Q.1 a. What do you mean by functional dependency? Discuss with suitable example.

Answer:

Functional dependencies play a key role in differentiating good database designs from bad database designs.

A **functional dependency** is a type of constraint that is a generalization of the notion of *key*,

Functional dependencies are constraints on the set of legal relations. They allow us to express facts about the enterprise that we are modeling with our database.

The notion of functional dependency generalizes the notion of superkey. Consider a relation schema R, and let $\alpha \subseteq R$ and $\beta \subseteq R$. The **functional dependency**

$\alpha \rightarrow \beta$

holds on schema R if, in any legal relation $r(R)$, for all pairs of tuples $t1$ and $t2$ in r such that $t1[\alpha] = t2[\alpha]$, it is also the case that $t1[\beta] = t2[\beta]$.

b. Define various properties of transaction.

Answer:

The main properties **ACID properties** of the transactions are:

• **Atomicity**. Either all operations of the transaction are reflected properly in the database, or none are.

• **Consistency**. Execution of a transaction in isolation (that is, with no other transaction executing concurrently) preserves the consistency of the database.

• **Isolation**. each transaction is unaware of other transactions executing concurrently in the system.

• **Durability**. After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

c. What is lock? What are the various types of locks used for concurrency control?

Answer:

There are various modes in which a data item may be locked. In this section, we restrict our attention to two modes:

1. Shared. If a transaction *Ti* has obtained a **shared-mode lock** (denoted by S) on item *Q*, then *Ti* can read, but cannot write, *Q*.

2. Exclusive. If a transaction *Ti* has obtained an **exclusive-mode lock** (denoted by X) on item *Q*, then *Ti* can both read andwrite *Q*.

A transaction requests a shared lock on data item *Q* by executing the lock-S(*Q*) instruction. Similarly, a transaction requests an exclusive lock through the lock-X(*Q*) instruction. A transaction can unlock a data item *Q* by the unlock(*Q*) instruction.

d. What is time stamp? How a system generates time stamp?

Answer:

With each transaction *Ti* in the system, we associate a unique fixed timestamp, denoted by TS(*Ti*).

This timestamp is assigned by the database system before the transaction

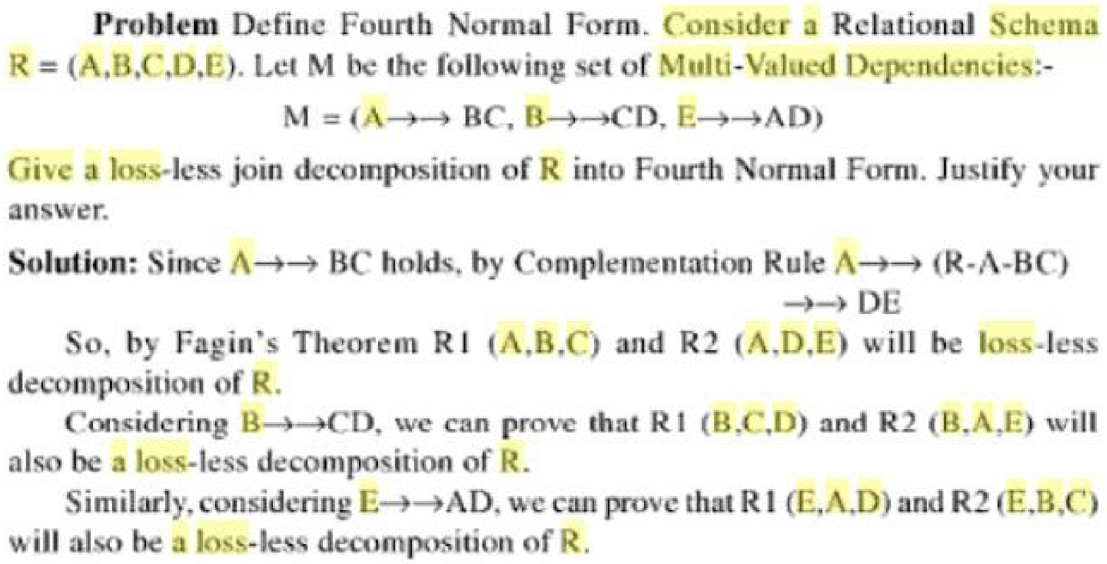
Ti starts execution. If a transaction *Ti* has been assigned timestamp TS(*Ti*), and a new transaction *Tj* enters the system, then TS(*Ti*) < TS(*Tj*).
if TS(*Ti*) < TS(*Tj*), then the system must ensure that the produced schedule is equivalent to a serial schedule in which transaction *Ti* appears before transaction *Tj* .
• **W-timestamp**(*Q*) denotes the largest timestamp of any transaction that executed write(*Q*) successfully.
• **R-timestamp**(*Q*) denotes the largest timestamp of any transaction that executed read(*Q*) successfully.

e. Explain the use of GROUP BY-clause and write an expression in SQL for the following query which is based on the relational schema mentioned in question number 2.
<u>Query:</u> For each department, retrieve the department number, the number of employees in the department, and their average salary.

Answer:
SQL has a GROUP BY-clause for specifying the grouping attributes, which *must also appear in the SELECT-clause*
SELECT DNO, COUNT (*), AVG (SALARY) FROM EMPLOYEE GROUP BY DNO
– In Q20, the EMPLOYEE tuples are divided into groups--each group having the same value for the grouping attribute DNO
– The COUNT and AVG functions are applied to each such group of tuples separately
– The SELECT-clause includes only the grouping attribute and the functions to be applied on each group of tuples
– A join condition can be used in conjunction with grouping

f. Consider a relational scheme R = (A, B, C, D, E) and following set of multi valued dependencies:
M = (A--> -->BC, B --> -->CD, E --> -->AD)
Give a lossless join decomposition of scheme R into fourth normal form.

Answer:


Problem Define Fourth Normal Form. Consider a Relational Schema R = (A,B,C,D,E). Let M be the following set of Multi-Valued Dependencies:-

$$M = (A \rightarrow\rightarrow BC,\ B \rightarrow\rightarrow CD,\ E \rightarrow\rightarrow AD)$$

Give a loss-less join decomposition of R into Fourth Normal Form. Justify your answer.

Solution: Since $A \rightarrow\rightarrow BC$ holds, by Complementation Rule $A \rightarrow\rightarrow (R\text{-}A\text{-}BC)$
$\rightarrow\rightarrow DE$

So, by Fagin's Theorem R1 (A,B,C) and R2 (A,D,E) will be loss-less decomposition of R.

Considering $B \rightarrow\rightarrow CD$, we can prove that R1 (B,C,D) and R2 (B,A,E) will also be a loss-less decomposition of R.

Similarly, considering $E \rightarrow\rightarrow AD$, we can prove that R1 (E,A,D) and R2 (E,B,C) will also be a loss-less decomposition of R.

g. What is weak entity set? Explain with suitable example. **(7 × 4)**

Answer:

An entity set that does not have a primary key is referred to as a **weak entity set**.

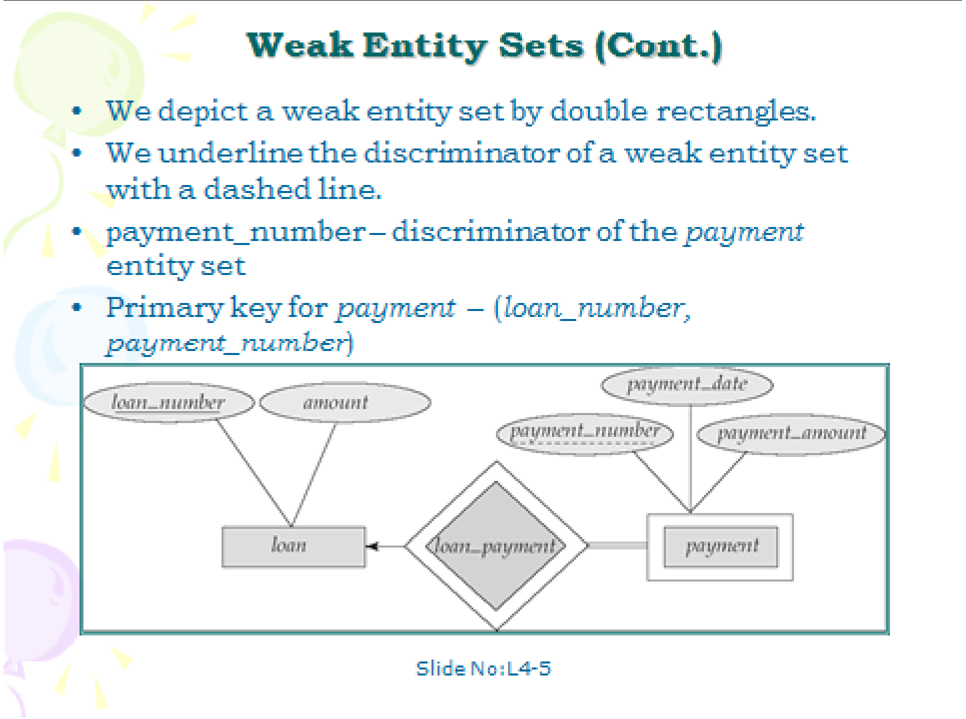
• The existence of a weak entity set depends on the existence of a **identifying entity set**

– it must relate to the identifying entity set via a total, one-to-many relationship set from the identifying to the weak entity set

– Identifying relationship depicted using a double diamond

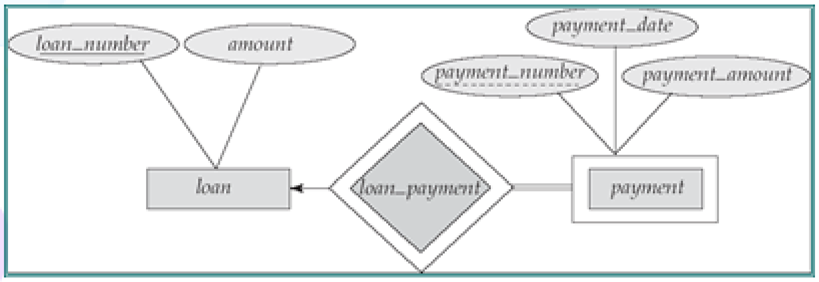
• The **discriminator** (*or partial key)* of a weak entity set is the set of attributes that distinguishes among all the entities of a weak entity set.

• The primary key of a weak entity set is formed by the primary key of the strong entity set on which the weak entity set is existence dependent, plus the weak entity set's discriminator.



Weak Entity Sets (Cont.)

- We depict a weak entity set by double rectangles.
- We underline the discriminator of a weak entity set with a dashed line.
- payment_number – discriminator of the *payment* entity set
- Primary key for *payment* – (*loan_number, payment_number*)



Slide No:L4-5

Q.2 a. Consider the following relational database schema **(3+3+3+3)**

EMPLOYEE

FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO

DEPARTMENT

DNAME	DNUMBER	MGRSSN	MGRSTARTDATE

DEPT_LOCATIONS

DNUMBER	DLOCATION

PROJECT

PNAME	PNUMBER	PLOCATION	DNUM

WORKS_ON

ESSN	PNO	HOURS

DEPENDENT

ESSN	DEPENDENT_NAME	SEX	BDATE	RELATIONSHIP

Write an expression in SQL for each of the following queries
(i) Retrieve the name and address of all employees who work for the 'Research' department.
(ii) For every project located in 'Stafford', list the project number, the controlling department number, and the department manager's last name, address, and birthdate.
(iii) Retrieve the name of each employee who works on all the projects controlled by department number 5.
(iv) Find the maximum salary, the minimum salary, and the average salary among all employees.

Answer:

(i) SELECT FNAME, LNAME, ADDRESS
FROM EMPLOYEE, DEPARTMENT
WHERE DNAME='Research' AND DNUMBER=DNO

Similar to a SELECT-PROJECT-JOIN sequence of relational algebra operations
(DNAME='Research') is a selection condition (corresponds to a SELECT operation in relational algebra)
(DNUMBER=DNO) is a join condition (corresponds to a SELECT operation in relational algebra)

OR

SELECT FNAME, LNAME, ADDRESS
FROM EMPLOYEE
WHERE DNO IN (SELECT DNUMBER
FROM DEPARTMENT
WHERE DNAME='Research')

(ii) SELECT PNUMBER, DNUM, LNAME, BDATE, ADDRESS
FROM PROJECT, DEPARTMENT, EMPLOYEE
WHERE DNUM=DNUMBER AND MGRSSN=SSN
AND PLOCATION='Stafford'

– In this, there are *two* join conditions
– The join condition DNUM=DNUMBER relates a project to its controlling department
– The join condition MGRSSN=SSN relates the controlling department to the employee who
manages that department

(iii) SELECT FNAME, LNAME
FROM EMPLOYEE
WHERE ((SELECT PNO
FROM WORKS_ON
WHERE SSN=ESSN) CONTAINS
(SELECT PNUMBER
FROM PROJECT
WHERE DNUM=5))

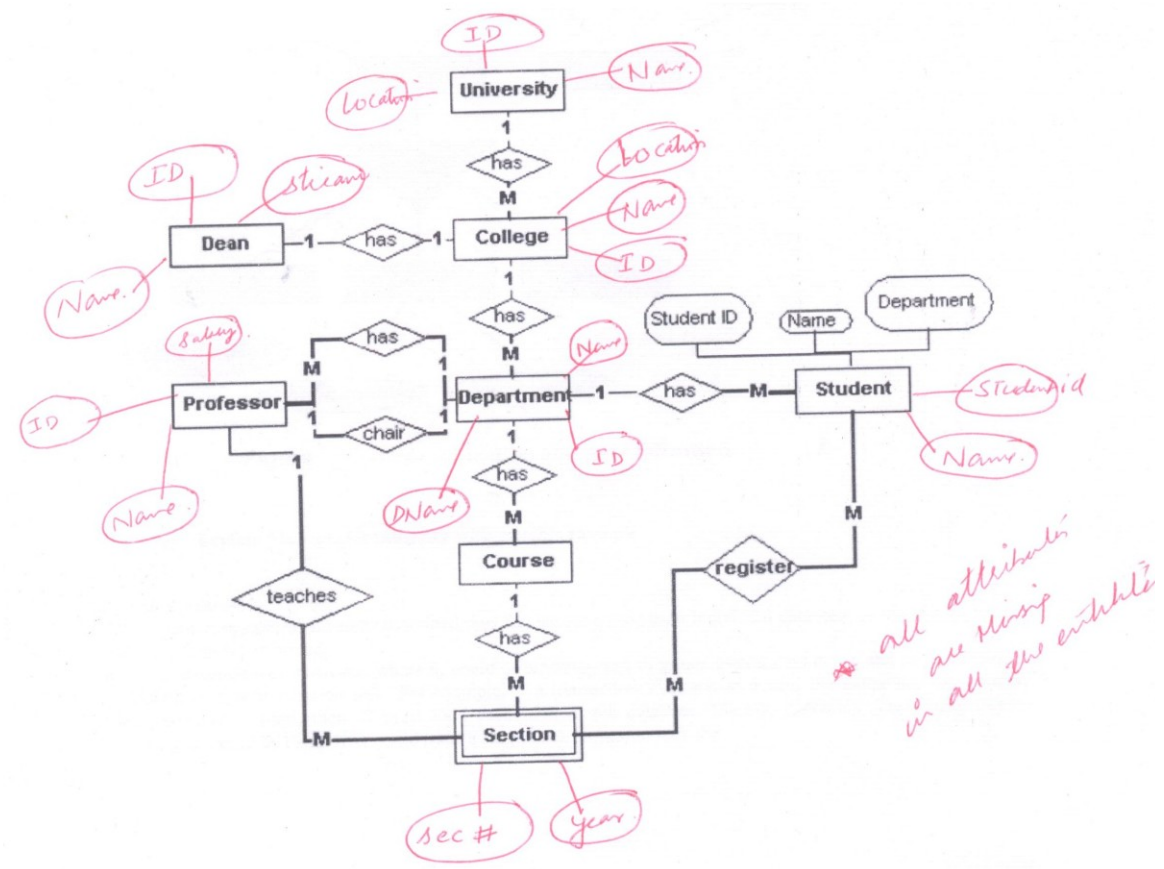
(iv) SELECT MAX (SALARY),
MIN (SALARY), AVG (SALARY)
FROM EMPLOYEE

b. Explain the function of ORDER BY and HAVING Clause with suitable example. **(3+3)**

Answer:
The ORDER BY clause is used to sort the tuples in a query result based on the values of some attribute(s)
Example: In above mentioned relational schema; Retrieve a list of employees and the projects each works in, ordered by the employee's department, and within each department ordered alphabetically by employee last name. SELECT DNAME, LNAME, FNAME, PNAME FROM DEPARTMENT, EMPLOYEE, WORKS_ON, PROJECT WHERE DNUMBER=DNO AND SSN=ESSN AND PNO=PNUMBER ORDER BY DNAME, LNAME
The default order is in ascending order of values
We can specify the keyword DESC if we want a descending order; the keyword ASC can be used to explicitly specify ascending order, even though it is the default
THE HAVING-CLAUSE
Sometimes we want to retrieve the values of these functions for only those *groups that satisfy certain conditions*
The HAVING-clause is used for specifying a selection condition on groups (rather than on individual tuples)
In above mentioned relational schema; For each project *on which more than two employees work* , retrieve the project number, project name, and the number of employees who work on that project. SELECT PNUMBER, PNAME, COUNT (*) FROM PROJECT, WORKS_ON WHERE PNUMBER=PNO GROUP BY PNUMBER, PNAME HAVING COUNT (*) > 2

Q.3 a. Draw the E-R diagram for the university system which includes information about students, department, professors, courses, which student are enrolled in which course, which professor are teaching in which courses, student grades, which courses a department offers. Consider suitable assumption wherever required. (10)

Answer:



b. **Explain the term Generalization and Specialization with suitable example.** **(4)**

Answer:

There are similarities between the *customer* entity set and the *employee* entity set in the sense that they have several attributes in common. This commonality can be expressed by **generalization**, which is a containment relationship that exists between a *higher-level* entity set and one or more *lower-level* entity sets.

Generalization is a simple inversion of specialization.

Generalization proceeds from the recognition that a number of entity sets share some common features

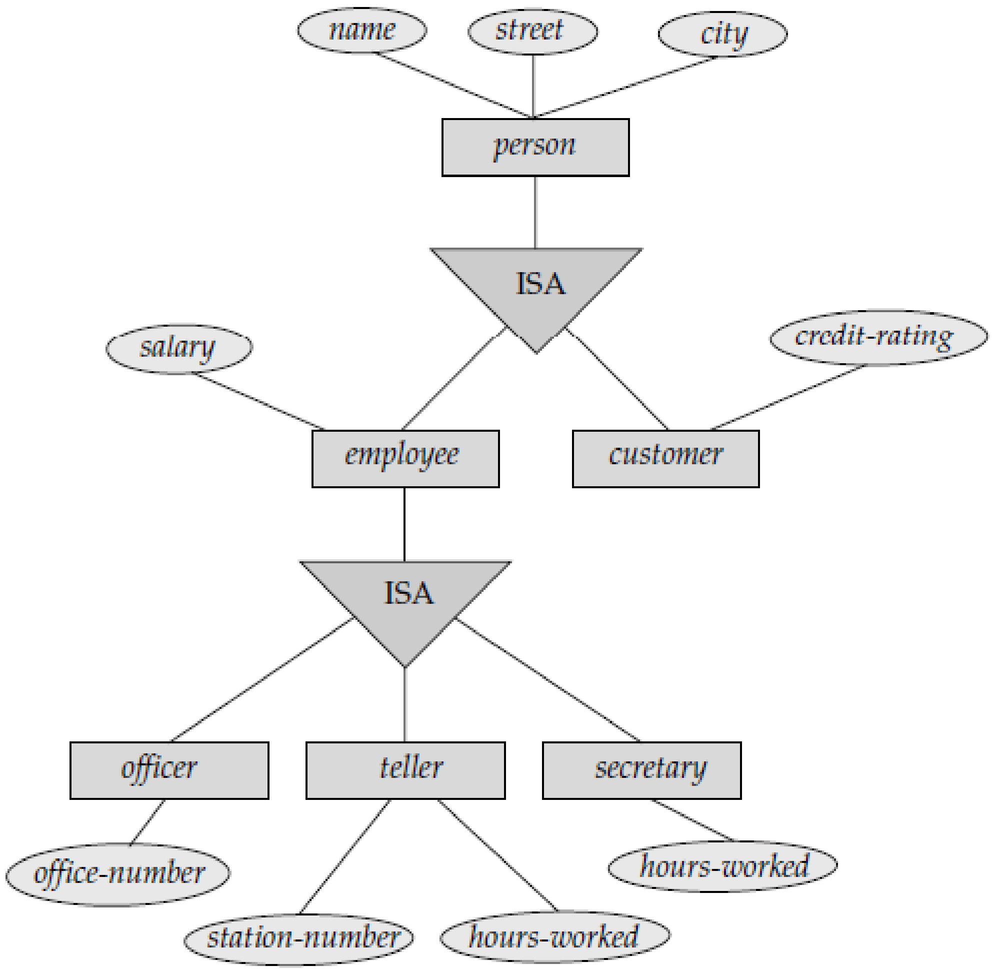


Figure Specialization and generalization.

c. Explain Multiple Granularity with suitable example. **(4)**

Answer:

Multiple Granularity

In the concurrency-control schemes described thus far, we have used each individual data item as the unit on which synchronization is performed.

There are circumstances, however, where it would be advantageous to group several data items, and to treat them as one individual synchronization unit. For example, if a transaction *Ti* needs to access the entire database, and a locking protocol is used, then *Ti* must lock each item in the database. Clearly, executing these locks istime consuming. It would be better if *Ti* could issue a *single* lock request to lock the

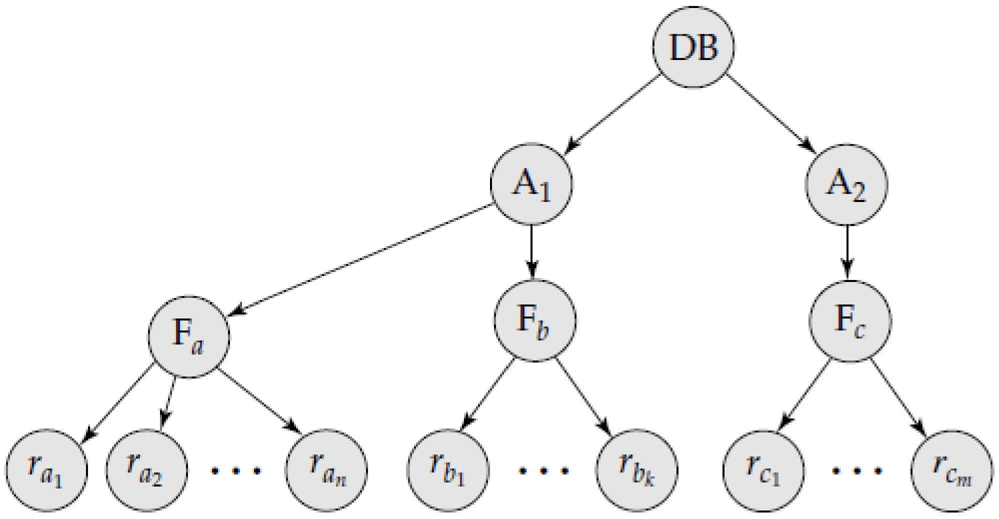


Figure Granularity hierarchy.

entire database. On the other hand, if transaction *Tj* needs to access only a few data items, it should not be required to lock the entire database, since otherwise concurrency is lost. What is needed is a mechanism to allow the system to define multiple levels of **granularity**.We can make one by allowing data items to be of various sizes and defining a hierarchy of data granularities, where the small granularities are nested within larger ones. Such a hierarchy can be represented graphically as a tree. Note that the tree that we describe here is significantly different from that used by the tree protocol.
A nonleaf node of the multiple-granularity tree represents the data associated with its descendants. In the tree protocol, each node is an independent data item.
As an illustration, consider the tree of Figure 16.16, which consists of four levels of nodes. The highest level represents the entire database. Below it are nodes of type *area*; the database consists of exactly these areas. Each area in turn has nodes of type *file* as its children. Each area contains exactly those files that are its child nodes. No file is in more than one area. Finally, each file has nodes of type *record*. As before, the file consists of exactly those records that are its child nodes, and no record can be present in more than one file.
Each node in the tree can be locked individually. As we did in the two-phase locking protocol,we shall use **shared** and **exclusive** lock modes. When a transaction locks a node, in either shared or exclusive mode, the transaction also has implicitly locked all the descendants of that node in the same lock mode. For example, if transaction *Ti* gets an **explicit lock** on file *Fc* of, in exclusive mode, then it has an **implicit lock** in exclusive mode all the records belonging to that file. It does not need to lock the individual records of *Fc* explicitly.

Q.4 a. What are the purposes of normalization? Explain 3NF, 4NF and BCNF with suitable example. (2+2+3+3)

Answer:

◆ **Normalization**: Process of decomposing unsatisfactory "bad" relations by breaking up their attributes into smaller relations

- **Normal form**: Condition using keys and FDs of a relation to certify whether a relation schema is in a particular normal form
- 2NF, 3NF, BCNF based on keys and FDs of a relation schema
- 4NF based on keys, multi-valued dependencies

Third Normal Form

- Definition
- **Transitive functional dependency** – if there a set of atribute Z that are neither a primary or candidate key and both X → Z and Y → Z holds.
- Examples:
- SSN → DMGRSSN is a transitive FD since

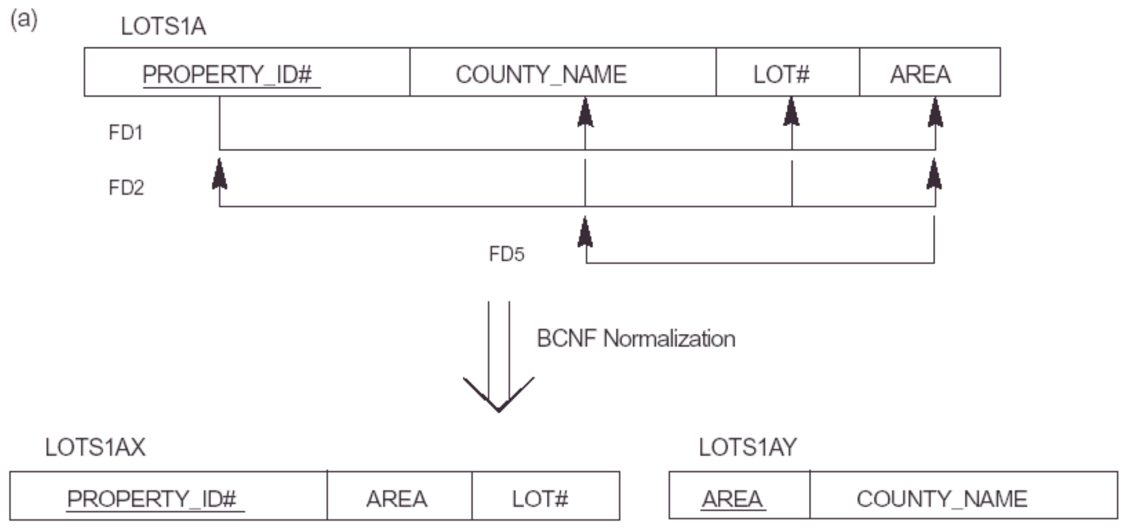
SSN → DNUMBER and DNUMBER → DMGRSSN hold

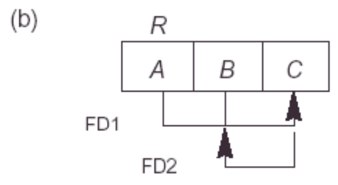
- SSN → ENAME is *non-transitive* since there is no set of attributes X where SSN → X and X → ENAME

A relation schema R is in **third normal form** (**3NF**) if it is in 2NF *and* no non-prime attribute A in R is transitively dependent on the primary key

BCNF (Boyce-Codd Normal Form)

- A relation schema R is in **Boyce-Codd Normal Form** (**BCNF**) if whenever an FD X → A holds in R, then X is a superkey of R
- Each normal form is strictly stronger than the previous one:
- Every 2NF relation is in 1NF
- Every 3NF relation is in 2NF
- Every BCNF relation is in 3NF
- There exist relations that are in 3NF but not in BCNF
- The goal is to have each relation in BCNF (or 3NF)





Forth Normal Form
A table isin 4NF ,if it is in BCNF and it contains no multi-valued deprndencies.
A relation schema R is in 4NF , with respect to a set of dependencies F (that includes FD and multivalued dependencies) if,for every multivalued dependency X->->Y in F_+ , X is a superkey for R.
For example

Faculty	Subject	Committee
John	DBMS	Placement
John	Networking	Placement
John	MIS	Placement
John	DBMS	Scholarship
John	Networking	Scholarship
John	MIS	Scholarship

Faculty	Course
John	DBMS
John	Networking
John	MIS

Faculty	Committee
John	Placement
John	Scholarship

b. Define Serializability, Conflict Serializability and view Serializability schedule with suitable example. (2+3+3)

Answer:

Serializability
The database system must control concurrent execution of transactions, to ensure that the database state remains consistent. Before we examine how the database system can carry out this task, we must first understand which schedules will ensure consistency, and which schedules will not.

Conflict Serializability
Let us consider a schedule *S* in which there are two consecutive instructions *Ii* and *Ij*, of transactions *Ti* and *Tj* , respectively (*i* _= *j*). If *Ii* and *Ij* refer to different data items, then we can swap *Ii* and *Ij* without affecting the results of any instruction in the schedule. However, if *Ii* and *Ij* refer to the same data item *Q*, then the order of the two steps maymatter. Since we are dealing with only read and write instructions, there are four cases that we need to consider:
1. *Ii* = read(*Q*), *Ij* = read(*Q*). The order of *Ii* and *Ij* does not matter, since the same value of *Q* is read by *Ti* and *Tj* , regardless of the order.
2. *Ii* = read(*Q*), *Ij* = write(*Q*). If *Ii* comes before *Ij*, then *Ti* does not read the value of *Q* that is written by *Tj* in instruction *Ij*. If *Ij* comes before *Ii*, then *Ti* reads
the value of *Q* that is written by *Tj*. Thus, the order of *Ii* and *Ij* matters.
3. *Ii* = write(*Q*), *Ij* = read(*Q*). The order of *Ii* and *Ij* matters for reasons similar to those of the previous case.
4. *Ii* = write(*Q*), *Ij* = write(*Q*). Since both instructions are write operations, theorder of these instructions does not affect either *Ti* or *Tj*

T_1	T_2
read(A)	
write(A)	
	read(A)
read(B)	
	write(A)
write(B)	
	read(B)
	write(B)

Figure Schedule

Faculty	**Committee**
John	Placement
John	Scholarship

Faculty	**Course**
John	DBMS
John	Networking
John	MIS

View Serializability

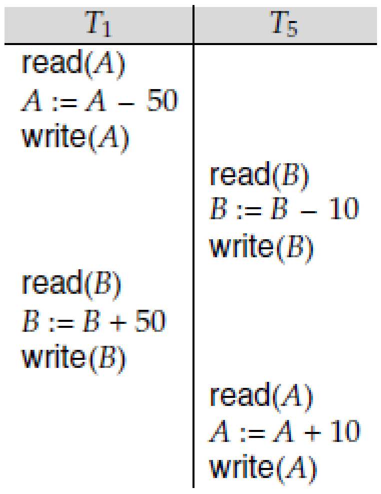
In this section, we consider a form of equivalence that is less stringent than conflict equivalence, but that, like conflict equivalence, is based on only the read and write operations of transactions.

Consider two schedules *S* and *S*_, where the same set of transactions participates in both schedules. The schedules *S* and *S*_ are said to be **view equivalent** if three conditions are met:

1. For each data item *Q*, if transaction *Ti* reads the initial value of *Q* in schedule *S*, then transaction *Ti* must, in schedule *S*_, also read the initial value of *Q*.

2. For each data item *Q*, if transaction *Ti* executes read(*Q*) in schedule *S*, and if that value was produced by a write(*Q*) operation executed by transaction *Tj* , then the read(*Q*) operation of transaction *Ti* must, in schedule *S*_, also read the value of *Q* that was produced by the same write(*Q*) operation of transaction *Tj* .

3. For each data item *Q*, the transaction (if any) that performs the final write(*Q*) operation in schedule *S* must perform the final write(*Q*) operation in schedule *S*.



T_1	T_5
read(A)	
$A := A - 50$	
write(A)	
	read(B)
	$B := B - 10$
	write(B)
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$A := A + 10$
	write(A)

Figure Schedule

Q.5 a. What are the various types of Distributed Database Systems? Explain fragmentation in distributed database systems. (4+4)

Answer:

Unlike parallel systems, in which the processors are tightly coupled and constitute a single database system, a distributed database system consists of loosely coupled sites that share no physical components. Furthermore, the database systems that run on each site may have a substantial degree of mutual independence.

Distributed databases as homogeneous or heterogeneous

Homogeneous and Heterogeneous Databases

In a **homogeneous distributed database**, all sites have identical database management system software, are aware of one another, and agree to cooperate in processing users' requests. In such a system, local sites surrender a portion of their autonomy in terms of their right to change schemas or database management system software.
That software must also cooperate with other sites in exchanging information about transactions, to make transaction processing possible across multiple sites.
In contrast, in a **heterogeneous distributed database**, different sites may use different schemas, and different database management system software. The sites may not be aware of one another, and they may provide only limited facilities for cooperation in transaction processing. The differences in schemas are often a major problem for query processing, while the divergence in software becomes a hindrance for processing transactions that access multiple sites.
Data Fragmentation
If relation *r* is fragmented, *r* is divided into a number of *fragments r*1, *r*2*, . . . , rn*. These fragments contain sufficient information to allow reconstruction of the original relation
r. There are two different schemes for fragmenting a relation: *horizontal* fragmentation and *vertical* fragmentation. Horizontal fragmentation splits the relation by assigning each tuple of *r* to one or more fragments. Vertical fragmentation splits the relation by decomposing the scheme *R* of relation *r*.
We shall illustrate these approaches by fragmenting the relation *account*, with the schema
Account-schema = (*account-number*, *branch-name*, *balance*)
In **horizontal fragmentation**, a relation *r* is partitioned into a number of subsets,
*r*1, *r*2*, . . . , rn*. Each tuple of relation *r* must belong to at least one of the fragments, so that the original relation can be reconstructed, if needed.
As an illustration, the *account* relation can be divided into several different fragments, each of which consists of tuples of accounts belonging to a particular branch.
If the banking system has only two branches—Hillside and Valleyview—then there are two different fragments:
*account*1 = σ*branch-name* = "Hillside" (*account*)
*account*2 = σ*branch-name* = "Valleyview" (*account*)
Horizontal fragmentation is usually used to keep tuples at the sites where they are used the most, to minimize data transfer.
In general, a horizontal fragment can be defined as a *selection* on the global relation *r*. That is,we use a predicate *Pi* to construct fragment *ri*:
ri = σ*Pi* (*r*)
The two types of fragmentation can be applied to a single schema; for instance, the fragments obtained by horizontally fragmenting a relation can be further partitioned vertically. Fragments can also be replicated. In general, a fragment can be replicated, replicas of fragments can be fragmented further, and so on.
Say we have this relation

customer_id	Name	Area	Payment Type	Sex
1	Bob	London	Credit card	Male
2	Mike	Manchester	Cash	Male
3	Ruby	London	Cash	Female

Horizontal Fragmentation are subsets of tuples (rows)
Fragment 1

customer_id	Name	Area	Payment Type	Sex
1	Bob	London	Credit card	Male
2	Mike	Manchester	Cash	Male

Fragment 2

customer_id	Name	Area	Payment Type	Sex
3	Ruby	London	Cash	Female

Vertical fragmentation are subset of attributes
Fragment 1

customer_id	Name	Area	Sex
1	Bob	London	Male
2	Mike	Manchester	Male
3	Ruby	London Female	

Fragment 2

customer_id | Payment Type
1 | Credit card
2 | Cash
3 | Cash

b. What is relational algebra? Explain its various operations. (2+4)

Answer:

The Relational Algebra

The relational algebra is a *procedural* query language. It consists of a set of operations that take one or two relations as input and produce a new relation as their result. The fundamental operations in the relational algebra are *select, project*, *union*, *set difference*,
Cartesian product, and *rename*. In addition to the fundamental operations, there are several other operations—namely, set intersection, natural join, division, and assignment.
We will define these operations in terms of the fundamental operations.

Fundamental Operations

The select, project, and rename operations are called *unary* operations, because they operate on one relation. The other three operations operate on pairs of relations and are, therefore, called *binary* operations

The Select Operation

The **select** operation selects tuples that satisfy a given predicate.We use the lowercase Greek letter sigma (σ) to denote selection. The predicate appears as a subscript to σ.
The argument relation is in parentheses after the σ. Thus, to select those tuples of the *loan* relation where the branch is "Perryridge," we write
σ*branch-name* ="Perryridge" (*loan*)

The Project Operation

Suppose we want to list all loan numbers and the amount of the loans, but do not care about the branch name. The **project** operation allows us to produce this relation.
The project operation is a unary operation that returns its argument relation, with certain attributes left out. Since a relation is a set, any duplicate rows are eliminated.
Projection is denoted by the uppercase Greek letter pi (Π).We list those attributes that we wish to appear in the result as a subscript to Π. The argument relation follows in parentheses. Thus, we write the query to list all loan numbers and the amount of the loan as
Π*loan-number, amount*(*loan*)

Composition of Relational Operations

The fact that the result of a relational operation is itself a relation is important. Consider the more complicated query "Find those customers who live in Harrison." We write:
Π*customer-name* (σ*customer-city* ="Harrison" (*customer*))

The Union Operation

Consider a query to find the names of all bank customers who have either an account or a loan or both. Note that the *customer* relation does not contain the information, since a customer does not need to have either an account or a loan at the bank. To answer this query, we need the information in the *depositor* relation and in the *borrower* relation .We know how to find the names of all customers with a loan in the bank:
Π*customer-name* (*borrower*)

The Set Difference Operation

The **set-difference** operation, denoted by −, allows us to find tuples that are in one relation but are not in another. The expression $r - s$ produces a relation containing those tuples in r but not in s.
We can find all customers of the bank who have an account but not a loan by writing
Π*customer-name* (*depositor*) − Π*customer-name* (*borrower*)

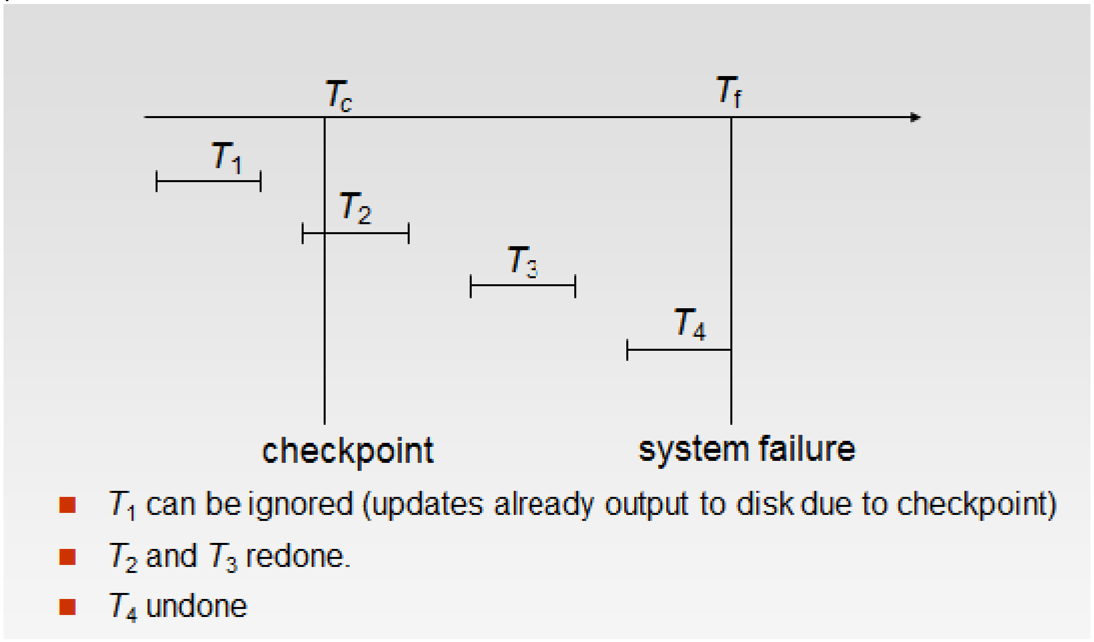
The Cartesian-Product Operation

The **Cartesian-product** operation, denoted by a cross ($\times$), allows us to combine information from any two relations.We write the Cartesian product of relations $r1$ and $r2$ as $r1 \times r2$.

c. Define check point and its impact on data base recovery. (4)

Answer:

Check points are those time points on which we save output all log records currently residing in main memory onto stable storage.
For data base recovery, we just know the timing of check point or means name of check point.

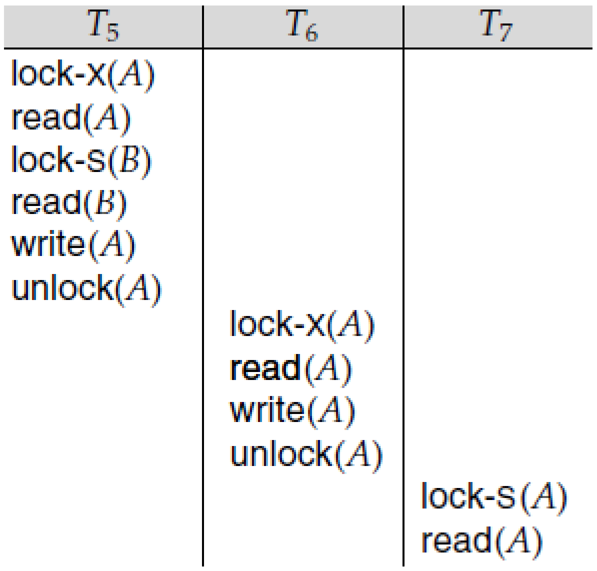


- T_1 can be ignored (updates already output to disk due to checkpoint)
- T_2 and T_3 redone.
- T_4 undone

We used checkpoints to reduce the number of log records that the system must scan when it recovers from a crash. Since we assumed no concurrency, it was necessary to consider only the following transactions during recovery:
• Those transactions that started after the most recent checkpoint
• The one transaction, if any, that was active at the time of the most recent checkpoint
The situation is more complex when transactions can execute concurrently, since several transactions may have been active at the time of the most recent checkpoint.
Restart Recovery
When the system recovers from a crash, it constructs two lists: The undo-list consists of transactions to be undone, and the redo-list consists of transactions to be redone.
The system constructs the two lists as follows: Initially, they are both empty.
The system scans the log backward, examining each record, until it finds the first <checkpoint> record:
• For each record found of the form <*Ti* commit>, it adds *Ti* to redo-list.
• For each record found of the form <*Ti* start>, if *Ti* is not in redo-list, then it adds *Ti* to undo-list.
When the system has examined all the appropriate log records, it checks the list *L* in the checkpoint record.

Q.6 a. Describe Two Phase Locking protocol with suitable example. (8)
Answer:
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 needed. Once the transaction releases a lock, it enters the shrinking phase, and it can issue no more lock requests.
For example, transactions *T*3 and *T*4 are two phase. On the other hand, transactions *T*1 and *T*2 are not two phase. Note that the unlock instructions do not need to appear at the end of the transaction. For example, in the case of transaction *T*3, we could move the unlock(*B*) instruction to just after the lock-X(*A*) instruction, and still retain the two-phase locking property.
We can show that the two-phase locking protocol ensures conflict serializability.

Consider any transaction. The point in the schedule where the transaction has obtained its final lock (the end of its growing phase) is called the **lock point** of the transaction. Now, transactions can be ordered according to their lock points—this ordering is, in fact, a serializability ordering for the transactions. We leave the proof as an exercise for you to do Two-phase locking does *not* ensure freedom from deadlock. Observe that transactions *T*3 and *T*4 are two phase, but, in schedule 2 (Figure), they are deadlocked.
Cascading rollback may occur under two-phase locking. As an illustration, consider the partial schedule of Figure 16.8. Each transaction observes the two-phase locking protocol, but the failure of *T*5 after the read(A) step of *T*7 leads to cascading rollback of *T*6 and *T*7.
Cascading rollbacks can be avoided by a modification of two-phase locking called the **strict two-phase locking protocol**. This protocol requires not only that locking be two phase, but also that all exclusive-mode locks taken by a transaction be helduntil that transaction commits. This requirement ensures that any data written by an uncommitted transaction are locked in exclusive mode until the transaction commits, preventing any other transaction from reading the data.
Another variant of two-phase locking is the **rigorous two-phase locking protocol**, which requires that all locks be held until the transaction commits.



T_5	T_6	T_7
lock-X(A)		
read(A)		
lock-S(B)		
read(B)		
write(A)		
unlock(A)		
	lock-X(A)	
	read(A)	
	write(A)	
	unlock(A)	
		lock-S(A)
		read(A)

Figure Partial schedule under two-phase locking.

b. Describe Deadlock with suitable example and also explain about recovery from the deadlock. (2+3)

Answer:
A system is in a deadlock state if there exists a set of transactions such that every transaction in the set is waiting for another transaction in the set. More precisely, there exists a set of waiting transactions *{T*0, *T*1*, . . ., Tn}* such that *T*0 is waiting for a data item that *T*1 holds, and *T*1 is waiting for a data item that *T*2 holds, and . . ., and *Tn*−1 is waiting for a data item that *Tn* holds, and *Tn* is waiting for a data item that *T*0 holds. None of the transactions can make progress in such a situation.
The only remedy to this undesirable situation is for the system to invoke some drastic action, such as rolling back some of the transactions involved in the deadlock.
Rollback of a transaction may be partial: That is, a transaction may be rolled back to the point where it obtained a lock whose release resolves the deadlock.
There are two principal methods for dealing with the deadlock problem. We canuse a **deadlock prevention** protocol to ensure that the system will *never* enter a deadlock state. Alternatively, we can allow the system to enter a deadlock state, and then try to recover by using a **deadlock detection** and **deadlock recovery** scheme. As we shall see, both methods may result in transaction rollback. Prevention is commonly used if

the probability that the system would enter a deadlock state is relatively high; otherwise, detection and recovery are more efficient.
Note that a detection and recovery scheme requires overhead that includes not only the run-time cost of maintaining the necessary information and of executing the detection algorithm, but also the potential losses inherent in recovery from a deadlock.
Recovery from Deadlock
When a detection algorithm determines that a deadlock exists, the system must **recover** from the deadlock. The most common solution is to roll back one or more transactions to break the deadlock. Three actions need to be taken:
1. Selection of a victim. Given a set of deadlocked transactions, we must determinewhich transaction (or transactions) to roll back to break the deadlock.Weshould roll back those transactions that will incur the minimum cost. Unfortunately, the term *minimum cost* is not a precise one. Many factors may determine the cost of a rollback, including
a. How long the transaction has computed, and how much longer the transaction will compute before it completes its designated task.
b. How many data items the transaction has used.
c. How many more data items the transaction needs for it to complete.
d. How many transactions will be involved in the rollback.
2. Rollback. Once we have decided that a particular transaction must be rolled back, we must determine how far this transaction should be rolled back.
The simplest solution is a **total rollback**: Abort the transaction and then restart it. However, it is more effective to roll back the transaction only as far as necessary to break the deadlock. Such **partial rollback** requires the system to maintain additional information about the state of all the running transactions. Specifically, the sequence of lock requests/grants and updates performed by the transaction needs to be recorded. The deadlock detection mechanism should decide which locks the selected transaction needs to release in order to break the deadlock. The selected transaction must be rolled back to the point where it obtained the first of these locks, undoing all actions it took after that point. The recovery mechanism must be capable of performing such partial rollbacks. Furthermore, the transactions must be capable of resuming execution after a partial rollback. See the bibliographical notes for relevant references.
3. Starvation. In a system where the selection of victims is based primarily on cost factors, it may happen that the same transaction is always picked as a victim. As a result, this transaction never completes its designated task, thus there is **starvation**.

c. Consider the universal relation;
R =(A,B,C,D,E,F,G,H,I,J) And the set of functional dependencies F as given below:
F{AB→C, A→DE, B→F, F→GH, D→IJ}
(i) Determine the key for R
(ii) Decompose R into second normal form **(3+2)**

Answer:
For determine the key of R firstly,we need to decompose the FD's an need to find out the Clouser of All attribute.
Decomosition:
AB→C
A→DE (A→D)(A→E)
B→F
F→GH (F→G) (F→H)
D→IJ (D→I) (D→J)
Clouser Set of determinate (left side values):
A_+={A,D,E,I,J}
B_+={B,F,G,H}
F_+={F,G,H}
D_+={D,I,J}
So no single attribute will be key.

AB+={A,B,C,D,E,F,G,H,I,J)
So key=AB
(ii) For come in 2NF it need that ,every left hand side attribute is part of candidate key(AB)
So decomposition
R1= {A,B,C,D,E,F}
R2= {F,D,G,H,I,J}
And different decompositions are also possible.

Q.7 Write the short notes on the following:

(i) Query Optimization
(ii) Time Stamp Based Concurrency Control
(iii) Relational Model

Answer:

(i) Alternative ways of evaluating a given query
a. Equivalent expressions
b. Different algorithms for each operation

An **evaluation plan** defines exactly what algorithm is used for each operation, and how the execution of the operations is coordinated.
Cost difference between evaluation plans for a query can be enormous
l E.g. seconds vs. days in some cases

Steps in **cost-based query optimization**
Generate logically equivalent expressions using **equivalence rules**
Annotate resultant expressions to get alternative query plans
Choose the cheapest plan based on **estimated cost**
Estimation of plan cost based on:
Statistical information about relations. Examples:
- number of tuples, number of distinct values for an attribute

Statistics estimation for intermediate results
- to compute cost of complex expressions

Cost formulae for algorithms, computed using statistics

(ii)

□ Each transaction is issued a timestamp when it enters the system. If an old transaction *Ti* has time-stamp TS(*Ti*), a new transaction *Tj* is assigned time-stamp TS(*Tj*) such that TS(*Ti*) <TS(*Tj*).

□ The protocol manages concurrent execution such that the time-stamps determine the serializability order.

□ In order to assure such behavior, the protocol maintains for each data *Q* two timestamp values:
□ **W-timestamp**(*Q*) is the largest time-stamp of any transaction that executed **write**(*Q*) successfully.

□ **R-timestamp**(*Q*) is the largest time-stamp of any transaction that executed **read**(*Q*) successfully.
□ The timestamp ordering protocol ensures that any conflicting **read** and **write** operations are executed in timestamp order.

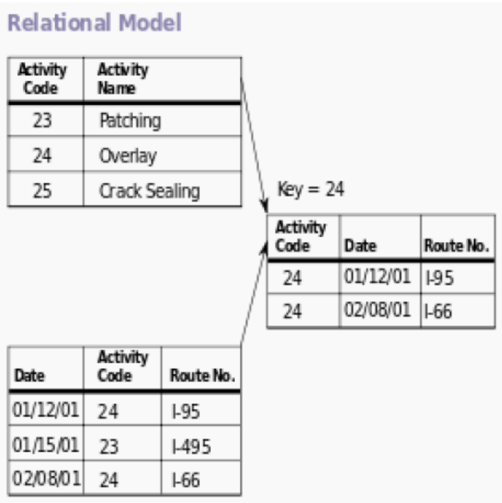
□ Suppose a transaction Ti issues a **read**(*Q*)
□ If TS(*Ti*) □ **W**-timestamp(*Q*), then *Ti* needs to read a value of *Q* that was already overwritten.
□ Hence, the **read** operation is rejected, and *Ti* is rolled back.
□ If TS(*Ti*)□ **W**-timestamp(*Q*), then the **read** operation is executed, and R-timestamp(*Q*) is set to **max**(R-timestamp(*Q*), TS(*Ti*)).
□ Suppose that transaction *Ti* issues **write**(*Q*).
□ If TS(*Ti*) < R-timestamp(*Q*), then the value of *Q* that *Ti* is producing was needed previously, and the system assumed that that value would never be produced.
□ Hence, the **write** operation is rejected, and *Ti* is rolled back.
□ If TS(*Ti*) < W-timestamp(*Q*), then *Ti* is attempting to write an obsolete value of *Q*.
□ Hence, this **write** operation is rejected, and *Ti* is rolled back.
□ Otherwise, the **write** operation is executed, and W-timestamp(*Q*) is set to TS(*Ti*).

Correctness of Timestamp-Ordering Protocol
□ The timestamp-ordering protocol guarantees serializability since all the arcs in the precedence graph are of the form:

$$\begin{tikzcd}
\text{transaction with smaller timestamp} \arrow[r] & \text{transaction with larger timestamp}
\end{tikzcd}$$

Thus, there will be no cycles in the precedence graph

(iii) The **relational model** for database management is a database model based on first-order predicate logic, first formulated and proposed in 1969 by Edgar F. Codd. In the relational model of a database, all data is represented in terms of tuples, grouped into relations. A database organized in terms of the relational model is a relational database.

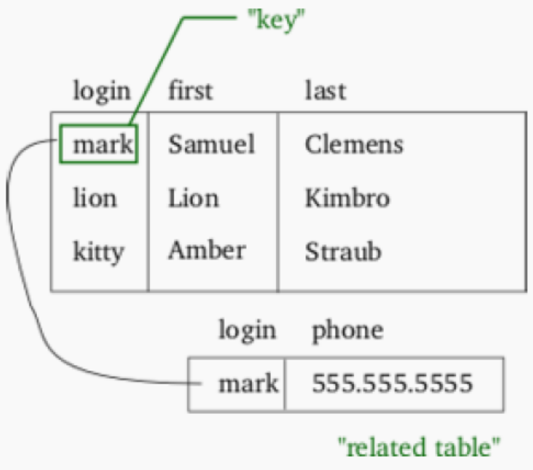


Activity Code	Activity Name
23	Patching
24	Overlay
25	Crack Sealing

Activity Code	Date	Route No.
24	01/12/01	I-95
24	02/08/01	I-66

Date	Activity Code	Route No.
01/12/01	24	I-95
01/15/01	23	I-495
02/08/01	24	I-66

Diagram of an example database according to the Relational model.[3]

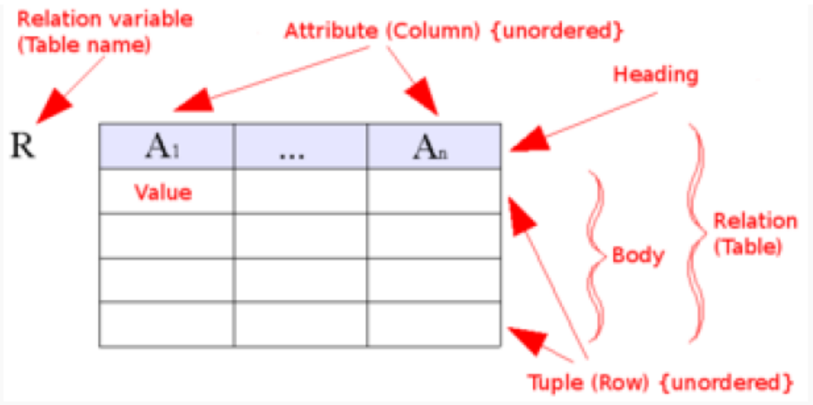


login	first	last
mark	Samuel	Clemens
lion	Lion	Kimbro
kitty	Amber	Straub

login	phone
mark	555.555.5555

In the relational model, related records are linked together with a "key".

The purpose of the relational model is to provide a declarative method for specifying data and queries: users directly state what information the database contains and what information they want from it, and let the database management system software take care of describing data structures for storing the data and retrieval procedures for answering queries.
Most relational databases use the SQL data definition and query language; these systems implement what can be regarded as an engineering approximation to the relational model. A *table* in an SQL database schema corresponds to a predicate variable; the contents of a table to a relation; key constraints, other constraints, and SQL queries correspond to predicates. However, SQL
The relational model's central idea is to describe a database as a collection of predicates over a finite set of predicate variables, describing constraints on the possible values and combinations of values. The content of the database at any given time is a finite (logical) model of the database, i.e. a set of relations, one per predicate variable, such that all predicates are satisfied. A request for information from the database (a database query) is also a predicate.



Relational model concepts.

Other models are the hierarchical model and network model. Some systems using these older architectures are still in use today in data centers with high data volume needs, or where existing systems are so complex and abstract it would be cost-prohibitive to migrate to systems employing the relational model; also of note are newer object-oriented databases.

TEXT BOOKS

1. Elmasri & Navathe, "Fundamental of Database Systems", Addison Wesley, 5th Edition, 2006

2. R Ramakrishnan & J Gehrke, Database Management Systems, McGraw Hill, Third Edition, 2002