Data Base Management Systems , VTU Question paper , June/July 2018

1a. Define DBMS. Discuss the characteristics of Database approach. 6M

Definition : A **database management system (DBMS)** is a collection of programs that enables users to create and maintain a database. The DBMS is hence a *general-purpose software system* that facilitates the processes of *defining*, *constructing*, and *manipulating* databases for various applications. **Defining** a database involves specifying the data types, structures, and constraints for the data to be stored in the database. **Constructing** the database is the process of storing the data itself on some storage medium that is controlled by the DBMS. **Manipulating** a database includes such functions as querying the database to retrieve specific data, updating the database to reflect changes in the mini world, and generating reports from the data.

The characteristics of Database approach.

1. Self-describing nature of a database system:
   A database system includes —in addition to the data stored that is of relevance to the organization— a complete definition/description of the database's structure and constraints. This **meta-data** (i.e., data about data) is stored in the so-called **system catalog**, which contains a description of the structure of each file, the type and storage format of each field, and the various constraints on the data (i.e., conditions that the data must satisfy).
   The system catalog is used not only by users but also by the DBMS software, which certainly needs to "know" how the data is structured/organized in order to interpret it in a manner consistent with that structure.
2. Insulation between programs and data, and data abstraction:
   DBMS access programs do not require changes in most cases. The structure of data files is stored in the DBMS catalog separately from the access programs. We call this property **program-data independence.** in a DBMS environment, we just need to change the description of records in the catalog to reflect the inclusion of the new data item; no programs are changed. The next time a DBMS program refers to the catalog, the new structure will be accessed and used.
   Data Abstraction:
   - A data model is used to hide storage details and present the users with a conceptual view of the database.
   - Programs refer to the data model constructs rather than data storage details
3. Support of multiple views of the data.
   A database typically has many users, each of whom may require a different perspective or **view** of the database. A view may be a subset of the database or it may contain **virtual data** that is derived from the database files but is not explicitly stored. Some users may not need to be aware of whether the data they refer to is stored or derived. A multiuser DBMS whose users have a variety of applications must provide facilities for defining multiple views. For example, one user of the database of Figure 01.02 may be interested only in the transcript of each student; the view for this user .A second user, who is interested only in checking that students have taken all the prerequisites of each course they register for, may require the view .

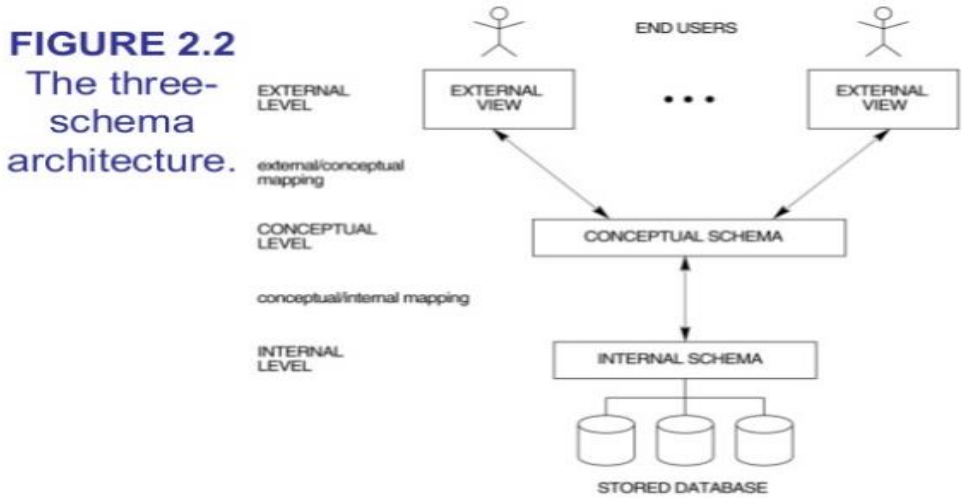
4. Sharing of data and multiuser transaction processing. :
   Arising from this is the need for concurrency control, which is supposed to ensure that several users trying to update the same data do so in a "controlled" manner so that the results of the updates are as though they were done in some sequential order. This gives rise to the concept of a transaction, which is a process that makes one or more accesses to a database and which must have the appearance of executing in isolation from all other transactions and of being atomic. Applications such as airline reservation systems are known as online transaction processing applications.

1 b. Describe three – scheme architecture, with a neat diagram. 4M

The goal of the three-schema architecture, is to separate the user applications and the physical database. In this architecture, schemas can be defined at the following three levels:

1. The internal level has an internal schema, which describes the physical storage structure of the database. The internal schema uses a physical data model and describes the complete details of data storage and access paths for the database.
2. The conceptual level has a conceptual schema, which describes the structure of the whole database for a community of users. The conceptual schema hides the details of physical storage structures and concentrates on describing entities, data types, relationships, user operations, and constraints. A high-level data model or an implementation data model can be used at this level.
3. The external or view level includes a number of external schemas or user views. Each external schema describes the part of the database that a particular user group is interested in and hides the rest of the database from that user group. A high-level data model or an implementation data model can be used at this level.

The three-schema architecture is a convenient tool for the user to visualize the schema levels in a database system. Most DBMSs do not separate the three levels completely, but support the three- schema architecture to some extent. Some DBMSs may include physical-level details in the conceptual schema. In most DBMSs that support user views, external schemas are specified in the same data model that describes the conceptual-level information. Some DBMSs allow different data models to be used at the conceptual and external levels.



1C. Discuss in detail about the advantages of DBMs over traditional file system. 6M

1.Controlling Redundancy
2.Restricting Unauthorized Access
3.Providing Persistent Storage for Program Objects and Data Structures.
4.Permitting Inference and Actions Using Rules
5 Providing Multiple User Interfaces
6 Representing Complex Relationships Among Data
7 Enforcing Integrity Constraints
8 Providing Backup and Recovery

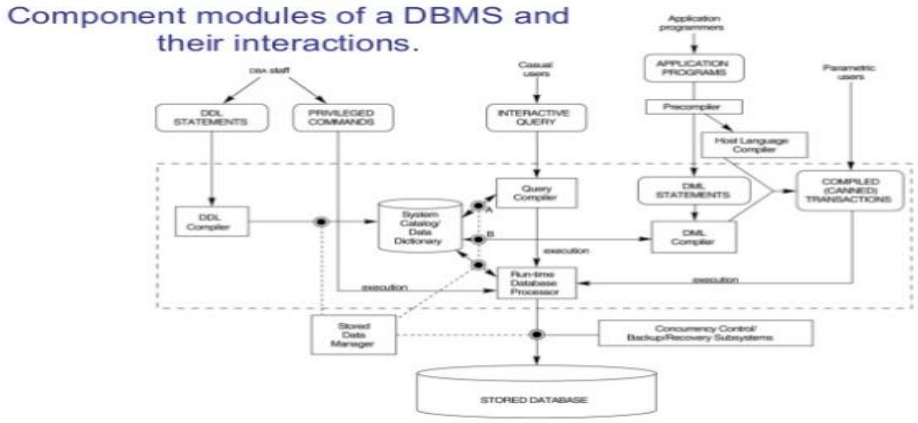
2a. Illustrate component modules of DBMS and their interactions, with a neat diagram. 6M

The database and the DBMS catalog are usually stored on disk. Access to the disk is controlled primarily by the **operating system (OS),** which schedules disk input/output. A higher-level **stored data manager** module of the DBMS controls access to DBMS information that is stored on disk, whether it is part of the database or the catalog. The dotted lines and circles marked A, B, C, D, and E in Figure illustrate accesses that are under the control of this stored data manager. The stored data manager may use basic OS services for carrying out low-level data transfer between the disk and computer main storage, but it controls other aspects of data transfer, such as handling buffers in main memory. Once the data is in main memory buffers, it can be processed by other DBMS modules, as well as by application programs.

The **DDL compiler** processes schema definitions, specified in the DDL, and stores descriptions of the schemas (meta-data) in the DBMS catalog. The catalog includes information such as the names of files, data items, storage details of each file, mapping information among schemas, and constraints, in addition to many other types of information that are needed by the DBMS modules. DBMS software modules then look up the catalog information as needed.

The **run-time database processor** handles database accesses at run time; it receives retrieval or update operations and carries them out on the database. Access to disk goes through the stored data manager. The **query compiler** handles high-level queries that are entered interactively. It parses, analyzes, and compiles or interprets a query by creating database access code, and then generates calls to the run-time processor for executing the code.

The **pre-compiler** extracts DML commands from an application program written in a host programming language. These commands are sent to the **DML compiler** for compilation into object code for database access. The rest of the program is sent to the host language compiler. The object codes for the DML commands and the rest of the program are linked, forming a canned transaction whose executable code includes calls to the runtime database processor.



2b. Briefly explain any two types of attributes in E-R model. 3M

- simple/atomic vs. composite
- single-valued vs. multi-valued (or set-valued)
- stored vs. derived (Note from instructor: this seems like an implementation detail that ought not be considered at this (high) level of abstraction.)

A composite attribute is one that is composed of smaller parts. An atomic attribute is indivisible or indecomposable.

Example 1: A Birth Date attribute can be viewed as being composed of (sub-)attributes month, day, and year (each of which would probably be viewed as being atomic).

To describe the structure of a composite attribute, one can draw a tree (as in the aforementioned Figure 7.4). In case we are limited to using text, it is customary to write its name followed by a parenthesized list of its sub-attributes. For the examples mentioned above, we would write

Birth Date(Month, Day, Year)

Address(StreetAddr (StrNum, StrName, AptNum), City, State, Zip)

Single- vs. multi-valued attribute: Consider a PERSON entity. The person it represents has (one) SSN, (one) date of birth, (one, although composite) name, etc. But that person may have zero or more academic degrees, dependents, or (if the person is a male living in Utah) spouses! How can we model this via attributes Academic Degrees, Dependents, and Spouses? One way is to allow such attributes to be multi-valued (perhaps set-valued is a better term), which is to say that we assign to them a (possibly empty) set of values rather than a single value.

To distinguish a multi-valued attribute from a single-valued one, it is customary to enclose the former within curly braces (which makes sense, as such an attribute has a value that is a set, and curly braces are traditionally used to denote sets).

2C.Define the following terms, with an example for each:
i)Entity set ii) Cardinality ratio iii) Participation iv) Weak Entity 7M

Entity set : An entity is an object that exists and is distinguishable from other objects. For instance,
John Harris with S.I.N. 890-12-3456 is an entity, as he can be uniquely identified as one particular person in the universe.

An entity may be concrete (a person or a book, for example) or abstract (like a holiday or a concept).

An entity set is a set of entities of the same type (e.g., all persons having an account at a bank).

Cardinality ratio : In database design, the cardinality or fundamental principle of one data table with respect to another is a critical aspect. The relationship of one to the other must be precise and exact between each other in order to explain how each table links together.

Participation : A participation constraint defines the number of times an object in an object class can participate in a connected relationship set. Every connection of a relationship set must have a participation constraint. However, participation constraints do not apply to relationships.

Weak entity set : An entity set that does not have a primary key is referred to as a weak entity set. The existence of a weak entity set depends on the existence of a strong entity set; it must relate to the strong set via a one-to-many relationship set.

3a. Summarize join operations in relational algebra. 8M

The **JOIN** operation, denoted by , is used to combine *related tuples* from two relations into single tuples. This operation is very important for any relational database with more than a single relation, because it allows us to process relationships among relations. To illustrate join, suppose that we want to retrieve the name of the manager of each department. To get the manager's name, we need to combine each department tuple with the employee tuple whose SSN value matches the MGRSSN value in the department tuple. We do this by using the JOIN operation, and then projecting the result over the necessary attributes, as follows:

$$DEPT\_MGR \leftarrow DEPARTMENT_{MGRSSN=SSN} EMPLOYEE$$

$$RESULT \leftarrow \pi_{DNAME, LNAME, FNAME}(DEPT\_MGR)$$

MGRSSN is a foreign key and that the referential integrity constraint plays a role in having matching tuples in the referenced relation EMPLOYEE. The general form of a JOIN operation on two relations R(A1, A2, . . ., An) and S(B1, B2, . . ., Bm) is: R<join condition>S
The result of the JOIN is a relation Q with n + m attributes Q(A1, A2, . . ., An, B1, B2, . . ., Bm) in that order; Q has one tuple for each combination of tuples—one from R and one from S—*whenever the combination satisfies the join condition.* This is the main difference between CARTESIAN PRODUCT and JOIN: in JOIN, only combinations of tuples *satisfying the join condition* appear in the result, whereas in the CARTESIAN PRODUCT *all* combinations of tuples

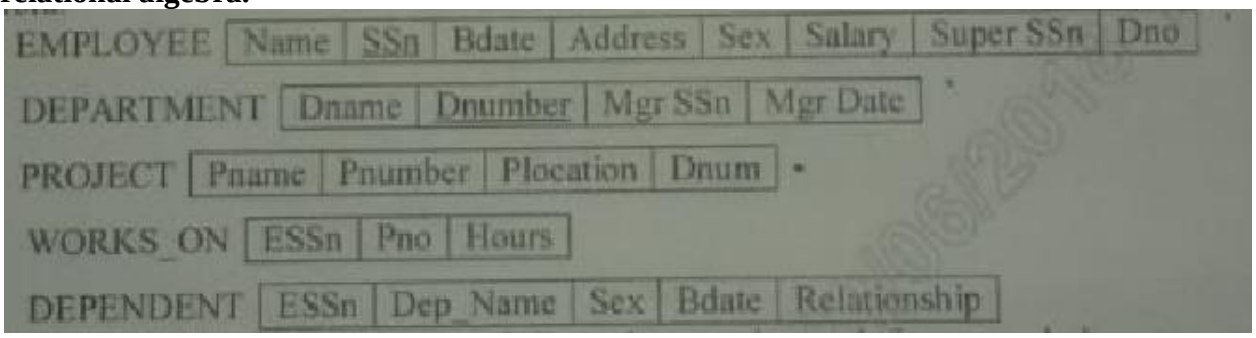
are included in the result. The join condition is specified on attributes from the two relations R and S and is evaluated for each combination of tuples. Each tuple combination for which the join condition evaluates to true is included in the resulting relation Q *as a single combined tuple.*
The most common JOIN involves join conditions with equality comparisons only. Such a JOIN, where the only comparison operator used is =, is called an **EQUIJOIN.**

new operation called **NATURAL JOIN**—denoted by *—was created to get rid of the second (superfluous) attribute in an EQUIJOIN condition (Note 10). The standard definition of NATURAL JOIN requires that the two join attributes (or each pair of join attributes) have the same name in both relations. If this is not the case, a renaming operation is applied first. In the following example, we first rename the DNUMBER attribute of DEPARTMENT to DNUM—so that it has the same name as the DNUM attribute in PROJECT—then apply NATURAL JOIN:

PROJ_DEPTãPROJECT * q(DNAME, DNUM,MGRSSN,MGRSTARTDATE)(DEPARTMENT)

The attribute DNUM is called the **join attribute.**

3b. Consider the following relational schema and answer the following queries using relational algebra.



i) Retrieve the name and address of all employees who work for research department.

$$RESEARCH\_DEPT \leftarrow \sigma_{DNAME='Research'}(DEPARTMENT)$$

$$RESEARCH\_EMPS \leftarrow (RESEARCH\_DEPT_{DNUMBER=DNO} EMPLOYEE)$$

$$RESULT \leftarrow \pi_{FNAME,\ LNAME,\ ADDRESS}(RESEARCH\_EMPS)$$

ii) Find the names of employees who work on all the projects controlled by Dnumber 5.

$$DEPT5\_PROJS(PNO) \leftarrow \pi_{PNUMBER}(\sigma_{DNUM=5}(PROJECT))$$

$$EMP\_PRJO(SSN, PNO) \leftarrow \pi_{ESSN,\ PNO}(WORKS\_ON)$$

$$RESULT\_EMP\_SSNS \leftarrow EMP\_PRJO \div DEPT5\_PROJS$$

$$RESULT \leftarrow \pi_{LNAME,\ FNAME}(RESULT\_EMP\_SSNS * EMPLOYEE)$$

iii) For every project located in 'Stafford ' list the project no, controlling department number, department managers, name, birth date.

STAFFORD_PROJS ← $\sigma_{PLOCATION='Stafford'}$(PROJECT)

CONTR_DEPT ← (STAFFORD_PROJS $\bowtie_{DNUM=DNUMBER}$ DEPARTMENT)

PROJ_DEPT_MGR ← (CONTR_DEPT $\bowtie_{MGRSSN=SSN}$ EMPLOYEE)

RESULT ← $\pi_{PNUMBER, DNUM, LNAME, ADDRESS, BDATE}$(PROJ_DEPT_MGR)

iv) Retrieve the names of employee who have no dependents.

ALL_EMPS ← π_{SSN}(EMPLOYEE)

EMPS_WITH_DEPS(SSN) ← π_{ESSN}(DEPENDENT)

EMPS_WITHOUT_DEPS ← (ALL_EMPS - EMPS_WITH_DEPS)

RESULT ← $\pi_{LNAME, FNAME}$(EMPS_WITHOUT_DEPS * EMPLOYEE)

4a. Demonstrate ER-to- Relational mapping algorithm, with an example. 8M

Step 1: Mapping of Regular Entity Types

Example: We create the relations EMPLOYEE, DEPARTMENT, and PROJECT in the relational schema corresponding to the regular entities in the ER diagram.

SSN, DNUMBER, and PNUMBER are the primary keys for the relations EMPLOYEE, DEPARTMENT, and PROJECT.

Step 2: Mapping of Weak Entity Types

Example: Create the relation DEPENDENT in this step to correspond to the weak entity type DEPENDENT. Include the primary key SSN of the EMPLOYEE relation as a foreign key attribute of DEPENDENT (renamed to ESSN). The primary key of the DEPENDENT relation is the combination {ESSN, DEPENDENT_NAME} because DEPENDENT_NAME is the partial key of DEPENDENT.

Step 3: Mapping of Binary 1:1 Relation Types

There are three possible approaches:

Foreign Key approach: Choose one of the relations-say S-and include a foreign key in S the primary key of T. It is better to choose an entity type with total participation in R in the role of S.

Example: 1:1 relation MANAGES is mapped by choosing the participating entity type DEPARTMENT to serve in the role of S, because its participation in the MANAGES relationship type is total.

Merged relation option: An alternate mapping of a 1:1 relationship type is possible by merging the two entity types and the relationship into a single relation. This may be appropriate when both participations are total.
Cross-reference or relationship relation option: The third alternative is to set up a third relation R for the purpose of cross-referencing the primary keys of the two relations S and T representing the entity types.
Step 4: Mapping of Binary 1:N Relationship Types.
Example: 1:N relationship types WORKS_FOR, CONTROLS, and SUPERVISION. For WORKS_FOR we include the primary key DNUMBER of the DEPARTMENT relation as foreign key in the EMPLOYEE relation and call it DNO.
Step 5: Mapping of Binary M:N Relationship Types.
Example: The M:N relationship type WORKS_ON from the ER diagram is mapped by creating a relation WORKS_ON in the relational database schema.
The primary keys of the PROJECT and EMPLOYEE relations are included as foreign keys in WORKS_ON and renamed PNO and ESSN, respectively. Attribute HOURS in WORKS_ON represents the HOURS attribute of the relation type. The primary key of the WORKS_ON relation is the combination of the foreign key attributes {ESSN, PNO}.
Step 6: Mapping of Multi valued attributes.
Example: The relation DEPT_LOCATIONS is created. The attribute DLOCATION represents the multi valued attribute LOCATIONS of DEPARTMENT, while DNUMBER- as foreign key-represents the primary key of the DEPARTMENT relation. The primary key of R is the combination of {DNUMBER, DLOCATION}.
Step 7: Mapping of N-ary Relationship Types.
Example: The relationship type SUPPY in the ER on the next slide.
This can be mapped to the relation SUPPLY shown in the relational schema, whose primary key is the combination of the three foreign keys {SNAME, PARTNO, PROJNAME}

4b. Illustrate unary relational operations, with appropriate syntax and example. 8M

SELECT and PROJECT are unary relational operations

The **SELECT** operation is used to select a *subset* of the tuples from a relation that satisfy a selection condition.

For example, to select the EMPLOYEE tuples whose department is 4, or those whose salary is greater than $30,000, we can individually specify each of these two conditions with a SELECT operation as follows: sDNO=4(EMPLOYEE)

sSALARY>30000(EMPLOYEE)

the SELECT operation is denoted by s<selection condition>(R) where the symbol s (sigma) is used to denote the SELECT operator, and the selection condition is a Boolean expression specified on the attributes of relation R. The SELECT operator is unary; that is, it is applied to a single relation. Moreover, the selection operation is applied to *each tuple individually;* hence, selection conditions cannot involve more than one tuple. The degree of the relation resulting from a SELECT operation is the same as that of R. The number of tuples in the resulting relation is always *less than or equal to* the number of tuples in R.

The PROJECT Operation

The **PROJECT** operation, selects certain *columns* from the table and discards the other columns. We use the PROJECT operation to *project* the relation over these attributes only. For example, to list each employee's first and last name and salary, we can use the PROJECT operation as follows: pLNAME, FNAME, SALARY(EMPLOYEE) .The general form of the PROJECT operation is p<attribute list>(R) where p (pi) is the symbol used to represent the PROJECT operation and <attribute list> is a list of attributes from the attributes of relation R. The result of the PROJECT operation has only the attributes specified in <attribute list> and *in the same order as they appear in the list.* Hence, its degree is equal to the number of attributes in <attribute list>.

If the attribute list includes only non key attributes of R, duplicate tuples are likely to occur; the PROJECT operation *removes any duplicate tuples,* so the result of the PROJECT operation is a set of tuples and hence a valid relation .This is known as duplicate elimination**.**

The number of tuples in a relation resulting from a PROJECT operation is always less than or equal to the number of tuples in R. If the projection list is a super key of R—that is, it includes some key of R—the resulting relation has the *same number* of tuples as R.

5a. Bring out the different clauses of SELECT-FROM-WHERE-GROUP-HAVING with an example for each. 08M

An SQL query can contain three types of clauses, the **select** clause, the **from** clause, and the **where** clause.

- The **select** clause is used to list the attributes desired in the result of a query.
- The **from** clause is a list of the relations to be accessed in the evaluation of the query.
- The **where** clause is a predicate involving attributes of the relation in the **from** clause. A typical SQL query has the form **select** *A*1, *A*2, . . . , *An* **from** *r*1, *r*2, . . . , *rm* **where** *P*;

  Each *Ai* represents an attribute, and each *ri* a relation. *P* is a predicate. If the **where** clause is omitted, the predicate *P* is **true**. The clauses must be written in the order **select**, **from**, **where**, the easiest way to understand the operations specified by the query is to consider the clauses in operational order: first **from**, then **where**, and then **select**. The **from** clause by itself defines a Cartesian product of the relations listed**.** A **select** clause of the form **select** * indicates that all attributes of the result relation of the **from** clause are selected.

When an SQL query uses grouping, it is important to ensure that the only attributes that appear in the **select** statement without being aggregated are those that are present in the **group by** clause. At times, it is useful to state a condition that applies to groups rather than to tuples. For example, we might be interested in only those departments where the average salary of the instructors is more than $42,000. This condition does not apply to a single tuple; rather, it applies to each group constructed by the **group by** clause. To express such a query, we use the **having** clause of SQL. SQL applies predicates in the **having** clause after groups have been formed, so aggregate functions may be used.

We express this query in SQL as follows:

select *dept name*, **avg** (*salary*) **as** *avg salary* **from** *instructor* **group by** *dept name* **having avg** (*salary*) > 42000;

As was the case for the **select** clause, any attribute that is present in the **having** clause without being aggregated must appear in the **group by** clause, otherwise the query is treated as erroneous. The meaning of a query containing aggregation, **group by**, or **having** clauses is defined by the following sequence of operations:

dept_name	*avg(avg_salary)*
Physics	91000
Elec. Eng.	80000
Finance	85000
Comp. Sci.	77333
Biology	72000
History	61000

The result relation for the query "Find the average salary of instructors in those departments where the average salary is more than $42,000."

1. As was the case for queries without aggregation, the **from** clause is first evaluated to get a relation.
2. If a **where** clause is present, the predicate in the **where** clause is applied on the result relation of the **from** clause.
3. Tuples satisfying the **where** predicate are then placed into groups by the **group by** clause if it is present. If the **group by** clause is absent, the entire set of tuples satisfying the **where** predicate is treated as being in one group.
4. The **having** clause, if it is present, is applied to each group; the groups that do not satisfy the **having** clause predicate are removed.
5. The **select** clause uses the remaining groups to generate tuples of the result of the query, applying the aggregate functions to get a single result tuple for each group.

To illustrate the use of both a **having** clause and a **where** clause in the same query, we consider the query

"For each course section offered in 2009, find the average total credits (*tot cred*) of all students enrolled in the section, if the section had at least 2 students."

select *course id*, *semester*, *year*, *sec id*, **avg** (*tot cred*) **from** *takes* **natural join** *student* **where** *year* = 2009 **group by** *course id*, *semester*, *year*, *sec id* **having count** (*ID*) >= 2;

5b. Explain set membership and set comparison operations for nested sub queries. 08M

Set Membership

SQL allows testing tuples for membership in a relation. The **in** connective tests for set membership, where the set is a collection of values produced by a **select** clause. The **not in** connective tests for the absence of set membership .As an illustration, reconsider the query

"Find all the courses taught in the both the Fall 2009 and Spring 2010 semesters".

select *course id* **from** *section* **where** *semester* = 'Spring' **and** *year*= 2010

We then need to find those courses that were taught in the Fall 2009 and that appear in the set of courses obtained in the sub query. We do so by nesting the sub query in the **where** clause of an outer query.

The resulting query is **select distinct** *course id* **from** *section* **where** *semester* = 'Fall' **and** *year*= 2009 **and** *course id* **in** (**select** *course id* **from** *section* **where** *semester* = 'Spring' **and** *year*= 2010); We use the **not in** construct in a way similar to the **in** construct.

For example, to find all the courses taught in the Fall 2009 semester but not in the Spring 2010 semester,

select distinct *course id* **from** *section* **where** *semester* = 'Fall' **and** *year*= 2009 **and** *course id* **not in** (**select** *course id* **from** *section* **where** *semester* = 'Spring' **and** *year*= 2010);

The **in** and **not in** operators can also be used on enumerated sets.

Set Comparison

Consider the query "Find the names of all instructors whose salary is greater than at least one instructor in the Biology department."

The phrase "greater than at least one" is represented in SQL by > **some**.

select *name* **from** *instructor* **where** *salary* > **some** (**select** *salary* **from** *instructor* **where** *dept name* = 'Biology'); The sub query: (**select** *salary* **from** *instructor* **where** *dept name* = 'Biology')

The > **some** comparison in the **where** clause of the outer **select** is true if the *salary* value of the tuple is greater than at least one member of the set of all salary values for instructors in Biology.

SQL also allows < **some**, <= **some**, >= **some**, = **some**, and <> **some** comparisons.

The construct > **all** corresponds to the phrase "greater than all." Using this construct, we write the query as follows: **select** *name* **from** *instructor* **where** *salary* > **all** (**select** *salary* **from** *instructor* **where** *dept name* = 'Biology'); As it does for **some**, SQL also allows < **all**, <= **all**, >= **all**, = **all**, and <> **all** comparisons.

<> **all** is identical to **not in**, whereas =**all** is *not* the same as **in**.

As another example of set comparisons, consider the query

"Find the departments that have the highest average salary."We begin by writing a query to find all average salaries, and then nest it as a sub query of a larger query that finds average salaries:

select *dept name* **from** *instructor* **group by** *dept name* **having avg** (*salary*) >= **all** (**select avg** (*salary*) **from** *instructor* **group by** *dept name*);

6a. Write a short note on granting and revoking of privileges in SQL. 5M

The SQL standard includes the privileges **select**, **insert**, **update**, and **delete**. The privilege **all privileges** can be used as a short form for all the allowable privileges. A user who creates a new relation is given all privileges on that relation automatically.

The **grant** statement is used to confer authorization. **grant** <privilege list> **on** <relation name or view name> **to** <user/role list>;

The *privilege list* allows the granting of several privileges in one command. The **select** authorization on a relation is required to read tuples in the relation. The following **grant** statement grants database users Amit and Satoshi **select** authorization on the *department* relation: **grant select on** *department* **to** *Amit, Satoshi*;

This allows those users to run queries on the *department* relation.

The **update** authorization on a relation allows a user to update any tuple in the relation. The

update authorization may be given either on all attributes of the relation or on only some. If **update** authorization is included in a **grant** statement, the list of attributes on which update authorization is to be granted optionally appears in parentheses immediately after the **update** keyword. If the list of attributes is omitted, the update privilege will be granted on all attributes of the relation.

This **grant** statement gives users Amit and Satoshi update authorization on the *budget* attribute of the *department* relation: **grant update** (*budget*) **on** *department* **to** *Amit*, *Satoshi*; The **insert** authorization on a relation allows a user to insert tuples into the relation.

The **insert** privilege may also specify a list of attributes; any inserts to the relation must specify only these attributes, and the system either gives each of the remaining attributes default values (if a default is defined for the attribute) or sets them to *null*. The **delete** authorization on a relation allows a user to delete tuples from a relation. The user name **public** refers to all current and future users of the system. Thus, privileges granted to **public** are implicitly granted to all current and future users.

By default, a user/role that is granted a privilege is not authorized to grant that privilege to another user/role. SQL allows a privilege grant to specify that the recipient may further grant the privilege to another user.

To revoke an authorization, we use the **revoke** statement. It takes a form almost identical to that of **grant**:

revoke <privilege list>
on <relation name or view name>
from <user/role list>;

Thus, to revoke the privileges that we granted previously, we write **revoke select on** *department* **from** *Amit*, *Satoshi*;

revoke update (budget) **on** *department* **from** *Amit*, *Satoshi*; Revocation of privileges is more complex if the user from whom the privilege is revoked has granted the privilege to another user.

6b. Consider the following schema and solve the following queries in SQL. 6M

BRANCH(**Branchid**,Branchname,HOD)
STUDENT(**USN**,Name,Address,Branchid,sem)
BOOK(Bookid,Bookname,Authorid,Publisher,Branchid)
AUTHOR(Authorid,Authorname,Country,age)
BORROW(USN,Bookid,Borrowed_Date)

I. List the details of Students who are all Studying in 2nd sem MCA.

```
SELECT * FROM STUDENT S, BRANCH B WHERE S.BRANCHID=B.BRANCHID
  AND S.SEM = '02' AND B.BRANCHNAME = 'MCA';
```

II. Display the student details who borrowed more than two books.

```
SELECT * FROM STUDENT WHERE USN IN (SELECT
                    USN FROM BORROW GROUP BY USN HAVING
COUNT(USN)>=2);
```

III. Display the student details who borrowed books of more than one Author.

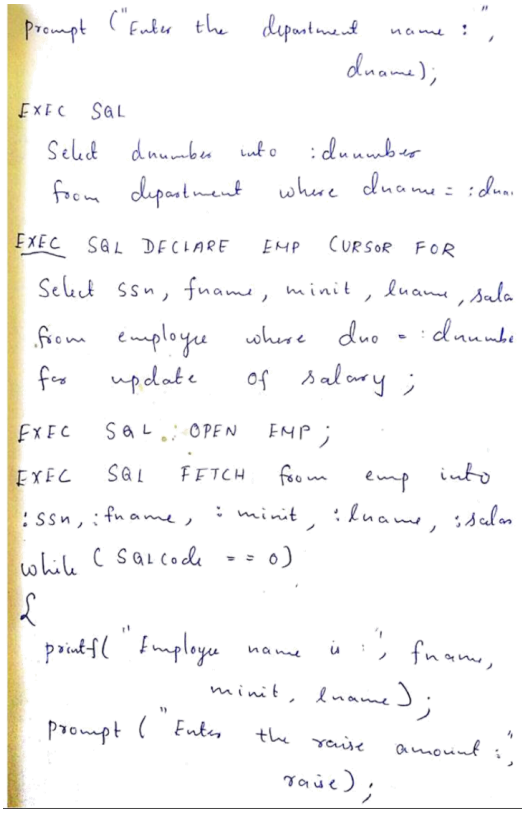
```
SELECT * FROM STUDENT  WHERE USN IN (SELECT USN FROM BORROW
BRW, BOOK K  WHERE BRW.BOOKID = K.BOOKID GROUP BY USN HAVING
                                        COUNT(AUTHORID)>1);
```

IV. Display the USN, Student name, Branch_name, Book_name of 2nd sem MCA Students.

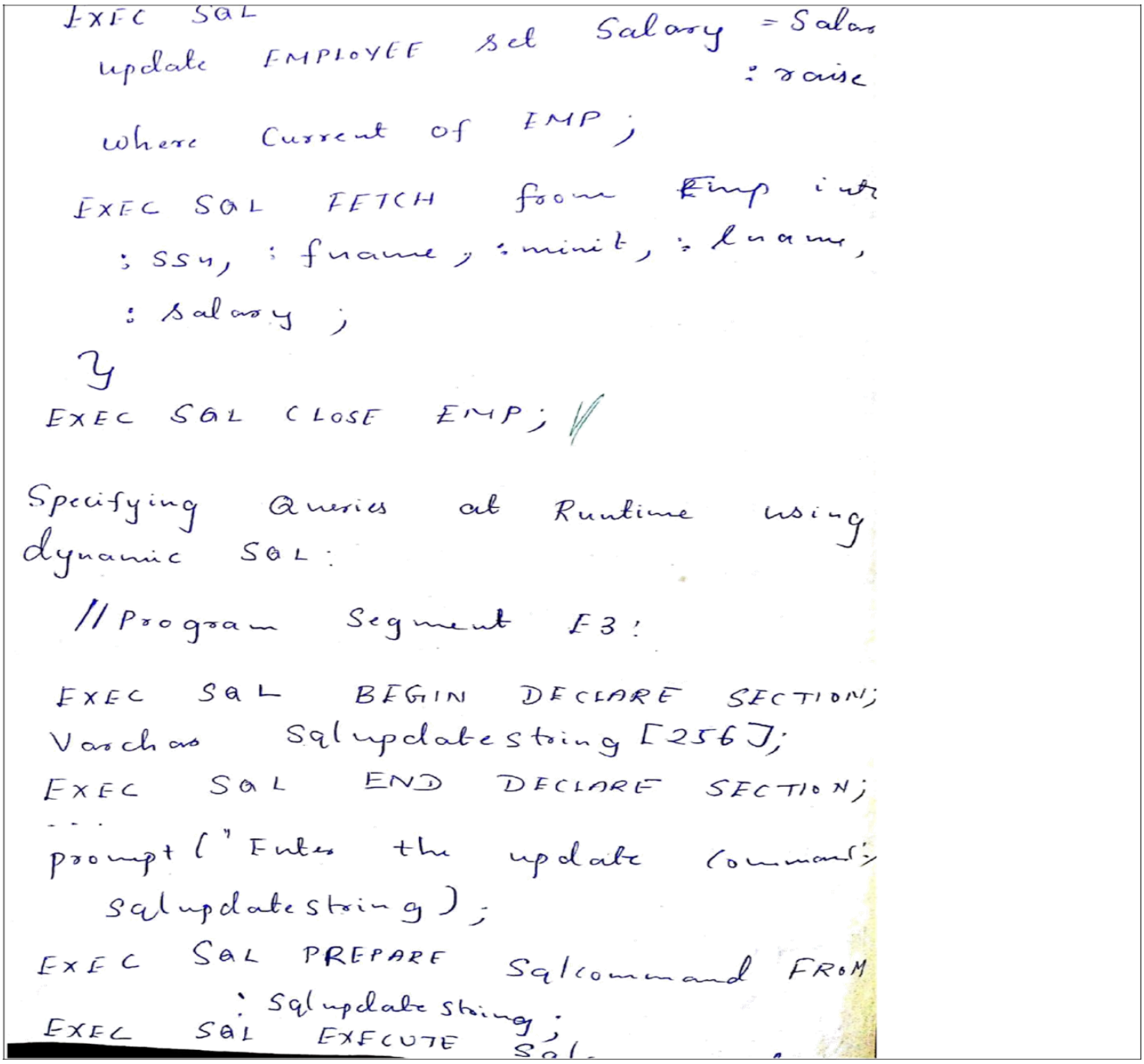
```
SELECT S.USN, S.NAME, B.BRANCHNAME, BK.BOOKNAME  FROM
  STUDENT S, BRANCH B, BOOK BK, BORROW BRW
        WHERE
              S.USN = BRW.USN AND
              S.BRANCHID = B.BRANCHID AND
              Bk.BOOKID = BRW.BOOKID AND     S.SEM = '02';
```

6c. Give a brief explanation on embedded SQL. **5M**

Specify the cursor using a DECLARE CURSOR statement.
Perform the query and build the result table using the OPEN statement.
Retrieve rows one at a time using the FETCH statement.
Process rows with the DELETE or UPDATE statements (if required).
Terminate the cursor using the CLOSE statement.



```
Prompt ("Enter the department name : ",
                        dname);
EXEC SQL
  Select dnumber into :dnumber
  from department where dname = :dna
EXEC SQL DECLARE EMP CURSOR FOR
  Select ssn, fname, minit, lname, sala
  from employee where dno = :dnumbe
  for update of salary;
EXEC SQL OPEN EMP;
EXEC SQL FETCH from emp into
:ssn, :fname, :minit, :lname, :salar
while (SQLcode == 0)
{
  printf("Employee name is : ", fname,
              minit, lname);
  Prompt ("Enter the raise amount :",
                    raise);
```



```
EXEC SQL
  update EMPLOYEE set Salary = Salary
                              : raise
  where Current of EMP;
EXEC SQL FETCH from Emp into
  : SSN, : fname, : minit, : lname,
  : salary ;
}
EXEC SQL CLOSE EMP;
```

Specifying Queries at Runtime using dynamic SQL:

```
// Program Segment E3:

EXEC SQL BEGIN DECLARE SECTION;
Varchar Sqlupdatestring [256];
EXEC SQL END DECLARE SECTION;
...
prompt ("Enter the update command:
   Sqlupdatestring);
EXEC SQL PREPARE Sqlcommand FROM
        : Sqlupdatestring;
EXEC SQL EXECUTE Sql
```

7a. Demonstrate the informal design guidelines for the relational schema. **8M**

INFORMAL DESIGN GUIDELINES FOR RELATIONAL SCHEMA

1.Semantics of the Attributes
2.Reducing the Redundant Value in Tuples.
3.Reducing Null values in Tuples.
4.Dissallowing spurious Tuples.

1. **Semantics of the Attributes** : Whenever we are going to form relational schema there should be some meaning among the attributes. This meaning is called semantics. This semantics relates one attribute to another with some relation. Eg: USN No Student name Sem

2. **Reducing the Redundant Value in Tuples** Mixing attributes of multiple entities may cause problems Information is stored redundantly wasting storage Problems with update anomalies Insertion anomalies Deletion anomalies Modification anomalies Student name Sem Eg: Dept No Dept Name If we integrate these two and is used as a single table i.e Student Table USN No Student name Sem Dept No Dept Name Here whenever if we insert the tuples there may be 'N' students in one department, so Dept No, Dept Name values are repeated 'N' times which leads to data redundancy. Another problem is updata anomalies ie if we insert new dept that has no students. If we delet the last student of a dept, then whole information about that department will be deleted If we change the value of one of the attributes of a particular table the we must update the tuples of all the students belonging to the department else Database will become inconsistent. Note: Design in such a way that no insertion ,deletion, modification anomalies will occur
3. **Reducing Null values in Tuples**. Note: Relations should be designed such that their tuples will have as few NULL values as possible Attributes that are NULL frequently could be placed in separate relations (with the primary key) Reasons for nulls: attribute not applicable or invalid attribute value unknown (may exist) value known to exist, but unavailable
4. **Disallowing spurious Tuples** Bad designs for a relational database may result in erroneous results for certain JOIN operations The "lossless join" property is used to guarantee meaningful results for join operations Note: The relations should be designed to satisfy the lossless join condition. No spurious tuples should be generated by doing a natural-join of any relations.

7b. Define Functional Dependency. List out the six inference rules of functional dependency. 4M

In relational database theory, a functional dependency is a constraint between two sets of attributes in a relation from a database. In other words, functional dependency is a constraint that describes the relationship between attributes in a relation.
Let A, B and C and D be arbitrary subsets of the set of attributes of the giver relation R, and let AB be the union of A and B. Then,⇒→
Reflexivity
If B is subset of A, then A → B
Augmentation
If A → B, then AC → BC
Transitivity:
If A → B and B → C, then A → C.
Projectivity or Decomposition Rule
If A → BC, Then A → B and A → C
Proof :
Step 1 : A → BC (GIVEN)
Step 2 : BC → B (Using Rule 1, since B ⊆ BC)
Step 3 : A → B (Using Rule 3, on step 1 and step 2)
Union or Additive Rule :
If A→B, and A→C Then A→BC.

Proof :
Step 1 : A → B (GIVEN)
Step 2 : A → C (given)
Step 3 : A → AB (using Rule 2 on step 1, since AA=A)
Step 4 : AB → BC (using rule 2 on step 2)
[5]
Step 5 : A → BC (using rule 3 on step 3 and step 4)

Pseudo Transitive Rule :

If A → B, DB → C, then DA → C
Proof :
Step 1 : A → B (Given)
Step 2 : DB → C (Given)
Step 3 : DA → DB (Rule 2 on step 1)
Step 4 : DA → C (Rule 3 on step 3 and step 2)

7c. Define Triggers. Brief about triggers with syntax and example. 4M

Triggers are stored programs, which are automatically executed or fired when some events occur. Triggers are, in fact, written to be executed in response to any of the following events −
A database manipulation (DML) statement (DELETE, INSERT, or UPDATE)
A database definition (DDL) statement (CREATE, ALTER, or DROP).
A database operation (SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN).
Triggers can be defined on the table, view, schema, or database with which the event is associated.
To demonstrate triggers we will be using the CUSTOMERS table we had created and used in the previous chapters −

Select * from customers;

```
+----+----------+-----+-----------+----------+
| ID | NAME | AGE | ADDRESS | SALARY |
+----+----------+-----+-----------+----------+
| 1 | Ramesh | 32 | Ahmedabad | 2000.00 |
| 2 | Khilan | 25 | Delhi | 1500.00 |
| 3 | kaushik | 23 | Kota | 2000.00 |
| 4 | Chaitali | 25 | Mumbai | 6500.00 |
| 5 | Hardik | 27 | Bhopal | 8500.00 |
| 6 | Komal | 22 | MP | 4500.00 |
+----+----------+-----+-----------+----------+
```

The following program creates a row-level trigger for the customers table that would fire for INSERT or UPDATE or DELETE operations performed on the CUSTOMERS table. This trigger will display the salary difference between the old values and new values –

```
CREATE OR REPLACE TRIGGER display_salary_changes
BEFORE DELETE OR INSERT OR UPDATE ON customers
FOR EACH ROW  WHEN (NEW.ID > 0)
DECLARE
sal_diff number;
```

```
BEGIN
        sal_diff := :NEW.salary - :OLD.salary;
        dbms_output.put_line('Old salary: ' || :OLD.salary);
        dbms_output.put_line('New salary: ' || :NEW.salary);
        dbms_output.put_line('Salary difference: ' || sal_diff);
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −
Trigger created.

8a. What is Normalization? Explain the 1NF, 2NF and 3NF with example. 10M

Normalization rule are divided into following normal form.
First Normal Form Second Normal Form Third Normal Form
First Normal Form (1NF)
As per First Normal Form, no two Rows of data must contain repeating group of information i.e each set of column must have a unique value, such that multiple columns cannot be used to fetch the same row. Each table should be organized into rows, and each row should have a primary key that distinguishes it as unique.
The Primary key is usually a single column, but sometimes more than one column can be combined to create a single primary key.
For example consider a table which is not in First normal form

Student Table :

Student	Age	Subject
Adam	15	Biology, Maths
Alex	14	Maths
Stuart	17	Maths

In First Normal Form, any row must not have a column in which more than one value is saved, like separated with commas. Rather than that, we must separate such data into multiple rows.
Student Table following 1NF will be :

Student	Age	Subject
Adam	15	Biology
Adam	15	Maths
Alex	14	Maths
Stuart	17	Maths

Using the First Normal Form, data redundancy increases, as there will be many columns with same data in multiple rows but each row as a whole will be unique.

Second Normal Form (2NF)

As per the Second Normal Form there must not be any partial dependency of any column on primary key. It means that for a table that has concatenated primary key, each column in the table that is not part of the primary key must depend upon the entire concatenated key for its existence. If any column depends only on one part of the concatenated key, then the table fails Second normal form.

In example of First Normal Form there are two rows for Adam, to include multiple subjects that he has opted for. While this is searchable, and follows First normal form, it is an inefficient use of space. Also in the above Table in First Normal Form, while the candidate key is {Student, Subject}, Age of Student only depends on Student column, which is incorrect as per Second Normal Form. To achieve second normal form, it would be helpful to split out the subjects into an independent table, and match them up using the student names as foreign keys.

Student	Age
Adam	15
Alex	14
Stuart	17

New Student Table following 2NF will be :

In Student Table the candidate key will be Student column, because all other column i.e Age is dependent on it.

New Subject Table introduced for 2NF will be :

Student	Subject
Adam	Biology
Adam	Maths
Alex	Maths
Stuart	Maths

In Subject Table the candidate key will be {Student, Subject} column. Now, both the above tables qualifies for Second Normal Form and will never suffer from Update Anomalies. Although there are a few complex cases in which table in Second Normal Form suffers Update Anomalies, and to handle those scenarios Third Normal Form is there.

Third Normal Form (3NF)

Third Normal form applies that every non-prime attribute of table must be dependent on primary key, or we can say that, there should not be the case that a non-prime attribute is determined by another non-prime attribute. So this transitive functional dependency should be removed from the table and also the table must be in Second Normal form. For example, consider a table with following fields.

Student_Detail Table :

Student_id	Student_name	DOB	Street	city	State	Zip

In this table Student_id is Primary key, but street, city and state depends upon Zip. The dependency between zip and other fields is called transitive dependency. Hence to apply 3NF, we need to move the street, city and state to new table, with Zip as primary key.

New Student_Detail Table :

Student_id	Student_name	DOB	Zip

Address Table :

Zip	Street	city	state

The advantage of removing transitive dependency is, the amount of data duplication is reduced. Data integrity achieved.

8b. Explain BCNF, with the help of an example. 06M

A relation schema R is in Boyce-Codd Normal Form (BCNF) if whenever an FD X -> A holds in R, then X is a super key of R. Every BCNF relation is in 3NF.

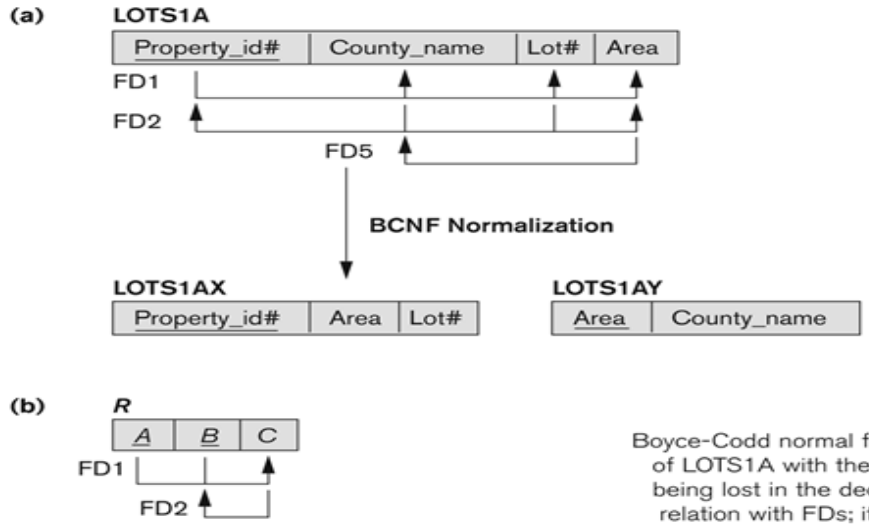


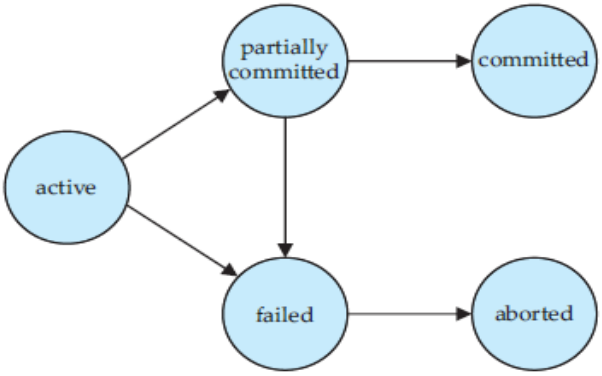
Figure 10.12
Boyce-Codd normal form. (a) BCNF normalization of LOTS1A with the functional dependency FD2 being lost in the decomposition. (b) A schematic relation with FDs; it is in 3NF, but not in BCNF.

9a. With the help of state transition diagram, explain the states of transaction execution. 8M

A transaction must be in one of the following states:

- **Active,** the initial state; the transaction stays in this state while it is executing.
- **Partially committed**, after the final statement has been executed.
- **Failed**, after the discovery that normal execution can no longer proceed.
- **Aborted**, after the transaction has been rolled back and the database has been restored to its state prior to the start of the transaction.
- **Committed,** after successful completion.

We say that a transaction has committed only if it has entered the committed state.



Similarly, we say that a transaction has aborted only if it has entered the aborted state. A transaction is said to have terminated if it has either committed or aborted. A transaction starts in the active state. When it finishes its final statement, it enters the partially committed state. At this point, the transaction has completed its execution, but it is still possible that it may have to be aborted, since the actual output may still be temporarily residing in main memory, and thus a hardware failure may preclude its successful completion.

9b. Define transaction. Explain ACID properties of transaction. 8M

To ensure integrity of the data, we require that the database system maintain the following properties of the transactions:

i)ACID Properties

Consistency: The consistency requirement here is that sum of A and B be unchanged by the execution of transaction. Without the consistency requirement, money could be created or destroyed by a transaction. It can be verified easily that if the database is consistent before an execution of the transaction, the database remains consistent after the execution of the transaction. Ensuring consistency for an individual transaction if the responsibility of the application programmer, who codes the transaction.

Durability: Once the execution of the transaction completes successfully, and the user who initiated the transaction has been notified that the transfer of funds has taken place, it must be the case that no system failure will result in a loss of data corresponding to this transfer of funds.

Atomicity: That is the reason for the atomicity requirement: if the atomicity property is present, all actions of the transaction are reflected in the database or none are. The database system keeps track of the old values on the disk, on which transaction performs a write and if the transaction does not complete, the database system restores the old values to make it appear that the transaction never occurred. Ensuring atomicity is the responsibility of the database system itself; it is handled by transaction-management component.

□□**Isolation:** Even if the consistency and atomicity properties are ensured for each transaction, if several transactions are executed concurrently, their operations may interleave in some undesirable way resulting in an inconsistent state.

The isolation property of a transaction ensures that the concurrent execution of transactions results in a system state that is equivalent to a state that could have been obtained had these transactions executed one at a time in some order.

Transactions access data using two operations:

Read(x), which transfers the data item x from the database to a local buffer belonging to the transaction that executed the read operation.

Write(x), which transfers the data item x from the local buffer of the transaction that executed the write back to the database.

In a real database system, the write operation does not necessarily result in the immediate update of the data on the disk. The write operation may be temporarily stored in memory and executed on the disk later. For now, assume write is done immediately.

Let Ti be a transaction that transfers $50 from account A to B. this transaction is
Ti: read(A);
A := A – 50;
Write(A);
Read(B);
B := B+50;
Write (B);

10a. Explain how to deal with deadlock in concurrent control mechanism. 8M
Lock based protocols:
1. Locks
2. Granting of locks
3. The 2-phase locking protocol
4. Implementation of locking

A lock is a mechanism to control concurrent access to data item.
Data items can be locked in 2 modes.
1. Shared
2. Exclusive

Exclusive(x) – Data item can be both read as well as written. X-lock is requested using lock-X instructions.
Shared(x) – Data item can only be read s-lock is requested using lock-s instructions.
Lock requests are made to concurrency control manager. The compatibility relation between the two modes of locking discussed in this section appears in the matrix comp of Figure. An element comp (A, B) of the matrix has the value true if and only if mode A is compatible with mode B.
S X
S True false , X false false
A transaction to unlock a data item immediately after its final access of that data item is not always desirable, since Serializability may not be ensured.
T1: lock-X(B);
Read (B);
B := B – 50;
Write(B);
[10]
Unlock(B);
Lock-X(A);
Read(A);
A := A + 50;
Write(A);
Unlock(A);
T2: lock-S(A);
Read (A);
Unlock(A);

Lock-S(B);
Read(B);
Unlock(B);
Display(A + B);

Granting of Lock:

When a transaction requests a lock on a data item in a particular mode, and no other transaction has a lock on the same data item in a conflicting mode, the lock can be granted. However, care must be taken to avoid the following scenario. Suppose a transaction T2 has a shared-mode lock on the data item.

Clearly, T1 has to wait for T2 to release the shared-mode lock. Meanwhile, a transaction T3 may request a shared-mode lock on the same data item. The lock request is compatible with the lock granted to T2, so T3 may be granted the shared-mode lock. At this point T2 may release the lock, but still T1 has to wait for T3 to finish. But again, there may be a new transaction T4 that requests a shared-mode lock on the same data item, and is granted the lock before T3 releases it.

In fact, it is possible that there is a sequence of transaction releases the lock a short while after it is granted, but T1 never gets the exclusive-mode lock on the data item. The transaction T1 may never make progress, and is said to be starved.

THE TWO – PHASE LOCKING PROTOCOL

One protocol that ensures Serializability is the **two-phase locking protocol**. This protocol requires that each transaction issue lock and unlock requests in two phases:

1. **Growing phase.** A transaction may obtain locks, but may not release any lock.
2. **Shrinking phase.** A transaction may release locks, but may not obtain any new locks.

Initially, a transaction is in the growing phase. The transaction acquires locks as need. Once the transaction releases a lock, it enters the shrinking phase, and it can issue no more lock request.

Implementation of locking:

A lock-manager can be implemented as a process that receives messages from transactions and sends messages in reply.

The lock manager process requests in the following way:

1. When a lock request message arrives, it adds a record to the end of the linked list for the data item, if the linked list is present. Otherwise it creates a new linked list, containing only the record for the request.
2. When the lock manager receives an unlock message from a transaction, it deletes the record for that data item in the linked list corresponding to that transactions.
3. If a transaction aborts, the lock manager deletes any waiting request made by the transactions.

10b. What is a deadlock? Explain the 2PL. 8M

THE TWO – PHASE LOCKING PROTOCOL

One protocol that ensures Serializability is the **two-phase locking protocol**. This protocol requires that each transaction issue lock and unlock requests in two phases:

1. **Growing phase.** A transaction may obtain locks, but may not release any lock.
2. **Shrinking phase.** A transaction may release locks, but may not obtain any new locks.

Initially, a transaction is in the growing phase. The transaction acquires locks as need. Once the transaction releases a lock, it enters the shrinking phase, and it can issue no more lock request.