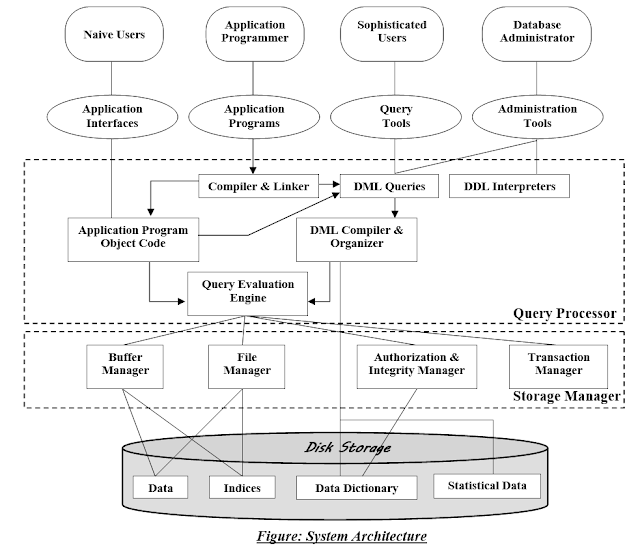
**11 a)**



**11 b)**

There are four different types of database-system users, differentiated by the way they expect to interact with the system. Different types of user interfaces have been designed for the different types of users.

1) Naïve Users

2) Application Programmers

3) Sophisticated Users

4) Specialized Users

**12 a)**

**Fundamental Relational algebra operations:**

SELECT ( )

PROJECT (π)

UNION (U)

SET DIFFERENCE (-)

CARTESIAN PRODUCT (X)

RENAME (ρ)

12 b) \*\*\*\*\*\*\*\*\*\*\*\*

**13 a)**

***Authorization:*** We may want to differentiate among the users as far as the type of access they are permitted on various data values in the database. These differentiations are expressed in terms of authorization. (The most common authorization is: *read authorization*, which allows reading, but not modification of data; *insert authorization*, which allows insertion of new data, but not modification of existing data; *update authorization*, which allows modification, but not deletion, of data; and *delete authorization*, which allows deletion of data. We may assign the user all, none, or a combination of these types of authorization).

Forms of authorization on parts of the database:

* + - **Read** - allows reading, but not modification of data.
    - **Insert** - allows insertion of new data, but not modification of existing data.
    - **Update** - allows modification, but not deletion of data.
    - **Delete** - allows deletion of data.

Forms of authorization to modify the database schema:

* + - **Index** - allows creation and deletion of indices.
    - **Resources** - allows creation of new relations.
    - **Alteration** - allows addition or deletion of attributes in a relation.
    - **Drop** - allows deletion of relations.

**Authorization Specification in SQL**

The **grant** statement is used to confer authorization

**grant** <privilege list> **on** <relation name or view name> **to** <user list>

<user list> is:

* a user-id
* **public**, which allows all valid users the privilege granted
* A role

Granting a privilege on a view does not imply granting any privileges on the underlying relations. The grantor of the privilege must already hold the privilege on the specified item (or be the database administrator).

**Example**

* + - **select:** allows read access to relation,or the ability to query using the view
      * Example: grant users *U*1, *U*2, and *U*3 **select** authorization on the *branch* relation:

**grant select on** department **to** *U*1*, U*2*, U*3

* + - **insert**: the ability to insert tuples
    - **update**: the ability to update using the SQL update statement
    - **delete**: the ability to delete tuples.
    - **all privileges**: used as a short form for all the allowable privileges

**13 b)**

##### Boyce-Codd Normal Form(BCNF):

A relation schema *R* is in BCNF with respect to a set *F* of functional dependencies if for all functional dependencies in *F*+ of the form

  **

where   *R* and **  *R*, at least one of the following holds:

1.   ** is trivial (i.e., **  )
2.  is a superkey for *R*

Example schema *not* in BCNF: *instr\_dept* (*ID, name, salary, dept\_name, building, budget* ) because *dept\_name* *building, budget* holds on *instr\_dept,* but *dept\_name* is not a super key **Decomposing a Schema into BCNF**

Suppose we have a schema *R* and a non-trivial dependency  ** causes a violation of BCNF. We decompose *R* into:

1. ( U  )

2. ( *R* - ( * -*  ) )

In our example,

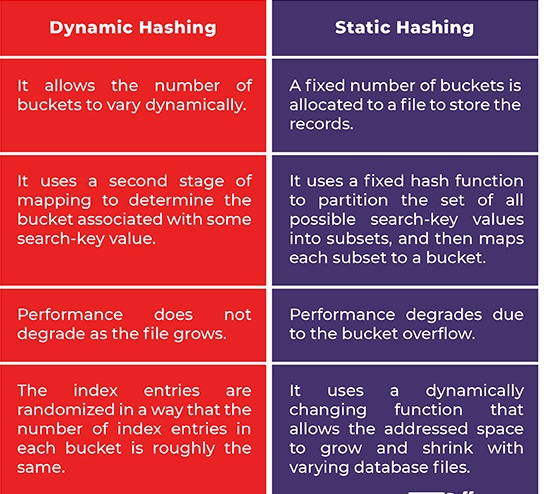
 = *dept\_name*

 = *building, budget*

and *inst\_dept* is replaced by

* + ( U  ) = ( *dept\_name, building, budget* )
  + ( *R* - (  *-*  ) ) = ( *ID, name, salary, dept\_name* )

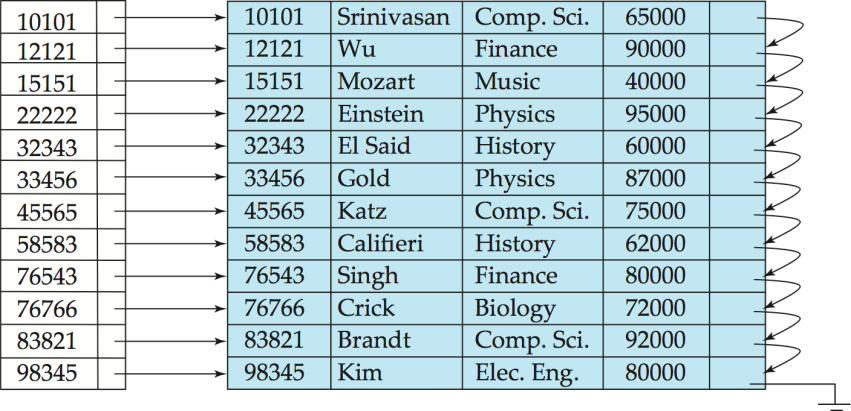
**Student can give any example of BCNF**

**14 a)**

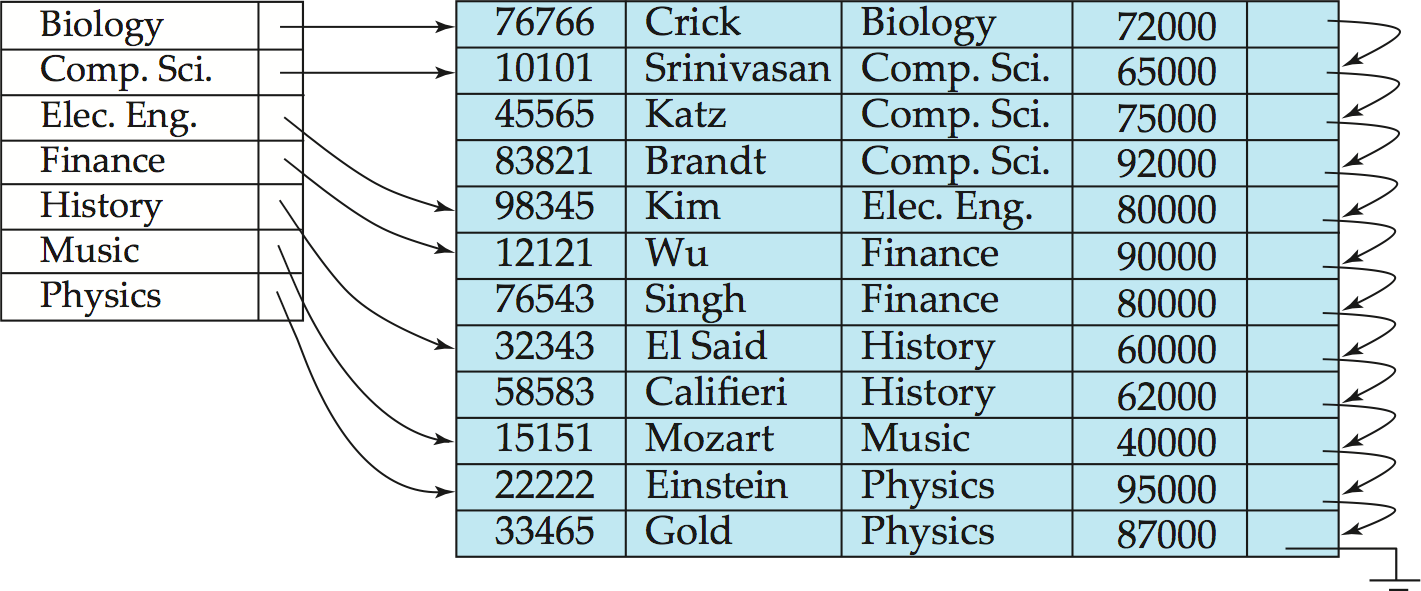
**14 b)**

Dense index — Index record appears for every search-key value in the file.

EX: Index on *ID* attribute of *instructor* relation

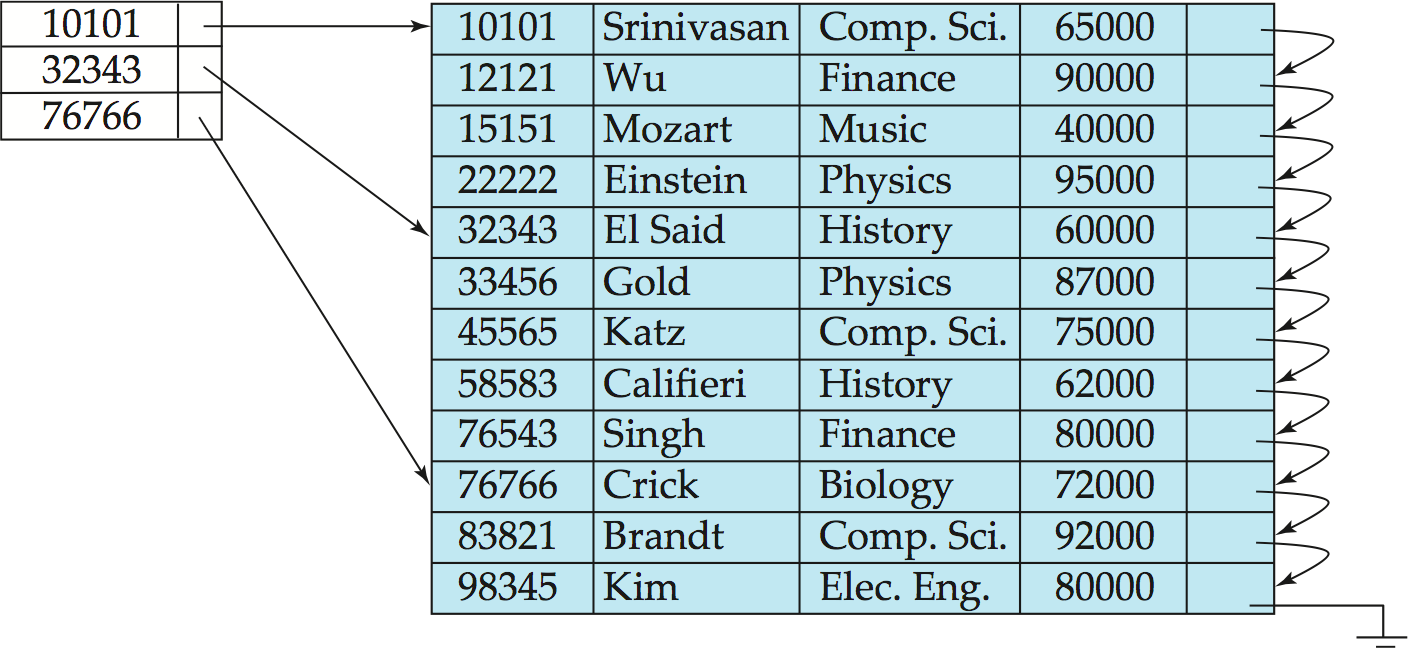


Dense index on *dept\_name*, with *instructor* file sorted on *dept\_name*



Sparse Index: contains index records for only some search-key values. It is Applicable when records are sequentially ordered on search-key

To locate a record with search-key value *K* we:

* + Find index record with largest search-key value < *K*
  + Search file sequentially starting at the record to which the index record points

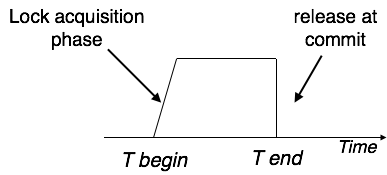
Compared to dense indices:

* + Less space and less maintenance overhead for insertions and deletions.
  + Generally slower than dense index for locating records.

**15 a)**

### Strict Two-Phase Locking

The first phase of Strict-2PL is same as 2PL. After acquiring all the locks in the first phase, the transaction continues to execute normally. But in contrast to 2PL, Strict-2PL does not release a lock after using it. Strict-2PL holds all the locks until the commit point and releases all the locks at a time.

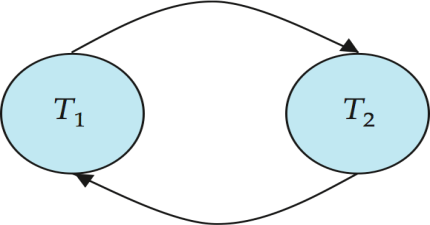


Strict-2PL does not have cascading abort as 2PL does.

**15 b)**

#### Testing for Serializability

* Consider some schedule of a set of transactions *T*1, *T*2, ..., *Tn*
* **Precedence graph** — a direct graph where the vertices are the transactions (names).
* We draw an arc from *Ti* to *Tj* if the two transaction conflict and *Ti* accessed the data item on which the conflict arose earlier.
* We may label the arc by the item that was accessed.



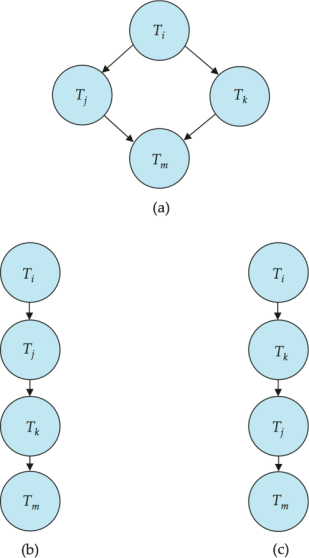
**Example**

**Example Schedule (Schedule A) + Precedence Graph**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| *T1* | *T2* | *T3* | *T4* | *T5* |
| read(Y) read(Z)  read(U)  read(U) write( U) | read(X)  read(Y) write(Y) | write(Z) | read(Y) write(Y) read(Z) write(Z) | read(V) read(W) read(W) | |

**Test for Conflict Serializability**

A schedule is conflict serializable if and only if its precedence graph is acyclic.

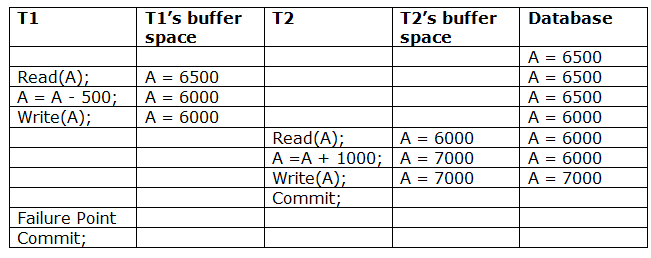
* Cycle-detection algorithms exist which take order *n*2 time, where *n* is the number of vertices in the graph.
  + (Better algorithms take order *n* + *e* where *e* is the number of edges.)
* If precedence graph is acyclic, the serializability order can be obtained by a *topological sorting* of the graph.
  + This is a linear order consistent with the partial order of the graph.
  + For example, a Serializability order for Schedule A would be *T*5  *T*1  *T*3  *T*2  *T*

**Test for View Serializability**

* The precedence graph test for conflict Serializability cannot be used directly to test for view Serializability.
  + Extension to test for view Serializability has cost exponential in the size of the precedence graph.
* The problem of checking if a schedule is view serializable falls in the class of *NP*-complete problems.
  + Thus existence of an efficient algorithm is *extremely* unlikely.
* However practical algorithms that just check some **sufficient conditions** for view Serializability can still be used.

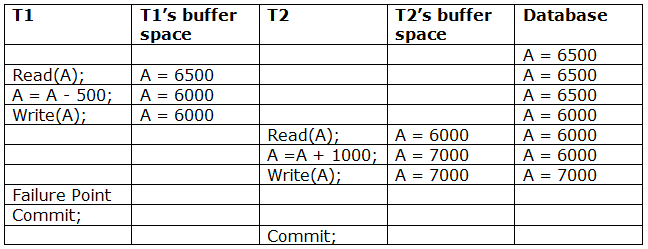
**16 a)**

Sometimes a transaction may not execute completely due to a software issue, system crash or hardware failure. In that case, the failed transaction has to be rollback. But some other transaction may also have used value produced by the failed transaction. So we also have to rollback those transactions.



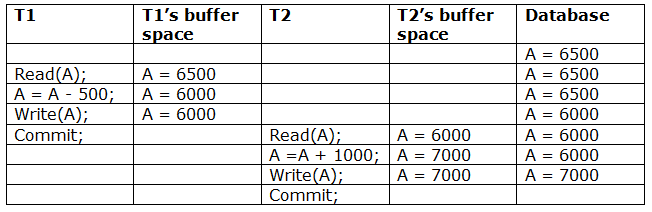
The above table 1 shows a schedule which has two transactions. T1 reads and writes the value of A and that value is read and written by T2. T2 commits but later on, T1 fails. Due to the failure, we have to rollback T1. T2 should also be rollback because it reads the value written by T1, but T2 can't be rollback because it already committed. So this type of schedule is known as irrecoverable schedule.

**Irrecoverable schedule:** The schedule will be irrecoverable if Tj reads the updated value of Ti and Tj committed before Ti commit.



The above table 2 shows a schedule with two transactions. Transaction T1 reads and writes A, and that value is read and written by transaction T2. But later on, T1 fails. Due to this, we have to rollback T1. T2 should be rollback because T2 has read the value written by T1. As it has not committed before T1 commits so we can rollback transaction T2 as well. So it is recoverable with cascade rollback.

**Recoverable with cascading rollback:** The schedule will be recoverable with cascading rollback if Tj reads the updated value of Ti. Commit of Tj is delayed till commit of Ti.



The above Table 3 shows a schedule with two transactions. Transaction T1 reads and write A and commits, and that value is read and written by T2. So this is a cascade less recoverable schedule.

**16 b)**

***Explain ARIES algorithm.***

Algorithm for Recovery and Isolation Exploiting Semantics (ARIES) is based on the Write Ahead Log (WAL) protocol. Every update operation writes a [log record](https://www.geeksforgeeks.org/log-based-recovery-in-dbms/) which is one of the following :

1. **Undo-only log record:**
2. **Redo-only log record:**
3. **Undo-redo log record:**  
   In it, every log record is assigned a unique and monotonically increasing log sequence number (LSN). Every data page has a page LSN field that is set to the LSN of the log record corresponding to the last update on the page. WAL requires that the log record corresponding to an update make it to stable storage before the data page corresponding to that update is written to disk. For performance reasons, each log write is not immediately forced to disk. A log tail is maintained in main memory to buffer log writes. The log tail is flushed to disk when it gets full. A transaction cannot be declared committed until the commit log record makes it to disk.

Once in a while the recovery subsystem writes a checkpoint record to the log. The checkpoint record contains the transaction table and the dirty page table. A master log record is maintained separately, in stable storage, to store the LSN of the latest checkpoint record that made it to disk. On restart, the recovery subsystem reads the master log record to find the checkpoint’s LSN, reads the checkpoint record, and starts recovery from there on.

The recovery process actually consists of 3 phases:

1. **Analysis:**  
   The recovery subsystem determines the earliest log record from which the next pass must start. It also scans the log forward from the checkpoint record to construct a snapshot of what the system looked like at the instant of the crash.
2. **Redo:**  
   Starting at the earliest LSN, the log is read forward and each update redone.
3. **Undo:**  
   The log is scanned backward and updates corresponding to loser transactions are undone.

Sometimes a transaction may not execute completely due to a software issue, system crash or hardware failure. In that case, the failed transaction has to be rollback. But some other transaction may also have used value produced by the failed transaction. So we also have to rollback those transactions.

**17 a)**

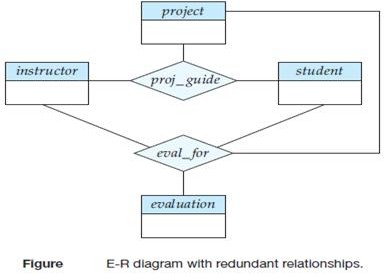
**Weak Entity Sets:** An entity set that is dependent on other entity set (or) an entity set that does not have a primary key is referred to as a **weak entity set**. An entity set that has a primary key is termed a **strong entity set**.

For a weak entity set to be meaningful, it must be associated with another entity set, called the **identifying** or **owner entity set**. The identifying entity set is said to **own** the weak entity set that it identifies. The relationship associating the weak entity set with the identifying entity set is called the **identifying relationship**. The identifying relationship is many-to-one from the weak entity set to the identifying entity set, and the participation of the weak e

**17 b)**

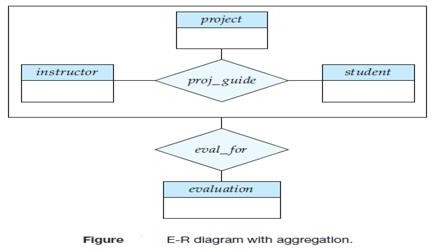
**Aggregation:** One limitation of the E-R model is that it cannot express relationships among relationships. To illustrate the need for such a construct, consider the ternary relationship proj\_guide, between an instructor, student and project. Now suppose that each instructor guiding a student on a project is required to file a monthly evaluation report. We model the evaluation report as an entity evaluation, with a primary key evaluation id. One alternative for recording the (student, project, instructor) combination to which an evaluation corresponds is to create a quaternary (4-way) relationship set eval\_for between instructor, student, project, and evaluation.

Using the basic E-R modeling constructs, we obtain the following E-R diagram.



It appears that the relationship sets proj\_guide and eval\_for can be combined into one single relationship set. Nevertheless, we should not combine them into a single relationship, since some instructor, student, project combinations may not have an associated evaluation.

There is ***redundant information*** in the resultant figure, however, since every instructor, student, project combination in eval\_for must also be in proj\_guide. The best way to model a situation such as the one just described is to use aggregation. **Aggregation** is an abstraction through which relationships are treated as higher-level entities. Thus, for our example, we regard the relationship set proj\_guide (relating the entity sets instructor, student, and project) as a higher-level entity set called proj\_guide. Such an entity set is treated in the same manner as is any other entity set.



We can then create a binary relationship eval\_for between proj\_*guide* and *evaluation* to represent which (*student*, *project*, *instructor*) combination an *evaluation* is for, as shown in the figure

**17 c)**

***Assertions:*** An assertion is *any condition that the database must always satisfy*. Domain constraints and referential integrity constraints are special forms of assertions. However, there are many constraints that we cannot express by using only these special forms. For example, “Every department must have at least five courses offered every semester” must be expressed as an assertion.