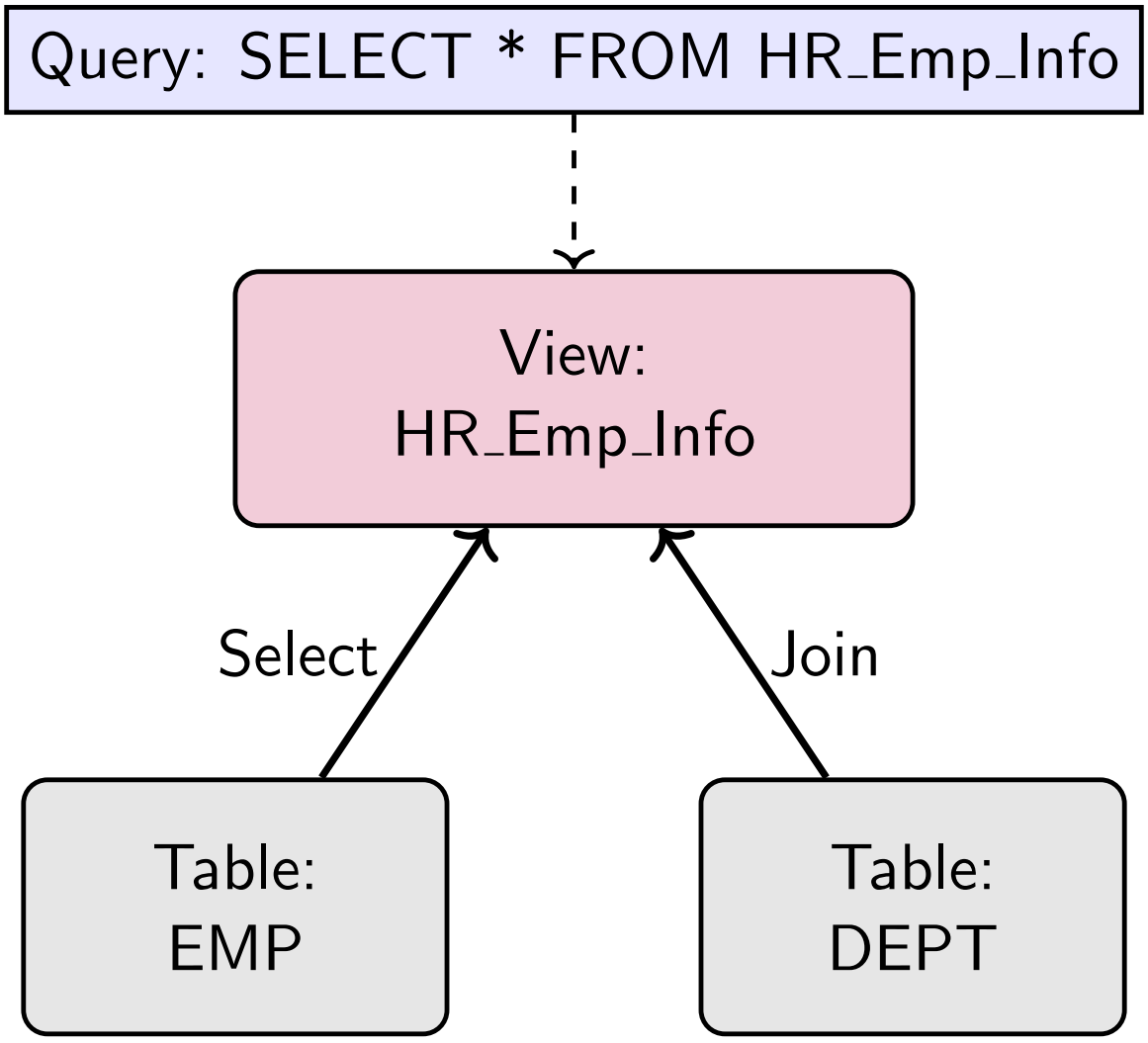
Problem 2: Sequences and Defaults II

- **Concept:** Surrogate keys management to ensure uniqueness decoupled from business data.

Problem 3: View Abstraction I

- **Scenario:** HR needs to frequently view Employee Names alongside their Department Names, but they should not have direct access to the \{}texttt{EMPLOYEE} salary data.
- **SQL Solution:**
- \{}texttt{CREATE VIEW HR\{}_Employee\{}_Info AS}
- \{}texttt{SELECT e.Emp\{}_Name, e.Email, d.Dept\{}_Name}
- \{}texttt{FROM EMPLOYEE e}
- \{}texttt{JOIN DEPARTMENT d ON e.Dept\{}_ID = d.Dept\{}_ID;}
- **Benefit:** Provides vertical security (hides the salary column) and simplifies complex JOIN logic for end-user queries.

Diagram: View Architecture I



Problem 4: Query Optimization with Indexes I

- **Scenario:** The HR application frequently searches employees by their \{}texttt{Email}. Currently, this performs a full table scan which is slow on large tables.
- **Problem:** Write a statement to optimize this search operation.
- **SQL Solution:**
- \{}texttt{CREATE INDEX idx\{}_emp\{}_email ON EMPLOYEE(Email);}
- **Explanation:** The B-tree index created on the \{}texttt{Email} column provides \{}(O(\{}log N)\{}) search time instead of \{}(O(N)\{}) full table scan.
- **Important Note:** Since \{}texttt{Email} was defined with a \{}texttt{UNIQUE} constraint during table creation, the database engine implicitly creates a unique index on it! Explicit creation is logically redundant here.

Problem 5: Enforcing Rules with Triggers I

- **Scenario:** Implement an audit mechanism. Whenever an employee's salary is updated, record the old salary, new salary, and timestamp in an \{}texttt{AUDIT\{}_LOG} table.
- **PL/SQL Solution:**
- \{}texttt{CREATE OR REPLACE TRIGGER trg\{}_salary\{}_audit}
- \{}texttt{AFTER UPDATE OF Salary ON EMPLOYEE FOR EACH ROW}
- \{}texttt{BEGIN}
- \{}texttt{ INSERT INTO AUDIT\{}_LOG (Emp\{}_ID, Old\{}_Sal, New\{}_Sal, Change\{}_Date)}
- \{}texttt{ VALUES (:OLD.Emp\{}_ID, :OLD.Salary, :NEW.Salary, SYSDATE);}
- \{}texttt{END;}

Problem 5: Enforcing Rules with Triggers II

- **Concepts applied:** Row-level trigger execution, \{}texttt{:OLD} and \{}texttt{:NEW} pseudorecords, and column-specific triggering.

Summary and Next Steps I

- **Summary of Practice Session:**
- 1. Modeled relational tables enforcing data integrity with Constraints.
- 2. Managed synthetic primary keys using Sequences.
- 3. Applied Views for security and query simplification.
- 4. Evaluated Indexing strategies and implicit index behaviors.
- 5. Designed a PL/SQL trigger for automated database auditing.
- **Next Steps:**
- Complete Lab Assignment 8: Implement these exact database objects in your local Oracle or PostgreSQL environment.

Agenda I

- 1. Recap of Integrity Constraints
- 2. Solving Complex Constraint Problems
- 3. View Creation and WITH CHECK OPTION
- 4. Sequence and Synonym Use-cases
- 5. Analyzing Index Performance via B-Trees

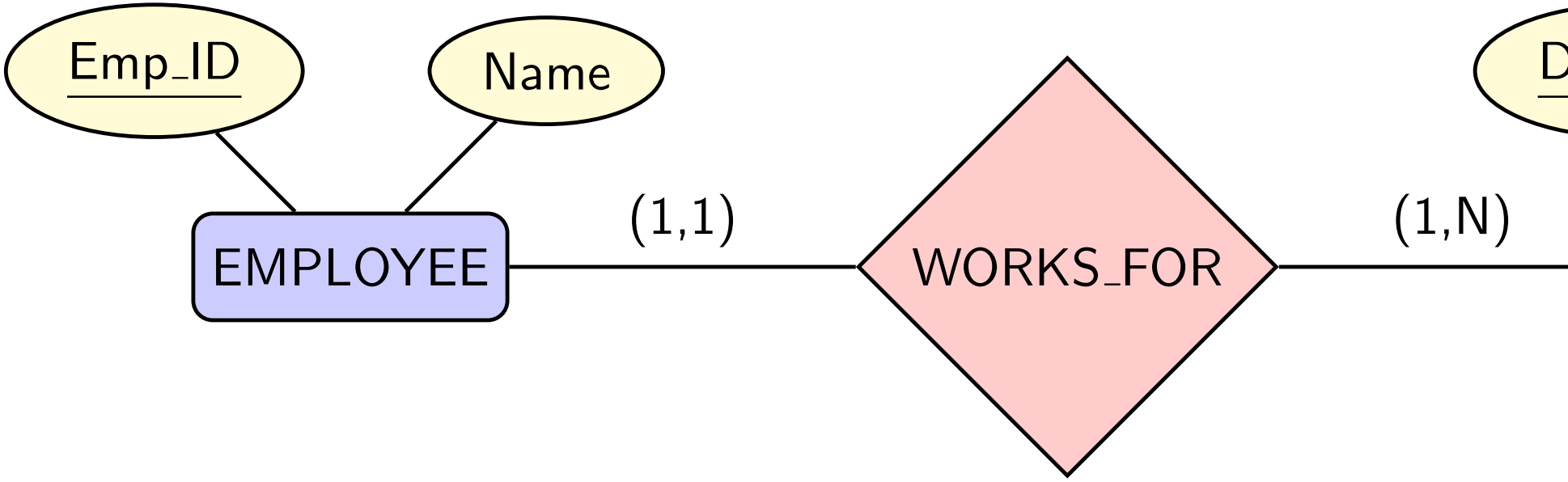
Problem 1: Complex Integrity Constraints I

- **Scenario**: Create an 'EMPLOYEES' table where an employee's salary must be greater than 30000, and the commission percentage should not exceed 15%. Also, the department ID must exist in the 'DEPARTMENTS' table, or be null.
- **Solution**: We implement this using a mix of CHECK, FOREIGN KEY, and domain constraints.
- CREATE TABLE Employees (
- Emp_ID INT PRIMARY KEY,
- Salary DECIMAL(10,2) CHECK (Salary ¿ 30000),
- Commission_Pct DECIMAL(3,2) CHECK (Commission_Pct ¡= 0.15),
- Dept_ID INT,

Problem 1: Complex Integrity Constraints II

- CONSTRAINT fk_dept FOREIGN KEY (Dept_ID) REFERENCES Departments(Dept_ID) ON DELETE SET NULL
-);
- Notice how ON DELETE SET NULL handles referential integrity when a department is removed without forcing the deletion of the employee.

Visualizing ER to Relational Constraints Mapping I



Problem 2: Enforcing Data Domain via Views I

- **Scenario**: HR wants a view for 'SALES' department employees (Dept_ID = 80). Any data modifications through this view MUST only belong to department 80.
- **Concept**: A simple view might allow inserting rows that fall outside the view's defining query. To prevent this, we use the WITH CHECK OPTION clause.
- CREATE OR REPLACE VIEW Sales_Emp_VW AS
- SELECT Emp_ID, Name, Salary, Dept_ID
- FROM Employees
- WHERE Dept_ID = 80
- WITH CHECK OPTION;
- **Analysis**: If a user runs 'INSERT INTO Sales_Emp_VW (Emp_ID, Name, Salary, Dept_ID) VALUES (101, 'John', 40000, 90);', it will FAIL with an ORA-01402 error (view WITH CHECK OPTION where-clause violation).

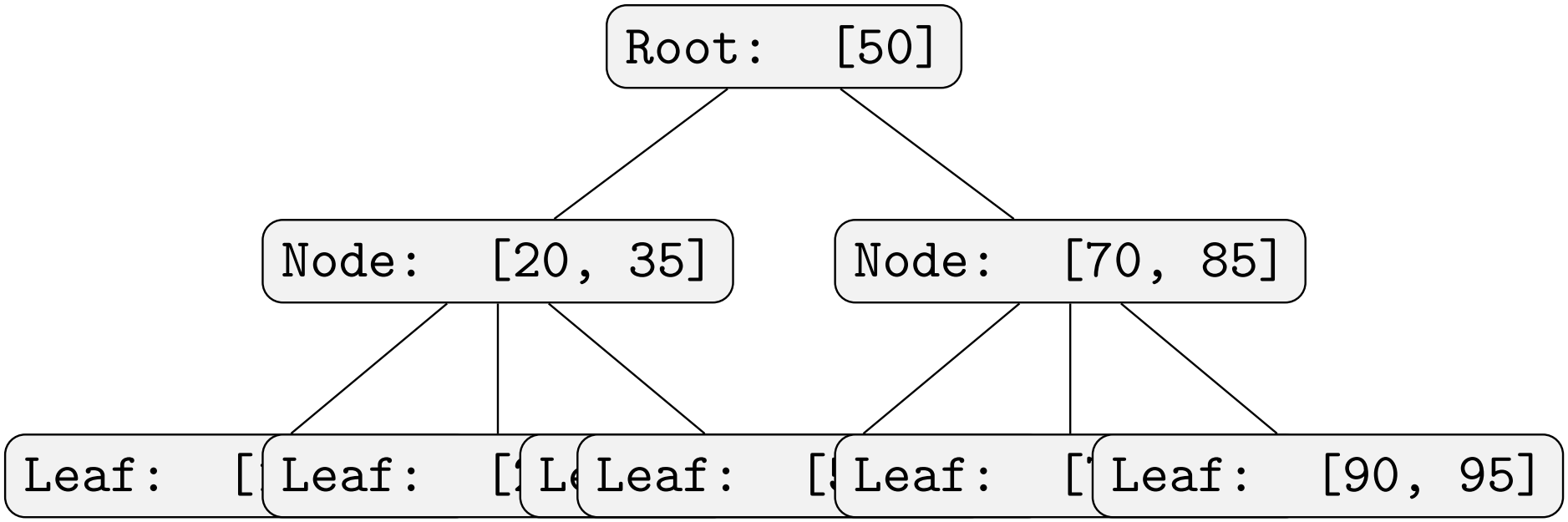
Problem 3: Sequence Management for Surrogate Keys I

- **Scenario**: We need to automatically generate primary keys for an 'ORDERS' table. The keys should start at 1000, increment by 1, and cache 20 values for performance.
- **Solution**: We utilize a SEQUENCE database object.
- CREATE SEQUENCE Order_Seq
- START WITH 1000
- INCREMENT BY 1
- CACHE 20
- NOCYCLE;
- **Implementation**: In an INSERT statement:
- INSERT INTO Orders (Order_ID, Order_Date) VALUES (Order_Seq.NEXTVAL, SYSDATE);

Problem 3: Sequence Management for Surrogate Keys II

- **Key Takeaway**: Caching improves performance by reducing database dictionary I/O. However, if the instance crashes unexpectedly, cached numbers might be permanently lost, resulting in gaps in the surrogate key sequence.

B-Tree Index Structure for Query Optimization I



Problem 4: Aliasing with Synonyms I

- **Scenario**: An application frequently queries ‘HR_ADMIN.EMPLOYEE_PAYROLL_DATA‘. Writing this long schema-prefixed name is cumbersome, and we want to abstract the underlying schema location.
- **Solution**: Create a PUBLIC synonym if all users need it, or a PRIVATE synonym for a specific database user.
- CREATE PUBLIC SYNONYM Payroll FOR HR_ADMIN.EMPLOYEE_PAYROLL_DATA;
- **Usage**: Authorized users can now simply execute: ‘SELECT * FROM Payroll;‘
- **Analysis**: Synonyms provide a layer of data independence. If the underlying table is moved to a different schema or database link, only the synonym definition needs updating. The application code using the synonym remains entirely unchanged.

Problem 5: Complex Indexing Strategies I

- **Scenario**: A reporting query filters employees by last name in uppercase: ‘SELECT * FROM Employees WHERE UPPER(Last_Name) = ’SMITH’;‘. A standard index on ‘Last_Name‘ is being bypassed by the optimizer.
- **Solution**: We must deploy a Function-Based Index to address this access pattern.
- CREATE INDEX emp_upper_name_idx ON Employees (UPPER(Last_Name));
- **Analysis**: The database optimizer cannot apply a standard B-Tree index if the indexed column is mutated by a function (like UPPER) in the WHERE predicate. The function-based index pre-computes the function result and stores it in the index tree, allowing for rapid retrieval.
- Note: Ensure that ‘QUERY_REWRITE_ENABLED‘ is active if required by your specific RDBMS version.

Conclusion & Practice Directives I

- **Summary of Database Object Problem Solving**:
- 1. Integrity Constraints: Crucial for maintaining ACID properties and logical data consistency at the engine level.
- 2. Views: Utilize WITH CHECK OPTION to enforce data boundaries and secure DML operations.
- 3. Sequences: Ideal for generating surrogate keys, but system designers must account for potential caching gaps.
- 4. Indexes: Deeply analyze query access patterns to accurately apply Standard versus Function-Based indexes.
- **Next Steps**: Complete Assignment 4. Focus specifically on constructing PL/SQL trigger blocks that mimic complex declarative constraints.

Agenda I

- 1. Basics of PL/SQL
- 2. PL/SQL Block Structure
- 3. Advantages of PL/SQL over SQL
- 4. Data Types Overview
- 5. The %TYPE Attribute
- 6. The %ROWTYPE Attribute
- 7. Execution Environment

Basics of PL/SQL I

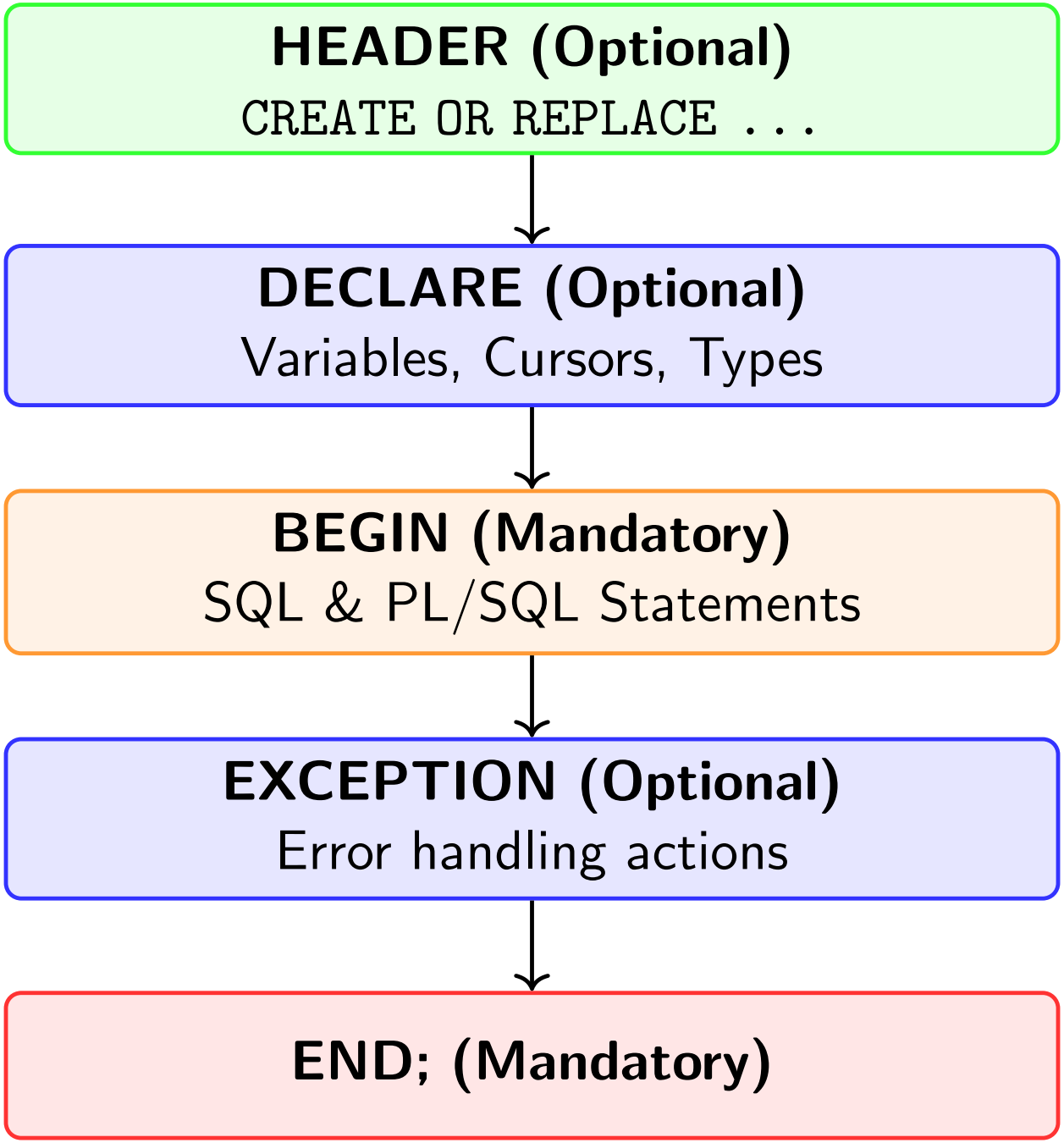
- PL/SQL (Procedural Language extensions to SQL) is Oracle Corporation's standard data access language for relational databases.
- It seamlessly integrates procedural constructs with SQL, allowing the creation of complex business logic inside the database.
- Key characteristics:
- • Block-structured language: Code is divided into logical blocks for modularity.
- • Tight integration with SQL: Supports all SQL data manipulation, cursor control, and transaction statements natively.
- • High Performance: Sends entire blocks of statements to the database engine simultaneously, drastically reducing network traffic compared to executing individual SQL statements.

Basics of PL/SQL II

- • Portability: PL/SQL applications can run on any OS and platform where an Oracle Database is running.

PL/SQL Block Structure I

PL/SQL Block Structure II



Advantages of PL/SQL over SQL I

- 1. Better Performance: SQL processes one statement at a time, generating network traffic for each. PL/SQL groups statements into a block, sending the entire block to the server in a single call.
- 2. Procedural Capabilities: Introduces conditional branching (IF-THEN-ELSE), loops (FOR, WHILE), and exception handling, which standard SQL lacks.
- 3. Reusability: Subprograms (Procedures, Functions) and Packages can be stored in the database and reused by multiple applications.
- 4. Portability: Any PL/SQL block can be moved to another Oracle Database on a different platform without modification.
- 5. Security: Applications can be granted execute privileges on a stored procedure without granting direct access to the underlying tables.

PL/SQL Data Types I

- PL/SQL supports a rich set of data types that mirror standard SQL types, along with specialized PL/SQL-only types:
- • Scalar Types: Hold a single value. Examples: NUMBER, VARCHAR2, BOOLEAN, DATE, CHAR.
- • Composite Types: Collections of scalar types. Examples: RECORD, TABLE, VARRAY.
- • Reference Types: Pointers to other data items. Examples: REF CURSOR.
- • LOB Types: Large Object types for holding massive amounts of unstructured data. Examples: BLOB, CLOB, BFILE.
- Special Anchor Attributes (Dynamic Typing):
- • %TYPE: Anchors a variable's data type to a column's data type.
- • %ROWTYPE: Anchors a record's structure to a complete table row.

The %TYPE Attribute I

- The %TYPE attribute is used to declare a variable with the exact same data type as a database column or another previously declared variable.
- Advantages:
- • You don't need to hardcode the exact data type of the underlying column.
- • If the database column definition changes (e.g., VARCHAR2(50) to VARCHAR2(100)), the PL/SQL variable automatically adapts at runtime without requiring code changes.
- Syntax Example:
- DECLARE
- v_emp_name employees.last_name%TYPE;
- v_salary employees.salary%TYPE;

The %TYPE Attribute II

- BEGIN
- SELECT last_name, salary INTO v_emp_name, v_salary
- FROM employees WHERE employee_id = 100;
- END;

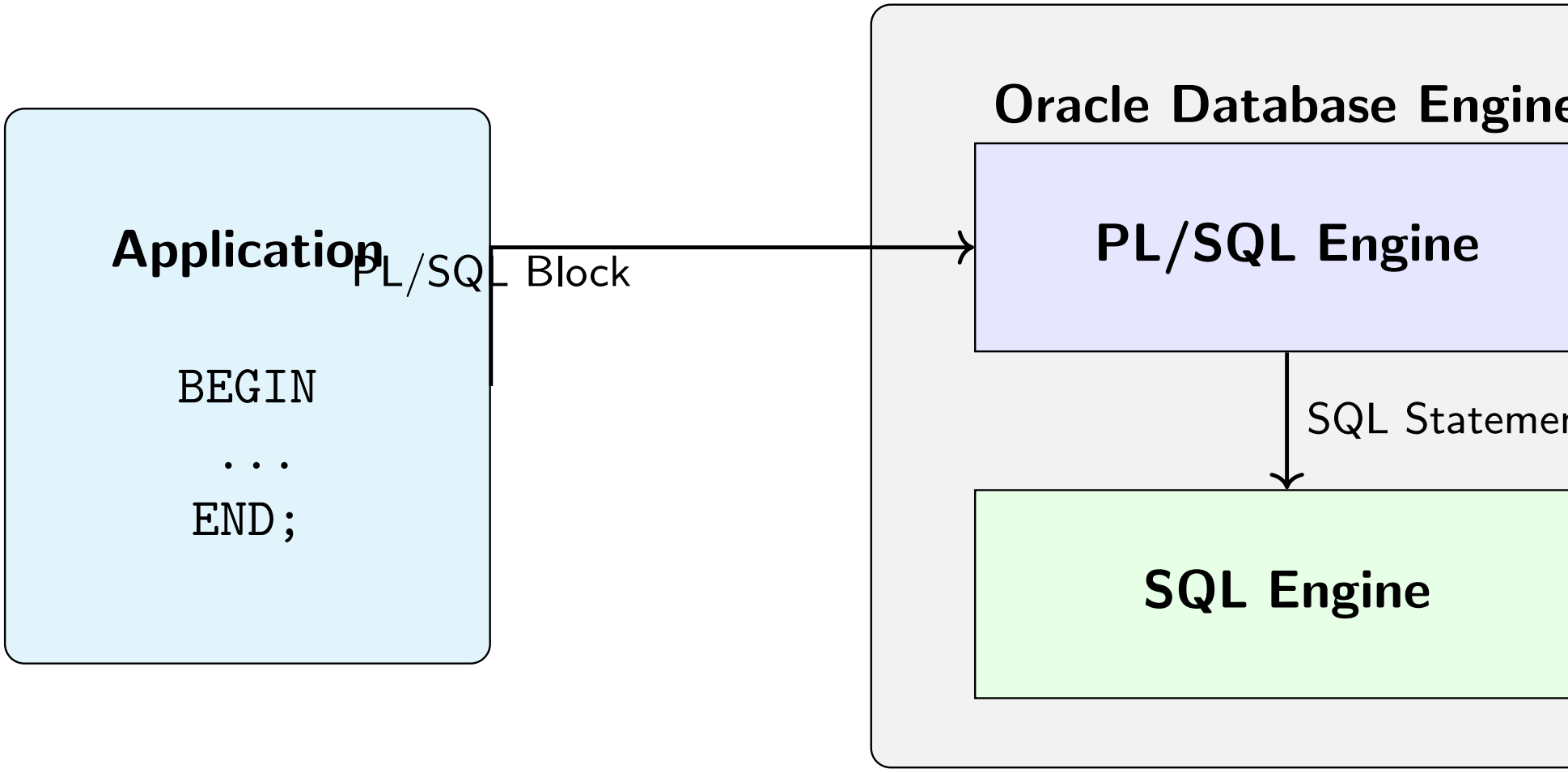
The %ROWTYPE Attribute I

- The %ROWTYPE attribute allows you to declare a composite record variable that represents an entire row of a table or a view.
- The fields in the record take their names and data types from the columns of the table.
- Advantages: Massive reduction in code verbosity and automatic adaptation to table schema changes (like adding or dropping columns).
- Syntax Example:
- DECLARE
- v_employee_record employees%ROWTYPE;
- BEGIN
- SELECT * INTO v_employee_record
- FROM employees WHERE employee_id = 100;

The %ROWTYPE Attribute II

- DBMS_OUTPUT.PUT_LINE('Name: ' ——
  v_employee_record.last_name);
- DBMS_OUTPUT.PUT_LINE('Salary: ' ——
  v_employee_record.salary);
- END;

PL/SQL Execution Environment I



Conclusion I

- • PL/SQL Bridges the gap between declarative database operations and procedural programming.
- • Using %TYPE and %ROWTYPE enforces data independence and builds resilient code that survives schema changes.
- • The PL/SQL Engine fundamentally optimizes database interaction by reducing network round-trips via block execution.
- • Mastering PL/SQL is essential for robust, secure, and high-performance Oracle Database applications.
- Next Lecture: PL/SQL Control Structures (IF statements, Loops) and explicit Cursors.

Agenda I

- Introduction and Basics of PL/SQL
- PL/SQL Block Structure
- Architecture of the PL/SQL Engine
- Data Types: %TYPE and %ROWTYPE Attributes
- Advantages of PL/SQL over Standard SQL
- SQL vs PL/SQL Execution Models

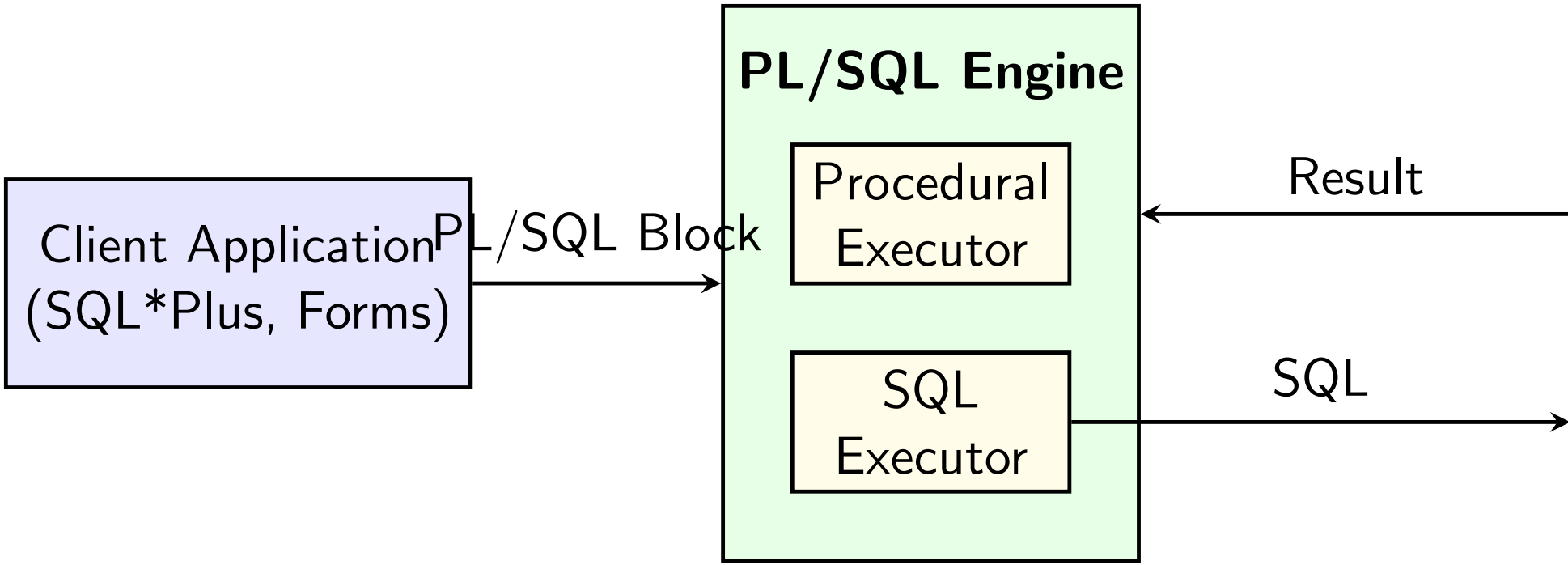
Basics of PL/SQL I

- PL/SQL (Procedural Language extension to SQL) is Oracle Corporation's procedural extension for SQL and the Oracle relational database. It integrates seamlessly with SQL, allowing both declarative and procedural paradigms.
- Unlike standard SQL which is a declarative, non-procedural language (you state WHAT you want, not HOW to get it), PL/SQL allows conditional branching (IF-THEN-ELSE), loops (FOR, WHILE), and variable declarations.
- Variables, constants, cursors, and exceptions can be defined and manipulated within the scope of a PL/SQL block, enabling complex business logic to reside directly within the database tier.
- PL/SQL code can be stored natively in the Oracle Database as Procedures, Functions, Packages, and Triggers, providing centralized logic and improved reusability.

PL/SQL Block Structure I

- A PL/SQL program unit is strictly defined by a block structure consisting of up to three sections: Declarative (optional), Executable (mandatory), and Exception Handling (optional).
- DECLARE section: Used to define memory-allocated variables, constants, cursors, and user-defined exceptions. E.g., ‘v_emp_id NUMBER(6);‘
- BEGIN section: Contains executable SQL and PL/SQL statements. This is the heart of the block where logic runs.
- EXCEPTION section: Captures and handles runtime errors globally for the block, preventing abrupt termination (e.g., ‘WHEN NO_DATA_FOUND THEN ...‘).
- Example snippet: ‘DECLARE‘ ‘ v_count NUMBER;‘ ‘BEGIN‘ ‘ SELECT COUNT(*) INTO v_count FROM employees;‘ ‘EXCEPTION‘ ‘ WHEN OTHERS THEN DBMS_OUTPUT.PUT_LINE(’Error’);‘ ‘END;‘

Architecture of the PL/SQL Engine I



Data Types: The %TYPE Attribute I

- The ‘%TYPE‘ attribute declares a variable with the exact same data type and size as a referenced database column or another previously declared variable.
- Syntax: ‘variable_name table_name.column_name%TYPE;‘
- It guarantees data type compatibility and allows the table schema to evolve without breaking the PL/SQL code (e.g., if a VARCHAR2 size increases from 50 to 100, the variable automatically adapts at runtime).
- Example: ‘v_last_name employees.last_name%TYPE;‘ ensures that ‘v_last_name‘ matches the exact definition of the ‘last_name‘ column in the ‘employees‘ table.
- Note: ‘%TYPE‘ inherits the data type and size, but does NOT inherit constraints such as NOT NULL or default values from the underlying column.

Data Types: The %ROWTYPE Attribute I

- The ‘%ROWTYPE‘ attribute declares a composite record variable that represents a complete row of a database table or a view.
- Syntax: ‘record_name table_name%ROWTYPE;‘
- A single ‘%ROWTYPE‘ variable can store all columns fetched from a table row, acting as an implicit structure where each column acts as a field of the record.
- Example: ‘DECLARE‘ ‘ rec_emp employees%ROWTYPE;‘ ‘BEGIN‘ ‘ SELECT * INTO rec_emp FROM employees WHERE employee_id = 100;‘ ‘ DBMS_OUTPUT.PUT_LINE(rec_emp.first_name —— ’ ’ —— rec_emp.last_name);‘ ‘END;‘
- This vastly reduces the number of variables required when selecting multiple columns and makes the code highly resilient to DDL changes like column additions or reordering.

Advantages of PL/SQL over SQL (Network Efficiency) I

- In standard SQL, each query requires a round-trip between the client application and the database server, leading to significant network latency for complex operations.
- PL/SQL encapsulates multiple SQL statements and procedural logic into a single block that is sent to the server in one payload.
- The PL/SQL engine executes the entire block locally on the database server, passing only internal calls between the procedural executor and the SQL statement executor natively.
- Only the final execution results or relevant output is returned to the client, drastically cutting down on network bandwidth usage and improving overall application throughput.

Advantages of PL/SQL (Modularity, Performance, Security) I

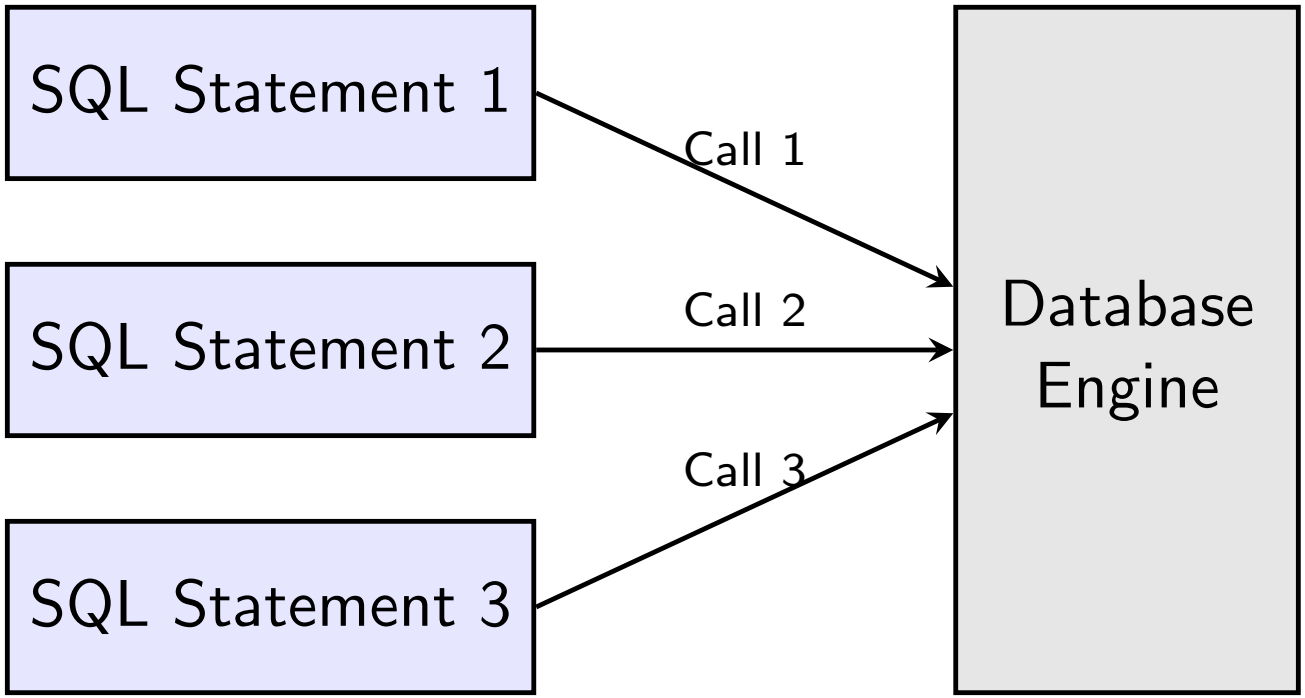
- Modularity: PL/SQL promotes modular programming. Logic can be grouped into independent functions and procedures, and further encapsulated into Packages with public and private interfaces.
- High Performance: Because PL/SQL code is parsed, compiled, and stored in the database's shared pool, subsequent executions skip the parsing phase. The engine uses a highly optimized bytecode format.
- Security & Abstraction: Applications can be granted execute privileges on PL/SQL procedures without needing direct SELECT/INSERT/UPDATE privileges on the underlying tables, creating a secure data access layer.

Advantages of PL/SQL (Modularity, Performance, Security) II

- Portability: Because PL/SQL is natively integrated into Oracle, any PL/SQL block or application will run seamlessly on any platform (Linux, Windows, Unix) that runs the Oracle Database.

SQL vs PL/SQL Execution Models I

Standard SQL



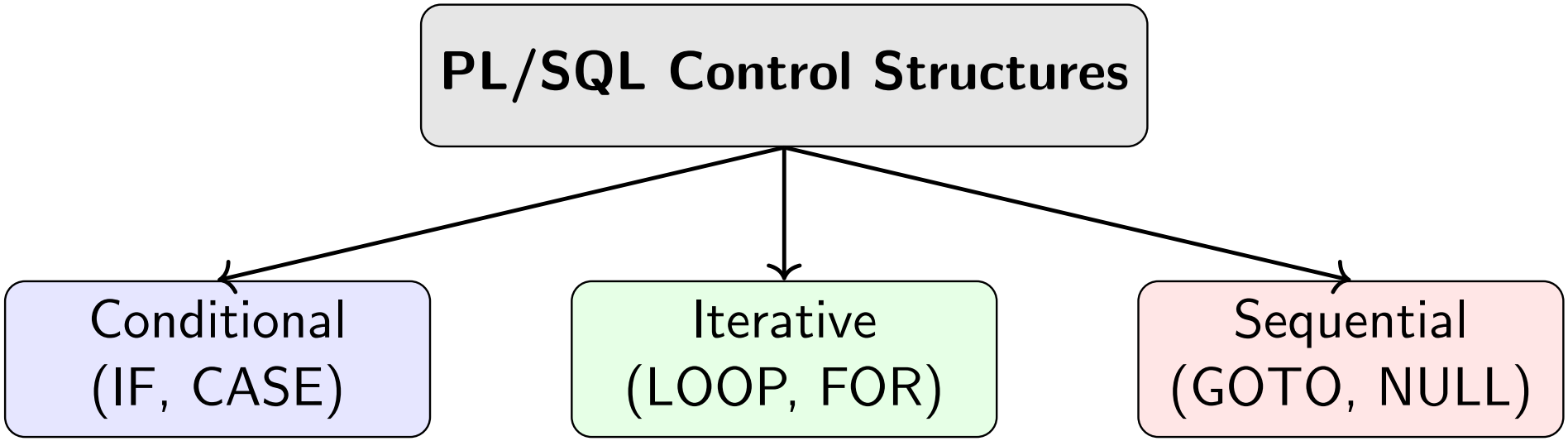
Agenda I

- 1. Introduction to PL/SQL Control Structures
- 2. Classification of Control Structures
- 3. Conditional Control: IF and CASE
- 4. Iterative Control Flowchart
- 5. Iterative Control: LOOP, WHILE, and FOR
- 6. Sequential Control: GOTO and NULL
- 7. Best Practices and Performance Considerations

Introduction to PL/SQL Control Structures I

- By default, PL/SQL executes statements sequentially from top to bottom.
- Control structures allow the manipulation of the logical execution flow using branches and loops.
- Without control structures, PL/SQL would be limited to linear processing. Their addition makes PL/SQL a Turing-complete programming language.
- Control statements can be nested indefinitely, but deep nesting can degrade code readability and maintainability.
- They are fundamentally classified into three categories: Conditional, Iterative, and Sequential controls.

Classification of Control Structures I



Conditional Statements: IF-THEN-ELSIF I

- The IF statement executes or skips a sequence of statements based on a boolean expression.
- PL/SQL utilizes three-valued logic (TRUE, FALSE, NULL). If a condition evaluates to NULL, it is treated identically to FALSE.
- Syntax utilizes 'ELSIF' (not ELSEIF or ELSE IF).
- “‘sql DECLARE v_salary NUMBER := 50000; BEGIN IF v_salary ¿ 100000 THEN DBMS_OUTPUT.PUT_LINE('Executive'); ELSIF v_salary ¿ 50000 THEN DBMS_OUTPUT.PUT_LINE('Manager'); ELSE DBMS_OUTPUT.PUT_LINE('Staff'); END IF; END; “‘
- Always ensure boolean evaluation handles potential NULL variables properly to avoid implicit logical bugs.

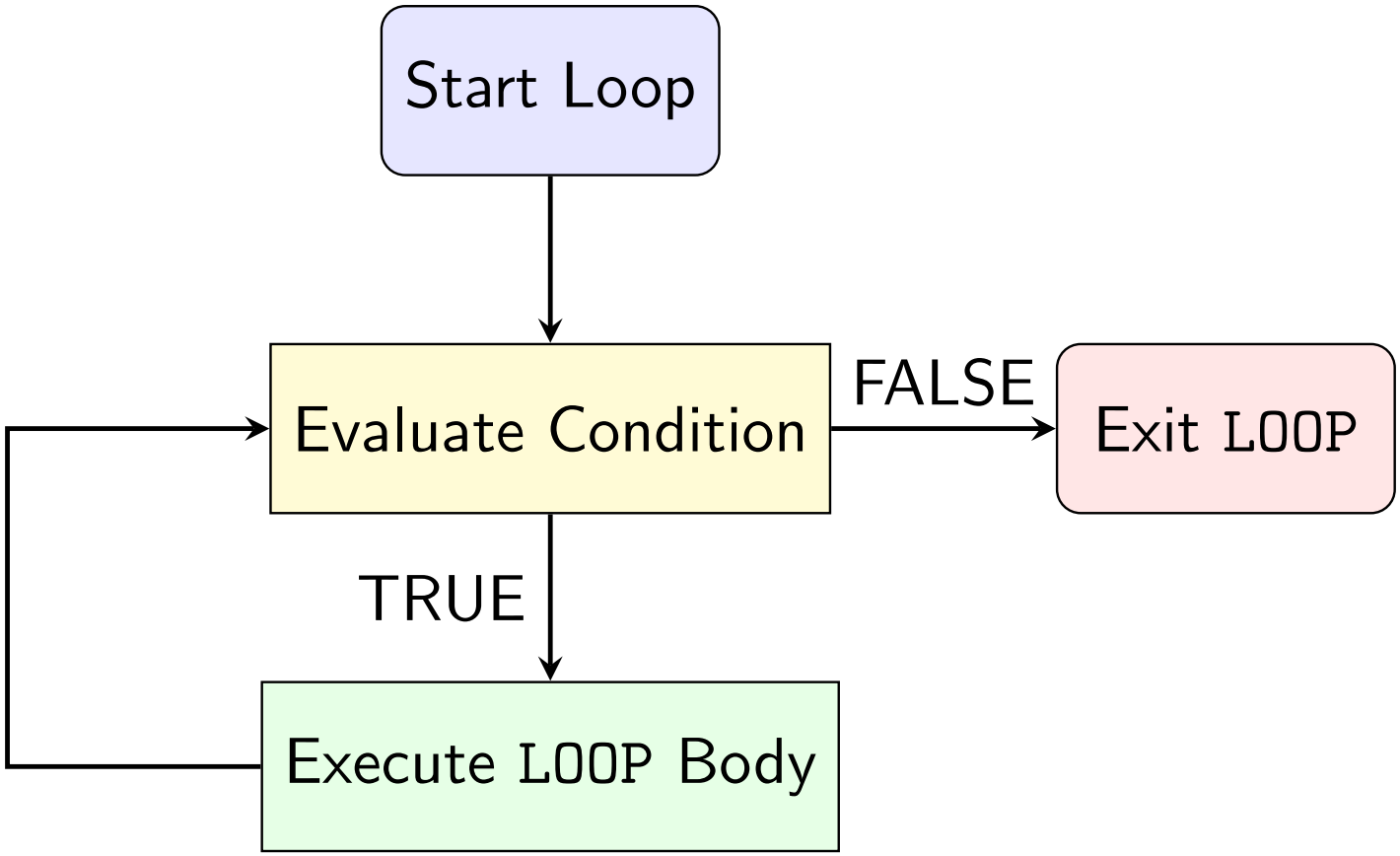
Conditional Statements: CASE I

- CASE provides a more readable alternative to complex, deeply nested IF-THEN-ELSIF blocks.
- Simple CASE: Compares a single selector expression against multiple literal values.
- Searched CASE: Evaluates multiple, independent boolean conditions sequentially. PL/SQL implements short-circuit evaluation for these conditions.
- “‘sql DECLARE v_grade CHAR(1) := ’B’; BEGIN CASE WHEN v_grade = ’A’ THEN DBMS_OUTPUT.PUT_LINE(’Excellent’); WHEN v_grade = ’B’ THEN DBMS_OUTPUT.PUT_LINE(’Good’); ELSE DBMS_OUTPUT.PUT_LINE(’Needs Improvement’); END CASE; END; “‘

Conditional Statements: CASE II

- Unlike SQL CASE expressions which return a value, PL/SQL CASE statements execute blocks. Missing an ELSE raises a CASE_NOT_FOUND exception.

Flowchart of an Iterative Loop (WHILE) I



Iterative Control: Basic, WHILE, and FOR Loops I

- Basic LOOP: Executes at least once. Relies on an 'EXIT WHEN' condition to break the iteration and prevent infinite loops.
- WHILE LOOP: Pre-evaluates conditions; the loop body will not execute at all if the initial condition is FALSE.
- FOR LOOP: Iterates over a defined bound of integers. The loop index is implicitly declared as a PLS_INTEGER, is read-only inside the loop, and goes out of scope after termination.
- “‘sql BEGIN FOR i IN REVERSE 1..3 LOOP DBMS_OUTPUT.PUT_LINE('Count: ' —— i); END LOOP; END; “‘
- Use cursor FOR loops for iterating over result sets to abstract away OPEN, FETCH, and CLOSE operations.

Sequential Control: GOTO and NULL I

- GOTO unconditionally alters execution flow by branching to a label defined as ¡¡label_name¿¿.
- Scope restrictions apply: You cannot use GOTO to jump into an IF block, a loop body, or a CASE statement from the outside.
- NULL is an executable statement that performs no action. It is often used as a placeholder during top-down development.
- ‘‘‘sql BEGIN IF v_status = 'INVALID' THEN NULL; – Implicitly ignore this state ELSE GOTO process_data; END IF; ¡¡process_data¿¿ DBMS_OUTPUT.PUT_LINE('Processing logic here...'); END; ‘‘‘
- NULL is notoriously used in EXCEPTION blocks (e.g., WHEN OTHERS THEN NULL;) - an anti-pattern known as exception swallowing.

Performance Considerations and Best Practices I

- Minimize GOTO Usage: GOTO statements create 'spaghetti code' that is difficult to maintain and debug. Prefer structured loops and conditionals.
- Avoid Implicit Conversions: Ensure conditions compare identical data types to utilize indexes efficiently and avoid excess CPU overhead.
- Beware of Context Switches: Executing SQL DML within a PL/SQL loop causes massive context switching overhead between the PL/SQL and SQL engines.
- Use Bulk SQL: Replace row-by-row FOR loops with BULK COLLECT and FORALL constructs to process large collections of data concurrently.
- Limit Nested Complexity: Deeply nested IF statements should be refactored into modular subprograms or converted to Searched CASE statements to enhance readability.

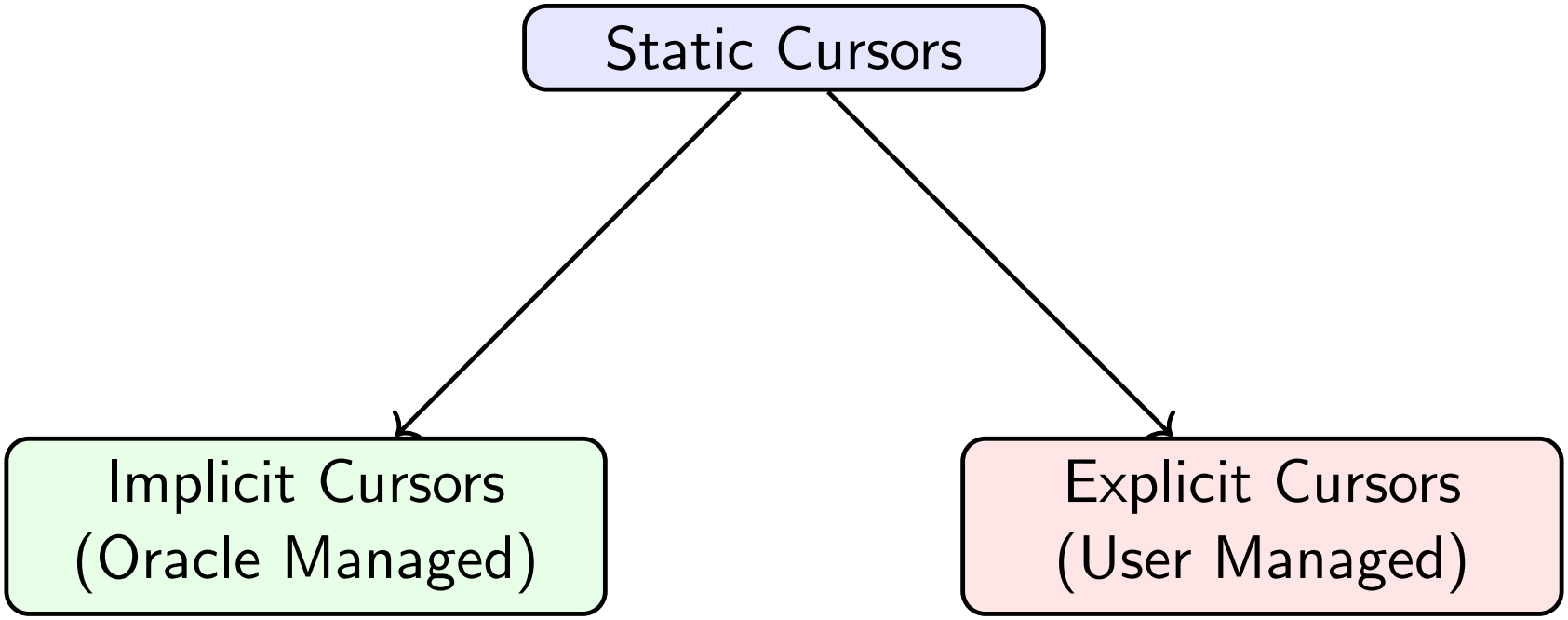
Agenda I

- 1. Introduction to Cursors
- 2. Context Areas and Active Sets
- 3. Static Cursors Classification
- 4. Implicit Cursors
- 5. Implicit Cursor Attributes
- 6. Explicit Cursors
- 7. Explicit Cursor Lifecycle
- 8. Practical Examples in PL/SQL

What is a Cursor? I

- Oracle Engine uses a work area for its internal processing to execute SQL statements. This work area is private to SQL operations and is called a Context Area.
- A Cursor is a pointer to this context area or memory location allocated by the database server.
- When an executable SQL statement is encountered, the engine allocates memory to process it. The cursor allows a PL/SQL block to name this context area, access the information, and control its processing.
- Active Set: The set of rows returned by a query. A cursor points to the current row being processed within this active set.

Classification of Static Cursors I



Implicit Cursors I

- Implicit cursors are automatically created by Oracle whenever an SQL statement is executed, assuming no explicit cursor is present.
- They are generated for all DML statements (INSERT, UPDATE, DELETE) and single-row SELECT ... INTO statements.
- Programmers have no programmatic control over implicit cursors (they cannot explicitly OPEN, FETCH, or CLOSE them).
- The cursor is named 'SQL' internally. We use this prefix 'SQL' followed by the attribute name to access information about the most recently executed implicit cursor.

Implicit Cursor Attributes I

- Oracle provides attributes to check the status of DML operations. Accessed via SQL%ATTRIBUTE_NAME:
- 1. SQL%ISOPEN: Always evaluates to FALSE because Oracle automatically closes implicit cursors after execution is complete.
- 2. SQL%FOUND: Returns TRUE if an INSERT, UPDATE, or DELETE affected one or more rows, or a SELECT INTO returned exactly one row.
- 3. SQL%NOTFOUND: The logical opposite of %FOUND. Returns TRUE if zero rows were affected.
- 4. SQL%ROWCOUNT: Returns the integer number of rows affected by the most recent SQL statement.

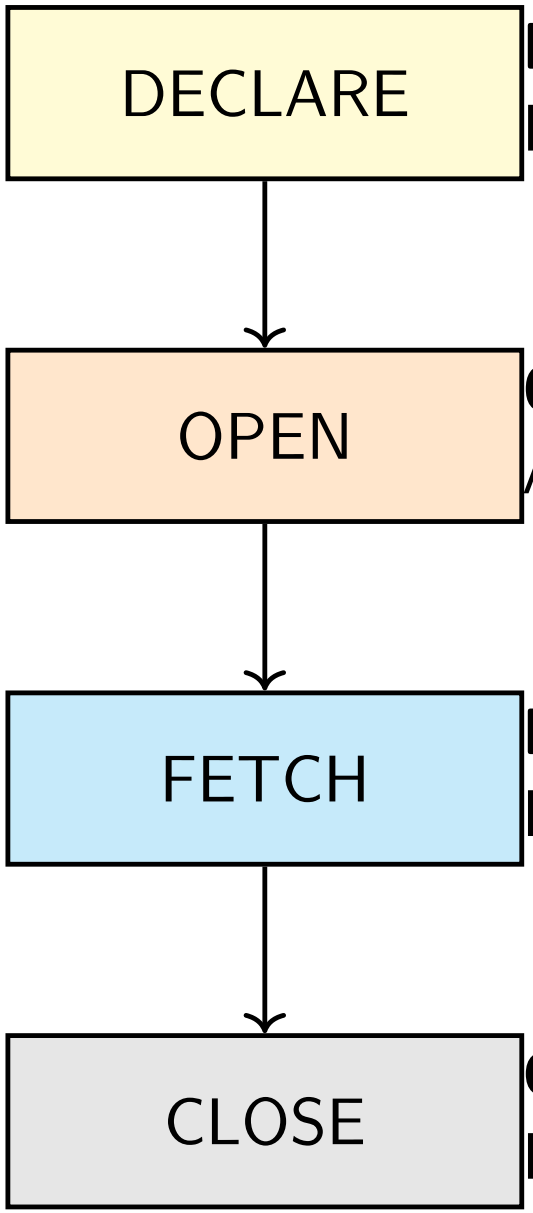
Example: Implicit Cursor I

- DECLARE
- v_emp_id employees.employee_id%TYPE := 101;
- BEGIN
- UPDATE employees SET salary = salary * 1.10 WHERE employee_id = v_emp_id;
- IF SQL%FOUND THEN
- DBMS_OUTPUT.PUT_LINE('Employee salary updated successfully. Rows updated: ' —— SQL%ROWCOUNT);
- ELSE
- DBMS_OUTPUT.PUT_LINE('Employee not found or no update performed.');
- END IF;
- END;

Explicit Cursors I

- Explicit cursors are defined by the programmer for queries that return more than one row.
- Since standard PL/SQL scalar variables can only hold one value at a time, queries returning multiple rows would raise a TOO_MANY_ROWS exception if handled by a basic SELECT INTO.
- Provides fine-grained control over the context area and the row processing loop.
- Explicit cursor attributes (e.g., cursor_name%FOUND, cursor_name%ROWCOUNT) function similarly to implicit attributes but are tied to a specific named cursor.

Explicit Cursor Lifecycle I



Declare
Define SQL query

Open
Allocate memory, find active set

Fetch
Load current row, advance pointer

Close
Free memory and resources

Example: Explicit Cursor I

- DECLARE
- CURSOR emp_cursor IS SELECT employee_id, last_name FROM employees WHERE department_id = 50;
- v_emp_id employees.employee_id%TYPE;
- v_name employees.last_name%TYPE;
- BEGIN
- OPEN emp_cursor;
- LOOP
- FETCH emp_cursor INTO v_emp_id, v_name;
- EXIT WHEN emp_cursor%NOTFOUND;
- DBMS_OUTPUT.PUT_LINE('ID: ' —— v_emp_id —— ', Name: ' —— v_name);
- END LOOP;
- CLOSE emp_cursor;
- END;

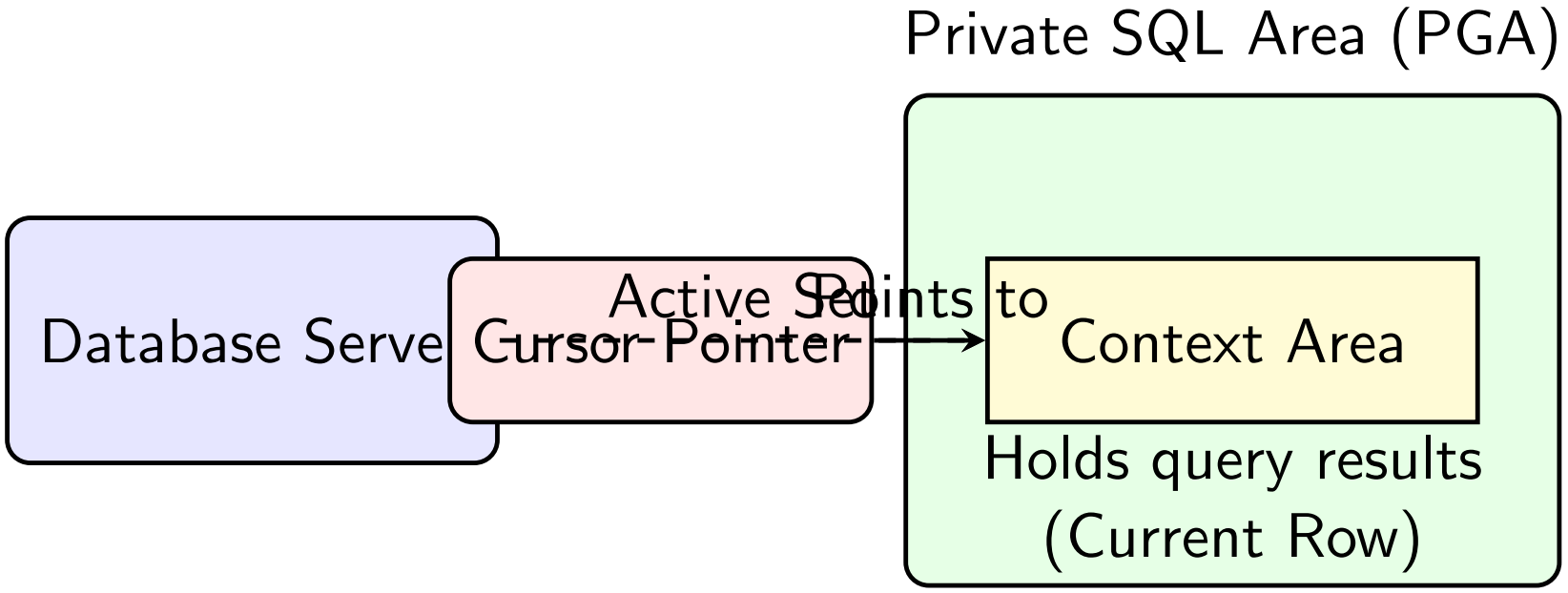
Agenda I

- Introduction to Cursors
- Implicit Cursors
- Explicit Cursors
- Cursor Lifecycle
- Examples and Best Practices

What is a Cursor? I

- A cursor is a temporary work area created in the system memory when a SQL statement is executed.
- It holds the rows (one or more) returned by a SQL statement, known as the 'active set'.
- Since PL/SQL does not natively support multi-row query results in simple variables, cursors act as pointers to the context area.
- They are essential for row-by-row processing, which is often required for complex business logic, data validation, and transformations.

Cursor Context Area Architecture I



Static Cursors Overview I

- Static cursors are predefined during compile time. Their SQL text is known and fixed when the PL/SQL block is compiled.
- Implicit Cursors: Automatically managed by Oracle for all DML (INSERT, UPDATE, DELETE) and single-row SELECT (SELECT INTO) statements. The programmer has no direct control over them.
- Explicit Cursors: Declared and managed explicitly by the programmer for multi-row SELECT statements. They offer precise control over the query execution.
- Common cursor attributes include %FOUND, %NOTFOUND, %ISOPEN, and %ROWCOUNT, utilized to check the status of the cursor operations.

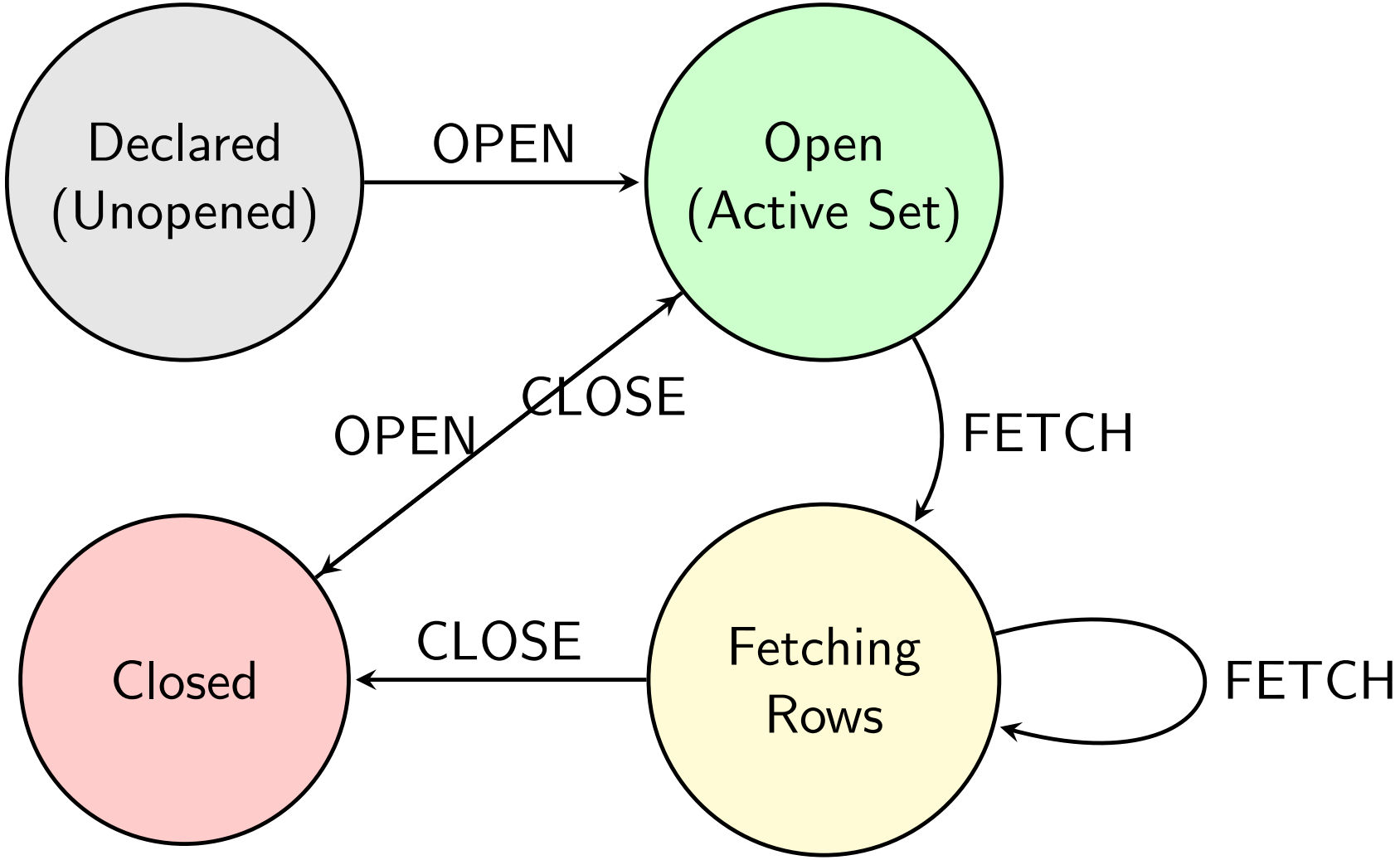
Implicit Cursors I

- Automatically created for DML statements and single-row SELECT ... INTO statements.
- The default name of the implicit cursor is 'SQL'. Attributes are referenced as SQL%ATTRIBUTE_NAME.
- SQL%ROWCOUNT: Returns the number of rows affected by the most recent DML operation.
- SQL%FOUND: Evaluates to TRUE if the most recent DML statement affected one or more rows.
- SQL%NOTFOUND: Evaluates to TRUE if the DML statement affected zero rows.
- SQL%ISOPEN: Always evaluates to FALSE because Oracle closes the implicit cursor immediately after execution.

Explicit Cursors: Lifecycle I

- 1. DECLARE: Defines the cursor with a name and the associated SELECT statement in the declarative section. Syntax: CURSOR cursor_name IS select_statement;
- 2. OPEN: Allocates memory for the context area, parses the statement, binds variables, and executes the query to identify the active set. Syntax: OPEN cursor_name;
- 3. FETCH: Retrieves the current row from the active set into variables and advances the cursor pointer to the next row. Syntax: FETCH cursor_name INTO variable_list;
- 4. CLOSE: Disables the cursor, releases the context area memory, and undefines the active set. Syntax: CLOSE cursor_name;

State Transitions of an Explicit Cursor I



Explicit Cursor: PL/SQL Example I

- DECLARE
- CURSOR emp_cursor IS SELECT employee_id, last_name FROM employees WHERE department_id = 30;
- v_emp_id employees.employee_id%TYPE;
- v_lname employees.last_name%TYPE;
- BEGIN
- OPEN emp_cursor; – Memory allocated, query executed
- LOOP
- FETCH emp_cursor INTO v_emp_id, v_lname; – Retrieve row
- EXIT WHEN emp_cursor%NOTFOUND; – Check if fetch was successful
- DBMS_OUTPUT.PUT_LINE('ID: ' —— v_emp_id —— ', Name: ' —— v_lname);

Explicit Cursor: PL/SQL Example II

- END LOOP;
- CLOSE emp_cursor; – Release memory
- END;

Best Practices for Cursors I

- Always explicitly CLOSE cursors to prevent memory leaks and 'maximum open cursors exceeded' (ORA-01000) errors.
- Use Cursor FOR Loops when iterating through all rows; they automatically handle declaring, opening, fetching, exiting, and closing without explicit commands.
- Parameterize explicit cursors to make them reusable across different inputs, minimizing hard parsing.
- Prefer set-based SQL operations (e.g., INSERT ... SELECT) over row-by-row cursor processing whenever possible to minimize context switching overhead.

Agenda I

- Introduction to Stored Procedures
- Creating and Executing Procedures
- Parameter Modes: IN, OUT, IN OUT
- Procedure Execution Flow
- Introduction to Stored Functions
- Creating and Executing Functions
- Procedures vs. Functions

Introduction to Stored Procedures I

- A Stored Procedure is a named PL/SQL block that performs one or more specific tasks.
- It is compiled and stored in the database schema, allowing for reuse and sharing among multiple applications.
- Advantages:
- - Improved Performance: Parsed and compiled once, reducing execution overhead.
- - Reduced Network Traffic: Instead of sending multiple SQL statements, a single call executes a batch of commands.
- - Security: Users can be granted execution privileges without needing direct access to underlying tables.
- - Modularity: Encapsulates complex business logic into manageable units.

Creating a Stored Procedure I

- Syntax: CREATE [OR REPLACE] PROCEDURE procedure_name [(parameter [,parameter])] IS [declaration_section] BEGIN executable_section [EXCEPTION exception_section] END [procedure_name];
- Example: Creating a procedure to update employee salary.
- CREATE OR REPLACE PROCEDURE update_salary (p_emp_id IN NUMBER, p_increment IN NUMBER) IS
- v_current_salary NUMBER;
- BEGIN
- SELECT salary INTO v_current_salary FROM employees WHERE employee_id = p_emp_id;
- UPDATE employees SET salary = salary + p_increment WHERE employee_id = p_emp_id;
- COMMIT;

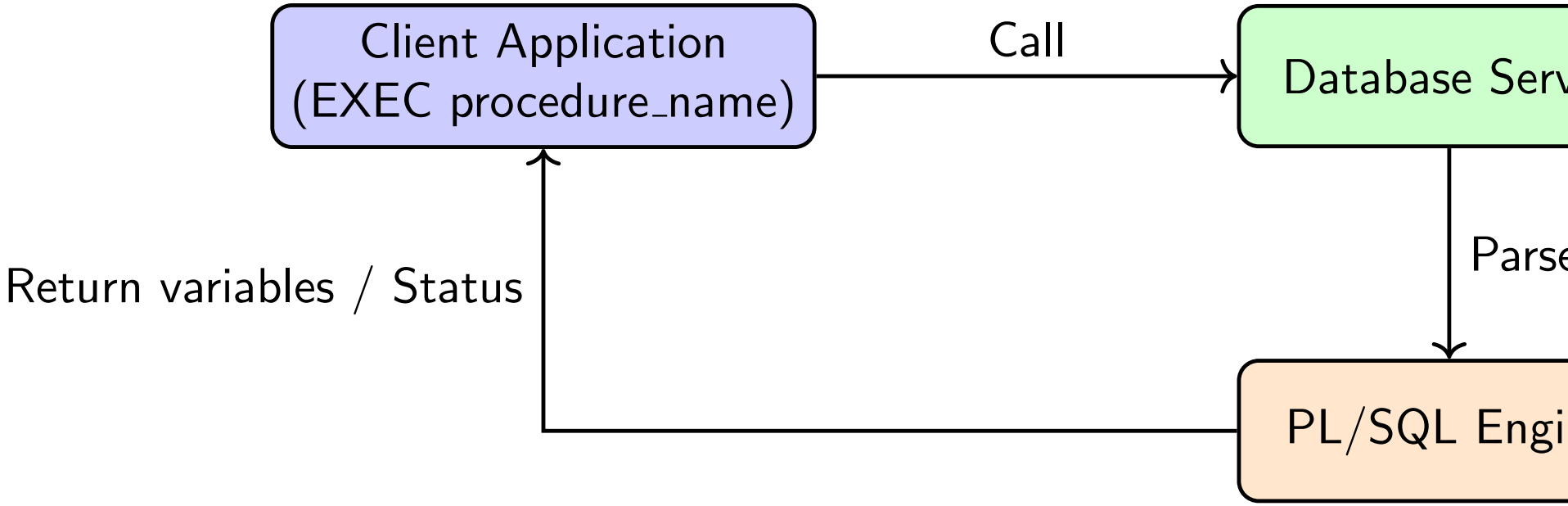
Creating a Stored Procedure II

- EXCEPTION
- WHEN NO_DATA_FOUND THEN
- DBMS_OUTPUT.PUT_LINE('Employee ID not found.');
- END update_salary;

Executing Stored Procedures & Parameter Modes I

- Stored procedures can be executed from an anonymous PL/SQL block, another procedure, or via the EXECUTE (EXEC) command.
- Execution Example: EXECUTE update_salary(101, 5000);
- Alternatively inside a block:
- BEGIN
- update_salary(102, 3000);
- END;
- Parameter Modes:
- - IN (Default): Passes a value to the subprogram. Cannot be modified inside the procedure.
- - OUT: Returns a value to the caller. Must be assigned a value inside the procedure.
- - IN OUT: Passes an initial value to the subprogram and returns an updated value to the caller.

Procedure Execution Flow I



Introduction to Stored Functions I

- A Stored Function is a named PL/SQL block that MUST return a single value.
- Functions are typically used to compute and return a value, unlike procedures which are used to perform an action.
- They can be called from within SQL statements (e.g., SELECT, WHERE, HAVING clauses), provided they do not violate database state rules (pragma restrictions).
- Advantages:
- - Simplifies complex calculations inside SQL queries.
- - Increases code reusability and readability.
- - Provides seamless integration between SQL and PL/SQL.

Creating and Executing Functions I

- Syntax: CREATE [OR REPLACE] FUNCTION function_name [(parameter [,parameter])] RETURN return_datatype IS [declaration_section] BEGIN executable_section RETURN return_value; [EXCEPTION exception_section] END [function_name];
- Example: Function to calculate annual bonus.
- CREATE OR REPLACE FUNCTION get_annual_bonus (p_emp_id IN NUMBER) RETURN NUMBER IS
- v_salary NUMBER;
- v_bonus NUMBER;
- BEGIN
- SELECT salary INTO v_salary FROM employees WHERE employee_id = p_emp_id;
- v_bonus := v_salary * 0.15;

Creating and Executing Functions II

- RETURN v_bonus;
- EXCEPTION
- WHEN NO_DATA_FOUND THEN RETURN 0;
- END get_annual_bonus;
- Execution inside SQL: SELECT employee_id, get_annual_bonus(employee_id) AS bonus FROM employees;

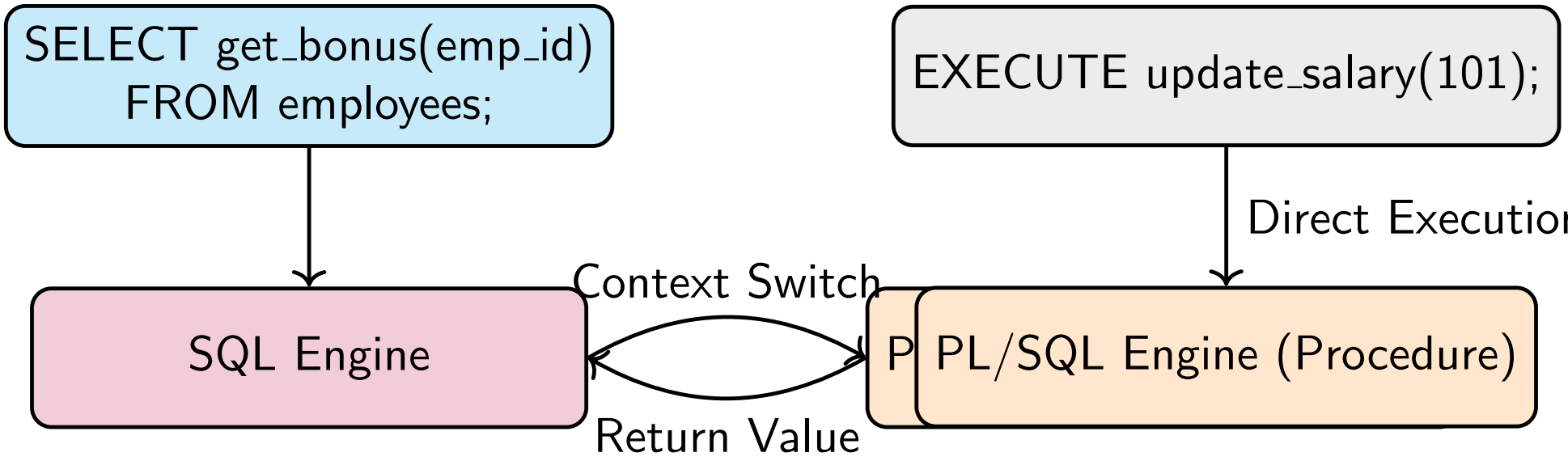
Procedures vs. Functions I

- 1. Return Value:
- - Procedure: Does not return a value via the RETURN statement (uses OUT parameters).
- - Function: MUST return a single value using the RETURN statement.
- 2. Invocation:
- - Procedure: Invoked as a standalone statement (EXECUTE or PL/SQL block).
- - Function: Invoked as part of an expression or directly within SQL statements.
- 3. Usage Intent:
- - Procedure: Designed to perform a specific action or business process (DML operations).

Procedures vs. Functions II

- - Function: Designed to compute and return a derived value without altering database state.
- 4. DML Operations:
- - Procedure: Can freely execute INSERT, UPDATE, DELETE.
- - Function: Cannot execute DML if called from a SQL SELECT query (mutating table issues).

Invoking Functions in SQL vs Procedures I



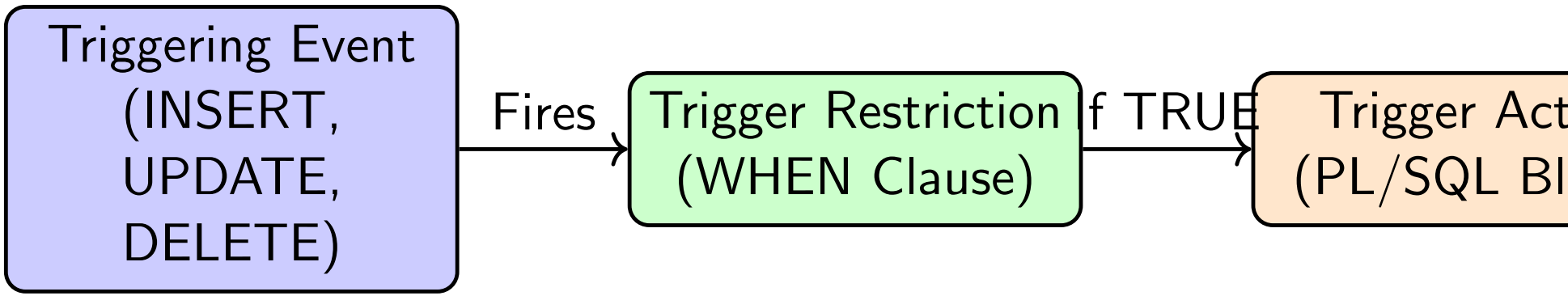
Agenda I

- Fundamentals of Triggers
- Types of Triggers: Timing
- Types of Triggers: Level
- Trigger Execution Hierarchy
- Trigger Examples and Best Practices

Fundamentals of Triggers I

- A trigger is a specialized stored PL/SQL program unit associated with a specific database table, view, or schema.
- Unlike standard procedures or functions, triggers execute (fire) implicitly and automatically whenever a predefined triggering event occurs (e.g., INSERT, UPDATE, or DELETE).
- Primary uses of Triggers:
- 1. Enforcing complex business rules and referential integrity that cannot be handled by standard declarative constraints.
- 2. Automatically generating derived column values (like timestamps or audit user tracking).
- 3. Auditing and logging database modifications to a separate tracking table.
- 4. Implementing synchronous replication of tables across systems.

Anatomy of a Trigger I



Types of Triggers: BEFORE vs AFTER I

- Timing defines WHEN the trigger fires relative to the triggering DML statement.
- BEFORE Triggers:
- - Execute immediately before the triggering DML statement modifies the data.
- - Allow the trigger body to inspect and potentially alter the newly submitted data.
- - Primarily used to validate data, calculate derived values, or conditionally prevent the DML operation by raising an application exception.
- AFTER Triggers:
- - Execute after the DML statement completes and the database block has been updated.

Types of Triggers: BEFORE vs AFTER II

- - Typically used for auditing, logging, or cascading updates to other dependent tables where knowing the final state of the updated data is required.

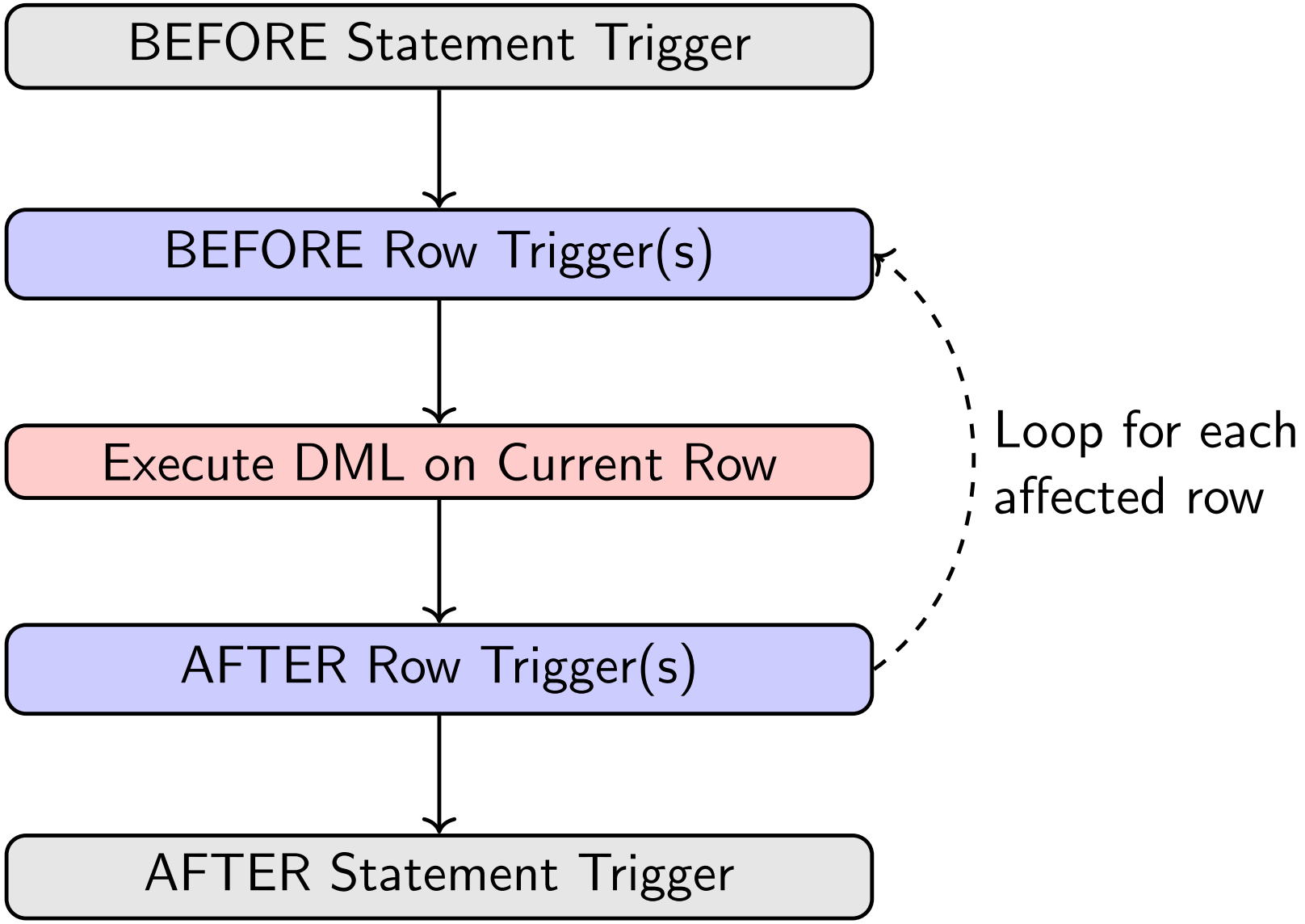
Types of Triggers: FOR EACH ROW vs STATEMENT I

- Level defines HOW MANY TIMES the trigger action executes for a given DML statement.
- STATEMENT-Level Triggers (Default):
- - Fire exactly once for the triggering event, regardless of how many rows are affected by the statement.
- - Even if a DELETE statement removes zero rows, a statement-level trigger still fires once.
- - Best for tasks that do not depend on specific data row changes, such as general security checks (e.g., time-of-day restrictions).
- ROW-Level Triggers (FOR EACH ROW):
- - Fire once for every individual row affected by the triggering DML statement.

Types of Triggers: FOR EACH ROW vs STATEMENT II

- - Defined explicitly using the 'FOR EACH ROW' clause.
- - Allow access to the pre-modification and post-modification data of the specific row using the pseudorecords :OLD and :NEW.

Trigger Execution Hierarchy I



Example: FOR EACH ROW Trigger I

- Scenario: Automatically track the last modification timestamp and user for an employee record update.
- CREATE OR REPLACE TRIGGER trg_emp_audit
- BEFORE INSERT OR UPDATE ON employees
- FOR EACH ROW
- BEGIN
- :NEW.last_modified_date := SYSDATE;
- :NEW.last_modified_by := USER;
- END;
- Explanation:
- - Because this is a BEFORE ROW trigger, the :NEW pseudorecord allows modifying the inbound column values prior to database write.
- - This logic would fail if written as an AFTER trigger, as :NEW values cannot be modified once the row is processed.

Example: FOR EACH STATEMENT Trigger I

- Scenario: Prevent any modifications to the salary table during non-business hours (weekends).
- CREATE OR REPLACE TRIGGER trg_secure_salary
- BEFORE INSERT OR UPDATE OR DELETE ON salaries
- BEGIN
- IF (TO_CHAR(SYSDATE, 'DY') IN ('SAT', 'SUN')) THEN
- RAISE_APPLICATION_ERROR(-20001, 'Salary changes are not allowed on weekends.');
- END IF;
- END;
- Explanation:
- - This is a statement-level trigger because the 'FOR EACH ROW' clause is omitted.

Example: FOR EACH STATEMENT Trigger II

- - It evaluates the condition exactly once per DML statement execution.
- - It is highly efficient for general security checks since it prevents the entire DML statement from proceeding before any rows are even touched.

Best Practices and the Mutating Table Error I

- 1. Minimize Trigger Code: Keep PL/SQL trigger bodies lightweight. Move complex logic to stored procedures and invoke them from the trigger.
- 2. Avoid Trigger Cascading: A trigger on Table A updating Table B, triggering an update on Table C, leads to unmanageable code and hidden performance bottlenecks.
- 3. Beware of Mutating Table Errors (ORA-04091):
- - Occurs when a row-level trigger attempts to query or modify the exact same table that is currently undergoing the DML operation.
- - The RDBMS flags the table state as inconsistent during the DML.

Best Practices and the Mutating Table Error II

- - Solution: Use Compound Triggers (Oracle 11g+) or a combination of statement and row-level triggers with PL/SQL collections to capture row data and defer table queries until the AFTER STATEMENT phase.

Agenda I

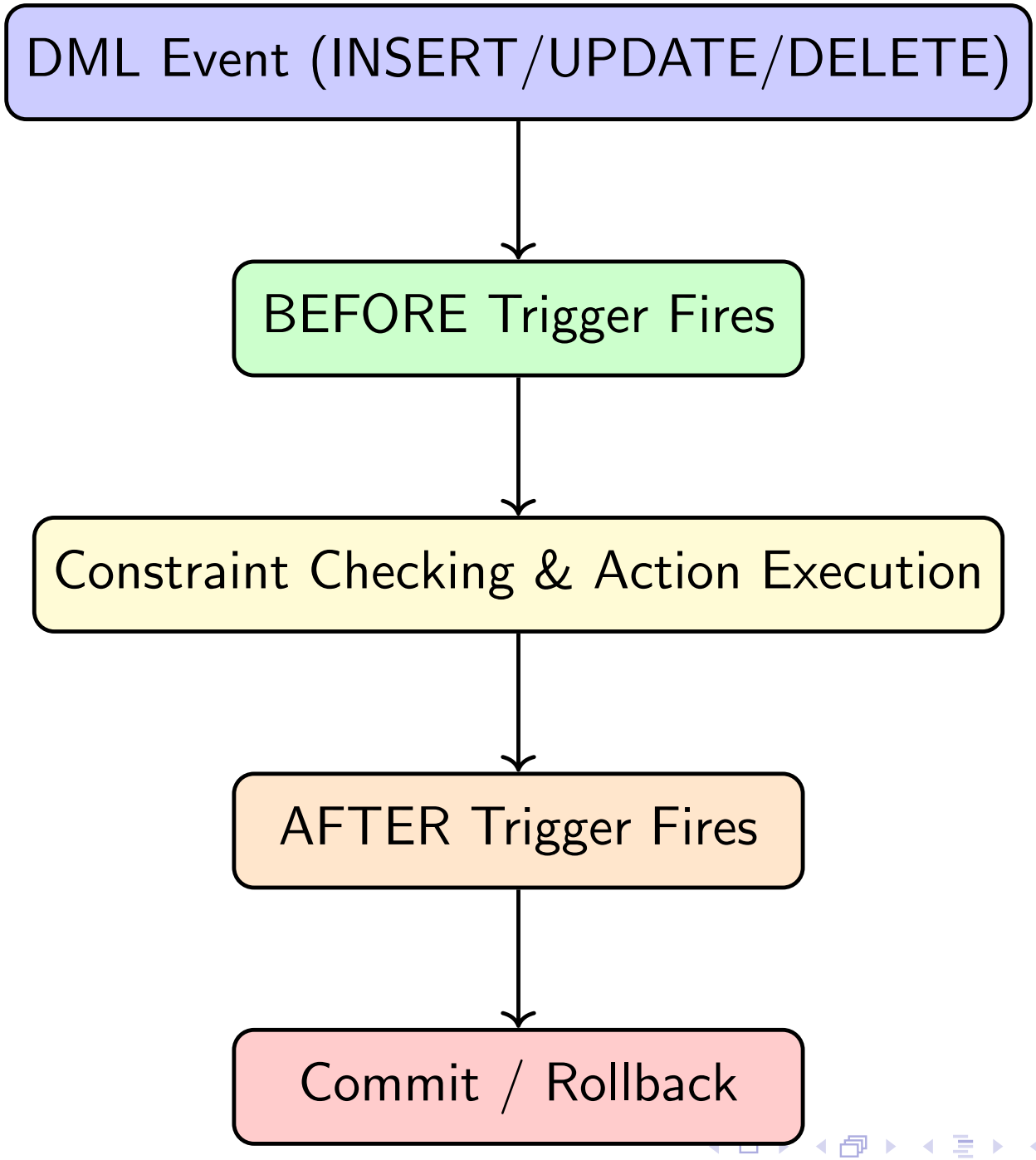
- 1. Fundamentals of Triggers
- 2. Trigger Execution Architecture
- 3. Types of Triggers: BEFORE vs AFTER
- 4. Statement-Level vs Row-Level Triggers
- 5. Syntax and PL/SQL Implementation
- 6. Practical Row-Level Auditing Example
- 7. Summary

Fundamentals of Triggers I

- A trigger is a stored subprogram in a database that automatically executes (fires) when a specified event occurs.
- Events can be DML operations (INSERT, UPDATE, DELETE), DDL operations (CREATE, ALTER, DROP), or database operations (LOGON, STARTUP).
- Unlike standard stored procedures, triggers cannot be invoked directly; they are entirely event-driven.
- Use Cases: Enforcing complex business rules, maintaining audit trails, preventing invalid transactions, and automatically generating derived column values.

Trigger Execution Flow I

Trigger Execution Flow II



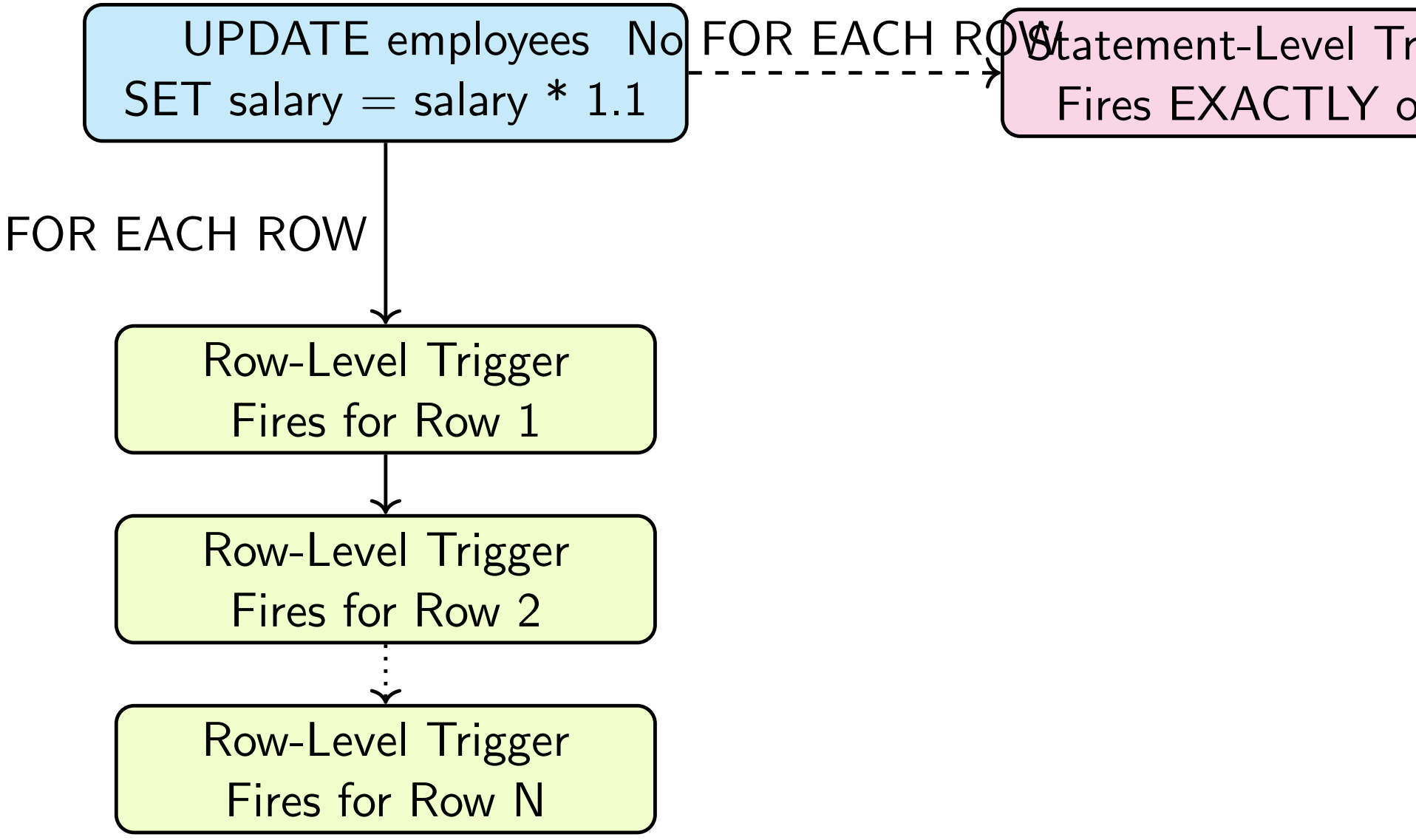
Types of Triggers: BEFORE I

- BEFORE triggers execute before the triggering DML statement operates on the target table.
- They are frequently used to validate or modify incoming data before it is permanently written to the database.
- Example use case: Checking if a newly inserted salary is within an acceptable range, or generating a primary key sequence from an autonomous transaction.
- In a BEFORE row-level trigger, you can modify the :NEW pseudo-record values directly, e.g., :NEW.salary := :NEW.salary * 1.1.

Types of Triggers: AFTER I

- AFTER triggers execute after the DML statement has completed its operation and all constraints have been checked.
- They are typically used to perform actions that depend on the successful completion of the triggering statement, such as logging or cross-table synchronization.
- Unlike BEFORE triggers, you CANNOT modify the :NEW pseudo-record in an AFTER trigger because the data has already been written and committed into the active transaction.
- Example use case: Inserting a new record into an audit_log table detailing the previous and newly inserted state of a critical row.

FOR EACH ROW vs FOR EACH STATEMENT I



FOR EACH ROW vs STATEMENT Details I

- Statement-Level Triggers (Default): Fire once per DML statement, regardless of how many rows are affected. Even if 0 rows are affected by an UPDATE, the statement trigger still fires exactly once.
- Row-Level Triggers (FOR EACH ROW): Fire once for every single row affected by the DML statement. If a query updates 100 rows, the trigger executes 100 times.
- Pseudo-records: Row-level triggers have access to :OLD (data prior to DML) and :NEW (data post DML) pseudo-records. Statement-level triggers DO NOT.
- Best Practice: Use statement-level triggers to enforce table-level security (e.g., preventing any updates during weekends) and row-level triggers for targeted data auditing.

Example: Row-Level Auditing Trigger I

- CREATE OR REPLACE TRIGGER trg_audit_salary
- AFTER UPDATE OF salary ON employees
- FOR EACH ROW
- BEGIN
- INSERT INTO salary_audit (emp_id, old_sal, new_sal, changed_date)
- VALUES (:OLD.employee_id, :OLD.salary, :NEW.salary, SYSDATE);
- END;

Summary I

- Triggers are implicit stored subprograms fired automatically by specific database events.
- BEFORE triggers validate or manipulate data prior to persistence; AFTER triggers handle post-transaction logging and cascading logic.
- Statement-level triggers fire once per query, whereas row-level triggers execute per affected tuple in the target table.
- Access to :OLD and :NEW values is restricted strictly to row-level triggers.

Agenda I

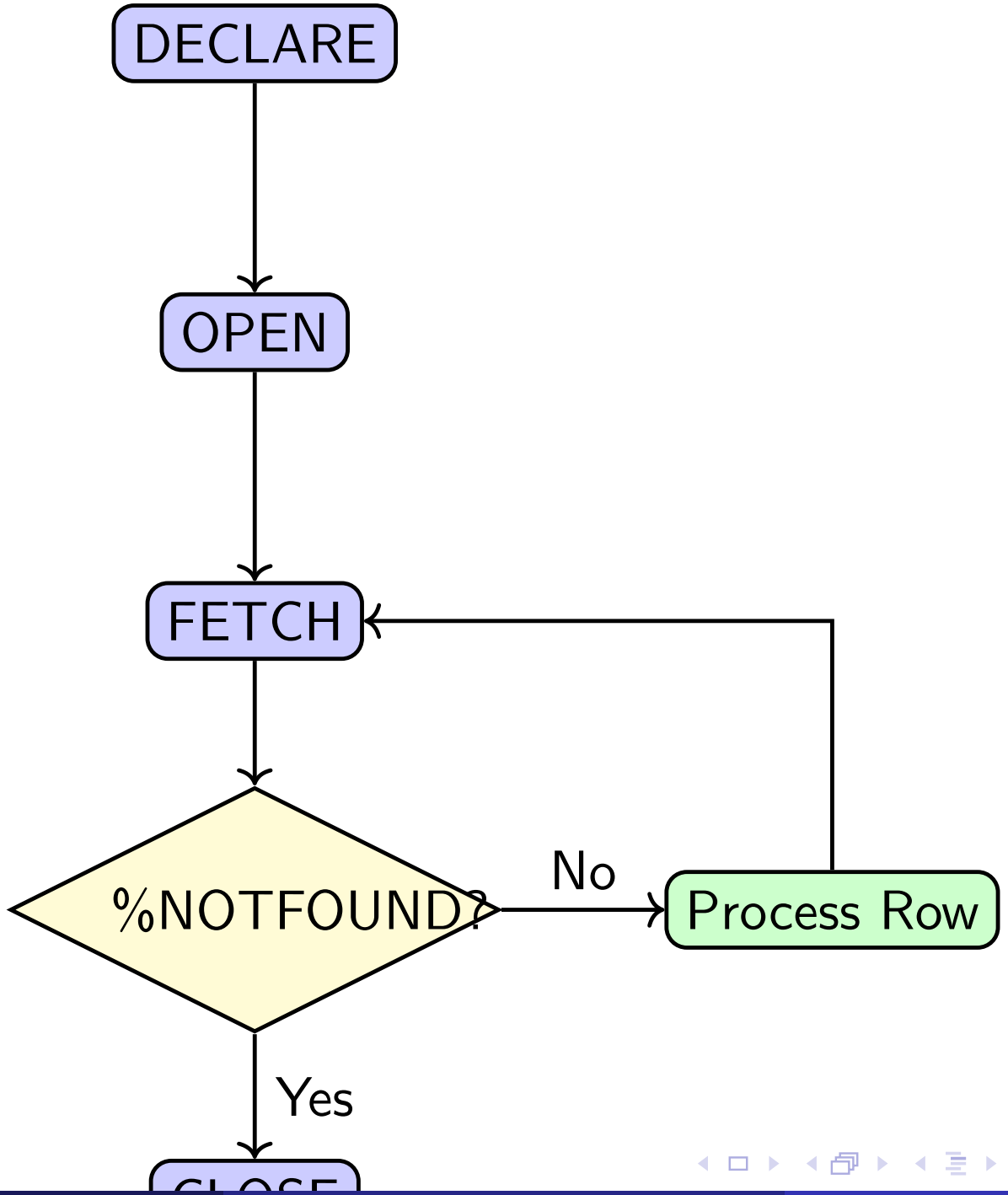
1. Problem Solving: Explicit Cursors and Loops
2. Understanding Cursor Execution (Diagram)
3. Problem Solving: Audit Logging Triggers
4. Trigger Execution Architecture (Diagram)
5. Problem Solving: Exception Handling
6. Complex Stored Procedures
7. The Mutating Table Problem
8. Summary & Best Practices

Problem 1: Cursor with Loop I

- Problem: Write a PL/SQL block to give a 10% salary increase to all employees in department 50.
- DECLARE
- CURSOR emp_cursor IS SELECT employee_id, salary FROM employees WHERE department_id = 50 FOR UPDATE OF salary;
- BEGIN
- FOR emp_record IN emp_cursor LOOP
- UPDATE employees SET salary = salary * 1.10 WHERE CURRENT OF emp_cursor;
- END LOOP;
- COMMIT;
- END;

Diagram: Explicit Cursor Lifecycle I

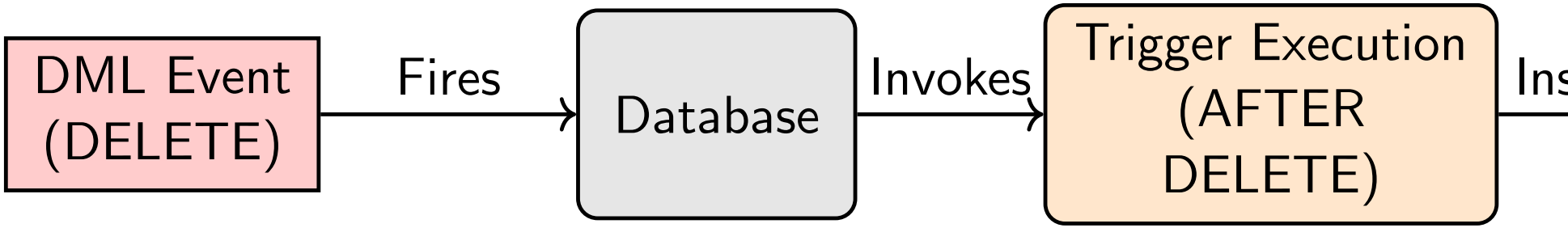
Diagram: Explicit Cursor Lifecycle II



Problem 2: Audit Logging Trigger I

- Problem: Create a trigger that logs every deletion from the ORDERS table into an ORDERS_AUDIT table.
- CREATE OR REPLACE TRIGGER audit_order_delete
- AFTER DELETE ON orders
- FOR EACH ROW
- BEGIN
- INSERT INTO orders_audit (order_id, deleted_by, deleted_date)
- VALUES (:OLD.order_id, USER, SYSDATE);
- END;

Diagram: Trigger Execution Architecture I



Problem 3: Exception Handling in PL/SQL I

- Problem: Handle exceptions appropriately when fetching a single employee record.
- DECLARE
- v_emp_name employees.last_name%TYPE;
- BEGIN
- SELECT last_name INTO v_emp_name FROM employees WHERE employee_id = 9999;
- EXCEPTION
- WHEN NO_DATA_FOUND THEN
- DBMS_OUTPUT.PUT_LINE('Error: Employee not found.');
- WHEN TOO_MANY_ROWS THEN
- DBMS_OUTPUT.PUT_LINE('Error: Multiple employees found.');

Problem 3: Exception Handling in PL/SQL II

- WHEN OTHERS THEN
- DBMS_OUTPUT.PUT_LINE('Unexpected error: ' —— SQLERRM);
- END;

Problem 4: Complex Stored Procedure I

- Problem: Create a procedure to transfer funds between two accounts, ensuring transaction integrity via explicit ROLLBACK.
- CREATE OR REPLACE PROCEDURE transfer_funds (p_from_acc INT, p_to_acc INT, p_amount NUMBER) AS
- BEGIN
- UPDATE accounts SET balance = balance - p_amount WHERE account_id = p_from_acc;
- IF SQL%ROWCOUNT = 0 THEN RAISE_APPLICATION_ERROR(-20001, 'Source account not found'); END IF;
- UPDATE accounts SET balance = balance + p_amount WHERE account_id = p_to_acc;

Problem 4: Complex Stored Procedure II

- IF SQL%ROWCOUNT = 0 THEN RAISE_APPLICATION_ERROR(-20002, 'Destination account not found'); END IF;
- COMMIT;
- EXCEPTION
- WHEN OTHERS THEN
- ROLLBACK;
- RAISE;
- END;

Problem 5: Mutating Table Exception I

- Concept: A mutating table exception (ORA-04091) occurs when a row-level trigger attempts to query or modify the same table that fired the trigger.
- This prevents a trigger from seeing an inconsistent state of the table.
- Solution Approaches:
- 1. PRAGMA AUTONOMOUS_TRANSACTION (use carefully, as it commits independently).
- 2. Compound Triggers (Oracle 11g+): Allows sharing state between statement and row-level sections to avoid mutating table issues.
- 3. Package State: Store row identifiers in a BEFORE statement trigger, populate them in a row trigger, and process them in an AFTER statement trigger.

Summary and Best Practices I

- Best Practice 1: Always use %TYPE and %ROWTYPE for robust and maintainable variable declarations.
- Best Practice 2: Prefer implicit cursors (FOR loop) over explicit OPEN/FETCH/CLOSE for cleaner and often faster code.
- Best Practice 3: Keep triggers small and highly efficient. Complex logic inside row-level triggers can cause massive performance bottlenecks on bulk DML operations.
- Best Practice 4: Handle expected exceptions explicitly. Reserve WHEN OTHERS for truly unexpected errors and always re-raise or log them properly.
- Best Practice 5: Test transaction boundaries strictly, ensuring ROLLBACKs occur in exception blocks for procedures that modify data.