Model Answer for ISE-II DBMS

2 Marks Questions

1. What role do joins play in combining data from multiple tables, and why are they essential for relational database queries?

- Joins in relational databases are used to combine data from multiple tables based on related columns, often involving primary and foreign keys. They are essential because:

i. Data Combination: Joins allow you to retrieve related data from different tables (e.g., combining customer and order details), which would otherwise be scattered across multiple tables.
ii. Data Integrity: By linking tables via joins, you can avoid data duplication, maintaining a clean and normalized database structure.
iii. Query Efficiency: Joins optimize queries, allowing for complex data retrieval with a single, efficient operation, rather than manually combining data from separate queries.

2. What is referential integrity, and how does it ensure consistency between related tables in a relational database?

- A value appearing in one relation (table) for given set of attributes also appears in another table for another set of attributes is known as referential integrity. It is used to maintain the consistency between two tables. The tuple in one relation refers only to existing tuple in another relation. It is a concept in relational databases that ensures consistency and accuracy in the relationships between tables. It is enforced through the use of foreign keys-columns in one table that reference the primary key of another table. It ensures that:

i. **Valid References**: Foreign keys in one table must reference existing primary keys in another table, preventing invalid or orphaned records.
ii. **Consistency on Updates/Deletes**: It prevents deleting or updating a record in a parent table if it would leave related records in a child table with invalid references. Actions like ON DELETE CASCADE or ON UPDATE CASCADE can be set to maintain relationships.
iii. **Data Accuracy**: It ensures that data relationships are maintained and prevents errors, ensuring the integrity and consistency of the database.

3. **Contrast Boyce-Codd Normal Form (BCNF) eith Third Normal Form (3NF). In your discussion , emphasize their differing approaches to handling functional dependencies and their implications for database design.**

- Boyce-Codd Normal Form (BCNF) and Third Normal Form (3NF) both aim to reduce redundancy and improve data integrity, but they differ in their treatment of functional dependencies:

i. **3NF**: A table is in 3NF if it is in 2NF and all non-key attributes are non-transitively dependent on the primary key. It allows non-prime attributes to determine other non-prime attributes if the determinant is a candidate key**.**
ii. **BCNF**: A table is in BCNF if every determinant (an attribute that determines other attributes) is a candidate key. BCNF is stricter than 3NF, removing cases where non-prime attributes can still functionally determine other attributes, even if 3NF is satisfied.
iii. **Key Difference**: BCNF addresses additional functional dependencies that 3NF does not, making BCNF more restrictive and ensuring higher data integrity by eliminating more potential anomalies in the schema.

4. **Discuss the mechanism through which 1NF effectively prevents data redundancy and promotes data integrity in relational databases.**

- First Normal Form (1NF) prevents data redundancy and promotes data integrity in relational databases by enforcing the following rules:

i. **Atomicity**: 1NF requires that each column contains atomic (indivisible) values, ensuring that each field stores only a single value, not lists or sets of values. This eliminates repeating groups within columns.
ii. **Consistent Structure**: It ensures that all rows have the same number of columns, and each column contains data of a single type, which reduces the risk of data inconsistencies and errors.
iii. **Elimination of Redundancy**: By storing each piece of information in a separate row (e.g., splitting multiple phone numbers into individual rows), 1NF reduces unnecessary duplication of data, improving storage efficiency and simplifying data maintenance.

5. Explain rollback and commit using suitable example consider a database of college.

- ROLLBACK and COMMIT are key operations in transaction management that ensure data integrity in a relational database.

i. **COMMIT**: This operation is used to permanently save all changes made during a transaction to the database. Once a commit is issued, the changes are visible to all other users and cannot be undone.

Example: In a college database, if a student named Charlie is successfully added to the Students table and registered for a course, a COMMIT would make these changes permanent.

ii. **ROLLBACK**: This operation is used to undo any changes made during a transaction. If an error occurs or if the transaction needs to be canceled, a ROLLBACK restores the database to its previous consistent state.

Example: If, after adding Charlie to the Students table, there is an error updating the Courses table, a ROLLBACK would undo the insertion of Charlie, ensuring no partial data is saved.

Let's see the above example in more detail with syntax:

- If a student named Charlie is successfully added to the Students table and registered for a course in the Courses table, the COMMIT operation saves these changes:

```
INSERT INTO Students (StudentID, Name, CourseID) VALUES (3, 'Charlie', 'C101');

UPDATE Courses SET RegisteredStudents = RegisteredStudents + 1 WHERE CourseID
= 'C101';

COMMIT;
```

- If Charlie was successfully added to the Students table but the update to the Courses table failed, a ROLLBACK would undo the insertion of Charlie to keep the database consistent:

```
ROLLBACK;
```

6. **What defines a transaction in a database system, and why are ACID properties essential for maintaining consistency and realiability?**

- A transaction is a sequence of one or more operations that are treated as a single unit of work. These operations must either be completed in their entirety or not executed at all, ensuring consistency in the database. In practical terms, a transaction might involve multiple actions such as updating user information, transferring funds between accounts, or modifying records in multiple tables. The **ACID** properties (Atomicity, Consistency, Isolation, Durability) are essential for the following reasons:

i. **Atomicity** ensures that a transaction is fully completed or fully rolled back, preventing partial updates that could corrupt the database.
ii. **Consistency** guarantees that a transaction takes the database from one valid state to another, preserving integrity constraints.
iii. **Isolation** ensures that transactions don't interfere with each other, preventing issues like dirty reads or conflicts.
iv. **Durability** ensures that once a transaction is committed, its changes are permanent, even in the case of system failures.

7. **Discuss the importance of Atomicity in the database design with suitable examples.**

- Atomicity is a crucial property in database design that ensures a transaction is treated as a single, indivisible unit. It guarantees that either all operations within the transaction are completed successfully, or none of them are, preventing partial updates that could lead to inconsistencies. The importance of Atomicity in the database design with suitable example are:

i. **Data Integrity**: Atomicity ensures the database remains in a consistent state, even in the event of a failure. For example, if a money transfer between two accounts fails midway, atomicity ensures that the transaction is rolled back, and no money is deducted without being credited to the other account.
ii. **Error Recovery**: If an error occurs, atomicity ensures that all changes made by the transaction are undone, simplifying recovery. For instance, if a payment transaction fails after deducting funds but before updating the order status, atomicity ensures the deduction is reversed.
iii. **Prevents Partial Updates**: Without atomicity, partial updates could leave the database in an inconsistent state. For example, an inventory update and order record may not be

completed together, leading to errors in stock levels or untracked sales. Atomicity prevents this by ensuring both operations succeed or fail together.

- **5 Marks Questions**

1. **Evaluate the importance of SQL sub-queries. How would you use a sub-query to find all flights operated by a specific airline with delays over 30 minutes.**

➢ SQL sub-queries (also known as nested queries) are SQL queries embedded within another query. They allow for more complex data retrieval, often simplifying the SQL syntax and making queries more readable and efficient. Sub-queries can be used in various parts of an SQL statement, such as the SELECT, FROM, or WHERE clauses. The importance of SQL sub-queries lies in their ability to:

i. **Simplify Complex Queries**: Sub-queries allow complex operations, such as filtering, aggregating, or joining data, to be done in a modular, step-by-step manner. This improves readability and makes the query easier to maintain.
ii. **Improve Query Flexibility**: By using sub-queries, you can perform operations that would otherwise require multiple JOINs or multiple SQL statements. They help extract intermediate results that can be reused in the outer query.
iii. **Data Encapsulation**: Sub-queries allow the encapsulation of one query within another, enabling you to isolate the logic for specific data extraction or manipulation, without affecting the main query's structure.
iv. **Conditional Filters**: Sub-queries are useful for performing conditions on columns based on the results of another query. This is especially helpful when you need to compare values with a dynamically calculated list or set.

Example: Using a Sub-query to Find Flights Operated by a Specific Airline with Delays Over 30 Minutes

Let's say we have two tables:

- flights: Contains flight details (e.g., flight number, airline, departure time).
- delays: Contains information about flight delays (e.g., flight number, delay duration).

We want to find all flights operated by a specific airline with delays over 30 minutes. We can achieve this by using a sub-query to first filter the flights that have delays over 30 minutes, and then select flights from a specific airline based on that. The SQL query is:

```
SELECT flight_number, airline, departure_time

FROM flights

WHERE flight_number IN (

   SELECT flight_number

   FROM delays

   WHERE delay_duration > 30

);
```

If we wanted to find flights operated by a specific airline, we could add an additional condition to the outer query:

```
SELECT flight_number, airline, departure_time

FROM flights

WHERE airline = 'SpecificAirline'

AND flight_number IN (

   SELECT flight_number

   FROM delays

   WHERE delay_duration > 30

);
```

2. **Describe the process and challenges of revoking user authorization in complex database systems that span across multiple applications or distributed architectures. Provide examples of potential issues that arise with orphaned permissions or inconsistent access control enforcement.**

- Revoking user authorization in complex database systems, especially those that span multiple applications or distributed architectures, presents several challenges. These systems often involve interconnected components, such as databases, APIs, and microservices, each with its own access control mechanisms. Ensuring that revocation of user access is consistent across all parts of the system is crucial for maintaining security and data integrity. The process of Revoking User Authorization is:

i. **Centralized vs. Decentralized Control**:

- In a centralized system, where user authentication and authorization are managed from a single point (like an Identity and Access Management (IAM) system), revocation is typically simpler. When a user is deactivated or their privileges are revoked, it can be done in one place, and the changes propagate across all connected systems and databases.
- In a decentralized or distributed system, each database or application may have its own access control mechanism, making revocation more complicated. Revoking a user's rights in one system may not automatically affect other systems, leading to inconsistent access.

ii. **Identifying the User's Permissions**:

- The first step in the revocation process is identifying all the places where the user has permissions. This includes direct access (e.g., specific user roles in a database) and indirect access (e.g., through group memberships, inherited permissions, or through applications that interact with the database).
- In distributed systems, where users might access data across multiple services, applications, and databases, this can be especially challenging. An inventory of all user access points is necessary to ensure that all permissions are correctly identified.

iii. **Revocation**:

- Once the user's access points are identified, the administrator or system revokes the permissions at each point (e.g., removing roles, altering database privileges, disabling API keys).
- In complex systems, this might involve adjusting permissions across databases (SQL, NoSQL), file systems, cloud services, and application interfaces.

iv. **Propagation of Changes**:

- In systems with replication or distributed databases, the revocation must be propagated to all copies of the data. Changes in access control at one site may not automatically update all others, especially in systems with asynchronous replication.
- Caching mechanisms and session management should also be considered, as old sessions or cached credentials might still allow access temporarily even after revocation.

v. **Audit and Verification**:

- After revocation, systems should verify that the user can no longer access the resources. This often involves logging and monitoring to check that the user's access is effectively blocked across all systems.
- Audit trails and logging play a vital role in tracking who accessed what data and ensuring revocation is fully effective.

Let us see the Challenges in Revoking User Authorization:

i. **Orphaned Permissions**:

- Orphaned permissions occur when a user is removed or their access rights are revoked in one part of the system, but their permissions persist in other places. This can lead to unintended access.
- For example, a user might have been granted access through a direct role in one database but may still retain access indirectly through inherited permissions or through linked services. If the role in the database is revoked but not the inherited permissions or access in an API or application layer, the user may still have access to sensitive data.
- **Example**: If a user is removed from a role in a central database, but that same user's access is still available via an application server, they may continue to interact with the database through the application interface.

ii. **Inconsistent Access Control Enforcement**:

- In distributed systems, access control mechanisms might not be uniformly enforced across all databases, applications, or microservices. One application may use role-based access control (RBAC), while another uses attribute-based access control (ABAC), leading to discrepancies in the user's access after revocation.
- **Example**: A user is revoked access from a SQL database, but an application server using the same database still maintains its internal access control list (ACL) where the user has

permissions. The user can still access the data even though their direct access was revoked in the database.

iii. **Replication and Propagation Delays**:

- In distributed architectures where databases are replicated across multiple nodes or regions (e.g., in a cloud environment or between different microservices), replication delays or asynchronous replication might cause the revocation not to be immediately reflected across all nodes.
- This can lead to temporary periods during which a revoked user still has access to resources, posing a security risk.
- **Example**: A user's access to a database is revoked on one server, but due to replication lag, the user's access is still valid on another server for some time.

iv. **Session and Caching Issues**:

- If user credentials are cached in local or session storage (e.g., session tokens, browser cookies), revoking access might not take effect immediately because the user is still authenticated.
- **Example**: A user's session is still active after their privileges are revoked, allowing them to continue accessing the system until their session expires or is manually terminated. This is particularly problematic in applications that do not have proper session invalidation mechanisms.

v. **Cross-Application Permissions**:

- In systems where multiple applications rely on a shared database or authorization service (e.g., single sign-on systems), revocation in one system might not propagate to others.
- **Example**: A user's access is revoked in the primary database, but the user still has access to a secondary application or service that interacts with the database and has its own local permissions, leading to inconsistent access.

vi. **Data Integrity and Incomplete Revocation**:

- In some systems, the lack of proper transaction management during revocation can lead to incomplete or partial revocation. For example, when revoking a user's access across multiple services or databases, some services may not properly commit the revocation, leading to data integrity issues.

- **Example**: A user is removed from a database role, but the revocation is not reflected in a related service that has a copy of the data. The user could still access and modify data indirectly through that service.

The solutions to address these challenges are:

i. **Centralized Identity and Access Management (IAM)**:

- Implementing a centralized IAM system (such as OAuth, OpenID Connect, or LDAP) can help enforce consistent user permissions across multiple services and databases. This reduces the risk of orphaned permissions and inconsistent enforcement.

ii. **Regular Audits and Access Reviews**:
- Periodically auditing and reviewing user access ensures that privileges are up to date and that any orphaned or unnecessary permissions are identified and revoked. Implementing role-based access control (RBAC) can also make audits easier and more manageable.

iii. **Session Management**:
- Ensuring proper session management, including immediate invalidation of user sessions and access tokens upon revocation, is key. This might involve global session revocation or setting short session timeouts to limit the window during which revoked users can retain access.

iv. **Replication and Synchronization**:
- To address replication delays, synchronous replication or more aggressive synchronization mechanisms can be implemented. This ensures that changes, such as revocations, propagate to all systems more quickly.

v. **Granular Access Control Mechanisms**:
- Using fine-grained access controls such as attribute-based access control (ABAC) and dynamic policy enforcement can ensure that access is continuously monitored and adjusted based on user attributes, behavior, and context.

Revoking user authorization in complex, distributed database systems is a challenging task due to the potential for orphaned permissions, inconsistent access control enforcement, session and caching issues, and delays in replication. Ensuring that revocations are propagated across all systems and that any existing sessions are properly invalidated is crucial for maintaining security. By implementing centralized identity management, auditing, and fine-grained access control mechanisms, organizations can mitigate these challenges and maintain a robust and secure authorization model.

3. Analyze the implications of adopting 3NF over 2NF in terms of data integrity, including the impact on database performance and real-world applications.

- When designing a relational database, normalization is crucial for ensuring data integrity and minimizing redundancy. Third Normal Form (3NF) and Second Normal Form (2NF) are stages of normalization that influence how data is organized and how the database performs. While both forms aim to reduce redundancy, 3NF offers a more refined approach than 2NF, with implications for data integrity, database performance, and real-world applications.

i. Data Integrity:

- **2NF and Data Integrity:**
  - **2NF** eliminates **partial dependencies**, meaning that all non-key attributes must depend on the entire primary key, rather than just part of it (in cases of composite keys). This reduces data redundancy and ensures more accurate data representation.
  - However, **2NF does not address transitive dependencies**, where non-key attributes depend on other non-key attributes, potentially leading to **update anomalies**. For instance, if a department name depends on the department ID, and this ID is stored in multiple student records, changing the department name in one record might result in inconsistency across others.
- **3NF and Data Integrity:**
  - **3NF** takes data integrity further by eliminating **transitive dependencies**, ensuring that non-key attributes depend only on the primary key. This minimizes the chances of **update, insert, and delete anomalies** by ensuring that attributes are only dependent on the primary key.
  - This improved data integrity makes 3NF ideal for **complex systems** where data consistency is critical (e.g., financial systems, inventory systems). It prevents redundant storage of data, ensuring that updates to data (like customer information) only need to happen once.

ii. Performance:

- **2NF and Performance:**
  - Since **2NF** only removes partial dependencies, the schema can still be relatively simple, often involving fewer tables than 3NF. This can lead to **better performance** for simpler queries, as fewer **JOIN** operations are required.

- However, **redundant data** (due to transitive dependencies) may still be stored, and updates or deletions could require extra processing to maintain data consistency.

- **3NF and Performance:**
  - **3NF**, while improving data integrity, can **hurt performance** due to the need for additional **JOINs** between tables. This is especially noticeable in large databases or systems with many interrelated tables, as retrieving data often involves multiple JOINs to fetch information across tables.
  - For example, retrieving student information along with their department and department head would require joining multiple tables (Student, Department, DepartmentHead), leading to slower queries, particularly in large datasets.

iii. Real-World Applications:

- **2NF in Real-World Applications:**
  - **2NF** is often suitable for **smaller applications** or systems with **less complex relationships**. For example, in a small inventory system or simple student management system, where the data relationships are not highly complex, **2NF** provides a balance between data consistency and performance.
  - However, in applications where data consistency is less critical, 2NF may be sufficient, allowing for **faster performance** at the expense of some data integrity.
- **3NF in Real-World Applications:**
  - **3NF** is ideal for **large-scale applications** where **data integrity** and **consistency** are paramount. For example, in a **financial system**, ensuring that customer information or transaction data is consistent across the system is critical. 3NF helps by removing redundancy and ensuring that updates to, say, customer address details, are done in one place only.
  - On the other hand, **performance might suffer** in such applications due to the complexity of queries and JOIN operations. In **data warehousing** or **reporting systems**, where read performance is crucial, selective **denormalization** may be used to enhance performance while maintaining a normalized structure for transactional data.

iv. Balancing Data Integrity and Performance:

- **Trade-Off Between Normalization and Performance:**
  - Adopting 3NF over 2NF enhances **data integrity** by eliminating transitive dependencies, which reduces redundancy and ensures more accurate data representation. However, this comes at the cost of **database performance**, as

more tables and complex JOIN operations are required, which can slow down query performance, especially in large systems.

- In **real-world applications**, this trade-off can be managed by using **denormalization** in performance-critical areas of the database, such as reporting, while maintaining 3NF in transactional parts of the system. This approach ensures that critical data integrity is maintained while optimizing for performance in high-demand areas.

Adopting 3NF over 2NF results in better data integrity by eliminating transitive dependencies, reducing redundancy, and preventing update anomalies. However, it can negatively impact database performance, as it requires more complex queries with multiple JOINs. In real-world applications, 3NF is preferable for large-scale systems requiring high data consistency, while 2NF may be suitable for smaller, less complex systems where performance is a higher priority. In many cases, a combination of normalization and denormalization is used to strike a balance between data integrity and system performance.

4. Discuss the functional dependencies that can cause violations of 2NF and 3NF, and outline strategies to mitigate these violations in database design.

- In relational database design, functional dependencies (FDs) describe the relationship between attributes, indicating that the value of one attribute is determined by another. Violations of 2NF (Second Normal Form) and 3NF (Third Normal Form) are often caused by improper handling of these dependencies. Let's explore the functional dependencies that cause violations of 2NF and 3NF, along with strategies for mitigating these issues.

i. Violations of 2NF (Partial Dependency)

Second Normal Form (2NF) requires that a table be in 1NF (First Normal Form) and that all non-prime attributes (attributes not part of a candidate key) are fully dependent on the entire primary key in the case of a composite key. A violation of 2NF occurs when a non-prime attribute depends only on a part of a composite primary key, which is known as a partial dependency.

- **Functional Dependency Causing Violation of 2NF:**

- **Partial Dependency**: If a composite primary key is used, and a non-prime attribute depends only on a subset of that key, the relation is not in 2NF.

  Example: Consider a table StudentCourses(StudentID, CourseID, Instructor, InstructorPhone) where StudentID, CourseID is the composite primary key:

- StudentID → Instructor
- CourseID → InstructorPhone

Here, Instructor is dependent only on StudentID (part of the composite key), and InstructorPhone is dependent only on CourseID. These are partial dependencies, as they only depend on part of the composite key, not the full key.

- **Mitigation Strategy for 2NF Violations:**

- **Decompose the Table**: To remove partial dependencies, split the table into smaller tables where non-prime attributes are fully dependent on the whole primary key.

For example:

a. StudentCourses(StudentID, CourseID)
b. Courses(CourseID, Instructor, InstructorPhone)

This decomposition removes the partial dependencies, ensuring that all non-prime attributes depend on the full primary key.

ii. Violations of 3NF (Transitive Dependency)

Third Normal Form (3NF) builds upon 2NF by requiring that a table be in 2NF, and that all non-prime attributes are non-transitively dependent on the primary key. A violation of 3NF occurs when a non-prime attribute depends on another non-prime attribute, rather than directly on the primary key. This is known as a transitive dependency.

- **Functional Dependency Causing Violation of 3NF:**

  - Transitive Dependency: A non-prime attribute depends on another non-prime attribute, which in turn depends on the primary key. This violates 3NF because the dependency is not directly on the primary key.

  Example: Consider a table EmployeeInfo(EmployeeID, Department, DepartmentHead, HeadPhone) where EmployeeID is the primary key:

  - EmployeeID → Department
  - Department → DepartmentHead
  - DepartmentHead → HeadPhone

  Here, DepartmentHead and HeadPhone depend on Department, which is a non-prime attribute, and Department depends on EmployeeID. Thus, there is a transitive dependency from DepartmentHead and HeadPhone through Department.

- **Mitigation Strategy for 3NF Violations:**

  - Decompose the Table: To remove transitive dependencies, split the table into smaller tables such that non-prime attributes are directly dependent on the primary key.

    For example:

a. EmployeeInfo(EmployeeID, Department)
b. DepartmentInfo(Department, DepartmentHead, HeadPhone)

In this way, the transitive dependency is eliminated, and the non-prime attributes (DepartmentHead and HeadPhone) are now only dependent on the Department attribute, which is a candidate key in DepartmentInfo.

iii. General Strategies to Mitigate Violations

To address 2NF and 3NF violations, the primary strategy is decomposition. Decomposing a table into smaller, more focused tables ensures that non-prime attributes depend only on the primary key, either directly (in 3NF) or fully (in 2NF).

Key Strategies:

- **Decompose Relations**: Split large tables that contain partial or transitive dependencies into smaller tables to isolate dependencies. For 2NF violations, focus on removing partial dependencies, and for 3NF violations, eliminate transitive dependencies.
- **Ensure Direct Dependency**: Ensure that non-prime attributes are directly dependent on the primary key, and avoid indirect dependencies through other non-prime attributes.
- **Identify Candidate Keys**: Properly identify candidate keys to ensure correct decomposition and minimize the risk of partial and transitive dependencies.
- **Maintain Data Integrity**: Proper normalization reduces data redundancy and improves consistency by ensuring that updates, deletions, and insertions do not lead to anomalies.

iv. Implications and Real-World Considerations

- **Redundancy and Inconsistency**: Functional dependencies that violate 2NF and 3NF can lead to **data redundancy** and **inconsistent data**, especially in large datasets, where redundant updates can lead to anomalies.
- **Complex Queries**: As normalization progresses to higher forms like 3NF, queries become more complex because they often involve multiple tables and **JOIN** operations. However, this is usually a worthwhile trade-off for ensuring data consistency and minimizing redundancy.
- **Denormalization**: In some performance-critical systems (e.g., reporting systems), **denormalization** (introducing some redundancy) may be used to improve query performance, especially in scenarios where read speed is more important than absolute data integrity.

Violations of 2NF and 3NF arise from partial and transitive dependencies, respectively, which cause redundancy and data anomalies. The primary strategy to mitigate these violations is decomposition, which divides a table into smaller, more focused tables to ensure that non-prime attributes depend solely on the primary key. By applying these strategies, databases can maintain data integrity and avoid anomalies, although performance considerations may occasionally necessitate denormalization in certain real-world applications.