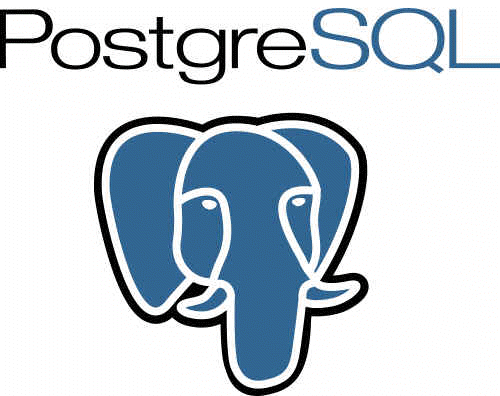
**Includes Solutions to DU DBMS – I Final Exam Questions of  
6 Years (2002-2007)**



**Prepared By**

**Sharafat Ibn Mollah Mosharraf**

**CSE, DU**

**12th Batch (2005-2006)**

DBMS – I

**Special Edition for CSEDU Students Students**

**TOUCH-*N*-PASS EXAM CRAM GUIDE SERIES**

Table of Contents

[Chapter 1: Introduction 1](#_Toc233702954)

[Theories 1](#_Toc233702955)

cHAPTER 2: [Entity-Relationship Model 4](#_Toc233702957)

[Questions and Answers 4](#_Toc233702958)

CHAPTER 3, 4: [Relational Model & SQL 9](#_Toc233702960)

[Points to be Remembered 9](#_Toc233702961)

[The Trick of Writing RA Expressions for Complex Queries 9](#_Toc233702962)

[Complete Concepts Problem 10](#_Toc233702963)

[General Structure of Query Statements 16](#_Toc233702964)

[Theories 18](#_Toc233702965)

CHAPTER 6: [Integrity & Security 29](#_Toc233702967)

[Questions and Answers 29](#_Toc233702968)

CHAPTER 7: [Relational Database Design 33](#_Toc233702970)

[Concepts 33](#_Toc233702971)

[Questions and Answers 33](#_Toc233702972)

CHAPTER 11: [Storage & File Structure 40](#_Toc233702974)

[Theories 40](#_Toc233702975)

cHAPTER 12: [Indexing and Hashing 43](#_Toc233702977)

[Concepts 43](#_Toc233702978)

[Questions and Answers 45](#_Toc233702979)

# Chapter 1

# Introduction

## Theories

|  |  |
| --- | --- |
| **1.1** | **What is DBMS? [*In-course 2007*; *2007*. *Marks: 1*]**  A database management system (DBMS) is a collection of interrelated data and a set of programs to access those data. The collection of data, usually referred to a database, contains information relevant to an enterprise. |
| **1.2** | **Mention some of the areas for database applications. *[In-course 1, 2005. Marks: 2]***   |  |  |  | | --- | --- | --- | | 1. Banking | 2. Airlines /Railways/Road Transport | 3. Universities | | 4. Credit Card Transaction | 5. Telecommunication | 6. Finance | | 7. Sales | 8. On-line Retailers | 9. Manufacturing | | 10. Human Resources | 11. Internet |  | |
| **1.3** | **List four significant differences between a file-processing system and a DBMS.**  Some main differences between a database management system and a file-processing system are:  • Both systems contain a collection of data and a set of programs which access that data. A database management system coordinates both the physical and the logical access to the data, whereas a file-processing system coordinates only the physical access.  • A database management system reduces the amount of data duplication by ensuring that a physical piece of data is available to all programs authorized to have access to it, whereas data written by one program in a file-processing system may not be readable by another program.  • A database management system is designed to allow flexible access to data (i.e., queries), whereas a file-processing system is designed to allow predetermined access to data (i.e., compiled programs).  • A database management system is designed to coordinate multiple users accessing the same data at the same time. A file-processing system is usually designed to allow one or more programs to access different data files at the same time. In a file-processing system, a file can be accessed by two programs concurrently only if both programs have read-only access to the file. |
| **1.4** | **What are the disadvantages of DBMS?**  Two disadvantages associated with database systems are listed below.  **a.** Setup of the database system requires more knowledge, money, skills, and time.  **b.** The complexity of the database may result in poor performance. |
| **1.5** | **Explain the difference between physical and logical data independence.**  Physical data independence is the ability to modify the physical scheme without making it necessary to rewrite application programs. Such modifications include changing from unblocked to blocked record storage, or from sequential to random access files.  Logical data independence is the ability to modify the conceptual scheme without making it necessary to rewrite application programs. Such a modification might be adding a field to a record; an application program’s view hides this change from the program. |
| **1.6** | **What are five main functions of a database administrator?**  Five main functions of a database administrator are:   1. To create the scheme definition 2. To define the storage structure and access methods 3. To modify the scheme and/or physical organization when necessary 4. To grant authorization for data access 5. To specify integrity constraints |
| **1.7** | **Classify database users. *[2004. Marks: 2]***  Database users are differentiated by the way they expect to interact with the system. There are four types of database users:   1. **Naive users –** are unsophisticated users who interact with the system by invoking one of the permanent application programs that have been written previously. 2. **Application programmers –** are computer professionals who write application programs. 3. **Sophisticated users –** interact with the system without writing programs. Instead, they form their requests in a database query language. 4. **Specialized users –** are sophisticated users who write specialized database applications that do not fit into the traditional data processing framework. |
| **1.8** | **What are the jobs of a DBA? [*In-course 2007*; *2007; 2004*. *Marks: 3*]**  The functions of a database administrator (DBA) include:   1. **Schema definition:** The DBA creates the original database schema by executing a set of data definition statements in the DDL. 2. **Storage structure and access method definition:** File organization (sequential, heap, hash, B+ tree), organization of records in a file (fixed length or variable length), index definition (ordered index, hash index). 3. **Schema and physical-organization modification:** The DBA carries out changes to the schema and physical organization to reflect the changing needs of the organization, or to alter the physical organization to improve the performance. 4. **Granting of authorization for data access:** By granting different types of authorization, the DBA can regulate which parts of the database various users can access. 5. **Specifying integrity constraints:** The DBA implements key declaration (primary key, foreign key), trigger, assertion, business rules of the organization. 6. **Acting as liaison with users.** 7. **Routine maintenance:** 8. Periodically backing up the database, either onto tapes or remote servers, to prevent loss of data in case of disasters. 9. Ensuring that enough disk space is available for normal operations and upgrading disk space as required. 10. Monitoring jobs running on the database and ensuring better performance. |
| **1.9** | **What can be done using DML? What are the classes of DML? *[In-course 1, 2005. Marks: 4]***  DML is a language that enables users to access or manipulate data as organized by appropriate data model. The types of accesses are:   1. The retrieval of information stored in the database – Query 2. The insertion of new information into the database – insert 3. The deletion of information from the database – delete 4. The modification of information stored in the database – update   **Classes of DML:**  There are basically two types:   1. **Procedural DMLs –** user specifies what data are required and how to get or compute the data. E.g. Relational Algebra. 2. **Nonprocedural / Declarative DMLs –** user specifies what data are required without specifying how to get or compute the data. E.g. SQL. |
| **1.10** | **How do you classify query languages? Give examples of each type. [*In-course 1, 2008; 2006*, *Marks: 2*]**  Query languages can be classified into two categories:   1. **Procedural:** Relational Algebra 2. **Non-procedural:** Tuple Relational Calculus, Domain Relational Calculus |
| **1.11** | **What are the differences between schema and instance? *[In-course 2007, Marks: 2]***   |  |  | | --- | --- | | **Schema** | **Instance** | | 1. The overall design of a database is called the database schema. | 1. The collection of information stored in a database at a particular moment is called an instance of the database. | | 1. A relation schema is a type definition. | 1. A relation is an instance of a schema. | | 1. Schemas are changed infrequently, if at all. | 1. Instances are changed frequently. | | 1. This corresponds to the variable declaration (with type definition) of a programming language. | 1. The values of the variable in a program at a point in time correspond to an instance of the database schema. | |
| **1.12** | **What is data dictionary? *[2004. Marks: 1]***  A data dictionary contains metadata (data about data). The data dictionary is considered to be a special type of table, which can only be accessed and updated by the database system itself (not a regular user). A database system consults the data dictionary before reading or modifying actual data. |
| **1.13** | **What are the components of query processor? *[In-course 1, 2005. Marks: 3]***  The components of query processor are:  **1. DDL interpreter:** Interprets DDL statements and records the definitions in the data dictionary.  **2. DML compiler:** Translates DML statements in a query language into an evaluation plan consisting of low-level instructions that the query evaluation engine understands.  **3. Query evaluation engine:** Executes low-level instructions generated by the DML compiler. |

# Chapter 2

# Entity-Relationship Model

## Questions and Answers

|  |  |
| --- | --- |
| **2.1** | **Why E-R model is used for data manipulation? *[2002. Marks: 2]***  E-R model is used for data manipulation because:   1. It can express the overall logical structure of a database graphically. 2. E-R diagrams are simple and clear. |
| **2.2** | **What is the basic difference between E-R diagram and Schema diagram? *[In-course 1, 2005. Marks: 1]***  The basic difference between E-R diagram and schema diagram is that E-R diagrams do not show foreign key attributes explicitly, whereas schema diagrams show them explicitly. |
| **2.3** | **Define the following:**   1. **Composite attribute *[2007, 2003. Marks: 1]*** 2. **Multivalued attribute *[2007, 2003. Marks: 1]*** 3. **Derived attribute *[2007, 2003; In-course 1, 2005. Marks: 1]***   ***Composite attributes:***  Attributes that can be divided into subparts are called composite attributes.  For example, the composite attribute *address* can be divided into attributes *street-number*, *street-name* and *apartment-number*.  ***Multivalued attributes:***  Attributes that have multiple values for a particular entity are called multivalued attributes.  For example, an *employee* may have multiple telephone numbers. So, the attribute *telephone-no* is a multivalued attribute.  ***Derived attribute:***  If the value of an attribute can be derived from the values of other related attributes or entities, then that attribute is called a derived attribute.  For example, if an entity set *employee* has two attributes *date-of-birth* and *age*, then the attribute *age* is a derived attribute as it can be derived from the attribute *date-of-birth*. |
| **2.4** | **Explain the difference between a weak entity set and a strong entity set. *[In-course 2, 2007; 2005; 2003. Marks: 2]***  A strong entity set has a primary key. All tuples in the set are distinguishable by that key. A weak entity set has no primary key unless attributes of the strong entity set on which it depends are included. Tuples in a weak entity set are partitioned according to their relationship with tuples in a strong entity set. Tuples within each partition are distinguishable by a discriminator, which is a set of attributes. |
| **2.5** | **Show with an example the association between a weak entity set and a strong entity set using E-R diagram. *[In-course 2, 2007; 2003. Marks: 1]***  2.PNG |
| **2.6** | **We can convert any weak entity set to a strong entity set by simply adding appropriate attributes. Why, then, do we have weak entity sets?**  We have weak entities for several reasons:   * We want to avoid the data duplication and consequent possible inconsistencies caused by duplicating the key of the strong entity. * Weak entities reflect the logical structure of an entity being dependent on another entity. * Weak entities can be deleted automatically when their strong entity is deleted. * Weak entities can be stored physically with their strong entities. |
| **2.7** | **What is the purpose of constraints in database? *[2002. Marks: 2]***  The purposes of constraints in database are:   1. To implement data check. 2. To centralize and simplify the database, so to make the development of database applications easier and more reliable. |
| **2.8** | **What are the constraints used in E-R model? *[In-course 2, 2007. Marks: 1]***  Constraints used in E-R model:   1. Cardinality Constraints 2. Participation Constraints 3. Key Constraints |
| **2.9** | **What participation constraints are used in E-R model? *[2006. Marks: 1]***  **OR, Explain the participation constraints in E-R model. *[In-course 2, 2007. Marks: 1]***  **OR, Explain with example the participation constraints in E-R model. *[2003. Marks: 3]***  The participation constraints used in E-R model are:   1. Total 2. Partial   The participation of an entity set *E* in a relationship set *R* is said to be *total* if every entity in *E* participates in at least one relationship in *R*. If only some entities in *E* participate in relationships in *R*, the participation of entity set *E* in relationship *R* is said to be *partial*.  For example, we expect every *loan* entity to be related to at least one *customer* through the *borrower* relationship. Therefore the participation of *loan* in the relationship set *borrower* is total.  In contrast, an individual can be a bank customer whether or not she has a loan with the bank. Hence, it is possible that only some of the *customer* entities are related to the *loan* entity set through the *borrower* relationship, and the participation of *customer* in the *borrower* relationship set is therefore partial. |
| **2.10** | **Explain the distinction between total and partial constraints.**  In a total design constraint, each higher-level entity must belong to a lower-level entity set. The same need not be true in a partial design constraint. For instance, some employees may belong to no work-team. |
| **2.11** | **Let *R* be binary relationship between *A* and *B* entity sets.**   1. **Show the mapping cardinalities using E-R diagrams. *[In-course 1, 2005. Marks: 2]*** 2. **How primary keys can be defined for the relationship set *R* for different mapping cardinalities? *[2006. Marks: 2]*** 3. **How can you combine the tables (if possible) for different mapping cardinalities? *[2004. Marks: 3]***   **1. *Mapping Cardinalities:***  *One-to-One*  A  B  R  A  B  R  *Many-to-Many*  A  B  R  A  B  R  *Many-to-One*  *One-to-Many*  **2. *Primary keys for R:***   1. One-to-One: PK*A* or PK*B* [PK*A* means *Primary Key* of the entity set *A*] 2. One-to-Many: PK*B* 3. Many-to-One: PK*A* 4. Many-to-Many: PK*A* and PK*B*   **3. *Combination of tables for different mapping cardinalities:***   1. One-to-One: – 2. One-to-Many: B and AB 3. Many-to-One: A and AB 4. Many-to-Many: – |
| **2.12** | **Draw the symbols of *identifying relationship set*, *discriminator of weak entity set*, *derived attribute* and *multivalued attribute* used in E-R model. *[2004. Marks: 2]***  R  *Identifying Relationship Set*  *Discriminator of*  *Weak Entity Set*  *Derived Attribute*  *Multivalued Attribute* |
| **2.13** | **Draw the E-R diagram for the following relation schemas: *[In-course 2, 2007. Marks: 1.5]***  ***Worker (worker\_id, worker\_name, hourly\_rate, skill\_type, supervisor\_id)***  ***Assignment (worker\_id, building\_id, start\_date, num\_days)***  ***Building (building\_id, address, building\_type)***  Worker  Building  Assignment |
| **2.14** | **Give the E-R diagram for the following database: *[In-course 1, 2005; 2003. Marks: 2]***  ***person (driver-id, name, address)***  ***car (license, model, year)***  ***accident (report-no, date, location)***  ***owns (driver-id, license)***  ***participated (driver-id, license, report-no, damage-amount)***  untitled.PNG |
| **2.15** | **What will be the tabular representation of the following E-R diagram? *[In-course 2, 2007; In-course 1, 2005. Marks: 2]***  **1.PNG**  *customer (customer-id, first-name, middle-initial, last-name, date-of-birth, street-number, street-name, apartment-number, city, state, zip-code)*  *customer-phone (phone-number, customer-id)* |
| **2.16** | **What will be the schema representation of the following E-R diagram? *[2007. Marks: 2]***  **2.PNG**  *loan (loan-number, amount)*  *loan-payment (payment-number, loan-number, payment-date, payment-amount)* |
| **2.17** | **We are interested to make a database for Railway Reservation System. (We will limit only for inter-city train between Dhaka and Sylhet.) Generally, a passenger takes flight of inter-city train that operates between Dhaka-Chittagong-Dhaka and Dhaka-Sylhet-Dhaka. Each train is identified by an ID and total seating capacity. Each train is assigned a leg instance (an instance of a flight on a specific date) for which we will keep number of compartments, number of available seats and date. Passenger reserves seat of each leg instance. For seat, we will keep seat ID and type. Each leg instance departs from a terminal and arrives to a terminal. We will keep departure time and arrival time; and for terminal, we will store its ID, name and city. For each passenger, we will store name, phone and age.**   1. **Develop a complete E-R diagram (including cardinalities). Make reasonable assumptions during your development phases, if needed and state them clearly.** 2. **Translate the E-R diagram into relations (tables). *[2005. Marks: 5 + 3]***   Terminal  Passenger  Assigned  Leg Instance  Reserves  Seat  Has  Train  Departs\_From  Arrives\_At  1  N  N  1  N  N  1  1  1  1  *Train* (*id, total\_seating\_capacity*)  *Terminal* (*id, name, city*)  *Leg\_Instance* (*date, train\_id, no\_of\_compartments, no\_of\_available\_seats*)  *Departure* (*terminal\_id, date, train\_id, departure\_time*)  *Arrival* (*terminal\_id, date, train\_id, arrival\_time*)  *Reservation* (*seat\_id, date, train\_id, seat\_type, passenger\_id, passenger\_name, age, phone*) |
| **2.18** | **A database is being constructed to keep track of the teams and games of a football league. A team has a number of players. For the team, we are interested to store team id, team name, address, date established, name of manager, and name of coach. For the player, we will store player id in team, date of birth, date joined, position etc. Each team plays games against other team in a round robin fashion. For each game, we will store game id, date held, score and attendance (an attribute to designate whether the participating teams have attended the game). Games are generally taking place at various stadiums of the country. For each stadium, we will keep its size, name and location.**   1. **Develop a complete E-R diagram (including cardinalities). Make reasonable assumptions during your development phases, if needed and state them clearly.** 2. **Translate the E-R diagram into relations (tables). *[2003. Marks: 6 + 4]***   Player  Team  Plays\_For  Game  Participates\_In  Stadium  Is\_Held\_In  1  N  N  1  2  1  *Team* (*team\_id, team\_name, date\_established, address, manager, coach*)  *Player* (*player\_id, team\_id, date\_of\_birth, date\_joined, position*)  *Stadium (name, location, size*)  *Game* (*id, date\_held, attendance, score, stadium\_name*)  *Team\_Game* (*team\_id, game\_id*) |

# 

# Chapter 3, 4

# Relational Model & SQL

## Points to be Remembered

|  |  |
| --- | --- |
| **3.1** | The order in which tuples or attributes appear in a relation is irrelevant, since a relation is a set of tuples – sorted or unsorted does not mater. |
| **3.2** | To represent string values, in RA, double quotes (" ") are used, whereas in SQL, single-quotes (' ') are used. |
| **3.3** | Note the difference in representation of the following operators in SQL and RA:   |  |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | --- | | **SQL** | >= | <= | < > | and | or | not | | **RA** | ≥ | ≤ | ≠ | ∧ | ∨ | ¬ | |
| **3.4** | In the projection operation, duplicate rows are eliminated in RA (as RA considers relations as *sets*); whereas SQL retains duplicate rows by default (since duplicate elimination is time consuming). To force the elimination of duplicate, a keyword distinct is inserted after select. |
| **3.5** | SQL does not allow the use of distinct with count(\*)(however, it can be used with count for a single attribute, e.g. count(distinct A)). distinct can be used with min and max, but result does not change. |
| **3.6** | If a where clause and having clause appear in the same query, SQL applies the predicate in the where clause first. Tuples satisfying the where predicate are then placed into groups by the group by clause. SQL then applies the having clause, if it is present, to each groups; it removes the groups that do not satisfy the having clause predicate. The select clause uses the remaining groups to generate the tuples of the result relation. |
| **3.7** | The input to sum and avg must be a collection of numbers, but the other aggregate functions (count, min and max) can operate on collection of non-numeric data types, such as string, as well. |
| **3.8** | Aggregate functions cannot be composed in SQL. Thus, we cannot use **max**(**avg**(…)). |
| **3.9** | Every *derived* table must have its own alias.  **Wrong:** **select** \* **from** **(select** x **from** y **where** p **=** q**)** **where** a **=** b;  **Right:** **select** \* **from** **(select** x **from** y **where** p **=** q**)** **as** new\_table **where** a **=** b; |
| **3.10** | The use of a null value in arithmetic and comparison operations causes several complications. The result of any arithmetic expression involving null returns null. So 5 + null returns null.  Any comparison with null (other than is null and is not null) returns unknown. So, 5 < null or null <> null or null = null returns unknown. |
| **3.11** | All aggregate functions except **count**(\*) ignore tuples with null values on the aggregated attributes. |
| **3.12** | If we use an arithmetic expression in the select clause, the resultant attribute does not have a name. |

## The Trick of Writing RA Expressions for Complex Queries

If you find that writing the RA expression of a query is getting difficult, then think of the query in terms of views. Views create multiple tables and *multiple* *simple* queries to perform a *single complex* query. As you’re able to write the RA expressions for simple queries, you’ll now be able to solve the complex queries.

For applications of this trick, see the following queries:

*Complete Concepts Problem* – query no. 20, 21, 23.

*Theory 4.10* – query no. 4

*Theory 4.11* – query no. 2 and 4

## Complete Concepts Problem

**Consider the database schema below:**

*employee (ename, street, city)*

*works (ename, cname, salary, jdate)*

*company (cname, city)*

*manages (ename, mname)*

***Note:* A manager is also an employee of a company.**

**Give *SQL* and *RA* expressions for the following queries:**

|  |  |  |
| --- | --- | --- |
| **Imp. Level** | **Diff.**  **Level** | **Queries** |
| 0 | 0 | 1. Find the names of all employees who work for First Bank Corporation. |
| 0 | 1 | 1. Find the names and cities of residence of all employees who work for First Bank Corporation. |
| 5 | 2 | 1. Find the names, street address, and cities of residence of all employees who work for First Bank Corporation and earn more than Tk. 30000. |
| 5 | 2 | 1. Find names, street addresses and cities of residence of all employees who work under manager Sabbir and who joined before January 01, 2009. |
| 1 | 1 | 1. Find the names of all employees in this database who live in the same city as the company for which they work. |
| 5 | 5 | 1. Find the names of all employees who live in the same city and on the same street as do their managers. |
| 3 | 3 | 1. Find the names of the employees living in the same city where Rahim is residing. |
| 0 | 0 | 1. Find the names of all employees in this database who do not work for First Bank Corporation. |
| 3 | 5 | 1. Find the names of all employees who earn more than ***every*** employee of Small Bank Corporation. |
| 5 | 5 | 1. Find the names of all employees who earn more than ***any*** employee of Small Bank Corporation. |
| 2 | 2 | 1. Assume the companies may be located in several cities. Find all companies located in every city in which Small Bank Corporation is located. |
| 5 | 4 | 1. Give all employees of First Bank Corporation a 10 percent salary raise. |
| 5 | 5 | 1. Give all managers in the database a 10% salary raise. |
| 1 | 5 | 1. Give all managers in this database a 10 percent salary raise, unless the salary would be greater than Tk.100,000. In such cases, give only a 3 percent raise. |
| 5 | 5 | 1. Increase the salary of employees by 10% for the companies those are located in Bogra. |
| 2 | 3 | 1. Modify the database so that Rahim now lives in Bhola. |
| 1 | 1 | 1. Delete all tuples in the *works* relation for employees of Small Bank Corporation. |
| 5 | 3 | 1. Delete records from *works* that contain employees living in Rajshahi. |
| 4 | 2 | 1. Display the average salary of each company except Square Pharma. |
| 5 | 4 | 1. Find the company with the most employees. |
| 4 | 4 | 1. Find the company that has the smallest payroll. |
| 4 | 3 | 1. Find the company with payroll less than Tk. 100000. |
| 3 | 5 | 1. Find those companies whose employees earn a higher salary, on average, than the average salary of Small Bank Corporation. |

**Note:** the ***Imp. Level*** column in the above table means how much important that query is for the exam (range: 0 – 5, where 0 means *not important at all* and 5 means *most important*); and the ***Diff. Level*** field means how difficult the problem is (range: 0-5, where 0 means *very easy* and 5 means *very difficult*).

**Sample Data**

|  |  |
| --- | --- |
| **Table Name** | **Data** |
| ***employee*** | |  |  |  | | --- | --- | --- | | ename | street | city | | Barkat | *x* | Bogra | | Jabbar | *x* | Comilla | | Jubayer | *u* | Faridpur | | Najmun Nahar | *y* | Sylhet | | Oronno | *z* | Dhaka | | Rafique | *z* | Rajshahi | | Rahim | *w* | Dhaka | | Sabbir | *v* | Chittagong | | Salam | *y* | Comilla | | Sharafat | *w* | Dhaka | |
| ***works*** | |  |  |  |  | | --- | --- | --- | --- | | ename | cname | salary | jdate | | Rahim | First Bank Corporation | 50000 | 2008-01-01 | | Barkat | First Bank Corporation | 40000 | 2007-01-01 | | Salam | First Bank Corporation | 60000 | 2009-07-01 | | Rafique | Small Bank Corporation | 30000 | 2009-06-08 | | Sharafat | First Bank Corporation | 80000 | 2005-06-01 | | Jabbar | Small Bank Corporation | 10000 | 2009-06-05 | | Najmun Nahar | Small Bank Corporation | 20000 | 2009-06-30 | | Oronno | The ONE Limited | 50000 | 2007-06-01 | | Jubayer | Square Pharma | 15000 | 2008-01-01 | | Sabbir | Vegabond Company | 100000 | 2001-01-01 | |
| ***company*** | |  |  | | --- | --- | | cname | city | | Anonymous IT | Chittagong | | Dream Tech | Chittagong | | First Bank Corporation | Dhaka | | JONS IT (Pvt.) Limited | Sylhet | | Small Bank Corporation | Dhaka | | Square Pharma | Bogra | | The ONE Limited | Dhaka | | Unique Softs | Dhaka | | Unknown Systems | Rajshahi | | Vegabond Company | Bogra | |
| ***manages*** | |  |  | | --- | --- | | ename | mname | | Rahim | Sharafat | | Barkat | Sharafat | | Salam | Sharafat | | Rafique | Oronno | | Jabbar | Oronno | | Najmun Nahar | Sabbir | | Jubayer | Sabbir | |

*employee (ename, street, city)*

*works (ename, cname, salary, jdate)*

*company (cname, city)*

*manages (ename, mname)*

|  |
| --- |
| 1. **Find the names of all employees who work for First Bank Corporation.**   **SQL:** **select** ename **from** works **where** cname **=** 'First Bank Corporation';  **RA: П** ename (**σ** cname = "First Bank Corporation" (works)) |
| 1. **Find the names and cities of residence of all employees who work for First Bank Corporation.**   **SQL:** **select** ename**,** city **from** employee **natural join** works  **where** cname **=** 'First Bank Corporation';  **RA: П** ename, city (**σ** cname = "First Bank Corporation" (employee **⋈** works)) |
| 1. **Find the names, street address, and cities of residence of all employees who work for First Bank Corporation and earn more than Tk. 30000.**   **SQL:** **select** ename**,** street**,** city **from** employee **natural join** works  **where** cname **=** 'First Bank Corporation' **and** salary > 30000;  **RA: П** ename, street, city (**σ** cname = "First Bank Corporation" **∧** salary > 30000 (employee **⋈** works)) |
| 1. **Find names, street addresses and cities of residence of all employees who work under manager Sabbir and who joined before January 01, 2009.**   **SQL:** **select** ename**,** street**,** city  **from** employee **natural join** works **natural join** manages  **where** mname **=** 'Sabbir' **and** jdate **<** '01**-**JAN**-**09';  **RA: П** ename, street, city (**σ** mname = "Sabbir" **∧** jdate < "01-jan-09" (employee **⋈** works **⋈** manages)) |
| 1. **Find the names of all employees in this database who live in the same city as the company for which they work.**   **SQL:** **select** ename **from** employee **natural join** works **natural join** company;  **RA: П** ename (employee **⋈** works **⋈** company) |
| 1. **Find the names of all employees who live in the same city and on the same street as do their managers.**   **SQL:** **select** employee**.**ename **from** employee **natural join** manages**,** employee **as** emp  **where** mname **=** emp**.**ename **and** employee**.**street **=** emp**.**street **and** employee**.**city **=** emp**.**city;  **RA: П** employee.ename  (**σ** mname = emp.ename **∧** employee.street = emp.street **∧** employee.city = emp.city (employee **⋈** manages **×** **ρ** emp (employee))) |
| 1. **Find the names of the employees living in the same city where Rahim is residing.**   **SQL:** **select** ename **from** employee **where** city **=** (  **select** city **from** employee **where** ename **=** 'Rahim'  );  **RA:** t ← **П** city (**σ** ename = "Rahim" (employee))  **П** ename (employee **⋈** t) |
| 1. **Find the names of all employees in this database who do not work for First Bank Corporation.**   **SQL:** **select** ename **from** works **where** cname **<>** 'First Bank Corporation';  **RA: П** ename (**σ** cname ≠ "First Bank Corporation" (works)) |

*employee (ename, street, city)*

*works (ename, cname, salary, jdate)*

*company (cname, city)*

*manages (ename, mname)*

|  |
| --- |
| 1. **Find the names of all employees who earn more than *every* employee of Small Bank Corporation.**   **SQL:** **select** ename **from** works **where** salary **>** (  **select** **max(**salary**)** **from** works **where** cname **=** 'Small Bank Corporation'  );  **RA:** t ← **G** **max(**salary**) as** max\_salary (**σ** cname = "Small Bank Corporation" (works))  **П** ename (**σ** salary > max\_salary (works **×** t))  ***OR,*** t1 ← **П** works.salary (**σ** works.salary < w.salary and w.cname = "Small Bank Corporation" (works **×** **ρ** w (works)))  t2 ← **П** salary(**σ** w.cname = "Small Bank Corporation" (works)) **–** t1  **П** ename (**σ** works.salary > t2.salary (works **×** t2)) |
| 1. **Find the names of all employees who earn more than *any* employee of Small Bank Corporation.**   **SQL:** **select** ename **from** works **where** salary **>** (  **select** **min(**salary**)** **from** works **where** cname **=** 'Small Bank Corporation'  );  **RA:** t ← **G** **min(**salary**) as** min\_salary (**σ** cname = "Small Bank Corporation" (works))  **П** ename (**σ** salary > min\_salary (works **×** t))  ***OR,*** t1 ← **П** works.salary (**σ** works.salary > w.salary and w.cname = "Small Bank Corporation" (works **×** **ρ** w (works)))  t2 ← **П** salary(**σ** w.cname = "Small Bank Corporation" (works)) **–** t1  **П** ename (**σ** works.salary > t2.salary (works **×** t2)) |
| 1. **Assume the companies may be located in several cities. Find all companies located in every city in which Small Bank Corporation is located.**   **SQL: select** cname **from** company **where** city **in (**  **select** city **from** company **where** cname **=** 'Small Bank Corporation'  **)**;  **RA:** city ← **П** city(**σ** cname = "Small Bank Corporation" (company))  **П** cname(company **⋈** city)  ***OR,* П** cname(company ÷ (**П** city(**σ** cname = "Small Bank Corporation" (company)))) |
| 1. **Give all employees of First Bank Corporation a 10 percent salary raise.**   **SQL:** **update** works **set** salary **=** salary \* 1.1 **where** cname **=** 'First Bank Corporation';  **RA:** t ← **П** ename, cname, salary \* 1.1, jdate (**σ** cname = "First Bank Corporation" (works))  works ← t **∪** (works – **σ** cname = "First Bank Corporation" (works)) |
| 1. **Give all managers in the database a 10% salary raise.**   **SQL:** **update** works **set** salary **=** salary \* 1.1 **where** ename **in** (  **select distinct** mname **from** manages  );  **RA:** t1 ← **П** works.ename, cname, salary, jdate (**σ** works.ename = mname (works **×** manages))  t2 ← **П** works.ename, cname, salary \* 1.1, jdate (t1)  works ← (works – t1) **∪** t2 |

*employee (ename, street, city)*

*works (ename, cname, salary, jdate)*

*company (cname, city)*

*manages (ename, mname)*

|  |
| --- |
| 1. **Give all managers in this database a 10 percent salary raise, unless the salary would be greater than Tk.100,000. In such cases, give only a 3 percent raise.**   **SQL:** **update** works **set** salary **=** **case**  **when** salary **\*** 1.1 **>** 100000 **then** salary \* 1.03  **else** salary **\*** 1.1  **end**  **where** ename **in** (  **select distinct** mname **from** manages  );  **RA:** t1 ← **П** works.ename, cname, salary, jdate (**σ** works.ename = mname (works **×** manages))  t2 ← **П** works.ename, cname, salary \* 1.03, jdate (**σ** t1.salary \* 1.1 > 100000 (t1))  t2 ← t2 **∪** (**П** works.ename, cname, salary \* 1.1, jdate (**σ** t1.salary \* 1.1 ≤ 100000 (t1)))  works ← (works – t1) **∪** t2 |
| 1. **Increase the salary of employees by 10% for the companies those are located in Bogra.**   **SQL:** **update** works **set** salary **=** salary \* 1.1 **where** cname **in** (  **select** cname **from** company **where** city **=** 'Bogra'  );  **RA:** t1 ← **П** ename, cname, salary, jdate (**σ** city = "Bogra" (works **⋈** company))  t2 ← **П** ename, cname, salary \* 1.1, jdate (t1)  works ← (works – t1) **∪** t2 |
| 1. **Modify the database so that Rahim now lives in Bhola.**   **SQL:** **update** employee **set** city **=** 'Bhola' **where** ename **=** 'Rahim';  **RA:** t ← **П** ename, street, "Bhola" (**σ** ename = "Rahim" (employee))  works ← (works – (**σ** ename = "Rahim" (employee))) **∪** t |
| 1. **Delete all tuples in the *works* relation for employees of Small Bank Corporation.**   **SQL:** **delete from** works **where** cname **=** 'Small Bank Corporation';  **RA:** works ← works – (**σ** cname = "Small Bank Corporation" (works))) |
| 1. **Delete records from *works* that contain employees living in Rajshahi.**   **SQL:** **delete from** works **where** ename **in** (  **select** ename **from** employee **where** city **=** 'Rajshahi'  );  **RA:** t ← **П** ename (**σ** city = "Rajshahi" (employee)))  works ← works – **П** ename, cname, salary, jdate (works **⋈** t) |
| 1. **Display the average salary of each company except Square Pharma.**   **SQL:** **select** cname**, avg(**salary**) from** works **where** cname **<>** 'Square Pharma' **group by** cname;  **RA:** cname**G** **avg(**salary**)** (**σ** cname ≠ "Square Pharma" (works)) |

*employee (ename, street, city)*

*works (ename, cname, salary, jdate)*

*company (cname, city)*

*manages (ename, mname)*

|  |
| --- |
| 1. **Find the company with the most employees.**   **SQL:** **select** cname**, count(distinct** ename**) from** works **group by** cname  **having count(distinct** ename**) >= all (**  **select count(distinct** ename**) from** works **group by** cname  **)**;  **RA:** t1 ← cname**G** **count(**ename**) as** num\_employees (works)  t2 ← **G max(**num\_employees**) as** num\_employees (t1)  **П** cname (t1 **⋈** t2) |
| 1. **Find the company that has the smallest payroll[[1]](#footnote-2). [*Similar to query 20*]**   **SQL:** **select** cname**, sum(**salary**) from** works **group by** cname  **having sum(**salary**) <= all (**  **select sum(**salary**) from** works **group by** cname  **)**;  **RA:** t1 ← cname**G** **sum(**salary**) as** payroll (works)  t2 ← **G min(**payroll**) as** payroll (t1)  **П** cname (t1 **⋈** t2) |
| 1. **Find the company with payroll less than Tk. 100000.**   **SQL:** **select** cname**, sum(**salary**) from** works **group by** cname **having sum(**salary**)** < 100000**;**  **RA:** t ← cname**G** **sum(**salary**)** (works)  **П** cname (**σ** payroll < 100000 (**ρ** c\_payroll (cname, payroll) (t))) |
| 1. **Find those companies whose employees earn a higher salary, on average, than the average salary of Small Bank Corporation.**   **SQL:** **select** cname **from** works **group by** cname  **having** **avg(**salary**)** > (  **select avg(**salary**)**  **from** works  **where** cname = 'Small Bank Corporation'  );  **RA:** t1 ← cname**G** **avg(**salary**)** (works)  t2 ← **σ** cname = "Small Bank Corporation" (t1)  **П** t3.cname (**σ** t3.avg\_salary > small-bank.avg\_salary (**ρ** t3 (cname, avg\_salary) (t1) **×** **ρ** small-bank (cname, avg\_salary) (t2))) |

## General Structure of Query Statements

**Legend:**

Choose *either* the statement *above*   
 the line *or* the statement *below*   
 the line at a time

**[***optional***]** You can use it or leave it

**,,,** Comma-separated list

**General Structure of CREATE TABLE statement:**

**CREATE TABLE** ***table-name*** **(**

**);**

***column\_data\_types*:**

1. **CHAR (***number\_of\_characters***)** *example*: CHAR(30)
2. **VARCHAR (***maximum\_number\_of\_characters***)** example: VARCHAR(255)
3. **INTEGER (***number\_of\_digits***)** *example:* INTEGER(10)
4. **(***total\_number\_of\_digits\_including\_decimals***,** *number\_of\_decimal\_digits***)** *example:* DECIMAL(5, 2) [*for 999.99*]
5. **DATE**
6. **TIME**
7. **DATETIME**

**Example of CREATE TABLE statement:**

**create table** account **(**

account\_no **char**(5),

branch\_name **varchar**(15),

balance **number**(10,2)**not null**,

**constraint** a\_pk **primary key**(account\_no),

**constraint** a\_fk **foreign key** (branch\_name) **references** branch(branch\_name),

**constraint** a\_chk1 **check** (balance>=0),

**constraint** a\_chk2 **check** (account\_no **like** 'A-%')

**);**

**General Structure of DROP TABLE statement:**

**DROP TABLE** *table-name***;**

**General Structure of INSERT statement:**

**INSERT INTO** *table-name* [**(***column-names*,,,**)**] **VALUES** [**(***values*,,,**)**] **;**

**Examples of INSERT statement:**

**insert into** account **values** **(**'a-101'**,** 'downtown'**,** 500**);**

**insert into** account **(**account\_no**,** branch\_name**,** balance**)**

**values** **(**'a-101'**,** 'downtown'**,** 500**);**

**General Structure of SELECT statement:**

**SELECT** [**DISTINCT**]

**FROM**

[**WHERE** *expression*]

[**GROUP BY** *column-name\_or\_alias\_or\_function*]

[**HAVING** *expression*]

[**ORDER BY** *column-name\_or\_alias\_or\_function* ]

[**LIMIT** ]**;**

**Example of SELECT statement:**

**select** account\_no**,** **avg(**balance**)** **as** average\_balance

**from** account **left outer join** depositor **using** **(**account\_no**)**

**where** balance **>** 1000

**group by** branch\_name

**having** **count(**account\_no**)** **>=** 10

**order by** average\_balance desc

**limit** 0**,** 100**;**

**General Structure of UPDATE statement:**

**UPDATE**

**SET** *column-name* **=** *column-value* ,,,

[**WHERE** expression] **;**

**Example of UPDATE statement:**

**update** account **set** balance **=** balance **\*** 1.1 **where** balance **>=** 100000;

## Theories

|  |  |
| --- | --- |
| **3.1** | **List two reasons why *null* values might be introduced into the database.**  *Nulls* may be introduced into the database because the actual value is either unknown or does not exist. For example, an employee whose address has changed and whose new address is not yet known should be retained with a *null* address. If employee tuples have a composite attribute *dependents*, and a particular employee has no dependents, then that tuple’s *dependents* attribute should be given a *null* value. |
| **3.2** | **List two reasons why we may choose to define a view.**   1. Security conditions may require that the entire logical database be not visible to all users. 2. We may wish to create a personalized collection of relations that is better matched to a certain user’s intuition than is the actual logical model. |
| **3.3** | **List two major problems with processing update operations expressed in terms of views.**  Views present significant problems if updates are expressed with them. The difficulty is that a modification to the database expressed in terms of a view must be translated to a modification to the actual relations in the logical model of the database.   1. Since the view may not have all the attributes of the underlying tables, insertion of a tuple into the view will insert tuples into the underlying tables, with those attributes not participating in the view getting null values. This may not be desirable, especially if the attribute in question is part of the primary key of the table. 2. If a view is a join of several underlying tables and an insertion results in tuples with nulls in the join columns, the desired effect of the insertion will not be achieved. In other words, an update to a view may not be expressible at all as updates to base relations. |
| **3.4** | **What are the conditions of updating a view?**   1. The fromclause has only one database relation. 2. The selectclause contains only attribute names of the relation and does not have any expression, aggregates or distinctspecifications. 3. Any attribute not listed in the **select** clause can be set to null. 4. The query does not have a group by or havingclause. |
| **3.5** | **What is materialized view? *[In-course-1, 2008; 2005. Marks: 1]***  A view which makes sure that if the actual relations used in the view definition change, the view is kept up-to-date, is called materialized view. |
| **3.6** | **Define the following:**   1. **Domain** 2. **Atomic Domain** 3. **Non-Atomic Domain** 4. **Tuple Variable** 5. **Domain:**   For each attribute, there is a set of permitted values, which are called domain (*D*) of that attribute. For the attribute *branch-name*, the domain is the set of all branch names.   1. **Atomic Domain:**   A domain is atomic if elements of the domain are considered to be indivisible parts. Example: set of integers: 23, 45, 5, 78 etc.   1. **Non-Atomic Domain:**   If elements of a domain can be divided into several parts, the domain is called non-atomic domain. Example: set of all sets of integers: {23, 12, 4; 5, 65, 4; 34, 23, 98}, employee-id: HR001, IT005   1. **Tuple Variable:**   A tuple variable is a variable whose domain is the set of all tuples. For example, t[account-number] = “A-101”, t[branch-name] = “Mirpur”. Alternatively, t[1], t[2] denote the value of tuple *t* on first and second attributes and so on. |
| **3.7** | **What are the fundamental operations used in Relational Algebra? What are the conditions for set (union, set-intersect and set-difference) operations in RA? *[2005. Marks: 1 + 1]***  **OR, What are the conditions for set operations to be valid? *[In-course 1, 2008. Marks: 1]***  The fundamental operations used in Relational Algebra are:   1. Select (unary) 2. Project (unary) 3. Rename (unary) 4. Cartesian Product (binary) 5. Union (binary) 6. Set-Difference (binary)   The conditions for set operations are:   1. The relations must be of the same arity. That means they must have the same number of attributes. 2. The domains of *i*th attribute of the first set and the *i*th attribute of the second set must be the same, for all *i*. |
| **3.8** | **What are the conditions for insertion?**  The conditions for insertion are:  1. The attribute values for inserted tuples must be members of the attribute’s domain.  2. Tuples inserted must be of the same arity. |
| **3.9** | **Why set-intersection operation is not included in fundamental relational algebra operations? [*In-course 2007*; *2007*. *Marks: 1*]**  Because set-intersection operation can be done using fundamental operations. If r1 and r2 are two sets, then their intersection can be expressed as:  r1 ∩ r2 = r1 – (r1 – r2) = r2 – (r2 – r1) |
| **3.10** | **Give an example of generalized projection. [*In-course 2007*, *Marks: 1*]**  **OR, Give a relational algebra expression to represent generalized projection. [*In-course 2008*, *Marks: 1*]**  **SQL:** **select** student\_name, marks + 5 **from** result;  **RA:** **П** student\_name, marks + 5 (result) |
| **3.11** | **Let r(R) and s(S) be two relations. Give the relational algebra expression for natural join (⋈) and the outer joins ( ̲̅⋈, ⋈ ̲̅ , ̲̅⋈ ̲̅ ) of the said relations. *[2005. Marks: 2]***  **r ⋈ s =** ∏R ∪ S (σ r.A1 = s.A1 Λ r.A2 = s.A2 ……. r.An = s.An (r × s)), where R ∩ S = {A1, A2, ……, An}  **r ̲̅⋈ s =** (r ⋈ s) ∪ (r – ∏R (r ⋈ s)) × {(null, null, …, null)}  **r ⋈ ̲̅ s =** (r ⋈ s) ∪ (s – ∏S (r ⋈ s)) × {(null, null, …, null)}  **r ̲̅⋈ ̲̅ s =** (r ⋈ s) ∪ (r – ∏R (r ⋈ s)) × {(null, null, …, null)} ∪ (s – ∏S (r ⋈ s)) × {(null, null, …, null)} |
| **3.12** | **The outer-join operations extend the natural-join operation so that tuples from the participating relations are not lost in the result of the join. Describe how the theta join operation can be extended so that tuples from the left, right, or both relations are not lost from the result of a theta join.**  **r ̲̅⋈θ s =** (r ⋈θ s) ∪ (r – ∏R (r ⋈θ s)) × {(null, null, …, null)}  **r ⋈ ̲̅θ s =** (r ⋈θ s) ∪ (s – ∏S (r ⋈θ s)) × {(null, null, …, null)}  **r ̲̅⋈ ̲̅θ s =** (r ⋈θ s) ∪ (r – ∏R (r ⋈θ s)) × {(null, null, …, null)} ∪ (s – ∏S (r ⋈θ s)) × {(null, null, …, null)} |
| **3.13** | **With example, show the difference between Cartesian product (×) and natural join (⋈). *[2005. Marks: 2]***  Let, R1 = (A, B) and R2 = (B, C) be two relation schema.  Again, let r1(R1) = {{a, 1}, {b, 2}} and r2(R2) = {{1, x}, {2, y}}  Then, r1 × r2 = {{a, 1, 1, x}, {a, 1, 2, y}, {b, 2, 1, x}, {b, 2, 2, y}}  And r1 ⋈ r2 = {{a, 1, x}, {b, 2, y}}  That is, the Cartesian product operation results in all the combinations of all the tuples from both tables, whereas the natural join operation results in only the tuple combinations from both tables where the values of the common attributes (in this example, the attribute ‘B’) are the same. |
| **3.14** | **For a given relation schema, *works (employee\_name, company\_name, salary)*, give a relational algebra expression using all aggregate functions where the grouping is done on company name. [*2007*, *Marks: 1*]**  company\_name**G** **sum(**salary**), avg(**salary**), max(**salary**), min(**salary**), count(**employee\_name**)** (works) |
| **3.15** | **Give the equivalent relational algebra expression of the following SQL form:**  **select *A*1, *A*2, …, *An* from *r*1, *r*2, …, *rn* where *P* *[2005. Marks: 1]***  ∏ A1, A2, …, An (σ P (r1 × r2 … × rn)) |
| **3.16** | **Write short notes on natural join, theta join and aggregate functions.**  **Natural Join:**  The natural join is a binary operation that allows us to combine certain selection and a Cartesian product into one operation. It is denoted by the “join” symbol ⋈.  The natural join operation forms:  i) A Cartesian product of two arguments  ii) Performs a selection forcing equality on those attributes that appear in both relation schemas  iii) Removes duplicate attributes  **Theta Join:**  The theta join operation is an extension to the natural join operation that allows us to combine a selection and a Cartesian product into a single operation. Consider relations r(R) and s(S); let θ be predicate on attributes in the schema R ∪ S. The theta join operation is defined as follows:  R ⋈θ S = σθ (r × s)  **Aggregate Functions:**  Aggregate functions take a collection of values and return a single value as a result. It is denoted by calligraphic G, G. For a collection of values {1, 1, 3, 4, 4, 11}:  1. **sum** returns the sum of the values: 24  2. **avg** returns the average of the values: 4  3. **count** returns the number of the elements in the collection: 6  4. **min** returns the minimum value of the collection: 1  5. **max** returns the maximum value of the collection: 11 |
| **3.17** | **With example, explain the importance of outer joins. [*In-course 2007*, *Marks: 2*]**  When joining two or more tables, if we want to keep all the records from one table and want to know which records from the other tables don’t match with them, then outer join can be used to solve the problem easily.  For example, if we want to know which records in two tables (e.g., x and y) do not match, then we can write the following query using outer join:  **select** \* **from** x **natural** **full outer join** y  **where** x.some\_attribute **is** null **or** y.some\_attribute **is null;** |
| **3.18** | **Let R = (A, B, C); and let r1 and r2 both be relations on schema R. Give an expression in SQL that is equivalent to each of the following queries. *[2003. Marks: 4]***   1. **r1 ∪ r2** 2. **r1 ∩ r2** 3. **r1 – r2** 4. **ΠAB (r1)** ⋈ **ΠBC (r2)** 5. **select** \* **from** r1 **union select** \* **from** r2**;** 6. **select** \* **from** r1 **intersect select** \* **from** r2 **;** 7. **select** \* **from** r1 **minus select** \* **from** r2**;** 8. **select** \* **from (select** A, B **from** r1**)** **as** x **natural join (select** B, C **from** r2**) as** y**;[[2]](#footnote-3)** |
| **3.19** | **Give names of the aggregate functions that ignore null values in their input collection. *[2004. Marks: 1]***  sum, avg, min, max |
| **3.20** | **What aggregate functions can be used for string type data? *[In-course 1, 2008]***  count, min, max |
| **3.21** | **With examples define the terms *Superkey*, *Candidate Key* and *Primary Key*. [*2006*, *Marks: 3*]**  **Superkey:**  A superkey is a set of one or more attributes that, taken collectively, allow us to identify uniquely a tuple in the relation. For example:  *Branch\_schema = (branch\_name, branch\_city, assets)*  In Branch\_schema above, {branch\_name}, {branch\_name, branch\_city}, {*all attributes*} are all superkeys.  ***Formal definition*:[[3]](#footnote-4)** Let *R* be a relation schema. If it is said that a subset *K* of *R* is a superkey of *R*, it restricts consideration to relations *r(R)* in which no two distinct tuples have the same values on all attributes in *K*. That is, if *t1 and t2* are in *r* and *t1 ≠ t2*, then *t1[K] ≠ t2[K]*.  **Candidate Key:**  The superkey, for which no proper subset is a superkey, is a candidate key.  For example, in Branch\_schema above, {branch\_name} is a candidate key.  **Primary Key:**  The primary key is a candidate key that is chosen by the database designer as the principal means of identifying tuples within a relation.  In the Branch\_schema above, {branch\_name} is a primary key. |
| **3.22** | **Define foreign key.**  A relation schema R2 may include among its attributes the primary key of another relation schema R1. This attribute is called a foreign key from R2, referencing R1. The relation r2 is also called the ***referencing relation*** of the foreign key dependency and r1 is called the ***referenced relation*** of the foreign key.  The attribute *branch\_name* in *Account-schema* is a foreign key from *Account\_schema* referencing *Branch\_schema*.  ***Formal definition*:** Let *r1(R1)* and *r2(R2)* be relations with primary keys *K1* and *K2*, respectively. It is said that a subset *α* of *R2* is a **foreign key** referencing *K1* in *r1* if it is required that, for every *t2* in *r2*, there must be a tuple *t1* in *r1* such that *t1[k1] = t2[α]*. |
| **3.23** | **Identify the relations among primary key, candidate key and super key. *[2003. Marks: 3]***  Primary Key ⊆ Candidate Key ⊆ Super Key |
| **3.24** | **Let R = (P, Q, R, S). If PQ and QS can uniquely identify a tuple in the relation r(R) separately, how many superkeys, candidate keys and primary keys are there? *[In-course 1, 2008. Marks: 2]***  **Super Keys:** 6 – {P, Q}, {P, Q, R}, {P, Q, S}, {Q, S}, {Q, R, S}, {P, Q, R, S}  **Candidate Keys:** 2 – {P, Q}, {Q, S}  **Primary Key:** 1 – either {P, Q}, or {Q, S} |
| **3.25** | **Why do we need the rename operation?**   1. Two relations in the from clause may have attributes with the same name, so an attribute name is duplicated in the result. 2. If we use an arithmetic expression in the select clause, the resultant attribute does not have a name. 3. If an attribute name can be derived from the base relation, we may want to change the attribute name in the result to make it more meaningful. |
| **3.26** | **Give the schema diagram for the following database: [*2006*, *Marks: 2*]**  ***book (ISBN, title, year, price)***  ***author (author-id, name, address, url)***  ***warehouse (code, address, phone)***  ***written-by (author-id, ISBN)***  ***stocks (code, ISBN, number)***  ***author written-by book***   |  |  |  |  |  | | --- | --- | --- | --- | --- | | author-id |  | author-id  ISBN |  | ISBN | | name  address  phone | title  year  price | |  |   ***warehouse stocks***   |  |  |  | | --- | --- | --- | | code |  | code  ISBN | | address  phone | | number | |
| **3.27** | **Draw the schema diagram for the following part of the bank database: *[In-course 1, 2008; In-course 2, 2007. Marks: 1.5]***  ***employee (employee-id, employee-name, street, city)***  ***branch (branch-name, branch-city, assets)***  ***job (title, level)***  ***works-on (employee-id, branch-name, title, salary)***  ***employee***   |  | | --- | | employee-id | | employee-name street  city |   ***branch***   |  | | --- | | branch-name | | branch-city  assets |   ***job***   |  | | --- | | title | | level |   ***works-on***   |  | | --- | | employee-id branch-name  title | | salary | |
| **3.28** | **Give the schema diagram for the following part of database: *[2004. Marks: 2]***  ***person (driver-id, name, address)***  ***car (license, model, year)***  ***accident (report-no, date, location)***  ***owns (driver-id, license)***  ***participated (driver-id, license, report-no, damage-amount)***  ***person***   |  | | --- | | driver-id | | name  address |   ***car***   |  | | --- | | license | | model  year |   ***accident***   |  | | --- | | report-no | | date  location |   ***owns***   |  | | --- | | driver-id  license |   ***participated***   |  | | --- | | driver-id  license  report-no | | damage-amount | |
| **4.1** | **What are the join types and conditions that are permitted in SQL? *[2005. Marks: 2]***  **Join types:** inner join, left outer join, right outer join, full outer join.  **Join conditions:** natural, on <*predicate*>, using (A1, A2, …, An). |
| **4.2** | **Show that, in SQL, <> all is identical to not in.**  Let the set S denote the result of an SQL subquery. We compare (*x* **<>** **all** *S*) with (*x* **not in** *S*). If a particular value *x*1 satisfies (*x*1 **<>** **all** *S*) then for all elements *y* of *S* *x*1 ≠ y. Thus, *x*1 is not a member of *S* and must satisfy (*x*1 **not in** *S*). Similarly, suppose there is a particular value *x*2 which satisfies (*x*2 **not in** *S*). It cannot be equal to any element *w* belonging to *S*, and hence (*x*2 **<> all** S) will be satisfied. Therefore, the two expressions are equivalent. |
| **4.3** | **Why duplicates are retained in SQL? *[2004. Marks: 1]***  Duplicates are retained in SQL because:   1. Eliminating them is costly. 2. Retaining duplicates is important in computing sum or average. |
| **4.4** | **What is the difference between ‘*Embedded SQL*’ and ‘*Dynamic SQL*’? *[2004. Marks: 2]***  Dynamic SQL component allows programs to construct and submit SQL queries at run time. In contrast, embedded SQL statements must be completely present at compile time; they are compiled by the embedded SQL preprocessor. |
| **4.5** | **Describe the circumstances in which you would choose to use embedded SQL rather than SQL alone or only a general-purpose programming language.**  Writing queries in SQL is typically much easier than coding the same queries in a general-purpose programming language. However not all kinds of queries can be written in SQL. Also non-declarative actions such as printing a report, interacting with a user, or sending the results of a query to a graphical user interface cannot be done from within SQL. Under circumstances in which we want the best of both worlds, we can choose embedded SQL or dynamic SQL, rather than using SQL alone or using only a general-purpose programming language.  Embedded SQL has the advantage of programs being less complicated since it avoids the clutter of the ODBC or JDBC function calls, but requires a specialized preprocessor. |
| **4.6** | **Consider the database schema below:**  ***employee (ename, street, city)***  ***emp\_company (ename, cname, salary, jdate)***  ***company (cname, city)***  ***manager (ename, mname, shift)***  ***Note:* A manager is also an employee of a company.**  **Give *SQL* and *RA* expressions for the following queries: [*In-course-1, 2007*. *Marks: 2.5 each*.]**   1. **Find names, street addresses and cities of residence of all employees who work under manager Sabbir and who joined before January 01, 2006.** 2. **Find the names of the employees living in the same city where Rahim is residing.** 3. **Display the average salary of each company except Square Pharma.** 4. **Increase the salary of employees by 10% for the companies those are located in Bogra.** 5. **Delete records from *emp\_company* that contain employees living in Rajshahi.** 6. *Similar to Complete Concepts Problem – Query No. 4* 7. *Similar to Complete Concepts Problem – Query No. 7* 8. *Similar to Complete Concepts Problem – Query No. 19* 9. *Similar to Complete Concepts Problem – Query No. 15* 10. *Similar to Complete Concepts Problem – Query No. 18* |
| **4.7** | **Consider the database schema below:**  ***employee (employee-id, employee-name, street, city)***  ***branch (branch-name, branch-city, assets)***  ***job (title, level)***  ***works (employee-id, branch-name, title, salary)***  **Give *SQL* and *RA* expressions for the following queries: [*2007*; *Marks: 2.5 each*.]**   1. **Find names, street addresses and cities of residence and job level of all employees who work for *Dhanmondi* branch and earn more than Tk. 10000.** 2. **Find the no. of employees and their total salaries for the branches located at *Khulna* city.** 3. **Give all *Executive*s of *Mirpur* branch a 10 percent salary raise.** 4. **Find the branches with payroll less than Tk. 100000.** 5. *Similar to Complete Concepts Problem – Query No. 3* 6. **SQL:** **select** **count(**employee-id**),** **sum(**salary**)** **from** branch **natural join** works   **where** branch-city **=** 'Khulna' **group by** branch-name;  **RA:** branch-name **G****count(**employee-id**), sum(**salary**)** (**σ** branch-city = "Khulna" (branch **⋈** works))   1. *Similar to Complete Concepts Problem – Query No. 12* 2. *Similar to Complete Concepts Problem – Query No. 22* |
| **4.8** | **Consider the database schema below:**  ***employee (person-name, street, city)***  ***works (person-name, company-name, salary)***  ***company (company-name, city)***  ***manages (person-name, manager-name)***  ***Note:* A manager is also an employee of a company.**  **Give *SQL* and *RA* expressions for the following queries: [*2006*; *Marks: 2.5 each*.]**   1. **Find names, street addresses and cities of residence of all employees who work for First Bank Corporation and earn more than Tk. 30000.** 2. **Find all employees in the database who earn more than any employee of Medium Bank Corporation.** 3. **Give all managers of First Bank Corporation in the database a 10% salary raise.** 4. **Find those companies whose employees earn a higher salary, on average, than the average salary of Small Bank Corporation.** 5. **Find the company that has the smallest payroll.** 6. *Similar to Complete Concepts Problem – Query No. 3* 7. *Similar to Complete Concepts Problem – Query No. 10* 8. *Similar to Complete Concepts Problem – Query No. 12* 9. *Similar to Complete Concepts Problem – Query No. 23* 10. *Similar to Complete Concepts Problem – Query No. 21* |
| **4.9** | **Consider the database schema below:**  ***client (client-no, name, address, city)***  ***product (product-no, description, profit-percent, qty-in-hand, reorder-level, cost-price)***  ***salesman (salesman-no, name, address, city, sale-amt)***  ***salesorder (order-no, order-date, client-no, del-add, salesman-no, del-date, order-status)***  ***order-detail (order-no, product-no, qty-ordered, qty-delivered)***  **Give *SQL* and *RA* expressions for the following queries: [*2005*; *Marks: 2.5 each*.]**   1. **Find the list of all clients who stay in cities Dhaka or Khulna.** 2. **Find the products with their description whose selling price is greater than 2000 and less than or equal to 5000. [Selling price can be found from cost-price and profit-percent]** 3. **Find the total ordered and delivered quantity for each product with a product range of P0035 to P0056.** 4. **Find the clients with their names and order numbers whose orders are handled by the salesman Mr. X.** 5. **Find the product no and description of non-moving products, i.e., products not being sold.** 6. **SQL:** **select** name **from** client **where** city **=** 'Dhaka' **or** city **=** 'Khulna';   ***OR:* select** name **from** client **where** city **in** **(**'Dhaka'**,** 'Khulna'**)**;  **RA: Π** name (**σ** city = "Dhaka" **⋁** city = "Khulna" (client))   1. **SQL:** **select** product-no**,** description **from** product   **where** **(**profit-percent **/** 100 **\*** cost-price **+** cost-price**)** **>** 2000  **and** **(**profit-percent **/** 100 **\*** cost-price **+** cost-price**)** **<=** 5000;  **RA: Π** product-no, description  (**σ****(**profit-percent **/** 100 **\*** cost-price **+** cost-price**)****>** 2000 **⋀****(**profit-percent **/** 100 **\*** cost-price **+** cost-price**)****<=** 5000 (product))   1. **SQL:** **select** **sum(**qty-ordered**),** **sum(**qty-delivered**)** **from** order-detail   **where** product-no **between** 'P0035' **and** 'P0056' **group by** product-no;  **RA:** product-no**G****sum(**qty-ordered**), sum(**qty-delivered**)** (**σ** product-no >= "P0035" **⋀** product-no <= "P0056" (order-detail))   1. **SQL:** **select** client.name**,** order-no **from** client**,** salesman**,** salesorder   **where** client.client-no **=** salesorder.client-no  **and** salesman.salesman-no **=** salesorder.salesman-no  **and** salesman.name **=** 'Mr. X';  **RA: Π** client.name, order-no  (**σ** client.client-no = salesorder.client-no **⋀** salesman.salesman-no = salesorder.salesman-no **⋀** salesman.name = "Mr. X"  (client × salesman × salesorder))   1. **SQL:** **select** product-no**,** description **from** product   **minus**  **select** product-no**,** description **from** product **natural join** order-detail;  **RA: Π** product-no, description (product) **-** **Π** product-no, description (product **⋈** order-detail)  ***OR,* SQL: select** product-no**,** description **from** product **left outer join** order-detail  **using** **(**product-no**)** **where** order-no **=** **null**;  **RA: Π** product-no, description (**σ** order-no = null (product  **̲̅⋈** order-detail)) |
| **4.10** | **Consider the part of a bank database schema below:**  ***Worker (worker-id, worker-name, hourly-rate, skill-type, supervisor-id)***  ***Assignment (worker-id, building-id, start-date, num-days)***  ***Building (building-id, address, building-type)***  **Notes: 1. skill-types are: Electric, Plumbing, Roofing, Framing etc.**  **2. building-types are: Office, Hospital, Residence, Warehouse etc.**  **3. supervisors are also workers of self-supervision.**  **Give *SQL* and *RA* expressions for the following queries: [*In-course 1, 2008; 2004*; *Marks: 2.5 each*.]**   1. **What are the skill types of workers assigned to building ‘B02’ (building-id)?** 2. **List the name of the workers assigned to ‘warehouse’ (building-type) buildings.** 3. **Find the no. of workers for each building where more than 5 workers are working for it.** 4. **Give 5% hourly wage increment for the workers working for ‘hospital’ buildings.** 5. **SQL:** **select** skill-type **from** worker **natural** **join** assignment   **where** building-id **=** 'B02';  **RA: Π** skill-type (**σ** building-id = "B02" (worker **⋈** assignment))   1. **SQL:** **select** worker-name   **from** worker **natural join** assignment **natural join** building  **where** building-type **=** 'warehouse';  **RA: Π** worker-name (**σ** building-type = "B02" (worker **⋈** assignment **⋈** building))   1. **SQL:** **select** building-id, **count(**worker-id**) from** assignment   **group by** building-id **having** **count(**worker-id**) >** 5;  **RA:** t1 ←building-id **G****count(**worker-id**) as** no-of-workers (assignment)  **σ** no-of-workers > 5 (t1)   1. **SQL:** **update** worker **set** hourly-rate **=** hourly-rate **\*** 1.05   **where** worker-id **in (**  **select** worker-id **from** assignment **natural join** building  **where** building-type **=** 'hospital'  **)**;  **RA:** t1 ← **П** worker-id (**σ** building-type = "hospital" (assignment **⋈** building))  t2 ← **П** worker-id, worker-name, hourly-rate, skill-type, supervisor-id (worker **⋈** t1)  t3 ← **П** worker-id, worker-name, hourly-rate \* 1.05, skill-type, supervisor-id (worker **⋈** t1)  worker ← (worker – t2) **∪** t3 |
| **4.11** | **Consider the part of a bank database schema below:**  ***Branch (branch-name, branch-city, assets)***  ***Customer (customer-name, customer-street, customer-city)***  ***Loan (loan-no, branch-name, amount)***  ***Borrower (customer-name, loan-no)***  ***Account (account-no, branch-name, balance)***  ***Depositor (customer-name, account-no)***  **Give *SQL* and *RA* expressions for the following queries: [*2003*; *Marks: 3 each*.]**   1. **Find all customers who have either an account or a loan (but not both) at the bank.** 2. **Find the average account balance of those branches where the total account balance for individual branch is greater than 160,000.** 3. **Find the number of depositors for each branch.** 4. **Find the branch that has the highest average balance.** 5. **SQL:** **select** customer-name   **from** depositor **natural** **full outer join** borrower  **where** loan-number **is** null **or** account-number **is null**  **RA: П** customer-name (**σ** loan-number = null **⋁** account-number = null (depositor  **̲̅⋈ ̲̅** borrower))   1. **SQL:** **select** branch-name, **avg(**balance**)** **from** account   **group by** branch-name **having** **sum(**balance**)** > 16000;  ***OR:* create view** sum\_bal **as**  **(select** branch-name **from** account **group by** branch-name  **having** **sum(**balance**)** > 16000**)**;  **create view** average\_bal **as**  **(select** branch-name**, avg(**balance**) as** avg\_bal **from** account  **group by** branch-name**)**;  **select** \* **from** sum\_bal **natural join** average\_bal;  **RA:** t1 ← branch-name **G****sum(**balance**) as** sum\_bal(account))  t2 ← **σ** sum\_bal > 160000 (t1)  t3 ← branch-name **G****avg(**balance**) as** avg\_bal(t2)   1. **SQL:** **select** **count(distinct** customer-name**)**   **from** depositor **natural join** account **group by** branch-name;  **RA:** branch-name **G****count(**customer-name**)** (depositor **⋈** account)   1. **SQL:** **select** branch-name **from** account **group by** branch-name   **having** **avg(**balance) >= **all**  **(select** **avg(**balance**)** **from** account **group by** branch-name**)**;  ***OR:* create view** avg\_balance **as**  **select** branch-name**, avg(**balance**) as** avg\_bal  **from** account **group by** branch-name;  **create view** max\_balance **as**  **select max(**avg\_bal**) as** max\_bal **from** avg\_bal;    **select** branch-name **from** avg\_balance, max\_balance **where** avg\_bal **=** max\_bal;  **RA:** t1 ← branch-name **G****avg(**balance**) as** avg\_bal (account)  t2 ← **G****max(**avg\_bal**) as** max\_bal (t1)  **Π** branch-name (**σ** avg\_bal = max\_bal (t1 × t2)) |
| **4.12** | **Consider the part of a company database schema below:**  ***employee (person-name, street, city)***  ***works (person-name, company-name, salary)***  ***company (company-name, city)***  ***manages (person-name, manager-name)***  **Give *SQL* and *RA* expressions for the following queries: [*In-course 1*, *2005*; *Marks: 2.5 each*.]**   1. **Find the names of all employees who live in the same city as the company for which they work.** 2. **Find all employees in the database who earn more than any employee of Beximco Textiles Limited.** 3. **Give all managers in the database a 10% salary raise, unless the salary would be greater than 100,000.** 4. **Find those companies whose employees earn a higher salary, on average, than the average salary of Beximco Textiles Limited.** 5. **Find the company with the smallest payroll.** 6. **Find the names, street addresses and cities of residence of all employees who work for Padma Textile Limited and earn more than 11,000.** 7. *Similar to Complete Concepts Problem – Query No. 5* 8. *Similar to Complete Concepts Problem – Query No. 10* 9. *Almost Similar to Complete Concepts Problem – Query No. 14*   **SQL:** **update** works **set** salary **=** salary **\*** 1.1  **where** salary **\*** 1.1 **<=** 100000 **and** person-name **in** (  **select distinct** manager-name **from** manages  );  **RA:** t1 ← **П** works.person-name, company-name, salary (**σ** works.person-name = manager-name (works **×** manages))  t2 ← **П** works.person-name, company-name, salary (**σ**salary \* 1.1 ≤ 100000 (t1))  t3 ← **П** works.person-name, company-name, salary \* 1.1 (t2)  works ← (works – t2) **∪** t3   1. *Similar to Complete Concepts Problem – Query No. 23* 2. *Similar to Complete Concepts Problem – Query No. 21* 3. *Similar to Complete Concepts Problem – Query No. 3* |

# Chapter 6

# Integrity & Security

## Questions and Answers

|  |  |
| --- | --- |
| **6.1** | **What integrity constraints are used in database? *[2004. Marks: 1]***  The integrity constraints used in database are:   1. Domain Constraints 2. Referential Integrity Constraints |
| **6.2** | **Define foreign key and dangling tuples. How foreign key defines acceptability of dangling tuples? *[2003. Marks: 4]***  Consider a pair of relations *r*(*R*) and *s*(*S*), and the natural join *r* ⋈ *s*. There may be a tuple *tr* in *r* that does not join with any tuple in *s*. That is, there is no *ts* in *s* such that *tr*[*R* ∩ *S*] = *ts*[*R* ∩ *S*]. Such tuples are called *dangling tuples*.  Foreign key defines acceptability of dangling tuples by permitting the use of *null* values. Attributes of foreign keys are allowed to be null, provided that they have not otherwise been declared to be non-null. If all the columns of a foreign key are non-null in a given tuple, the usual definition of foreign-key constraints is used for that tuple. If any of the foreign-key columns is null, the tuple is defined automatically to satisfy the constraint. |
| **6.3** | **SQL allows a foreign-key dependency to refer to the same relation, as in the following example:**  **create table** *manager*  **(***employee-name* **char**(20),  *manager-name* **char**(20),  **primary key** *employee-name*,  **foreign key** (*manager-name*) **references** *manager* **on delete cascade)**  **Here, *employee-name* is a key to the table *manager*, meaning that each employee has at most one manager. The foreign-key clause requires that every manager also be an employee. Explain exactly what happens when a tuple in the relation *manager* is deleted.**  The tuples of all employees of the manager, at all levels, get deleted as well.  This happens in a series of steps. The initial deletion will trigger deletion of all the tuples corresponding to direct employees of the manager. These deletions will in turn cause deletions of second level employee tuples, and so on, till all direct and indirect employee tuples are deleted. |
| **6.4** | **What is a trigger and what are its parts? *[2002. Marks: 3]***  **OR, With example define trigger. *[2006, Marks: 2. 2004, Marks: 1]***  A *trigger* is a statement that the system executes automatically as a side effect of a modification to the database.  The parts of a trigger are:   1. An event – which causes the trigger to be checked 2. A condition – that must be satisfied for trigger execution to proceed. 3. The actions – that are to be taken when the trigger executes |
| **6.5** | **Consider the following schemas:**  ***Account =* (*Account\_number, Branch name, Balance*)**  ***Depositor =* (*Customer\_id, Account\_number*)**  **Write an SQL trigger to carry out the following action: after update on account for each owner of the account, if the account balance is negative, delete the owner from the account and depositor relation. *[2007. Marks: 2]***  **create trigger** *check-delete-trigger* **after delete on** *Account*  **referencing old row as** *orow*  **for each row**  **delete from** *depositor*  **where** *Depositor.Customer\_id* **not in**  (**select** *Customer\_id* **from** *Depositor*  **where** *Account\_number* < > *orow.Account\_number*)  **end** |
| **6.6** | **Define assertion with example. *[2004. Marks: 1]***  An assertion is a predicate expressing a condition that we wish the database always to satisfy.  For example, the following assertion ensures that the assets value for the Perryridge branch is equal to the sum of all the amounts lent by the Perryridge branch.  **create assertion** *perry* **check**  (**not exists** (**select** \* **from** *branch* **where** *branch-name* = ′Perryridge′ **and**  *assets* < > (**select sum** (*amount*) **from** *loan* **where** *branch-name* = ′Perryridge′))) |
| **6.7** | **Write an assertion for the bank database to ensure that the assets value for the Perryridge branch is equal to the sum of all the amounts lent by the Perryridge branch.**  *See the example of assertion in Question and Answer 6.6*. |
| **6.8** | **What is database security? What security measures are taken to protect the database? *[2003. Marks: 4]***  *Database security* refers to protection from malicious access.  To protect the database, the following levels of security measures are taken:   * **Database system.** Some database-system users may be authorized to access only a limited portion of the database. Other users may be allowed to issue queries, but may be forbidden to modify the data. It is the responsibility of the database system to ensure that these authorization restrictions are not violated. * **Operating system.** No matter how secure the database system is, weakness in operating system security may serve as a means of unauthorized access to the database. * **Network.** Since almost all database systems allow remote access through terminals or networks, software-level security within the network software is as important as physical security, both on the Internet and in private networks. * **Physical.** Sites with computer systems must be physically secured against armed or surreptitious entry by intruders. * **Human.** Users must be authorized carefully to reduce the chance of any user giving access to an intruder in exchange for a bribe or other favors. |
| **6.9** | **What are the different forms of authorizations used in SQL? *[2005. Marks: 2]***  The different forms of authorizations used in SQL are:   1. Authorization on data (instance) – *read*, *insert*, *delete* and *update* authorizations 2. Authorization on database schema – *index*, *resource*, *alteration* and *drop* authorizations |
| **6.10** | **What is the use of role in database? *[2004. Marks: 2]***  Authorizations can be granted to roles, in exactly the same fashion as they are granted to individual users. Each database user is granted a set of roles (which may be empty) that he or she is authorized to perform. |
| **6.11** | **What is authorization graph? *[2006. Marks: 1]***  The graph representing the passing of authorization from one user to another is called an *authorization graph*.  untitled.PNG  Figure: Authorization graph. |
| **6.12** | **How can you relate views with authorization? *[2004. Marks: 2]*** |
| **6.13** | **What do you understand by authentication? *[2004. Marks: 1]***  *Authentication* refers to the task of verifying the identity of a person / software connecting to a database. |
| **6.14** | **What are the different forms of authentication used in the database? *[2006. Marks: 1]***  The different forms of authentication used in database are:   1. Password-based authentication 2. Challenge-response system |
| **6.15** | **What are the properties of a good encryption technique? *[2005. Marks: 1]***  The properties of a good encryption technique are:   1. It is relatively simple for authorized users to encrypt and decrypt data. 2. It depends not on the secrecy of the algorithm, but rather on a parameter of the algorithm called the *encryption key*. 3. Its encryption key is extremely difficult for an intruder to determine. |
| **6.16** | **How public key encryption maintains security? *[2005. Marks: 2]***  If user *U*1 wants to store encrypted data, *U*1 encrypts them using public key *E*1 and decryption requires the private key *D*1.  If user *U*1 wants to share data with *U*2, *U*1 encrypts the data using *E*2, the public key of *U*2. Since only user *U*2 knows how to decrypt the data (using *D*2), information is transferred securely. |
| **6.17** | **Describe challenge-response system for authentication. *[2004. Marks: 3]***  In the challenge-response system, the database system sends a challenge string to the user. The user encrypts the challenge string using a secret password as encryption key, and then returns the result. The database system can verify the authenticity of the user by decrypting the string with the same secret password, and checking the result with the original challenge string. This scheme ensures that no passwords travel across the network.  Public-key systems can be used for encryption in challenge–response systems. The database system encrypts a challenge string using the user’s public key and sends it to the user. The user decrypts the string using her private key, and returns the result to the database system. The database system then checks the response. This scheme has the added benefit of not storing the secret password in the database, where it could potentially be seen by system administrators. |
| **6.18** | **What are the advantages of encrypting data stored in the database?**   1. Encrypted data allows authorized users to access data without worrying about other users or the system administrator gaining any information. 2. Encryption of data may simplify or even strengthen other authorization mechanisms. For example, distribution of the cryptographic key amongst only trusted users is both, a simple way to control read access, and an added layer of security above that offered by views. |
| **6.19** | **Perhaps the most important data items in any database system are the passwords that control access to the database. Suggest a scheme for the secure storage of passwords. Be sure that your scheme allows the system to test passwords supplied by users who are attempting to log into the system.**  A scheme for storing passwords would be to encrypt each password, and then use a hash index on the user-id. The user-id can be used to easily access the encrypted password. The password being used in a login attempt is then encrypted and compared with the stored encryption of the correct password.  An advantage of this scheme is that passwords are not stored in clear text and the code for decryption need not even exist. |

# Chapter 7

# Relational Database Design

## Concepts

|  |  |
| --- | --- |
| **7.1** | **How to decompose a relation into BCNF with dependency preservation**  Let, R be the relation and FD be the set of functional dependencies.   1. Check the first functional dependency, α → β. If α is *trivial* or a *superkey*, then R is in BCNF. So, go for the next functional dependency. 2. If α is not a superkey, then decompose R into R1 = (α, β) and R2 = (R – β). 3. Repeat the steps 1 and 2 for each decomposed relation until it is found that *each functional dependency holds for at least one of the relations*. 4. If you find one or more functional dependencies that don’t hold for any of the relations, then start over again by reordering the elements of FD. |

## Questions and Answers

|  |  |
| --- | --- |
| **7.1** | **Define functional dependency.**  Consider a relation schema R, α and β are set of attributes of R and let α ⊆ R and β ⊆ R. The functional dependency α → β holds on schema R if, in any legal relation r(R), for all pairs of tuples t1 and t2 in r such that t1[α] = t2[α], it is also the case that t1[β] = t2[β]. |
| **7.2** | **Define superkey using functional dependency. *[2004. Marks: 2]***  A set of attributes K is a superkey of a relation schema R if K → R. That is K is a superkey if, whenever t1[K] = t2[K], it is also the case that t1[R] = t2[R] (i.e., t1 = t2). |
| **7.3** | **Define trivial functional dependency. *[In-course 2, 2007; In-course 2, 2008. Marks: 1]***  A functional dependency of the form α → β is **trivial** if β ⊆ α. For example, A → A, AB → A etc. |
| **7.4** | **Why certain functional dependencies are called trivial functional dependencies?**  Certain functional dependencies are called trivial functional dependencies because they are satisfied by all relations. |
| **7.5** | **List all nontrivial functional dependencies (with no common attributes) satisfied by the following relation: *[In-course 2, 2007; In-course 2, 2008; 2005; Marks: 2]***   |  |  |  | | --- | --- | --- | | **A** | **B** | **C** | | **a1** | **b1** | **c1** | | **a1** | **b1** | **c2** | | **a2** | **b1** | **c1** | | **a2** | **b1** | **c3** |   A → B C → B AC → B |
| **7.6** | **List all nontrivial functional dependencies satisfied by the following relation: *[2003. Marks: 3]***   |  |  |  |  | | --- | --- | --- | --- | | **A** | **B** | **C** | **D** | | **a1** | **b1** | **c1** | **d1** | | **a1** | **b2** | **c1** | **d2** | | **a2** | **b2** | **c2** | **d2** | | **a2** | **b2** | **c2** | **d3** | | **a3** | **b3** | **c2** | **d4** |   A → C AB → C AD → C CD → A ABD → C BCD → A  D → B AD → B BC → A CD → B ACD → B |
| **7.7** | **Show functional dependencies to indicate the following: *[2004, 2007. Marks: 2]***   1. **A one-to-one relationship set exists between entity sets *account* and *customer*.** 2. **A many-to-one relationship set exists between entity sets *account* and *customer*.**   **Where:**  ***customer (customer-name, customer-street, customer-city)***  ***account (account-number, balance)***  Let Pk(r) denote the primary key attribute of a relation r.   1. The functional dependencies **Pk(account) → Pk (customer)** and **Pk(customer) → Pk(account)** indicate a one-to-one relationship, because any two tuples with the same value for account must have the same value for customer, and any two tuples agreeing on customer must have the same value for account. 2. The functional dependency **Pk(account) → Pk(customer)** indicates a many-to-one relationship since any account value which is repeated will have the same customer value, but many account values may have the same customer value. |
| **7.8** | **Why Armstrong’s axioms are called ‘sound’ and ‘complete’? *[2005. Marks: 1]***  Armstrong’s axioms are *sound*, because they do not generate any *incorrect* functional dependencies.  They are *complete*, because, for a given set F of functional dependencies, they allow us to generate *all* F+. |
| **7.9** | **Use the definition of functional dependency to argue that each of Armstrong’s axioms (reflexivity, augmentation, and transitivity) is sound.**  The definition of functional dependency is: α → β holds on R if in any legal relation r(R), for all pairs of tuples t1 and t2 in *r* such that t1[α] = t2[α], it is also the case that t1[β] = t2[β].  ***Reﬂexivity rule*:** If α is a set of attributes, and β ⊆ α, then α → β.  Assume ∃ t1 and t2 such that t1[α] = t2[α]  t1[β] = t2[β] [since β ⊆ α]  α → β [Definition of FD]  ***Augmentation rule*:** If α → β, and γ is a set of attributes, then γα → γβ.  Assume ∃ t1,t2 such that t1[γα] = t2[γα]  t1[γ] = t2[γ] [γ ⊆ γα]  t1[α] = t2[α] [α ⊆ γα]  t1[β] = t2[β] [Definition of α → β]  t1[γβ] = t2[γβ] [γβ = γ ∪ β]  γα → γβ [Definition of FD]  ***Transitivity rule*:** If α → β and β → γ, then α → γ.  Assume ∃ t1,t2 such that t1[α]= t2[α]  t1[β] = t2[β] [Definition of α → β]  t1[γ] = t2[γ] [Definition of β → γ]  α → γ [Definition of FD] |
| **7.10** | **Use Armstrong’s axioms to prove the soundness of Union, Decomposition and Pseudotransitivity rules. *[In-course 2, 2007; In-course 2, 2008; 2007; Marks: 3. 2003, Marks: 5]***  ***Union rule*:** If α → β and α → γ then α → βγ.  We derive:  α → β [Given]  αα → αβ [Augmentation rule]  α → αβ …(i) [Union of identical sets]  Again, α → γ [Given]  αβ → γβ …(ii) [Augmentation rule]  α → βγ [From (i) and (ii) using transitivity rule and set union commutativity]  ***Decomposition rule*:** If α → βγ, then α → β and α → γ.  We derive:  α → βγ [Given]  βγ → β [Reflexivity rule]  α → β [Transitivity rule]  Again, βγ → γ [Reflexive rule]  α → γ [Transitive rule]  ***Pseudotransitivity rule*:** If α → β and γβ → δ, then αγ → δ.  We derive:  α → β [Given]  αγ → γβ [Augmentation rule and set union commutativity]  γβ → δ [Given]  αγ → δ [Transitivity rule] |
| **7.11** | **Consider the following proposed rule for functional dependencies: If *α → β* and *δ → β*, then *α → δ*. Prove that this rule is not sound by showing a relation r that satisfies *α → β* and *δ → β*, does not satisfy *α → δ*. *[2004. Marks: 2]***  Consider the following relation *r* :   |  |  |  | | --- | --- | --- | | **A** | **B** | **C** | | a1 | b1 | c1 | | a1 | b1 | c2 |   Let, α = A, β = B, γ = C.  From the above relation, we see that A → B and C → B (i.e., α → β and γ → β). However, it is not the case that A → C (i.e., α → γ) since the same *A* (α) value is in two tuples, but the *C* (γ) value in those tuples disagree. |
| **7.12** | **Define *closure of attribute sets*, α+.**  Let α be a set of attributes. We call the set of all attributes functionally determined by α under a set *F* of functional dependencies the closure of α under *F*, and we denote it by α+. |
| **7.13** | **What are uses of ‘closure of attribute sets’, α+? *[In-course 2, 2007; In-course 2, 2008; 2007; 2004; Marks: 2]***   1. To test if α is a superkey, we compute α+ and check if α+ contains all attributes of R. If it contains, α is a superkey of R. 2. For a given set of F, we can check if a functional dependency α → β holds (or is in F+), by checking if β ⊆ α+. That is, we compute α+ by using attribute closure and then check if it contains β. 3. 3. It gives us an alternate way to compute F+: For each γ ⊆ R, we find the closure γ+, and for each S ⊆ γ+, we output a functional dependency γ → S. |
| **7.14** | **Compute the closure of the attribute/s to list the candidate key/s for relation schema R = (A, B, C, D, E) with functional dependencies F = {A → BC, CD → E, B → D, E → A} *[2005. Marks: 2]*** |
| **7.15** | **Define canonical cover, FC. *[2004. Marks: 2]***  A canonical cover FC for F is a set of dependencies such that F logically implies all dependencies in FC and FC logically implies all dependencies in F. Furthermore, FC must have the following properties:   1. No functional dependency in FC contains an extraneous attribute. 2. Each left side of a functional dependency in FC is unique. That is, there are no two dependencies α1 → β1 and α2 → β2 in FC such that α1 = α2. |
| **7.16** | **What is the advantage of using canonical cover, FC? *[In-course 2, 2007, Marks: 1]***  The advantage of using canonical cover is that the effort spent in checking for dependency violations can be minimized. |
| **7.17** | **What are the design goals for relational database design? *[In-course 2, 2007; 2005 Marks: 1]***  **Explain why each is desirable.**  The design goals for relational database design are:   1. Lossless-join decompositions 2. Dependency preserving decompositions 3. Minimization of repetition of information   They are desirable so we can maintain an accurate database, check correctness of updates quickly, and use the smallest amount of space possible. |
| **7.18** | **Explain what is meant by *repetition of information*, *inability to represent information* and *loss of information*. Explain why each of these properties may indicate a bad relational database design. *[2006. Marks: 4.5]***  *Repetition of information* is a condition in a relational database where the values of one attribute are determined by the values of another attribute in the same relation, and both values are repeated throughout the relation. This is a bad relational database design because it increases the storage required for the relation and it makes updating the relation more difficult.  *Inability to represent information* is a condition where a relationship exists among only a proper subset of the attributes in a relation. This is bad relational database design because all the unrelated attributes must be filled with null values otherwise a tuple without the unrelated information cannot be inserted into the relation.  *Loss of information* is a condition of a relational database which results from the decomposition of one relation into two relations and which cannot be combined to recreate the original relation. It is a bad relational database design because certain queries cannot be answered using the reconstructed relation that could have been answered using the original relation. |
| **7.19** | **Explain the condition for lossless-join decomposition. *[2004. Marks: 2]***  Let R be a relation schema, F be a set of functional dependencies on R; and R1 and R2 form a decomposition of R.  The decomposition is a lossless-join decomposition of R if at least one of the following functional dependencies are in F+ :  1. R1 ∩ R2 → R1  2. R1 ∩ R2 → R2  In other words, if R1 ∩ R2 (the attribute involved in the natural join) forms a superkey of either R1 or R2, the decomposition of R is lossless-join decomposition. |
| **7.20** | **Suppose that we decompose the schema R = (A, B, C, D, E) into (A, B, C) and (A, D, E). Show that this decomposition is lossless-join decomposition if the following set F of functional dependencies holds: *[2006. Marks: 2]***  **A → BC**  **CD → E**  **B → D**  **E → A**  A decomposition {R1, R2} is a lossless-join decomposition if  R1 ∩ R2 → R1  or R1 ∩ R2 → R2.  Let R1 = (A, B, C), R2 = (A, D, E), and R1 ∩ R2 = A.  Since A is a candidate key (because, the closure of A is R),  R1 ∩ R2 → R1. |
| **7.21** | **Suppose that we decompose the schema R = (A, B, C, D, E) into (A, B, C) and (C, D, E). Show that this decomposition is *not* lossless-join decomposition if the following set F of functional dependencies holds: *[2006. Marks: 2]***  **A → BC**  **CD → E**  **B → D**  **E → A**  Let, r be a relation as follows:   |  |  |  |  |  | | --- | --- | --- | --- | --- | | **A** | **B** | **C** | **D** | **E** | | a1 | b1 | c1 | d1 | e1 | | a2 | b2 | c1 | d2 | e2 |   With *R*1 = (A, B, C) and *R*2 = (C, D, E) :  Π*R*1 (*r*) would be:   |  |  |  | | --- | --- | --- | | **A** | **B** | **C** | | a1 | b1 | c1 | | a2 | b2 | c1 |   Π*R*2 (*r*) would be:   |  |  |  | | --- | --- | --- | | **C** | **D** | **E** | | c1 | d1 | e1 | | c1 | d2 | e2 |   Π*R*1 (*r*) ⋈ Π*R*2 (*r*) would be:   |  |  |  |  |  | | --- | --- | --- | --- | --- | | **A** | **B** | **C** | **D** | **E** | | a1 | b1 | c1 | d1 | e1 | | a1 | b1 | c1 | d2 | e2 | | a2 | b2 | c1 | d1 | e1 | | a2 | b2 | c1 | d2 | e2 |   Clearly, Π*R*1 (*r*) ⋈ Π*R*2 (*r*) ≠ *r*.  Therefore, this is a lossy join. |
| **7.22** | **Deduce the condition for dependency preservation using restrictions for decomposing a given schema R and a set of FDs F. Decompose the schema R = (A, B, C, D, E) with functional dependencies F = { A → B, BC → D } into BCNF with dependency preservation. *[2005. Marks: 2 + 1]***  Let F be a set of functional dependencies on schema R. Let R1, R2, ..., Rn be a decomposition of R.  The *restriction* of F to Ri is the set of all functional dependencies in F+ that include only attributes of Ri.  The set of restrictions F1, F2, ..., Fn is the set of dependencies that can be checked efficiently.  Let F*'* = F1 ∪ F2 ∪ ... ∪ Fn.  F*'* is a set of functional dependencies on schema R, but in general, F*'* ≠ F  However, it may be that F*'* + = F+.  If this is so, then every functional dependency in F is implied by F*'*, and if F*'* is satisfied, then F must also be satisfied.  Therefore, thecondition for dependency preservation using restrictions for decomposing a given schema R and a set of FDs F is that F*'* + = F+.  ***Decomposition of the schema R*:**  We change the order of the FDs in F such that F = {BC → D, A → B}.  Now, the FD BC → D holds on R, but BC is not a superkey. So, we decompose R into  R1 = (B, C, D) and R2 = (A, B, C, E)  R1 is in BCNF. However, the FD A → B holds on R2, but A is not a superkey. So, we decompose R2 into  R3 = (A, B) and R4 = (A, C, E)  Now, R3 and R4 both are in BCNF. [R4 is in BCNF as only trivial functional dependencies exist in R4]  So, the final decomposed relations are: R1 = (B, C, D), R3 = (A, B) and R4 = (A, C, E). |
| **7.23** | **Consider a relation schema R = (A, B, C, D) and with functional dependencies F = {A → BC, B → D, D → B}. Show the BCNF decomposition of the above schema with dependency preservation with causes. *[In-course 2, 2008; Marks: 2]***  The FD A → BC holds on R, and A is a superkey (∵ A → BC and B → D, ∴ A → BCD, or, A → ABCD).  Therefore, we go for the next FD, B → D. This holds on R, but B is not a superkey. So, we decompose R into  R1 = (B, D) and R2 = (A, B, C)  Now, both R1 and R2 are in BCNF as R1 satisfies B → D and R2 satisfies A → BC.  So, the final decomposition of R is: R1 = (B, D) and R2 = (A, B, C). |
| **7.24** | **What are the differences between BCNF and 3NF? *[In-course 2, 2008; 2002, 2004, 2007. Marks: 3]***  For a functional dependency α → β, 3NF allows this dependency in a relation if each attribute *A* in β – α is contained in any candidate key for R. However, BCNF does not allow this condition.  It is always possible to find a dependency-preserving lossless-join decomposition that is in 3NF. However, it is not always possible to find such decomposition that is in BCNF.  Repetition of information occurs in 3NF, whereas no repetition of information occurs in BCNF. |
| **7.25** | **In designing a relational database, why might we choose a non-BCNF design?**  BCNF is not always dependency preserving. Therefore, we may want to choose another normal form (specifically, 3NF) in order to make checking dependencies easier during updates. This would avoid joins to check dependencies and increase system performance. |
| **7.26** | **What is multivalued dependency? Give an example. *[2002. Marks: 4]***  Let R be a relation schema and let α ⊆ R and β ⊆ R. The multivalued dependency α→→β holds on R if in any legal relation r(R), for all pairs of tuples t1 and t2 in r such that t1[α] = t2[α], there exist tuples t3 and t4 in r such that:  t1[α] = t2[α] = t3[α] = t4[α]  t3[β] = t1[β]  t4[β] = t2[β]  t3[R – β] = t2[R – β]  t4[R – β] = t1[R – β]  ***Example*:**   |  |  |  | | --- | --- | --- | | **Name** | **Address** | **Car** | | Tom | North Road | Toyota | | Tom | Oak Street | Honda | | Tom | North Road | Honda | | Tom | Oak Street | Toyota |   In the above relation, **Name →→ Address** and **Name →→ Car**. |
| **7.27** | **Give an example of a relation schema R and a set of dependencies that R is in BCNF, but not in 4NF. *[2006. Marks: 2]***  R = (*loan-no, amount, customer-name, customer-address*)  F = {*loan-no → amount*}  If we assume that each customer might have more than one addresses, then the functional dependency *customer-name → customer-address* cannot be enforced. Thus, R is in BCNF. However, it is not in 4NF as it contains multivalues for *customer-address* and therefore there occurs repetition of information in the *loan-no* and *amount* fields. |
| **7.28** | **An employee database is to hold information about employees, the department they are in and the skills which they hold. The attributes to be stored are:**  **(*emp-id, emp-name, emp-phone, dept-name, dept-phone, dept-mgrid, skill-id, skill-name, skill-date, skill-level*)**  **An employee may have many skills such as word-processing, typing, librarian... The date on which the skill was last tested and the level displayed at that test are recorded for the purposes of assigning work and determining salary. An employee is attached to one department and each department has a unique manager.**   1. **Derive a functional dependency set for the above database, stating clearly any assumptions that you make.** 2. **Derive a set of BCNF relations, indicating the primary key of each relation.**   ***[2002. Marks: 4 + 4]*** |

# Chapter 11

# Storage & File Structure

## Theories

|  |  |
| --- | --- |
| **11.1** | **Define the term: RAID *[In-course 2, 2008; 2003, 2005. Marks: 1]***  RAID (*Redundant Array of Independent Disks*) is a technology which makes use of two or more hard drives in order to improve performance, reliability or create larger data volumes. |
| **11.2** | **What is mirroring / shadowing? *[2004, 2005. Marks: 1]***  The simplest but most expensive approach to introduce redundancy is to duplicate every disk. This technique is called *mirroring*or*shadowing*. |
| **11.3** | **What is striping?**  In context of RAID, striping means splitting and writing data across multiple drives to increase throughput.  There are two types of striping:   1. **Bit-level Striping:** consists of splitting the bits of each *byte* across multiple disks. 2. **Block-level Striping:** stripes *blocks* across multiple disks. |
| **11.4** | **How RAID improves reliability via redundancy? *[2006. Marks: 2]***   1. The chance that at least one disk out of a set of *N* disks will fail is much higher than the chance that a specific disk will fail. 2. Redundancy stores extra information that is not needed normally, but that can be used in the event of failure of a disk to rebuild the lost information. |
| **11.5** | **How RAID improves performance via parallelism? *[In-course 2, 2008; Marks: 2]***   1. With disk mirroring, the rate at which read requests can be handled is doubled, since read requests can be sent to either disk. 2. The transfer rate of each read is the same as in a single disk system, but the number of reads per unit time is doubled. |
| **11.6** | **What are the advantages of having large number of disks in a system? *[2004. Marks: 2]***   1. Improving the rate at which data can be read or written, if the disks are operated in parallel. Several independent reads or writes can also be performed in parallel. 2. Improving the reliability of data storage – because redundant information can be stored on multiple disks. Thus failure of one disk does not lead to loss of information. |
| **11.7** | **What are the factors for choosing a RAID level? *[2003. Marks: 2]***   1. Monetary cost of extra disk storage requirements. 2. Performance requirements in terms of number of I/O operations. 3. Performance when a disk has failed. 4. Performance during rebuild (while the data in a failed disk is being rebuilt on a new disk). 5. How many disks should be in an array? 6. How many bits should be protected by each parity bit? |
| **11.8** | **Which RAID level is used for storage of log files in a database? Justify your answer. *[2007. Marks: 2]***  RAID 1 is used for storage of log files in a database, because for storing log files, fault tolerance is needed on a limited volume of data (the limit is the capacity of 1 disk). |
| **11.9** | **What are possible ways of organizing the records in files? What does reorganization do? *[2004. Marks: 4 + 1]***  **OR, Classify file organization. Why reorganization is required in sequential file organization? *[2003. Marks: 2 + 1]***  **OR, What are different types of organization of records in files? What do you understand by reorganization? *[2006. Marks: 3 + 1]***  ***File organization*:**   1. **Heap file organization:** Any record can be placed anywhere in the file where there is space for the record. There is no ordering of records. Typically, there is a single file for each relation. 2. **Sequential file organization:** Records are stored in sequential order, based on the value of a “search key” of each record. 3. **Hashing file organization:** A hash function is computed on some attribute of each record. The result of the function specifies in which block of the file the record should be placed. 4. **Multitable Clustering file organization:** Records of several different relations can be stored in the same file. Related records of the different relations are stored on the same block so that one I/O operation fetches related records from all the relations.   ***Reorganization*:**  The sequential file organization will work well if relatively few records need to be stored in overflow blocks. Eventually, however, the correspondence between search-key order and physical order may be totally lost. In such cases sequential processing will become mush less efficient. At this point, the file should be *reorganized* so that it is once again physically in sequential order. |
| **11.10** | **Describe slotted page structure for organizing records within a single block. *[In-course 2, 2008; 2004. Marks: 4]***  8  Figure 11.10: Slotted page structure.  In slotted page structure, there is a header at the beginning of each block, containing:  1. No of record entries in the header  2. The end of free space in the block  3. An array whose entries contain the location and size of each record.  The actual records are allocated contiguously in the block, starting from the end of the block. The free space in the block is contiguous, between the final entry in the header file and the first record.  If a record is inserted, space is allocated for it at the end of the free space and an entry containing its size and location is added to the header.  If a record is deleted, the space it occupies is freed and its entry is set to deleted. Further, the records in the block before the deleted record are moved, so that the free space created by deletion gets occupied and all free space is again between the final entry in the header array and the first record. The end-of-free-space pointer in the header is appropriately updated as well.  Records can be grown and shrunk by similar techniques, as long as there is space in the block. The cost of moving the record is not so high, since the size of a block is limited: A typical value is 4KB.  The slotted page structure requires that there be no pointers that point directly to records. Instead, pointers must point to the entry in the header that contains the actual location of the record. This level of indirection allows records to be moved to prevent fragmentation of space inside a block, while supporting indirect pointers to the record. |
| **11.11** | **How variable-length records are represented by fixed-length records? *[2005. Marks: 2]***  *Not present at sir’s lecture. Therefore, it is assumed to be not important for exam. However, the answer is in the book – topic no. 11.6.2.2, page no. 420 (according to 4th edition).* |
| **11.12** | **Consider a relational database with two relations:**  ***course (course-name, room, instructor)***  ***enrollment (course-name, student-name, grade)***  **Define instances of these relations for two courses, each of which enrolls three students. Give the file structure of these relations that uses clustering. *[2004. Marks: 2]***  ***Instances of the given relations:***   |  |  |  |  | | --- | --- | --- | --- | | **course-name** | **room** | **instructor** |  | | Java | 420 | SMH | *c*1 | | OS | 320 | MHK | *c*2 |   ***course* relation**   |  |  |  |  | | --- | --- | --- | --- | | **course-name** | **student-name** | **grade** |  | | Java | X | A | *e*1 | | Java | Y | B | *e*2 | | Java | Z | C | *e*3 | | OS | X | A | *e*4 | | OS | Y | B | *e*5 | | OS | Z | C | *e*6 |   ***enrollment* relation**  ***Clustering file structure of these relations:***  Block 0 contains: *c*1, *e*1, *e*2 and *e*3  Block 1 contains: *c*2, *e*4, *e*5 and *e*6 |
| **11.13** | **Give one advantage and disadvantage of the following strategies for storing a relational database: *[2007. Marks: 1 + 1]***   1. **Store each relation in one file** 2. **Store multiple relations / the entire database in one file** 3. Advantages of storing a relation as a file include using the file system provided by the OS, thus simplifying the DBMS, but incur the disadvantage of restricting the ability of the DBMS to increase performance by using more sophisticated storage structures. 4. By using one file for multiple relations, these complex structures can be implemented through the DBMS, but this increases the size and complexity of the DBMS. |
| **11.14** | **Define seek time. *[In-course 2, 2008. Marks: 1]***  The time it takes for a disk read/write head to move to a specific data track is called *seek time*. |

# Chapter 12

# Indexing and Hashing

## Concepts

|  |  |
| --- | --- |
| **12.1** |  |
| **12.2** | **Search Key**  An attribute or set of attributes used to look up records in a ﬁle is called a *search key*.  **Primary / Clustering Index**  If the ﬁle containing the records is sequentially ordered, a *primary index* is an index whose search key also deﬁnes the sequential order of the ﬁle.  **Secondary / Non-Clustering Index**  Indices whose search key speciﬁes an order different from the sequential order of the ﬁle are called *secondary indices*.  **Index-Sequential Files**  Files that are ordered sequentially on some search key and have a primary index on that search key are called *index-sequential files*.  **Dense Index**  *Dense index* is the index where an index record appears for every search-key value in the file.  **Sparse Index**  *Sparce index* is the index where an index record appears for only some of the search-key values in the file.  **Multilevel Index**  An index with two or more levels is called a *multilevel index*.  **B+ Tree**  A *B+ tree* is a type of index which takes the form of a balanced tree in which every path from the root of the tree to a leaf of the tree is of the same length.  In a B+ tree, each non-leaf node in the tree has between ⎡*n* / 2⎤ and *n* children, where *n* is ﬁxed for a particular tree. Each leaf has between ⎡(*n* – 1) / 2⎤ and *n* – 1 values. The ranges of values in each leaf do not overlap.  **B-Tree**  A *B-tree* index is similar to B+ tree index except that search-key values in a B-tree appear only onece.  **Hashing**  *Hashing* is a process of generating an index or address basing on some data. A hash function is used to compute the location of a record from a given search-key value.  **Hash Index**  A *hash index* is an index which organizes the search keys, with their associated pointers, into a hash file structure.  **Hash Function**  A *hash function* is any well-defined procedure or mathematical function which converts a large, possibly variable-sized amount of data into a small datum, usually a single integer that may serve as an index into an array.  The values returned by a hash function are called *hash values*, *hash codes*, *hash sums*, or simply *hashes*. |
| **12.3** | **Advantages of Dense Index**  It is generally faster to locate a record if we have a dense index rather than a sparse index.  **Advantages of Sparse Index**  However, sparse indices have advantages over dense indices in that they require less space and they impose less maintenance overhead for insertions and deletions.  **Advantages of Multilevel Index**  Searching for records with a multilevel index requires signiﬁcantly fewer I/O operations than does searching for records by binary search.  **Advantages of Primary Index**  A sequential scan in primary index order is efﬁcient because records in the ﬁle are stored physically in the same order as the index order.  **Advantages of Secondary Index**  Secondary indices improve the performance of queries that use keys other than the search key of the primary index.  **Disadvantages of Secondary Index**  Secondary indices impose a signiﬁcant overhead on modiﬁcation of the database.  **Disadvantages of Index-Sequential File Organization**  The main disadvantage of the index-sequential ﬁle organization is that performance degrades as the ﬁle grows, both for index lookups and for sequential scans through the data. Although this degradation can be remedied by reorganization of the ﬁle, frequent reorganizations are undesirable.  **Disadvantages of B+ Tree**   1. Imposes performance overhead on insertion and deletion. 2. Adds space overhead – Since nodes may be as much as half empty (if they have the minimum number of children), there is some wasted space.   **Advantages of B-Tree**   1. May use less tree nodes than a corresponding B+ tree. 2. Sometimes it is possible to find the desired value before reaching a leaf node.   **Disdvantages of B-Tree**   1. Only a small fraction of desired values are found before reaching a leaf node. 2. Fewer search-keys appear in non-leaf nodes; hence, fan-out is reduced. Thus, B-trees typically have greater depth than a corresponding B+ tree. 3. Insertion and deletion are more complicated than in B+ trees. 4. Implementation is harder than B+ trees, since leaf and non-leaf nodes are of different sizes.   **Advantages of Hashing**   1. Allows to avoid accessing an index structure. 2. Provides a way of constructing indices. |

## Questions and Answers

|  |  |
| --- | --- |
| **12.1** | **Define the followings:**   1. **Primary Index *[2006, Marks: 0.5. 2004, Marks: 1.5]*** 2. **Secondary Index *[2006. Marks: 0.5]*** 3. **Sparse Index *[2006. Marks: 0.5]*** 4. **Dense Index *[2006. Marks: 0.5]*** 5. **B+ tree *[2004. Marks: 1.5]*** 6. **Hashing *[2005, Marks: 1. 2004, Marks: 1]***   *See Concept 12.2.* |
| **12.2** | **Since indices speed query processing, why might they not be kept on several search keys?**  Reasons for not keeping several search indices include:   1. Every index requires additional CPU time and disk I/O overhead during inserts and deletions. 2. Indices on non-primary keys might have to be changed on updates, although an index on the primary key might not (this is because updates typically do not modify the primary key attributes). 3. Each extra index requires additional storage space. 4. For queries which involve conditions on several search keys, efficiency might not be bad even if only some of the keys have indices on them. Therefore database performance is improved less by adding indices when many indices already exist. |
| **12.3** | **What are the differences between a *primary index* and a *secondary index*? *[2005, Marks: 2. 2003, Marks: 3]***   |  |  | | --- | --- | | **Primary Index** | **Secondary Index** | | 1. If the ﬁle containing the records is sequentially ordered, a *primary index* is an index whose search key also deﬁnes the sequential order of the ﬁle. | 1. Indices whose search key speciﬁes an order different from the sequential order of the ﬁle are called *secondary indices*. | | 1. There can be only one primary index for a relation. | 1. There can be many secondary indices for a relation. | | 1. A sequential scan in primary index order is efﬁcient. | 1. Performance of sequential scan in secondary index order is poor. | | 1. Can be dense or sparse. | 1. Can be only dense. | |
| **12.4** | **Clustering indices may allow faster access to data than a non-clustering index affords. When must we create a non-clustering index despite the advantages of a clustering index? Explain your answer. *[2007. Marks: 2]***  If we need to lookup a record using a search-key other than the search-key on which the file is stored sequentially, then we must create a non-clustering index to improve the performance of look-up. |
| **12.5** | **When is it preferable to use a dense index rather than a sparse index? Explain your answer. *[2006, Marks: 2. 2004, Marks: 3]***  It is preferable to use a dense index instead of a sparse index when the file is not sorted on the indexed field (such as when the index is a secondary index) or when the index file is small compared to the size of memory. |
| **12.6** | **Why is sparse index used in database? *[2002. Marks: 4]***  Sparse index is used in database because:   1. It requires less space. 2. It imposes less maintenance overhead for insertions and deletions. |
| **12.7** | **What is the purpose of multilevel indexing? *[2005. Marks: 1]***  The purpose of multilevel indexing is to reduce I/O operations on indices when records are searched. |
| **12.8** | **Is it possible in general to have two primary indices on the same relation for different search keys? Explain your answer. *[2007. Marks: 2]***  In general, it is not possible to have two primary indices on the same relation for different keys because the tuples in a relation would have to be stored in different order to have same values stored together. We could accomplish this by storing the relation twice and duplicating all values, but for a centralized system, this is not efficient. |
| **12.9** | **Consider the following *dense primary index file* corresponding to the sequential file *Account* sorted on the attribute *branch\_name*.**   |  |  | | --- | --- | | *Branch\_name* | *Pointer* | | Adabor |  | | C.O. |  | | Dhanmondi |  | | Mirpur |  | | Motijheel |  |   ***Index file***   |  |  |  | | --- | --- | --- | | *Account\_no* | *Branch\_name* | *Balance* | | A-9 | Adabor | 300 | | A-1 | Adabor | 500 | | A-5 | C. O. Bazar | 560 | | A-8 | Dhanmodi | 590 | | A-3 | Dhanmodi | 420 | | A-2 | Mirpur | 600 | | A-4 | Mirpur | 520 | | A-10 | Mirpur | 120 | | A-6 | Motijheel | 600 | | A-7 | Motijheel | 200 |   ***Account file***  **Now make necessary modification to the index file after deletion of the record for the account no ‘A-5’ and then ‘A-2’. *[2007. Marks: 1 + 1]***  ***Index file after deletion of the record for the account no ‘A-5’:***   |  |  | | --- | --- | | *Branch\_name* | *Pointer* | | Adabor |  | | Dhanmondi |  | | Mirpur |  | | Motijheel |  |  |  | | --- | | A-9 | | A-8 | | A-2 | | A-6 |   ***Index file after deletion of the record for the account no ‘A-2’:***   |  |  | | --- | --- | | *Branch\_name* | *Pointer* | | Adabor |  | | Dhanmondi |  | | Mirpur |  | | Motijheel |  |  |  | | --- | | A-9 | | A-8 | | A-4 | | A-6 | |
| **12.10** | **Let *R* = (*A*, *B*, *C*) is a relation schema with *A* as candidate key. The relation *r*(*R*) is sorted on attribute *C*. Draw two secondary indices on candidate key *A* and non-candidate key *B* filling data for different attributes. *[2007. Marks: 3]***   |  |  | | --- | --- | | ***A*** | ***Pointer*** | | A-1 |  | | A-2 |  | | A-3 |  | | A-4 |  | | A-5 |  |   ***Index file on A***   |  |  |  | | --- | --- | --- | | ***A*** | ***B*** | ***C*** | | A-4 | a | 100 | | A-1 | b | 200 | | A-5 | a | 300 | | A-2 | d | 400 | | A-3 | c | 500 |   ***Relation file***   |  |  | | --- | --- | | ***B*** | ***Pointer*** | | a |  | | b |  | | c |  | | d |  |   ***Index file on B*** |
| **12.11** | **Consider a sequential file sorted with non-primary key. Draw a secondary index on the file for a search key which is the primary key of the file. *[In-course 2, 2005. Marks: 2]***  *See Question and Answer 12.10 – Index file on A.* |
| **12.12** | **What are the disadvantages of index-sequential file? *[In-course 2, 2005. Marks: 1]***  The main disadvantage of the index-sequential ﬁle organization is that performance degrades as the ﬁle grows, both for index lookups and for sequential scans through the data. Although this degradation can be remedied by reorganization of the ﬁle, frequent reorganizations are undesirable. |
| **12.13** | **Construct a B+ tree for the following set of key values and for the (i) *four* (ii) *six* (iii) *eight* pointers that will fit in one node: *[2006, Marks: 3. 2003, Marks: 2. (each)]***  **2, 3, 5, 7, 11, 17, 19, 23, 29, 31**  (i) 1.PNG  (ii) 2.PNG  (iii) 3.PNG |
| **12.14** | **For each B+ tree of *Question and Answer* 12.13, show the form of the tree after each of the following series of operations:**   1. **Insert 9** 2. **Insert 10** 3. **Insert 8** 4. **Delete 23** 5. **Delete 19**  |  |  |  | | --- | --- | --- | | **Structure** | **Operation** | **Form of Tree After Operation** | | (i) | 1. Insert 9 | 1.PNG | | 2. Insert 10 | 1.PNG | | 3. Insert 8 | 1.PNG | | 4. Delete 23 | 1.PNG | | 5. Delete 19 | 1.PNG | | (ii) | 1. Insert 9 | 1.PNG | | 2. Insert 10 | 1.PNG | | 3. Insert 8 | 1.PNG | | 4. Delete 23 | 1.PNG | | 5. Delete 19 | 1.PNG | | (iii) | 1. Insert 9 | 1.PNG | | 2. Insert 10 | 1.PNG | | 3. Insert 8 | 1.PNG | | 4. Delete 23 | 1.PNG | | 5. Delete 19 | 1.PNG | |
| **12.15** | **Construct a B+ tree for the following set of key values:**  **3, 4, 6, 9, 10, 11, 12, 13, 20, 22, 23, 31, 35, 36, 38, 41**  **Assume that the tree is initially empty and values are added in ascending order. The number of pointers that will fit in one node is *four*. *[2004. Marks: 4]***  12  35  6  10  20  23  38  3  4  6  9  10  11  12  13  20  22  23  31  38  41  35  36 |
| **12.16** | **For a B+ tree structure, search key value size = 12 bytes, pointer size = 8 bytes, block size = 388 bytes and there are 1000 search key values. How many nodes are required to access for an index lookup for the worst case? *[In-course 2, 2005. Marks: 2]***  Given,  Search-key value size, *Sk* = 12 bytes  Pointer size, *Sp* = 8 bytes  Block size, *Sb* = 388 bytes  Number of search-key values in file, *K* = 1000  ∴ Maximum number of pointers in a node, *n* = ⎣*Sb* / (*Sk* + *Sp*)⎦  = ⎣388 / (12 + 8)⎦  = 19  ∴ Number of nodes required to access for an index lookup = ⎡log ⎡*n* / 2⎤ (*K*)⎤  = ⎡log ⎡19 / 2⎤ (1000)⎤  = 3  **Answer:** 3. |
| **12.17** | **If an index structure occupies 1000 disk blocks, how many block reads will be required in the best case and the worst case to find a desired index entry, if no index entry is in the overflow block? *[In-course 2, 2005. Marks: 2]***  In the best case, 1 block read will be required to find a desired index entry.  In the worst case, ⎡log 2 (1000)⎤ or 10 block reads will be required. |
| **12.18** | **Describe the format of nodes of B+ tree. *[2002. Marks: 2]***  **Non-leaf nodes:** Each non-leaf node in the tree has between ⎡*n* / 2⎤ and *n* pointers, where *n* is ﬁxed for a particular tree.  **Root node:** The root node can hold fewer than ⎡*n* / 2⎤ pointers, but not less than two pointers unless the tree consists of only one node.  **Leaf nodes:** Each leaf has between ⎡(*n* – 1) / 2⎤ and *n* – 1 values. The ranges of values in each leaf do not overlap.  **Figure:** B+ tree node format. [*n* = 3] |
| **12.19** | **What is the fan-out of B+ tree indexing? *[2002. Marks: 2]***  **OR, What is fan-out of a node? *[In-course 2, 2005. Marks: 1]***  The number of pointers in a node of a B+ tree is called the *fan-out* of the node.  A non-leaf node has a fan-out between ⎡*n* / 2⎤ and *n*. The root node can hold fewer than ⎡*n* / 2⎤ pointers, but not less than two pointers unless the tree consists of only one node. |
| **12.20** | **Why are the leaf nodes of a B+ tree chained together? *[2007, In-course 2, 2005. Marks: 1]***  **OR, Why are nodes of a B+ tree at the leaf level linked? *[2002. Marks: 2]***  The leaf nodes of a B+ tree are chained together to allow for efficient *sequential processing* of the file. |
| **12.21** | **What are the differences between B+ tree structure and in-memory tree structure? *[2007; In-course 2, 2005; Marks: 2]***   |  |  | | --- | --- | | **B+ Tree Structure** | **In-Memory Tree Structure (Binary Tree)** | | 1. Each node is large – typically a disk block – and a node can have a large number of pointers – 200 to 400. | 1. Each node is small and has at most 2 pointers. | | 1. Fat and short. | 1. Thin and tall. | | 1. If there are *K* search-key values in the file, the path for a lookup is no longer than  ⎡log ⎡*n* / 2⎤ (*K*)⎤. | 1. In a balanced binary tree, the path can be of length ⎡log 2 (*K*)⎤. | |
| **12.22** | **What are the differences between B-tree and B+ tree? *[2002, Marks: 3. In-course 2, 2005, Marks: 4]***   |  |  | | --- | --- | | **B+ Tree** | **B-Tree** | | 1. Some search key values may appear twice. | 1. Search key values appear only once. | | 1. Contains redundant storage of search-key values. | 1. Eliminates redundant storage. | | 1. Takes more space than a corresponding B-tree. | 1. Takes less space than a corresponding B+ tree. | | 1. No additional pointer for search key is needed. | 1. As search-keys in non-leaf nodes appear nowhere else in the B-tree, an additional pointer field for each search key in a non-leaf node must be included. | |
| **12.23** | **What are the causes of bucket overflow in a hash file organization? *[2006, Marks: 1. 2005, 2004, 2003, Marks: 2]***  The causes of bucket overflow are :   1. **Insufficient buckets.** Our estimate of the number of records that the relation will have was too low, and hence the number of buckets allotted was not sufficient. 2. **Skew in the distribution of records to buckets.** This may happen either because there are many records with the same search key value, or because the hash function chosen did not have the desirable properties of uniformity and randomness. |
| **12.24** | **What can be done to reduce the occurrence of bucket overflows? *[2006. Marks: 1]***  To reduce the occurrence of overflows, we can:   1. Choose the hash function more carefully, and make better estimates of the relation size. 2. If the estimated size of the relation is *nr* and number of records per block is *fr*, allocate  (*nr* / *fr*) × (1 + *d*) buckets instead of (*nr* / *fr*) buckets. Here *d* is a fudge factor, typically around 0.2. Some space is wasted: about 20% of the space in the buckets will be empty. But the benefit is that some of the skew is handled and the probability of overflow is reduced. |
| **12.25** | **How bucket overflows are handled? *[2003. Marks: 3]***  Bucket overflows can be handled using two techniques:   1. **Closed Hashing.** If records must be inserted into a bucket and the bucket is already full, they are inserted into overflow buckets which are chained together in a linked list. 2. **Open Hashing.** In this technique, the set of buckets is fixed, and there are no overflow chains. Instead, if a bucket is full, the system inserts records in some other bucket in the initial set of buckets. When a new entry has to be inserted, the buckets are examined, starting with the hashed-to slot and proceeding in some *probe sequence*, until an unoccupied slot is found. The probe sequence can be any of the following:    1. **Liner probing.** The interval between probes is fixed (usually 1).    2. **Quadratic probing.** The interval between probes increases by some constant (usually 1) after each probe.    3. **Double hashing.** The interval between probes is computed by another hash function. |
| **12.26** | **What are the causes of skew? *[In-course 2, 2005. Marks: 2]***  Skew can occur for two reasons:   1. Multiple records may have the same search key. 2. The chosen hash function may result in non-uniform distribution of search keys. |
| **12.27** | **For a customer relation, *n*customer = 40000 and *f*customer = 50. If the fudge factor is 0.2, how many buckets will be required to reduce bucket overflow? *[2007, In-course 2, 2005 (similar) Marks: 2]***  Given,  *n*customer = 40000  *f*customer = 50  Fudge factor, *d* = 0.2  ∴ Number of buckets required = (*n*customer / *f*customer) × (1 + *d*)  = (40000 / 50) × (1 + 0.2)  = 960  **Answer:** 960. |
| **12.28** | **Why is hash structure not the best choice for a search key on which range queries are likely? *[2006. Marks: 1]***  A range query cannot be answered efficiently using a hash index; we will have to read all the buckets. This is because key values in the range do not occupy consecutive locations in the buckets; they are distributed uniformly and randomly throughout all the buckets. |
| **12.29** | **Give a comparison of static hashing and dynamic hashing. *[2004. Marks: 3]***  In static hashing, the set of bucket addresses is fixed. As databases grow or shrink over time, use of static hashing results in degradation of performance or wastage of space.  In dynamic hashing, the hash function can be modified dynamically to accommodate the growth or shrinkage of the database. |
| **12.30** | **Compare closed and open hashing. *[2007, In-course 2, 2005. Marks: 2]***   |  |  | | --- | --- | | **Closed Hashing** | **Open Hashing** | | 1. On bucket overflow, records are inserted into overflow buckets which are chained together in a linked list. | 1. Records are inserted in some other bucket in the initial set of buckets using a probe sequence (linear probing, quadratic probing etc.). | | 1. Deletion under closed hashing is simple. | 1. Deletion under open hashing is troublesome. | | 1. Preferable in database systems as insertion-deletion occurs there frequently. | 1. Preferable in compilers and assemblers as they perform only lookup and insertion operations in their symbol tables. | |
| **12.31** | **What are the limitations of hashing? *[In-course 2, 2005. Marks: 2]***  The limitations of hashing are:   1. Hash function must be chosen when the system is implemented and it cannot be changed easily thereafter if the file being indexed grows or shrinks. 2. Since the hash function maps search-key values to a fixed set of bucket addresses, space is wasted if the set of buckets is made large to handle further growth of the file. 3. If the set of buckets is too small, they will contain records of many different search-key values and bucket overflow can occur. As the file grows, performance suffers. |
| **12.32** | **Discuss the use of the hash function in identifying a bucket to search. *[2002. Marks: 3]***  Since it cannot be known at design time precisely which search-key values will be stored in the file, such hash function should be chosen that assigns search-key values to buckets in such a way that the distribution has these qualities:  **1. Uniform:** The distribution is uniform. The hash function assigns each bucket the same number of search-key values from the set of all possible search-key values.  **2. Random:** The distribution is random. In the average case, each bucket will have nearly the same number of values assigned to it regardless of the actual distribution of the search-key values.  Hash functions require careful design. A bad hash function may result in lookup taking time proportional to the number of search keys in the file. A well-designed function gives an average-case lookup time that is a small constant, independent of the number of search-keys in the file. |

1. **Payroll:** The total amount of money paid by a company as salary for all the employers. [↑](#footnote-ref-2)
2. **Note:** Every derived table must have its own alias. So, the aliases *x* and *y* must be put to execute the query successfully. [↑](#footnote-ref-3)
3. **Note:** At exam, don’t write formal definitions if the marks are little, for example in this question (only 3 marks for 3 definitions). [↑](#footnote-ref-4)