**Code No.**

**METHODIST COLLEGE OF ENGINEERING & TECHNOLOGY (An Autonomous Institution)**

**M.B.A III-Semester (Regular) Examination, Feb/March -2023**

**Subject: DATA BASE MANAGEMENT SYSTEM**

**Time: 3 hours Max.Marks:60**

**Note: Missing data, if any, maybe suitably assumed.**

**PART-A**

**Answer All the questions.**

|  |  |
| --- | --- |
| **Q.No** | **Questions** |
| **1. a** | Data Definition Language (DDL) is a subset of SQL. It is a language for describing data and its relationships in a database.  You can generate DDL in a script for database objects to:   * Keep a snapshot of the database structure * Set up a test system where the database acts like the production system but contains no data * Produce templates for new objects that you can create based on existing ones. For example, generate the DDL for the Customer table, then edit the DDL to create the table Customer\_New with the same schema. |
| **b** | The Tuple Relational Calculus (TRC) is used to select tuples from a relation. The tuples with specific range values, tuples with certain attribute values, and so on can be selected. In TRC, the variables represent the tuples from specified relations. A tuple is a single element of relation. In database terms, it is a row. Notation:  {T | P (T)} or {T | Condition (T)} Example : {T | EMPLOYEE (T) AND T.DEPT\_ID = 10} |
| **c** | Hashing is **a DBMS technique for searching for needed data on the disc without utilizing an index structure**. The hashing method is basically used to index items and retrieve them in a DB since searching for a specific item using a shorter hashed key rather than the original value is faster. It can be nearly hard to search all index values through all levels of a large database structure and then get to the target data block to obtain the needed data. Hashing is a method for calculating the direct position of an information record on the disk without the use of an index structure.To generate the actual address of a data record, hash functions containing search keys as parameters are used. |
| **d** | Concurrency control concept comes under the Transaction in database management system (DBMS). It is a procedure in DBMS which helps us for the management of two simultaneous processes to execute without conflicts between each other, these conflicts occur in multi user systems.  Concurrency can simply be said to be executing multiple transactions at a time. It is required to increase time efficiency. If many transactions try to access the same data, then inconsistency arises. Concurrency control required to maintain consistency data.  For example, if we take ATM machines and do not use concurrency, multiple persons cannot draw money at a time in different places. This is where we need concurrency.  Advantages  The advantages of concurrency control are as follows −   * Waiting time will be decreased. * Response time will decrease. * Resource utilization will increase. * System performance & Efficiency is increased. |
| **e** | In the database, Transaction Control Language (TCL) commands are used to govern transactions. This command is used to manage changes to DML statements. TCL allows you to create logical transactions by combining your statements. Read more on the commands and transactions in TCL.  Some of the TCL commands are listed below: COMMIT Command: This command is used to save all the transactions in the DB.  Syntax:  COMMIT;  For Example,  UPDATE Student SET DOB=’2005-03-27’ WHERE Stu\_Name=’Joey’;  COMMIT;  Thus, this example would insert the DOB in the given table, which has the name = Joey and then COMMIT these changes in the DB. ROLLBACK Command: The “rollback” term refers to the method of undoing changes. Thus, this command could only be used in order to reverse transactions that occurred since the last ROLLBACK or COMMIT command. All the modifications must be cancelled in case any SQL grouped statements produce a certain error.  Syntax:  ROLLBACK;  For Example:  UPDATE Student SET DOB=’2005-03-27’ WHERE Stu\_Name=’Joe’;  ROLLBACK;  The example given above would insert the DOB in a table that has name = Joey and then ROLLBACK these changes in the DB. Thus, this type of operation would not create an impact on the table. SAVEPOINT Command: It is used to roll back a certain transaction to a certain point rather than the entire transaction.  Syntax:  SAVEPOINT Name\_of\_Savepoint;  This command is used exclusively among all transactions in order to create SAVEPOINT.  The ROLLBACK command is used to undo a set of given transactions.  Here is the syntax for the rollback to savepoint command in a code:  ROLLBACK TO Name\_of\_Savepoint;  Example:  SAVEPOINT S1; // created savepoint  DELETE FROM Student WHERE Stu\_Name = ‘Joey’; //deleted  SAVEPOINT S2; // created savepoint |
|  |  |

**PART-B**

**Answer Any Five questions**.

|  |  |  |
| --- | --- | --- |
| **Q.No.** |  | **Questions** |
| **2.** | **a** | Entity relationship (ER) models are based on the real-world entities and their relationships. It is easy for the developers to understand the system by simply looking at the ER diagram. ER models are normally represented by ER-diagrams. Components ER diagram basically having three components:   * **Entities** − It is a real-world thing which can be a person, place, or even a concept. For Example: Department, Admin, Courses, Teachers, Students, Building, etc are some of the entities of a School Management System. * **Attributes** − An entity which contains a real-world property called an attribute. For Example: The entity employee has the property like employee id, salary, age, etc. * **Relationship** − Relationship tells how two attributes are related. For Example: Employee works for a department.   An entity has a real-world property called attribute and these attributes are defined by a set of values called domain.  **Example 1**  In a university,   * A student is an entity, * University is the database, * Name and age and sex are the attributes. * The relationships among entities define the logical association between entities.   **Example 2**  Given below is another example of ER:  https://www.tutorialspoint.com/assets/questions/media/53818/employee_and_department.jpg  In the above example,  Entities − Employee and Department.  Attributes −   * Employee − Name, id, Age, Salary * Department − Dept\_id, Dept\_name   The two entities are connected using the relationship. Here, each employee works for a department. Features of ER The features of ER Model are as follows −   * **Graphical Representation is Better Understanding** − It is easy and simple to understand so it can be used by the developers to communicate with the stakeholders. * **ER Diagram** − ER diagrams are used as a visual tool for representing the model. * **Database Design** − This model helps the database designers to build the database.   **Advantages**  The advantages of ER are as follows −   * The ER model is easy to build. * This model is widely used by database designers for communicating their ideas. * This model can easily convert to any other model like network model, hierarchical model etc. * It is integrated with the dominant relational model.   **Disadvantages**  The disadvantages of ER are as follows −   * There is no industry standard for developing an ER model. * Information might be lost or hidden in the ER model. * There is no Data Manipulation Language (DML). * There is limited relationship representation. |
| **b** | * SELECT—To retrieve data. * INSERT—To add data. * UPDATE—To modify data. * DELETE—To delete data. * CALL—To invoke an SQL invoked procedure or table procedure. |
| **3.** | **a** | Basics of Relational model:  Relational Algebra is a procedural query language that takes relations as an input and returns relations as an output. There are some basic operators which can be applied in relation to producing the required results which we will discuss one by one. We will use STUDENT\_SPORTS, EMPLOYEE, and STUDENT relations as given in Table 1, Table 2, and Table 3 respectively to understand the various operators.  **Table 1: STUDENT\_SPORTS**   |  |  | | --- | --- | | **ROLL\_NO** | **SPORTS** | | 1 | Badminton | | 2 | Cricket | | 2 | Badminton | | 4 | Badminton |   **Table 2: EMPLOYEE**   |  |  |  |  |  | | --- | --- | --- | --- | --- | | **EMP\_NO** | **NAME** | **ADDRESS** | **PHONE** | **AGE** | | 1 | RAM | DELHI | 9455123451 | 18 | | 5 | NARESH | HISAR | 9782918192 | 22 | | 6 | SWETA | RANCHI | 9852617621 | 21 | | 4 | SURESH | DELHI | 9156768971 | 18 |   **Table 3: STUDENT**   |  |  |  |  |  | | --- | --- | --- | --- | --- | | **ROLL\_NO** | **NAME** | **ADDRESS** | **PHONE** | **AGE** | | 1 | RAM | DELHI | 9455123451 | 18 | | 2 | RAMESH | GURGAON | 9652431543 | 18 | | 3 | SUJIT | ROHTAK | 9156253131 | 20 | | 4 | SURESH | DELHI | 9156768971 | 18 |   ***Selection operator (σ):***Selection operator is used to selecting tuples from a relation based on some condition. Syntax:  **σ (Cond)(Relation Name)**  Extract students whose age is greater than 18 from STUDENT relation given in Table 3  **σ (AGE>18)(STUDENT)**  **[Note:**SELECT operator does not show any result, the projection operator must be called before the selection operator to generate or project the result. So, the correct syntax to generate the result is**:** ∏(σ (AGE>18)(STUDENT))]  **RESULT:**   |  |  |  |  |  | | --- | --- | --- | --- | --- | | **ROLL\_NO** | **NAME** | **ADDRESS** | **PHONE** | **AGE** | | 3 | SUJIT | ROHTAK | 9156253131 | 20 |   ***Projection Operator (∏):*** Projection operator is used to project particular columns from a relation. Syntax:  **∏(Column 1,Column 2….Column n)(Relation Name)**  Extract ROLL\_NO and NAME from STUDENT relation given in Table 3  **∏(ROLL\_NO,NAME)(STUDENT)**  **RESULT:**   |  |  | | --- | --- | | **ROLL\_NO** | **NAME** | |  | RAM | | 2 | RAMESH | | 3 | SUJIT | | 4 | SURESH |   **Note:**If the resultant relation after projection has duplicate rows, it will be removed. For Example  ∏(ADDRESS)(STUDENT) will remove one duplicate row with the value DELHI and return three rows.  ***Cross Product(X):*** Cross product is used to join two relations. For every row of Relation1, each row of Relation2 is concatenated. If Relation1 has m tuples and and Relation2 has n tuples, cross product of Relation1 and Relation2 will have m X n tuples. Syntax:  **Relation1 X Relation2**  To apply Cross Product on STUDENT relation given in Table 1 and STUDENT\_SPORTS relation given in Table 2,  **STUDENT X STUDENT\_SPORTS**  **RESULT:**   |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | **ROLL\_NO** | **NAME** | **ADDRESS** | **PHONE** | **AGE** | **ROLL\_NO** | **SPORTS** | | 1 | RAM | DELHI | 9455123451 | 18 | 1 | Badminton | | 1 | RAM | DELHI | 9455123451 | 18 | 2 | Cricket | | 1 | RAM | DELHI | 9455123451 | 18 | 2 | Badminton | | 1 | RAM | DELHI | 9455123451 | 18 | 4 | Badminton | | 2 | RAMESH | GURGAON | 9652431543 | 18 | 1 | Badminton | | 2 | RAMESH | GURGAON | 9652431543 | 18 | 2 | Cricket | | 2 | RAMESH | GURGAON | 9652431543 | 18 | 2 | Badminton | | 2 | RAMESH | GURGAON | 9652431543 | 18 | 4 | Badminton | | 3 | SUJIT | ROHTAK | 9156253131 | 20 | 1 | Badminton | | 3 | SUJIT | ROHTAK | 9156253131 | 20 | 2 | Cricket | | 3 | SUJIT | ROHTAK | 9156253131 | 20 | 2 | Badminton | | 3 | SUJIT | ROHTAK | 9156253131 | 20 | 4 | Badminton | | 4 | SURESH | DELHI | 9156768971 | 18 | 1 | Badminton | | 4 | SURESH | DELHI | 9156768971 | 18 | 2 | Cricket | | 4 | SURESH | DELHI | 9156768971 | 18 | 2 | Badminton | | 4 | SURESH | DELHI | 9156768971 | 18 | 4 | Badminton |   ***Union (U):***Union on two relations R1 and R2 can only be computed if R1 and R2 are **union compatible** (These two relations should have the same number of attributes and corresponding attributes in two relations have the same domain). Union operator when applied on two relations R1 and R2 will give a relation with tuples that are either in R1 or in R2. The tuples which are in both R1 and R2 will appear only once in the result relation. Syntax:  **Relation1 U Relation2**  Find the person who is either student or employees, we can use Union operators like:  **STUDENT U EMPLOYEE**  **RESULT:**   |  |  |  |  |  | | --- | --- | --- | --- | --- | | **ROLL\_NO** | **NAME** | **ADDRESS** | **PHONE** | **AGE** | | 1 | RAM | DELHI | 9455123451 | 18 | | 2 | RAMESH | GURGAON | 9652431543 | 18 | | 3 | SUJIT | ROHTAK | 9156253131 | 20 | | 4 | SURESH | DELHI | 9156768971 | 18 | | 5 | NARESH | HISAR | 9782918192 | 22 | | 6 | SWETA | RANCHI | 9852617621 | 21 |   ***Minus (-):*** Minus on two relations R1 and R2 can only be computed if R1 and R2 are **union compatible**. Minus operator when applied on two relations as R1-R2 will give a relation with tuples that are in R1 but not in R2. Syntax:  **Relation1 - Relation2**  Find the person who is a student but not an employee, we can use minus operator like:  **STUDENT - EMPLOYEE**  **RESULT:**   |  |  |  |  |  | | --- | --- | --- | --- | --- | | **ROLL\_NO** | **NAME** | **ADDRESS** | **PHONE** | **AGE** | | 2 | RAMESH | GURGAON | 9652431543 | 18 | | 3 | SUJIT | ROHTAK | 9156253131 | 20 |   ***Rename(ρ):***Rename operator is used to giving another name to a relation. Syntax:  **ρ(Relation2, Relation1)**  To rename STUDENT relation to STUDENT1, we can use rename operator like:  **ρ(STUDENT1, STUDENT)**  If you want to create a relation STUDENT\_NAMES with ROLL\_NO and NAME from STUDENT, it can be done using rename operator as:  **ρ(STUDENT\_NAMES, ∏(ROLL\_NO, NAME)(STUDENT))** |
| **4.** | **a** | **SQL Join** statement is used to combine data or rows from two or more tables based on a common field between them. Different types of Joins are as follows:   * INNER JOIN * LEFT JOIN * RIGHT JOIN * FULL JOIN   Consider the two tables below:  **Student**  Screenshot from 2016-12-19 12-53-29  **StudentCourse**  table5  The simplest Join is INNER JOIN. ****A. INNER JOIN**** The INNER JOIN keyword selects all rows from both the tables as long as the condition is satisfied. This keyword will create the result-set by combining all rows from both the tables where the condition satisfies i.e value of the common field will be the same.  **Syntax**:  SELECT table1.column1,table1.column2,table2.column1,....  FROM table1  INNER JOIN table2  ON table1.matching\_column = table2.matching\_column;  **table1**: First table.  **table2**: Second table  **matching\_column**: Column common to both the tables.  ***Note****: We can also write JOIN instead of INNER JOIN. JOIN is same as INNER JOIN.*  https://blog.codinghorror.com/content/images/uploads/2007/10/6a0120a85dcdae970b012877702708970c-pi.png  **Example Queries(INNER JOIN)**  This query will show the names and age of students enrolled in different courses.  SELECT StudentCourse.COURSE\_ID, Student.NAME, Student.AGE FROM Student  INNER JOIN StudentCourse  ON Student.ROLL\_NO = StudentCourse.ROLL\_NO;  **Output**:  table2 ****B. LEFT JOIN**** This join returns all the rows of the table on the left side of the join and matches rows for the table on the right side of the join. For the rows for which there is no matching row on the right side, the result-set will contain *null*. LEFT JOIN is also known as LEFT OUTER JOIN.  **Syntax:**  SELECT table1.column1,table1.column2,table2.column1,....  FROM table1  LEFT JOIN table2  ON table1.matching\_column = table2.matching\_column;  table1: First table.  table2: Second table  matching\_column: Column common to both the tables.  ***Note****: We can also use LEFT OUTER JOIN instead of LEFT JOIN, both are the same.*  https://i.stack.imgur.com/VkAT5.png  **Example Queries(LEFT JOIN)**:  SELECT Student.NAME,StudentCourse.COURSE\_ID  FROM Student  LEFT JOIN StudentCourse  ON StudentCourse.ROLL\_NO = Student.ROLL\_NO;  **Output**:  table3 ****C. RIGHT JOIN**** RIGHT JOIN is similar to LEFT JOIN. This join returns all the rows of the table on the right side of the join and matching rows for the table on the left side of the join. For the rows for which there is no matching row on the left side, the result-set will contain *null*. RIGHT JOIN is also known as RIGHT OUTER JOIN.  **Syntax:**  SELECT table1.column1,table1.column2,table2.column1,....  FROM table1  RIGHT JOIN table2  ON table1.matching\_column = table2.matching\_column;  table1: First table.  table2: Second table  matching\_column: Column common to both the tables.  ***Note****: We can also use RIGHT OUTER JOIN instead of RIGHT JOIN, both are the same.*  https://media.geeksforgeeks.org/wp-content/uploads/20220515095048/join.jpg    **Example Queries(RIGHT JOIN)**:  SELECT Student.NAME,StudentCourse.COURSE\_ID  FROM Student  RIGHT JOIN StudentCourse  ON StudentCourse.ROLL\_NO = Student.ROLL\_NO;  **Output:**  table6 ****D. FULL JOIN**** FULL JOIN creates the result-set by combining results of both LEFT JOIN and RIGHT JOIN. The result-set will contain all the rows from both tables. For the rows for which there is no matching, the result-set will contain *NULL* values.  https://i.stack.imgur.com/3Ll1h.png  **Syntax:**  SELECT table1.column1,table1.column2,table2.column1,....  FROM table1  FULL JOIN table2  ON table1.matching\_column = table2.matching\_column;  table1: First table.  table2: Second table  matching\_column: Column common to both the tables.  **Example Queries(FULL JOIN)**:  SELECT Student.NAME,StudentCourse.COURSE\_ID  FROM Student  FULL JOIN StudentCourse  ON StudentCourse.ROLL\_NO = Student.ROLL\_NO;  **Output:**   | NAME | COURSE\_ID | | --- | --- | | HARSH | 1 | | PRATIK | 2 | | RIYANKA | 2 | | DEEP | 3 | | SAPTARHI | 1 | | DHANRAJ | NULL | | ROHIT | NULL | | NIRAJ | NULL | | NULL | 4 | | NULL | 5 | | NULL | 4 | |
| **5.** | **a** | ACID Properties in DBMS   A **transaction** is a single logical unit of work that accesses and possibly modifies the contents of a database. Transactions access data using read and write operations.  In order to maintain consistency in a database, before and after the transaction, certain properties are followed. These are called **ACID** properties. https://media.geeksforgeeks.org/wp-content/cdn-uploads/20191121102921/ACID-Properties.jpg****Atomicity:**** By this, we mean that either the entire transaction takes place at once or doesn’t happen at all. There is no midway i.e. transactions do not occur partially. Each transaction is considered as one unit and either runs to completion or is not executed at all. It involves the following two operations.  —**Abort**: If a transaction aborts, changes made to the database are not visible.  —**Commit**: If a transaction commits, changes made are visible.  Atomicity is also known as the ‘All or nothing rule’.    Consider the following transaction **T** consisting of **T1** and **T2**: Transfer of 100 from account **X** to account **Y**.  https://media.geeksforgeeks.org/wp-content/uploads/11-6.jpg  If the transaction fails after completion of **T1** but before completion of **T2**.( say, after **write(X)** but before **write(Y)**), then the amount has been deducted from **X** but not added to **Y**. This results in an inconsistent database state. Therefore, the transaction must be executed in its entirety in order to ensure the correctness of the database state. Consistency: This means that integrity constraints must be maintained so that the database is consistent before and after the transaction. It refers to the correctness of a database. Referring to the example above,  The total amount before and after the transaction must be maintained.  Total **before T** occurs = **500 + 200 = 700**.  Total **after T occurs** = **400 + 300 = 700**.  Therefore, the database is **consistent**. Inconsistency occurs in case **T1** completes but **T2** fails. As a result, T is incomplete. Isolation: This property ensures that multiple transactions can occur concurrently without leading to the inconsistency of the database state. Transactions occur independently without interference. Changes occurring in a particular transaction will not be visible to any other transaction until that particular change in that transaction is written to memory or has been committed. This property ensures that the execution of transactions concurrently will result in a state that is equivalent to a state achieved these were executed serially in some order.  Let **X**= 500, **Y** = 500.  Consider two transactions **T** and **T”.**  https://media.geeksforgeeks.org/wp-content/uploads/20210402015259/isolation-300x137.jpg  Suppose **T** has been executed till **Read (Y)** and then **T’’** starts. As a result, interleaving of operations takes place due to which **T’’** reads the correct value of **X** but the incorrect value of **Y** and sum computed by  **T’’: (X+Y = 50, 000+500=50, 500)**  is thus not consistent with the sum at end of the transaction:  **T: (X+Y = 50, 000 + 450 = 50, 450)**.  This results in database inconsistency, due to a loss of 50 units. Hence, transactions must take place in isolation and changes should be visible only after they have been made to the main memory. Durability: This property ensures that once the transaction has completed execution, the updates and modifications to the database are stored in and written to disk and they persist even if a system failure occurs. These updates now become permanent and are stored in non-volatile memory. The effects of the transaction, thus, are never lost.  **Some important points:**   | **Property** | **Responsibility for maintaining properties** | | --- | --- | | Atomicity | Transaction Manager | | Consistency | Application programmer | | Isolation | Concurrency Control Manager | | Durability | Recovery Manager |   The **ACID** properties, in totality, provide a mechanism to ensure the correctness and consistency of a database in a way such that each transaction is a group of operations that acts as a single unit, produces consistent results, acts in isolation from other operations, and updates that it makes are durably stored. |
| **6.** | **a** | A **View** in SQL as a logical subset of data from one or more tables. Views are used to restrict data access. A View contains no data of its own but it is like a window through which data from tables can be viewed or changed. The table on which a View is based is called BASE Tables.  There are 2 types of Views in SQL: Simple View and Complex View. **Simple views** can only contain a single base table. **Complex views** can be constructed on more than one base table. In particular, complex views can contain: join conditions, a group by clause, order by clause.  The key differences between Simple and Complex types of Views are as follows:   |  |  |  | | --- | --- | --- | | **S. No.** | **Simple View** | **Complex View** | | 1. | Contains only one single base table or is created from only one table. | Contains more than one base table or is created from more than one table. | | 2. | We cannot use group functions like MAX(), COUNT(), etc. | We can use group functions. | | 3. | Does not contain groups of data. | It can contain groups of data. | | 4. | DML operations could be performed through a simple view. | DML operations could not always be performed through a complex view. | | 5. | INSERT, DELETE and UPDATE are directly possible on a simple view. | We cannot apply INSERT, DELETE and UPDATE on complex view directly. | | 6. | Simple view does not contain group by, distinct, pseudocolumn like rownum, columns defined by expressions. | It can contain group by, distinct, pseudocolumn like rownum, columns defined by expressions. | | 7. | Does not include NOT NULL columns from base tables. | NOT NULL columns that are not selected by simple view can be included in complex view. | | 8. | In simple view, no need to apply major associations because of only one table. | In complex view, because of multiple tables involved general associations required to be applied such as join condition, group by or a order by clause. | | 9. | **Example:**  CREATE VIEW Employee AS SELECT Empid, Empname FROM Employee WHERE Empid = ‘030314’; | **Example:**  CREATE VIEW EmployeeByDepartment AS  SELECT e.emp\_id, d.dept\_id, e.emp\_name FROM Employee e, Department d WHERE e.dept\_id=d.dept\_id; | |
| **7.** | **a** | **Data Manipulation Operations**  When SQL is used in a host language program, you will need to perform data manipulation. There are several ways that the program can take advantage of SQL DML in CA IDMS.  SQL DML Statements  Use the following SQL statements in data manipulation operations:   * SELECT—To retrieve data * INSERT—To add data * UPDATE—To modify data * DELETE—To delete data * CALL—To invoke an SQL invoked procedure or table procedure.   SQL data manipulation statements provide the following capabilities:   * One statement can manipulate data in many rows * One statement can perform both computation and data manipulation * One statement can retrieve data from many tables   Consequently, you have several options for performing each type of data manipulation. |
| **b** | A **Relational DataBase Management System (RDBMS)** is a software that −   * Enables you to implement a database with tables, columns and indexes. * Guarantees the Referential Integrity between rows of various tables. * Updates the indexes automatically. * Interprets an SQL query and combines information from various tables.   RDBMS Terminology   * **Database** − A database is a collection of tables, with related data. * **Table** − A table is a matrix with data. A table in a database looks like a simple spreadsheet. * **Column** − One column (data element) contains data of one and the same kind, for example the column postcode. * **Row** − A row (= tuple, entry or record) is a group of related data, for example the data of one subscription. * **Redundancy** − Storing data twice, redundantly to make the system faster. * **Primary Key** − A primary key is unique. A key value cannot occur twice in one table. With a key, you can only find one row. * **Foreign Key** − A foreign key is the linking pin between two tables. * **Compound Key** − A compound key (composite key) is a key that consists of multiple columns, because one column is not sufficiently unique. * **Index** − An index in a database resembles an index at the back of a book. * **Referential Integrity** − Referential Integrity makes sure that a foreign key value always points to an existing row. |
| **8.** | **a** | The steps involved in processing a query appear in Figure. The basic steps are: 1. Parsing and translation. 2. Optimization. 3. Evaluation. Before query processing can begin, the system must translate the query into a usable form. A language such as SQL is suitable for human use, but is ill suited to be the system’s internal representation of a query. A more useful internal representation is one based on the extended relational algebra. Thus, the first action the system must take in query processing is to translate a given query into its internal form. This translation process is similar to the work performed by the parser of a compiler. In generating the internal form of the query, the parser checks the syntax of the user’s query, verifies that the relation names appearing in the query are names of the relations in the database, and so on. The system constructs a parse-tree representation of the query, which it then translates into a relational-algebra expression. If the query was expressed in terms of a view, the translation phase also replaces all uses of the view by the relational-algebra expression that defines the view.1 Most compiler texts cover parsing in detail    Given a query, there are generally a variety of methods for computing the answer. For example, we have seen that, in SQL, a query could be expressed in several different ways. Each SQL query can itself be translated into a relationalalgebra expression in one of several ways. Furthermore, the relational-algebra representation of a query specifies only partially how to evaluate a query; there are usually several ways to evaluate relational-algebra expressions. As an illustration, consider the query: select salary from instructor where salary < 75000; This query can be translated into either of the following relational-algebra expressions:    Further, we can execute each relational-algebra operation by one of several different algorithms. For example, to implement the preceding selection, we can search every tuple in instructor to find tuples with salary less than 75000. If a B+-tree index is available on the attribute salary, we can use the index instead to locate the tuples. To specify fully how to evaluate a query, we need not only to provide the relational-algebra expression, but also to annotate it with instructions specifying how to evaluate each operation. Annotations may state the algorithm to be used  for a specific operation, or the particular index or indices to use. A relationalalgebra operation annotated with instructions on how to evaluate it is called an evaluation primitive. A sequence of primitive operations that can be used to evaluate a query is a query-execution plan or query-evaluation plan. Figure 12.2 illustrates an evaluation plan for our example query, in which a particular index (denoted in the figure as “index 1”) is specified for the selection operation. The query-execution engine takes a query-evaluation plan, executes that plan, and returns the answers to the query. |
| **b** | Serializable schedules are the ideal way to ensure consistency, but in our dayto-day lives, we don’t impose such stringent requirements. A Web site offering goods for sale may list an item as being in stock, yet by the time a user selects the item and goes through the checkout process, that item might no longer be available. Viewed from a database perspective, this would be a nonrepeatable read. As another example, consider seat selection for air travel. Assume that a traveler has already booked an itinerary and now is selecting seats for each flight. Many airline Web sites allow the user to step through the various flights and choose a seat, after which the user is asked to confirm the selection. It could be that other travelers are selecting seats or changing their seat selections for the same flights at the same time. The seat availability that the traveler was shown is thus actually changing, but the traveler is shown a snapshot of the seat availability as of when the traveler started the seat selection process. Even if two travelers are selecting seats at the same time, most likely they will select different seats, and if so there would be no real conflict. However, the transactions are not serializable, since each traveler has read data that was subsequently updated by the other traveler, leading to a cycle in the precedence graph. If two travelers performing seat selection concurrently actually selected the same seat, one of them would not be able to get the seat they selected; however, the situation could be easily resolved by asking the traveler to perform the selection again, with updated seat availability information. It is possible to enforce serializability by allowing only one traveler to do seat selection for a particular flight at a time. However, doing so could cause significant delays as travelers would have to wait for their flight to become available for seat selection; in particular a traveler who takes a long time to make a choice could cause serious problems for other travelers. Instead, any such transaction is typically broken up into a part that requires user interaction, and a part that runs exclusively on the database. In the example above, the database transaction would check if the seats chosen by the user are still available, and if so update the seat selection in the database. Serializability is ensured only for the transactions that run on the database, without user interaction. |
| **9.** | **a** | SQL provides a create table like extension to support this task:  **create table temp instructor like instructor;**  The above statement creates a new table temp instructor that has the same schema as instructor. When writing a complex query, it is often useful to store the result of a query as a new table; the table is usually temporary. Two statements are required, one to create the table (with appropriate columns) and the second to insert the query result into the table. SQL:2003 provides a simpler technique to create a table containing the results of a query. For example the following statement creates a table t1 containing the results of a query.  **create table t1 as**  **(select \***  **from instructor**  **where dept name= ’Music’)**  **with data;**  By default, the names and data types of the columns are inferred from the query result. Names can be explicitly given to the columns by listing the column names after the relation name. As defined by the SQL:2003 standard, if the with data clause is omitted, the table is created but not populated with data. However many implementations populate the table with data by default even if the with data clause is omitted.  Note that several implementations support the functionality of create table ... like and create table ... as using different syntax; see the respective system manuals for further details. The above create table ... as statement closely resembles the create view statement and both are defined by using queries. The main difference is that the contents of the table are set when the table is created, whereas the contents of a view always reflect the current query result. |

**\*\*\*\*\*\***