SURESHOT TOPICS DBMD

IMPORTANT/PYQ-TAGGED TOPICS & CONCEPTS (DBMD UNITS 1-4) - EXTENDED DETAIL (WITH ADVANCED DML)

UNIT-I: CONCEPTUAL DATA MODELLING

1. **Overview of Database Systems Architecture and Components [PYQ Q1a, Q2a]**
    - **ANSI/SPARC Three-Schema Architecture [PYQ - Important for data independence]:**
        - **Purpose:** To achieve program-data independence (insulating applications from changes in data storage or logical structure).
        - **Levels:**
            - **External Schema (User Views/Subschemas):** An individual user's or application's view of the relevant portions of the database. Multiple external schemas can exist for the same database.
            - **Conceptual Schema (Global View):** A community view representing the entire database structure (entities, relationships, constraints) independent of physical storage. It hides the details of physical storage structures.
            - **Internal Schema (Physical View):** Describes the physical storage structures, access paths (e.g., indexes, hashing), and file organization used by the database. It is technology-dependent.
        - **Data Independence:**
            - **Logical Data Independence:** The immunity of external schemas (application programs) to changes in the conceptual schema. Changes to the conceptual schema (e.g., adding an attribute) should not require changes to existing external schemas or application programs that do not use the new data.
            - **Physical Data Independence:** The immunity of the conceptual schema (and consequently external schemas) to changes in the internal schema. Changes to the physical storage (e.g., changing file organization, adding an index) should not require changes to the conceptual or external schemas.
    - **Database System vs. DBMS [PYQ Q1a]:**
        - **Database:** A self-describing collection of interrelated data, which includes both the data itself and metadata (data about data).
            - **Types:** Single-user (desktop), Multi-user (workgroup, enterprise), Distributed Database (DDB), Data Warehouse.
        - **Database Management System (DBMS):** General-purpose software that facilitates the processes of defining, constructing, manipulating, and sharing databases among various

users and applications.

- **DBMS Components [PYQ Q1a]:**
  - **Query Languages (e.g., SQL):** Allow users to retrieve and manipulate data.
  - **Report Generators:** Tools to produce formatted reports from database data.
  - **Security, Integrity, Backup & Recovery facilities:** Mechanisms to protect data, ensure consistency, and recover from failures.
  - **Data Definition Language (DDL):** Used to define the database structure and schema (e.g., `CREATE TABLE`, `ALTER TABLE`).
  - **Data Manipulation Language (DML):** Used to access and manipulate data (e.g., `INSERT`, `SELECT`, `UPDATE`, `DELETE`).
  - **Data Control Language (DCL):** Used to manage permissions and access rights (e.g., `GRANT`, `REVOKE`).
  - **Data Dictionary/Repository:** A system catalog that stores metadata, including definitions of data elements, schemas, constraints, and access authorizations.
- **Limitations of File-Processing Systems [PYQ - Implied in understanding DB advantages]:**
  - **Lack of data integrity:** Data can be inconsistent, incorrect, or incomplete due to uncontrolled redundancy and lack of constraints.
  - **Lack of standards:** Difficulty in enforcing organization-wide naming conventions, data access protocols, and data protection standards.
  - **Lack of flexibility/maintainability:** Changes to data structures often require significant programming effort across multiple applications.
  - **Root Causes:**
    - **Lack of data integration:** Data is often separated and isolated in different files for different applications.
    - **Lack of program-data independence:** The structure of data files is embedded within application programs, making changes difficult.

2. **Database Design Life Cycle [PYQ Q2b]:**
   - **1. Requirements Specification:** Analysts review existing documents, conduct user interviews, and gather information to identify objectives, data requirements, and process specifications (business rules).
   - **2. Conceptual Data Modeling (Technology-Independent):** Describes the data structure of the database without considering physical storage details. Focuses on capturing user-specified business rules and semantics. The primary product is a Conceptual Schema (e.g., ER/EER model).
     - **Includes Presentation Layer:** For communication with users (often a simplified ER diagram).

        - **Includes Design-Specific Layer:** A more detailed conceptual model for database designers.
        - **Validation:** Crucial step to ensure the model accurately reflects user requirements.
    - **3. Logical Data Modeling (Technology-Dependent):** Transforms the conceptual schema into a schema compatible with a chosen data model (e.g., relational, network, hierarchical). Product: Logical Schema. Normalization is typically performed during this phase to reduce redundancy and improve data integrity.
    - **4. Physical Data Modeling (DBMS-Specific):** Specifies the internal storage structures, access strategies (e.g., indexes, file organization), and other physical implementation details using the tools and features of a particular DBMS.
3. **Conceptual Data Modelling**
    - **ER Modeling [PYQ Q3b, Q4a]:** A graphical technique for representing the entities, attributes, and relationships within a real-world domain.
        - **Unique Identifiers (Keys) [Related to PYQ Q1b]:** Attribute(s) whose values uniquely identify each entity instance within an entity type. These are typically underlined in ER diagrams.
            - **Superkey:** Any set of attributes that, taken collectively, uniquely identifies an entity instance. May contain redundant attributes.
            - **Candidate Key:** A minimal superkey; no proper subset of a candidate key is a superkey. An entity type can have multiple candidate keys.
            - **Primary Key:** The candidate key selected by the database designer to be the main unique identifier for an entity type.
            - **Key attribute:** An attribute that is part of any candidate key.
    - **EER Modeling (Enhanced Entity-Relationship) [PYQ Q3b, Q4a]:** Extends the basic ER model with additional constructs to represent more complex relationships and data semantics.
        - **Superclass/Subclass (SC/sc) Relationship (Is-A Relationship):**
            - **Superclass (SC):** A generic entity type that has one or more distinct subgroups (subclasses).
            - **Subclass (sc):** A specialized entity type that is a subgroup of a superclass. It inherits attributes and relationships from the superclass (Type Inheritance) and can also have its own specific attributes/relationships.
            - **Cardinality:** Always 1:1 between an SC instance and its corresponding sc instance.
            - **Participation:** Participation of a subclass in an SC/sc relationship is always total.
        - **Specialization and Generalization [PYQ Q3b]:** Two perspectives for defining SC/sc relationships.
            - **Specialization:** A top-down process of identifying distinct subgroups (subclasses) within a superclass based on distinguishing characteristics.

        - **Generalization:** A bottom-up process of identifying common features among several entity types and forming a more general superclass.
        - **Constraints on Specialization/Generalization:**
            - **Disjointness:** Specifies whether subclasses are mutually exclusive.
                - **Disjoint (d):** An SC instance can be a member of at most one subclass.
                - **Overlapping (o):** An SC instance can be a member of more than one subclass.
            - **Completeness (Totalness):** Specifies whether every SC instance must belong to at least one subclass.
                - **Total (double line from SC to circle):** Every SC instance must be a member of some subclass.
                - **Partial (single line from SC to circle):** An SC instance may not belong to any subclass.
            - **Predicate-defined (condition-defined) subclass:** Subclass membership is determined by a condition on an attribute of the superclass.
            - **User-defined subclass:** Subclass membership is explicitly specified by the user or application, not based on a condition.
    - **Hierarchy and Lattice [PYQ Q3b]:**
        - **Specialization Hierarchy:** A subclass participates as a subclass in only one SC/sc relationship, resulting in a tree-like structure (single inheritance). A subclass inherits attributes from its direct parent and all its ancestors up the hierarchy.
        - **Specialization Lattice (Multiple Inheritance):** A subclass (shared subclass) can be a subclass in more than one SC/sc relationship, inheriting attributes and relationships from multiple superclasses. This results in a graph-like (lattice) structure.
    - **Categorization [PYQ Q3b]:** A subclass (called a category) that represents a collection of objects that is a subset of the UNION of two or more superclasses, where these superclasses are of *different* entity types. A category instance must be an instance of at least one of its superclasses. Selective inheritance (inherits from only one of its defining superclasses for a given instance) is typical. Indicated by 'U' in a circle. Total/Partial participation of superclasses in the category, while category participation in the relationship is total.
- **Modeling Complex Relationships (Ch 5) [PYQ Q3b]:** (EER constructs are central to this for PYQ context)
    - **Ternary Relationship Type:** A relationship involving three distinct entity types. (min, max) cardinality notation is crucial for precision. Can sometimes be conceptualized as a base entity itself.
    - **Beyond Ternary (Cluster Entity Type):** Grouping of related entity types and their relationships into a higher-level abstract entity (cluster). Useful for representing complex objects or for layered ERDs.

- **Design Issues in ER & EER Modeling [PYQ Q3b]:**
  - Choosing between representing a concept as an entity type or an attribute.
  - Choosing between binary or higher-degree (ternary, N-ary) relationships.
  - Deciding when to use weak entity types.
  - Deciding when to use EER constructs: Specialization/Generalization vs. Categorization, single vs. multiple inheritance.
  - Handling M:N relationships and multi-valued attributes (typically through decomposition into new entity types in the Design-Specific ER model).
  - **Validation of Conceptual Design (Connection Traps):** Errors in ERD structure that can lead to misinterpretation or incorrect data retrieval.
    - **Fan Trap:** Occurs when a model represents a relationship between entity types, but the pathway between certain entity occurrences is ambiguous due to multiple 1:N relationships fanning out from a central entity.
    - **Chasm Trap:** Occurs when a model suggests the existence of a relationship between entity types, but the pathway does not exist for certain entity occurrences, often due to missing information caused by optional participation in relationships along the path.
    - Resolving traps often involves restructuring the ERD, possibly by adding new relationships or re-evaluating existing ones.

UNIT-II: LOGICAL DATA MODELLING

1. **Overview of Relational Data Model**
   - **Integrity Constraints [PYQ - Key concepts are important]:** Rules ensuring data accuracy and consistency.
     - **Key Constraints [PYQ Q1b]:**
       - **Superkey:** An attribute or set of attributes that uniquely identifies a tuple within a relation.
       - **Candidate Key:** A minimal superkey (no proper subset is a superkey). A relation can have multiple.
       - **Primary Key:** The candidate key chosen to uniquely identify tuples. Underlined in relation schema.
       - **Alternate Key:** Candidate keys that are not chosen as the primary key.
     - **Entity Integrity Constraint:** No component of a primary key value can be null. This ensures that every tuple is uniquely identifiable.
     - **Referential Integrity Constraint [PYQ Q1b]:** Ensures consistency among tuples in two (or the same) relations.
       - **Foreign Key (FK):** An attribute or set of attributes in one relation (the referencing relation) whose values are required to match the values of a candidate key (usually the

primary key) of some tuple in another (or the same) relation (the referenced relation), or the FK values must be wholly null.

- **Actions on violation (when a referenced tuple is deleted or its PK is updated):**
  - `RESTRICT` / `NO ACTION`: The delete/update operation on the referenced table is rejected if there are referencing tuples.
  - `CASCADE`: The delete/update operation on the referenced tuple is cascaded to all referencing tuples (e.g., referencing tuples are also deleted/updated).
  - `SET NULL`: The foreign key values in the referencing tuples are set to NULL.
  - `SET DEFAULT`: The foreign key values in the referencing tuples are set to their predefined default value.

2. **Mapping ER Model to a Logical Schema [PYQ Q1d, Q4a]:**
   - **Mapping N-ary Relationships (Higher Degree) [PYQ Q5b]:** An N-ary relationship is mapped by creating a new relation. The primary key of this new relation is formed by the combination of the primary keys of all the participating entity types. These PKs from participating entities act as foreign keys in the new relation. Any attributes of the N-ary relationship become attributes of this new relation.
3. **Mapping EER Model to a Logical Schema [PYQ Q4a]:**
   - **Mapping Specialization/Generalization Options:**
     - **Option 1 (Multiple Relations - SC and sc's):** Create a relation for the superclass (SC) and a separate relation for each subclass (sc). The primary key of the SC relation is also the primary key of each sc relation, and it acts as a foreign key in each sc relation referencing the SC. Best for disjoint subclasses with many specific attributes.
     - **Option 2 (Single Relation - SC only):** Create one relation for the superclass (SC) that includes all attributes of the SC and all attributes of all its subclasses. Use NULL values for attributes not applicable to a particular instance. A type attribute is added to distinguish among the subclass instances. Good for overlapping subclasses or when subclasses have few specific attributes.
     - **Option 3 (Multiple Relations - sc's only):** Create a relation for each subclass (sc). Each of these relations includes all attributes of the SC (inherited) plus the specific attributes of that sc. This option is viable only if the specialization is total and disjoint. May lead to redundancy of SC attributes.
   - **Mapping Aggregation [PYQ Q5b]:** The "whole" or aggregate entity type is mapped to a relation. The primary keys of the "part" entity types (which are superclasses in the EER aggregation construct) are included as foreign keys in the relation for the aggregate entity.
4. **Normalization [PYQ Q5a]:** A systematic process of organizing data in a database to reduce data redundancy and improve data integrity by decomposing relations into smaller, well-structured relations.
   - **Anomalies (without normalization):**

        - **Insertion Anomaly:** Difficulty in inserting data for one entity without having data for another related entity (e.g., cannot add a new course unless a student is enrolled).
        - **Deletion Anomaly:** Unintended loss of data about one entity when data about another entity is deleted (e.g., deleting the last student enrolled in a course might delete the course information itself).
        - **Modification/Update Anomaly:** Need to change redundant data in multiple places, leading to inconsistencies if not all occurrences are updated correctly.
    - **Normal Forms:**
        - **1NF (First Normal Form):** All attribute values in a relation must be atomic (indivisible). No repeating groups or multi-valued attributes within a single cell. (A relation in the relational model is, by definition, in 1NF).
        - **2NF (Second Normal Form):** A relation is in 2NF if it is in 1NF and all non-key attributes are fully functionally dependent on the entire primary key. This means there are no partial dependencies (where a non-key attribute depends on only a part of a composite primary key).
        - **3NF (Third Normal Form):** A relation is in 3NF if it is in 2NF and there are no transitive dependencies. A transitive dependency occurs when a non-key attribute depends on another non-key attribute, which in turn depends on the primary key.
        - **BCNF (Boyce-Codd Normal Form):** A stricter version of 3NF. A relation is in BCNF if and only if every determinant (an attribute or set of attributes on which some other attribute is fully functionally dependent) is a superkey (or a candidate key).
        - **4NF (Fourth Normal Form):** A relation is in 4NF if it is in BCNF and it has no non-trivial multi-valued dependencies (MVDs) unless the determinant of the MVD is a superkey. An MVD X ->> Y means that the set of Y values associated with a given X value is determined by X alone, independently of any other attributes Z in the relation.
        - **5NF (Fifth Normal Form / Project-Join Normal Form - PJ/NF):** A relation is in 5NF if it is in 4NF and it has no join dependencies (JDs) that are not implied by its candidate keys. A join dependency means that the relation can be losslessly decomposed into a set of projections and then reconstructed by joining these projections.

UNIT-III: DATABASE IMPLEMENTATION AND PHYSICAL DATABASE DESIGN

1. **Database Creation using SQL [PYQ Q6a]:**
    - **Data Definition Using SQL (SQL/DDL):** Standard language for defining database structure. Major DDL constructs are `CREATE`, `ALTER`, `DROP`.
    - **Base Table Specification in SQL/DDL:**
        - `CREATE TABLE` Statement: Defines a new base table that will be physically stored. Syntax: `CREATE TABLE table_name (column_definition_1, column_definition_2, ..., [table_constraint_1, ...]);`

        - **Column Definition:** `column_name data_type [DEFAULT default_value] [column_constraint_list]`
        - **Constraints (Column-level or Table-level):**
            - `NOT NULL`: Ensures a column cannot have null values.
            - `UNIQUE`: Ensures all values in a column (or set of columns) are unique. Can define alternate keys.
            - `PRIMARY KEY`: Specifies the primary key for the table (implies `NOT NULL` and `UNIQUE`).
            - `FOREIGN KEY ... REFERENCES ...`: Defines a foreign key and the referenced table/columns.
            - `CHECK (condition)`: Specifies a condition that must be true for every row in the table.
    - **Referential Triggered Actions (Deletion/Update Rules) [PYQ - Implied in DDL]:** Specified as part of a `FOREIGN KEY` constraint definition.
        - `ON DELETE {CASCADE | SET NULL | SET DEFAULT | RESTRICT | NO ACTION}`: Defines action when a referenced row is deleted.
        - `ON UPDATE {CASCADE | SET NULL | SET DEFAULT | RESTRICT | NO ACTION}`: Defines action when a referenced key is updated.
    - `ALTER TABLE` Statement: Modifies an existing table structure.
        - `ADD [COLUMN] column_definition`
        - `DROP [COLUMN] column_name {CASCADE | RESTRICT}`
        - `ALTER [COLUMN] column_name SET DEFAULT default_value | DROP DEFAULT`
        - `ADD table_constraint_definition`
        - `DROP CONSTRAINT constraint_name {CASCADE | RESTRICT}`
    - `DROP TABLE` Statement: Deletes a table definition and all its data.
        - `DROP TABLE table_name {CASCADE | RESTRICT}`: `CASCADE` drops the table and also any views, FK constraints, etc., that reference it. `RESTRICT` prevents dropping if other objects reference it.
    - **Advanced Data Manipulation using SQL (Ch 11) [PYQ - from Unit III/IV notes context, likely related to DML in practice/queries]:**
        - **Relational Algebra Concepts (Recap):** Understanding operators like Select (σ), Project (π), Union (∪), Intersection (∩), Difference (-), Cartesian Product (×), Join (⋈ - Equijoin, Natural Join, Theta Join, Outer Joins), Divide (÷), Aggregate Functions is fundamental to complex SQL.
        - **SQL Queries Based on a Single Table:**
            - `SELECT [DISTINCT | ALL] column_list FROM table_name [WHERE condition] [GROUP BY column_list] [HAVING condition] [ORDER BY column_list [ASC|DESC]];`
            - **Selection (`WHERE`):** Filters rows based on a condition.

- **Projection (`SELECT column_list`):** Selects specified columns. `DISTINCT` removes duplicate rows from the result.
- **Expressions in `SELECT` and `WHERE`:** Using arithmetic or string operations.
- **Operators:** `BETWEEN`, `IN`, `NOT IN`, `LIKE` (pattern matching: `%` for any string, `_` for single char, `ESCAPE` character).
- **Handling Null Values (`IS NULL`, `IS NOT NULL`):** Nulls in arithmetic/comparisons yield unknown. Aggregate functions (except `COUNT(*)`) ignore nulls.
- **Aggregate Functions (`COUNT`, `SUM`, `AVG`, `MAX`, `MIN`):** Perform calculations on a set of values.
- **Grouping (`GROUP BY`):** Groups rows with the same values in specified columns for aggregate functions.
- **Filtering Groups (`HAVING`):** Filters groups created by `GROUP BY` based on a condition (similar to `WHERE` for rows).

- **SQL Queries Based on Binary Operators (Joins):**
  - **Cartesian Product (`CROSS JOIN` or comma in `FROM` clause with no join condition):** Combines every row of one table with every row of another.
  - **Inner Joins:** Return rows when there is at least one match in both tables.
    - `FROM table1 [INNER] JOIN table2 ON join_condition;` (General form).
    - `FROM table1 JOIN table2 USING (common_column_list);` (Joins on common columns with same names).
    - `FROM table1 NATURAL JOIN table2;` (Joins on all identically named columns).
  - **Self Joins:** Joining a table to itself (requires table aliases).
  - **N-way Joins:** Joining more than two tables.
  - **Outer Joins:** Include rows that do not have matching values in the other table, filling missing columns with NULLs.
    - `LEFT [OUTER] JOIN`: Keeps all rows from the left table.
    - `RIGHT [OUTER] JOIN`: Keeps all rows from the right table.
    - `FULL [OUTER] JOIN`: Keeps all rows from both tables.
  - **Set Theoretic Operators (`UNION`, `INTERSECT`, `MINUS / EXCEPT`):** Combine results of two queries. Tables must be union-compatible (same number of columns, compatible data types).
    - `UNION ALL`: Keeps duplicate rows. `UNION` removes duplicates.
- **Subqueries (Nested Queries):** Queries embedded within other SQL queries.
  - **In `WHERE` clause:**

        - **Single-row subquery:** Returns one value; used with standard comparison operators (=, <, >, etc.).
        - **Multiple-row subquery:** Returns a list of values; used with `IN`, `NOT IN`, `ANY` (`> ANY` = greater than min, `< ANY` = less than max, `= ANY` equiv. to `IN`), `ALL` (`> ALL` = greater than max, `< ALL` = less than min).
    - **Correlated Subqueries:** Inner query depends on the outer query for its execution (executed once for each row of outer query). Often uses `EXISTS`, `NOT EXISTS` (`EXISTS`: true if subquery returns at least one row).
    - **In `FROM` clause (Inline Views / Derived Tables):** Subquery result treated as a temporary table.
    - **In `SELECT` clause (Scalar Subqueries):** Subquery must return a single value.
    - **In `HAVING` clause:** Subquery used in condition for filtering groups.
  - **Views:** A virtual table based on the result-set of a stored query. Does not store data itself but presents data from one or more underlying tables.
    - `CREATE VIEW view_name [(column_list)] AS subquery [WITH [CASCADED | LOCAL] CHECK OPTION];`
    - **Purpose:** Simplify complex queries, provide security (restrict access), present data in a customized way, logical data independence.
    - **Updating Views:** Possible only for simple views (usually based on a single table, no aggregates, no `GROUP BY`, no `DISTINCT`). `WITH CHECK OPTION` ensures DML on view doesn't cause rows to disappear from view's defining condition.

2. **Database Programming [PYQ Q7a, Q7b]:**
   - **Types/Approaches to Database Programming:**
     - **API-based Database Access (e.g., JDBC, ODBC, ADO.NET) [IMP]:** Uses an Application Programming Interface (API) – a set of functions, classes, and protocols – provided as a library for a specific programming language. The application makes calls to this API to connect to the database, send SQL statements (often as strings), and process results. More portable than Embedded SQL, supports dynamic SQL easily.
   - **Cursors [PYQ Q7a]:** A mechanism to process query results row by row within an application program, bridging the gap between set-oriented SQL and row-oriented host languages (addressing the impedance mismatch).
     - **Declaration:** `DECLARE cursor_name [INSENSITIVE] [SCROLL] CURSOR FOR SELECT_statement [ORDER BY ...] [FOR READ ONLY | FOR UPDATE [OF column_list]];`
       - `INSENSITIVE`: Cursor operates on a snapshot of the data.
       - `SCROLL`: Allows fetching rows in non-sequential order.
       - `FOR READ ONLY`: Cursor cannot be used for updates/deletes.

    - `FOR UPDATE [OF column_list]`: Cursor can be used for updates/deletes, optionally restricted to specified columns.
- **Operations:**
    - `OPEN cursor_name;`: Executes the query associated with the cursor and positions the cursor before the first row.
    - `FETCH cursor_name INTO :host_variable_list;`: Retrieves the current row pointed to by the cursor into host language variables and advances the cursor to the next row.
    - `UPDATE ... WHERE CURRENT OF cursor_name;`: Modifies the row currently pointed to by the cursor (if the cursor is updatable).
    - `DELETE ... WHERE CURRENT OF cursor_name;`: Deletes the row currently pointed to by the cursor (if the cursor is updatable).
    - `CLOSE cursor_name;`: Releases the resources associated with the cursor.
- **Properties:** Scrollable (can move forward/backward), Insensitive (operates on a snapshot of data as it existed when opened), Updatable (allows modification of data through the cursor).
- **Need for Cursors in Database Programming:** SQL is set-oriented, while many host languages are record-oriented. Cursors allow host programs to process multi-row SQL query results one row at a time.
- **Types of Cursors (Based on Properties and Behavior - often DBMS-specific variations exist):**
    - **Read-Only Cursor:** Allows only fetching data; no updates/deletions permitted.
    - **Updatable Cursor:** Allows fetching, modifying (`UPDATE ... WHERE CURRENT OF`), and deleting (`DELETE ... WHERE CURRENT OF`) rows. Query defining must be simple (e.g., single table, no aggregates).
    - **Scrollable Cursor:** Allows flexible movement within the result set (`NEXT` (default), `PRIOR`, `FIRST`, `LAST`, `ABSOLUTE n`, `RELATIVE n`).
    - **Insensitive Cursor (Snapshot Cursor):** Operates on a temporary copy (snapshot) of data as it existed when opened; changes by other transactions are not visible. Provides a static view.
    - **Sensitive Cursor:** Attempts to reflect changes made to underlying data by other transactions after the cursor was opened. Degree of sensitivity varies.
    - **Keyset-Driven Cursor:** (A type of sensitive cursor) Set of keys for qualifying rows is fixed at open. Changes to non-key values of rows in keyset are visible. Deleted rows appear as "holes." New rows inserted by others are typically not seen.
    - **Forward-Only Cursor (Non-Scrollable):** Default type; rows can only be fetched sequentially from the first to the last. No backward movement.
    - **Static Cursor:** Similar to an insensitive cursor; operates on a snapshot.
    - **Dynamic Cursor:** The most "sensitive" type; all committed changes (inserts, updates, deletes) made by others are visible. Order, membership, and values of rows can change.

- **Triggers [PYQ Q7b]:** Procedural code automatically executed by the DBMS in response to certain DML events (INSERT, UPDATE, DELETE) on a specified table.
  - **ECA Model (Event-Condition-Action):**
    - **Event:** The DML operation (e.g., `INSERT` on `ORDERS` table) that fires the trigger.
    - **Condition:** (Optional) A Boolean expression; the trigger fires only if this condition is true.
    - **Action:** The PL/SQL block or procedure to be executed when the trigger fires and the condition (if any) is met.
  - **Syntax:** `CREATE [OR REPLACE] TRIGGER trigger_name {BEFORE | AFTER} {INSERT | DELETE | UPDATE [OF column_list]} ON table_name [FOR EACH ROW] [WHEN (condition)] BEGIN ... END;`
    - `BEFORE | AFTER`: Specifies whether the trigger action executes before or after the DML event.
    - `INSERT | DELETE | UPDATE [OF column_list]`: Specifies the DML event(s). `UPDATE OF` for specific columns.
    - `FOR EACH ROW`: Indicates a row-level trigger.
    - `WHEN (condition)`: Specifies the optional condition for the trigger to fire.
  - **Row-level trigger (`FOR EACH ROW`):** Fires once for each row affected by the DML statement. Can access `:OLD` (values before DML) and `:NEW` (values after DML for INSERT/UPDATE) row values.
  - **Statement-level trigger (default if `FOR EACH ROW` omitted):** Fires once per DML statement, regardless of how many rows are affected. Cannot access `:OLD` or `:NEW`.
  - **Uses:** Enforcing complex integrity constraints beyond standard SQL constraints, auditing data changes, maintaining derived data automatically, logging events, implementing complex business rules.
  - **More Types of Trigger, additional perspectives or specialized:**
    - **INSTEAD OF Triggers:** Used primarily with views, especially complex views that are not inherently updatable (e.g., views involving joins, aggregates, `GROUP BY`, `DISTINCT`). When a DML operation is attempted on the view, the `INSTEAD OF` trigger fires *instead* of the DML operation on the view. The trigger's code then defines how to translate this operation into appropriate DML operations on the underlying base tables.
    - **DDL Triggers:** Fire in response to DDL events (e.g., `CREATE TABLE`, `ALTER TABLE`, `DROP TABLE`). Used for auditing schema changes, enforcing naming conventions. Not standard SQL, DBMS-specific.
    - **Logon/Logoff Triggers (System-Level Triggers):** Fire when a user session connects to or disconnects from the database. Used for auditing sessions, setting session parameters. DBMS-specific.
    - **Database Event Triggers (Server-Level Triggers):** Fire in response to database-level events like startup, shutdown, or specific errors at server level. Used for admin tasks,

error logging. DBMS-specific.

- **Compound Triggers (Oracle specific):** Allows defining actions for multiple timing points (BEFORE STATEMENT, BEFORE EACH ROW, AFTER EACH ROW, AFTER STATEMENT) for a single DML event on a table within a single trigger body. Simplifies logic and variable sharing.

UNIT-IV: DATABASE TUNING, MAINTENANCE, AND SECURITY

1. **Database Tuning and Maintenance**
   - **Introduction to Database Tuning [PYQ Q9c]:** The process of optimizing database performance to meet user requirements and service level agreements. It is an iterative process involving:
     - **Monitoring:** Observing database performance metrics.
     - **Identifying Bottlenecks:** Finding areas of poor performance.
     - **Diagnosing:** Determining the cause of bottlenecks.
     - **Implementing Changes:** Adjusting aspects like physical design, conceptual schema, queries, application code, DBMS parameters, hardware.
     - **Measuring:** Assessing the impact of changes.
     - **Workload Analysis:** Understanding the types and frequencies of queries and updates, performance goals for each type, identifying accessed relations, attributes, selection/join conditions, and their selectivity.
   - **Clustering and Indexing [PYQ Q8a, Q9a]:**
     - **Indexing:** Databases store vast data; without indexing, retrieving specific data requires a full table scan (reading every row), which is slow for large tables. Indexing provides a shortcut. An index is a separate data structure (usually stored on disk) that:
       1. Is built on one or more columns (the index key).
       2. Stores index key values and pointers (e.g., rowids) to the actual data rows.
       3. Is organized for fast searching (e.g., B+-tree, hash function).
     - **Guidelines for Index Selection (Central to understanding "why" and "how"):**
       - **1. Whether to Index - The Obvious Points:** Index attributes frequently used in `WHERE` clauses (for selections and joins), `ORDER BY` clauses, and `GROUP BY` clauses. Do not build an index unless some query will benefit. Choose indexes that benefit multiple queries.
       - **2. Choice of Search Key - For Single Attribute Indexes:**
         - **Exact Match Conditions (e.g., `column = value`):** Both Hash indexes and B+-tree indexes are efficient. Hash often faster if no range queries.
         - **Range Conditions (e.g., `column > value`, `column BETWEEN val1 AND val2`):** B+-tree indexes are essential (store keys sorted). Hash indexes are not suitable.

- **3. Multi-Attribute Search Keys - Composite Indexes:** Create if queries frequently have conditions on columns together (e.g., `WHERE LName = 'Smith' AND FName = 'John'`). Order of columns is critical: `(A, B, C)` supports queries on `(A)`, `(A,B)`, `(A,B,C)`. Place most selective/equality-used attribute first.
- **4. Whether to Cluster:** At most one index per table can be a clustered index (physical order of data rows matches index order). Highly beneficial for range queries on the clustered key and joins on PK if clustered.
- **5. Hash versus Tree Index - Revisited:** B+-tree: default, good for range/equality. Hash Index: best for exact key equality searches, fast point lookups; not for range/partial keys.
- **6. Balancing Cost of Index Maintenance vs. Benefit of Query Speedup:** Indexes speed `SELECT` but slow DML (`INSERT`, `DELETE`, `UPDATE`) as indexes also need updating. Be selective for frequently updated tables. More indexes for read-heavy tables.

- **Types of Indexing (Index Structures and Properties):**
  - **Primary Index (often Clustered):** An index whose search key specifies the sequential order of the file (the ordering key field). The data file is ordered by this key. At most one per data file. If built on PK, often called a clustered index.
  - **Clustered Index:** The physical order of data rows in the table is the same as the order of the index key values. Only one per table. Very efficient for range queries on the clustering key.
  - **Secondary Index (Non-Clustered Index):** An index whose search key specifies an order different from the sequential order of the file. Data rows not physically ordered by this key. Multiple per table. Index entries point to actual data rows (e.g., via rowids).
  - **B+-Tree Index:** Most common type. Balanced tree, all leaf nodes at same depth, leaf nodes linked for range searching. Supports equality/range.
  - **Hash Index:** Uses a hash function to compute bucket/page address for a key value. Very fast for exact equality. Not for range queries.
  - **Unique Index:** DBMS enforces no two rows have the same value for indexed column(s). PK indexes are always unique.
  - **Composite Index (Multi-column Index):** Index on two or more columns. Column order matters.
  - **Covering Index:** Non-clustered index containing all columns required by a query (in `SELECT` and `WHERE`). Allows index-only scan.
  - **Bitmap Index:** For columns with low cardinality (few distinct values). Uses a bitmap per distinct value. Efficient for complex AND/OR queries on such columns.
  - **Function-Based Index (Expression Index):** Index built on the result of a function or expression (e.g., `UPPER(LastName)`).
  - **Spatial Index (e.g., R-tree, Quad-tree):** Used for indexing spatial data types (points, lines, polygons).

        - **Full-Text Index:** Used for indexing text data (`VARCHAR(MAX)`, `CLOB`) to support fast searching for words/phrases.
        - **Index-Only Plans:** If all columns needed by a query are in an index, DBMS answers query by only accessing index, not table. Very efficient.
    - **Clustering [PYQ]:** Physically storing related data records close together on disk (ideally in the same or adjacent blocks) to minimize Disk I/O when accessing them together.
        - **How it Works:** Achieved by organizing data rows based on values of one or more columns (clustering key), often via a clustered index.
        - **Types of Clustering / Implementations:**
            - **Clustered Index (Intra-Table Clustering):** Physical storage order of data rows in a table matches the logical order of the clustered index key.
            - **Co-clustering (Inter-Table Clustering / Multi-Table Clustering):** Physically interleaving rows from two or more related tables on disk, based on their join key (e.g., PK-FK).
            - **Hash Clustering:** Rows are placed into physical storage buckets based on a hash function applied to a clustering key.
        - **Benefits of Clustering:** Reduced Disk I/O, improved query performance (especially range queries on clustered index and joins with co-clustering).
        - **Drawbacks of Clustering:** Higher DML maintenance (page splits), only one clustered index/table, full table scans might be slower if pages not dense.
        - **Tools to Assist in Index Selection:** Database Tuning Advisors / Wizards (e.g., DB2 Index Advisor, SQL Server Index Tuning Wizard) analyze workload, suggest candidate indexes, estimate benefits ("what-if" analysis).
- **De-normalization [PYQ]:** Intentionally adding controlled redundancy to a normalized schema to boost specific query performance by reducing joins. It's the reverse of normalization.
    - **Rationale/Use Case:** Highly normalized schemas can lead to slow queries due to many joins. Denormalization pre-joins or duplicates data for critical, frequently run queries that are performance bottlenecks.
    - **Trade-off: Benefit:** Faster read/query performance for targeted queries. **Cost/Drawback:** Increased storage (data redundancy), risk of update/insert/delete anomalies and data inconsistencies if duplicated data isn't managed carefully, more complex DML operations and application logic to maintain consistency.
    - **Guideline:** Apply selectively *after* normalization and *only if* indexing/query tuning isn't sufficient for critical query performance.
    - **Types of Denormalization / Techniques:**
        - **Pre-joining Tables:** Adding attributes from a "one-side" table to a frequently joined "many-side" table.

        - **Storing Derived/Calculated Values:** Storing pre-computed values (e.g., totals, averages) that are expensive to calculate on-the-fly.
        - **Combining Tables:** Merging tables with a 1:1 relationship or a very tight, frequently accessed 1:N relationship (few "many" side records per "one" side).
        - **Repeating Groups (Limited Use):** Using multiple columns for a fixed, small number of similar attributes instead of a separate table. Generally discouraged due to inflexibility.
        - **Creating Reporting Tables/Data Marts:** Building separate, denormalized tables specifically for analytical queries and reporting, populated from the normalized operational database.
    - **Database Tuning (Conceptual Schema, Queries, Views) [PYQ]:**
        - **Tuning Conceptual Schema:**
            - Settling for a weaker normal form (e.g., 3NF instead of BCNF if BCNF decomposition impacts critical queries).
            - Denormalization (as above).
            - Vertical partitioning (splitting a table's columns into multiple tables).
            - Horizontal partitioning (splitting a table's rows into multiple tables, e.g., by region, date).
        - **Tuning Queries and Views:**
            - Rewriting SQL queries to be more efficient (e.g., avoiding unnecessary joins, using sargable predicates - predicates that can use an index).
            - Optimizing view definitions.
            - Understanding and influencing the query optimizer's plan (e.g., using hints, ensuring statistics are up-to-date).
2. **Database Security [PYQ Q8b]:**
    - **Introduction to Database Security:** Protecting the database against unauthorized access, modification, or destruction.
    - **Objectives:**
        - **Secrecy/Confidentiality:** Preventing disclosure of information to unauthorized users.
        - **Integrity:** Ensuring data is accurate and consistent, and preventing unauthorized modification.
        - **Availability:** Ensuring authorized users can access data when needed.
    - **[PYQ Q8b]** Security considerations should be integrated throughout the database modeling and design process (Conceptual, Logical, Physical levels), not just as an afterthought.
        - **1. Conceptual Level (ER/EER Modeling):** Identify Sensitive Data, Define Access Privileges Conceptually (user roles, data needs), Consider Views for Abstraction, Data Ownership.
        - **2. Logical Level (Relational Schema Design):** View Design for Security (expose only necessary columns/rows), Plan Granular Privileges (`SELECT`, `INSERT`, etc.), Design for Separation of Duties.

- **3. Physical Level (Implementation in DBMS):** Implement using `GRANT`/`REVOKE`, RBAC, Stored Procedures, Triggers for auditing, MAC, Encryption, Auditing features, Authentication, Network Security.

- **Access Control:** Mechanisms to control who can access what data and perform what operations.
  - **Discretionary Access Control (DAC) [PYQ - Implied in GRANT/REVOKE]:** Access control policy based on privileges (access rights) granted to users or roles, typically by object owners or Database Administrators (DBAs).
    - **Objects:** Tables, views, columns, stored procedures, etc.
    - **Privileges:** `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `REFERENCES` (for FKs), `USAGE` (for domains, schemas), `EXECUTE` (for procedures/functions) etc.
    - `GRANT` Statement: `GRANT privilege_list ON object_name TO user_list [WITH GRANT OPTION];`
      - `WITH GRANT OPTION`: Allows the grantee to further grant the received privilege(s) to other users.
    - `REVOKE` Statement: `REVOKE [GRANT OPTION FOR] privilege_list ON object_name FROM user_list [CASCADE | RESTRICT];`
      - `GRANT OPTION FOR`: Revokes only the grant option, not the privilege itself.
      - `CASCADE`: Revokes the privilege from the user and from any users to whom this user granted it (propagates revocation).
      - `RESTRICT`: Fails if the privilege was passed on by the user from whom it is being revoked (default in some systems).
    - **Roles:** A named group of privileges. Privileges are granted to a role, and then the role is granted to users. Simplifies privilege management. `CREATE ROLE`, `DROP ROLE`, `GRANT role TO user`.
  - **Mandatory Access Control (MAC):** Access control policy based on system-wide policies that cannot be changed by individual users/owners. Uses security classifications (labels) for data objects (e.g., Top Secret, Secret, Confidential, Unclassified) and clearance levels for subjects (users/processes).
    - **Bell-LaPadula Model:** A formal MAC model primarily for ensuring confidentiality.
      - **Simple Security Property (No Read Up):** A subject S can read an object O only if the clearance level of S is greater than or equal to the classification level of O (`class(S) >= class(O)`).
      - ***-Property (Star Property - No Write Down):** A subject S can write to an object O only if the clearance level of S is less than or equal to the classification level of O (`class(S) <= class(O)`).

        - **Multilevel Relations and Polyinstantiation:** Storing data with different security levels within the same table. Polyinstantiation allows multiple tuples with the same primary key but different classification levels to exist, visible to users with appropriate clearance, to avoid covert channels (indirect ways of inferring higher-level information).
        - **DoD Security Levels:** Examples include C2, B1, which represent different levels of trust and security features in operating systems/DBMSs.
    - **Views as Security Mechanism:** Virtual tables based on queries that can restrict users to specific rows (via `WHERE` clause in view definition) and specific columns (via `SELECT` list in view definition) of underlying base tables. Users are granted privileges on the view, not directly on the base tables, thus enforcing access control by limiting their data visibility and operational scope.