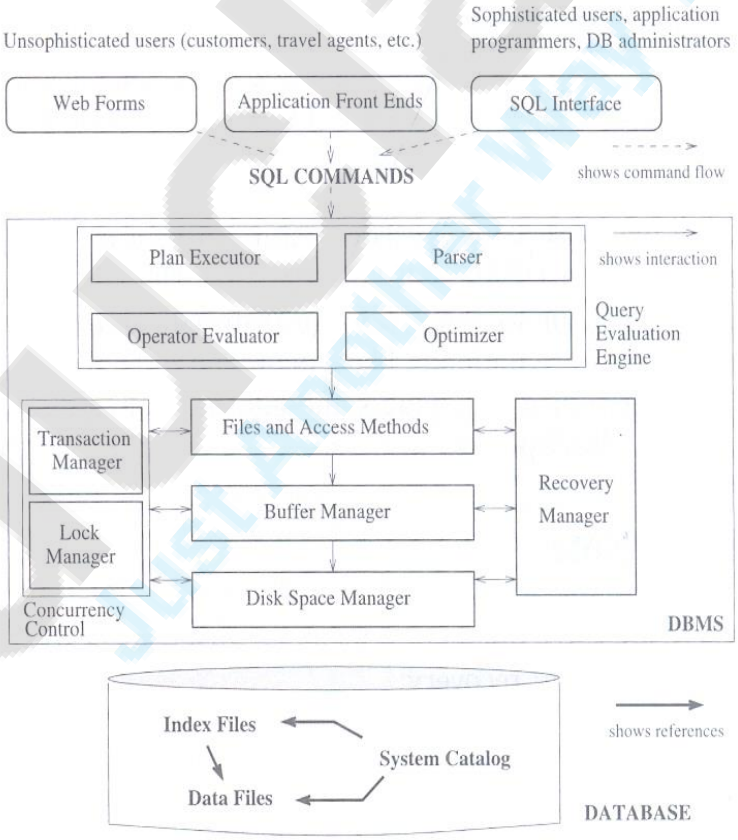
Module 1

1. What is DBMS.

- DBMS contains information about a particular enterprise
- Collection of large interrelated data
- Set of programs to create, maintain and access a database.
- An environment that is both convenient and efficient to use
- Ex. MySQL, Microsoft SQL Server, Oracle, Sybase, Microsoft Access
- Models real world enterprise
    i. Entities – students, faculty, courses
    ii. Relationships between entities – students' enrollment in courses, faculty teaching courses.



2. Explain architecture of DBMS.

- Query Evaluation Engine
    - When a user issues a query, the parsed query is presented to a query optimizer
    - Query optimization uses information about how the data is stored to produce an efficient execution plan for evaluating the query
    - Execution plan is a blueprint for evaluating a query, represented as a tree of relational operators
- File and Access Methods

- This layer supports the concept of file, which is a collection of pages or a collection of records
- It also organizes the information within the page

- Buffer Manager
  - Buffer manager brings pages in from disk to main memory as needed in response to read requests
- Disk Space Manager
  - Deals with the management of space on disk where data is stored
  - Higher layers allocate, deallocate, read and write pages through this layer
- Concurrency Control
  - Transaction Manager - ensures that transaction request and release locks according to a suitable locking protocol and schedules the execution transaction
  - Lock Manager - keeps track of requests for locks and grants locks on database objects when they become available
- Recovery Manager
  - Responsible for maintaining a log and restoring the system to a consistent state after a crash

3. How is DBMS different from conventional file system?

In conventional file system approach, each user defines and implements the file needed for a specific application as a part of programming the application.

For Example :

One user maintains the file of a student and their grades. Another user maintains the file of the students fees and their payments. Both needs the basic details and maintain separate files in it and programs to manipulate these files. This redundancy in defining and storing data results in wasted storage space and in redundant effort to maintain common data up-to-date.

Limitations of File Processing System

- Data Redundancy
  - Storing the same data multiple times in the database.
  - Leads to higher storage and access cost
- Data Inconsistency
  - All copies of data may not be updated properly. Ex. Part Information of a product is recorded differently in different files.
- Difficulty in accessing data
  - May have to write a new application program to satisfy an unusual request.
  - Ex. find all customers with the same postal code. Could generate this data manually, but a long job.
- Data Isolation
  - Data in different files. Data in different formats.
  - Difficult to write new application programs
- Integrity Problems

- Integrity constraints (ex. account balance > 0) are part of program code
- Difficult to add new constraints or change existing ones

- Atomicity of updates
  - Failures may leave database in an inconsistent state with partial updates carried out
  - Example: Transfer of funds from one account to another should either complete or not happen at all
- Concurrent Access by multiple users
  - Want concurrency for faster response time.
  - Uncontrolled concurrent accesses can lead to inconsistencies
  - E.g. two customers withdrawing funds from the same account at the same time - account has $500 in it, and they withdraw $100 and $50. The result could be $350, $400 or $450 if no protection.
- Security Problems
  - Every user of the system should be able to access only the data they are permitted to see.
  - E.g. payroll people only handle employee records, and cannot see customer accounts; tellers only access account data and cannot see payroll data.
  - Difficult to enforce this with application programs.

Above mentioned problems are resolved by using DBMS.

In File System, files are used to store data while, collections of databases are utilized for the storage of data in DBMS. Although File System and DBMS are two ways of managing data, DBMS has many advantages over File Systems. Typically when using a File System, most tasks such as storage, retrieval and search are done manually and it is quite tedious whereas a DBMS will provide automated methods to complete these tasks. Because of this reason, using a File System will lead to problems like data integrity, data inconsistency and data security, but these problems could be avoided by using a DBMS. Unlike File System, DBMS are efficient because reading line by line is not required and certain control mechanisms are in place.

4. What is Data Model and name various models.

Data model Describes structure of a database. It defines how data is connected to each other and how they are processed and stored inside the system. It is collection of conceptual tool for describing data, data relationship, data semantics & consistency constraints.

Types of Data Model

- Network Data Model
- Hierarchical Data Model
- Relational Data Model
- The Entity-Relationship Model
- Object- Based data Model
- Semi structured Data Model (XML)

5. Responsibilities of DB A.

DBA (Database Administrator) is responsible for

i. Schema Definition: Creates original database schema by executing a set of data definition statements in DDL.

ii. Storage structure and access method definition

iii. Security and Authorization: responsible for ensuring that unauthorized data access is not permitted

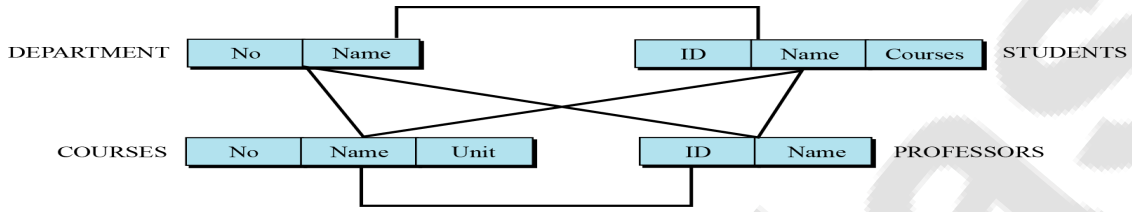
iv. Data Availability and recovery from failures: must ensure that if the system fails, user can continue to access as much of the uncorrupted data as possible

v. Database Tuning: responsible for modifying the database

vi. Maintenance: Periodic back-up, ensuring performance, Upgrading disk space if required.

6. Network data model – advantages/ disadvantages

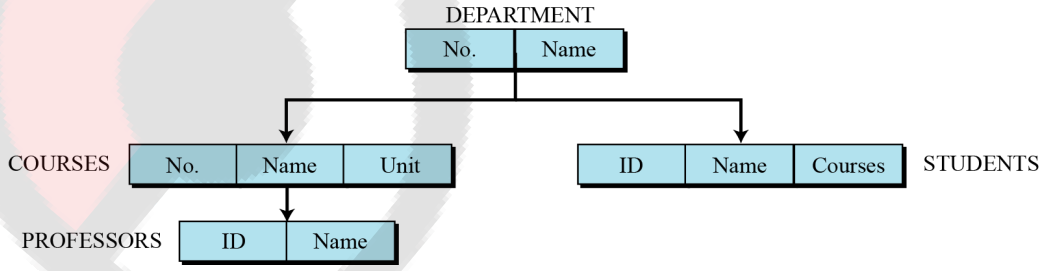
In network model, entities are organized in a graph, in which some entities can be accessed through several path. Relationships among data are represented by links.



- Advantages
  - Network Model is able to model complex relationships
  - Can handle most situations for modeling using record types and relationship types
  - Flexible pointer
- Disadvantages
  - Navigational and procedural nature of processing
  - Restructuring can be difficult

7. Hierarchical data model – advantages/ disadvantages

- Data is organized as an inverted tree
- Each data has only one parent but can have several children.
- At the top of the hierarchy, there is one data, which is called the root.



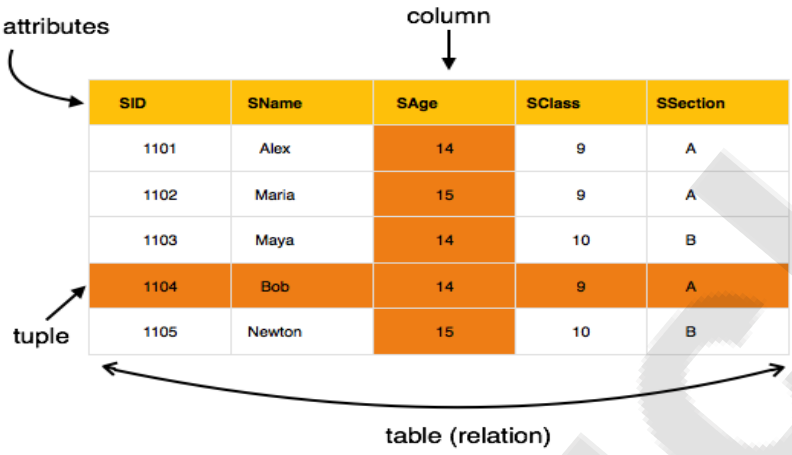
- Advantages
  - Simple to construct and operate on
  - Easy to search and add new branches

- Corresponds to a number of natural hierarchically organized domains - e.g., personnel organization in companies

- Disadvantages
  - Deletion of parent from the database also deletes all the child nodes
  - Difficult to depict all types of relationships

8. Relational data model – advantages/ disadvantages

Data is organized in two-dimensional tables called relations. Table (collection of table) is used to represent data and relationships among those data. Each row in a relation contains a unique value. Each column in a relation contains values from a same domain.



- Advantages
  - Conceptual simplicity
  - Design, implementation, maintenance and usage ease
  - Flexibility
  - Complex query
  - Security
- Disadvantages
  - Hardware overheads
  - Ease of design leads to bad design

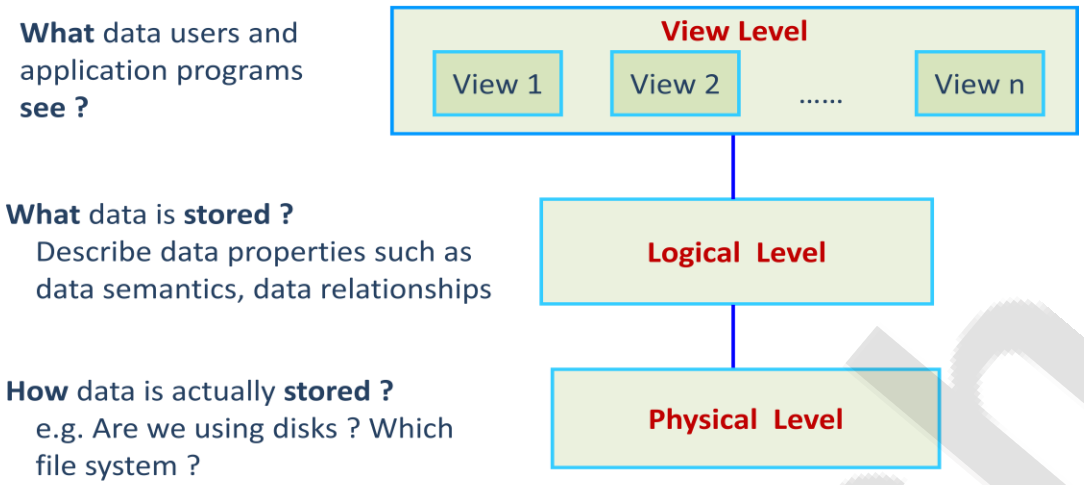
9. What are levels of abstraction in a database?

a. The data in a DBMS is described at three levels of abstraction:

i. Physical Level

ii. Logical Level

iii. View Level



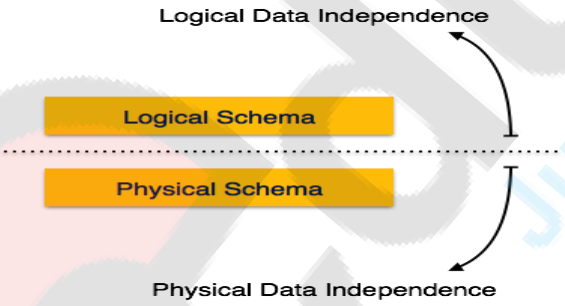
10. Difference Logical – Physical Independence

Physical Data Independence

- The ability to change or modify a physical schema(physical storage structures or devices) without changing logical schema
- Modification at the physical level are occasionally necessary to improve performance
- Easier to achieve as applications depend on the logical schema

Logical Data Independence

- The ability to change or modify a logical schema without changing external schema or application program
- Modification at the logical level necessary whenever the logical structure of the database is altered
- Alterations in conceptual schema may include addition or deletion of fresh entities, attributes or relationships
- More difficult to achieve



Module 2

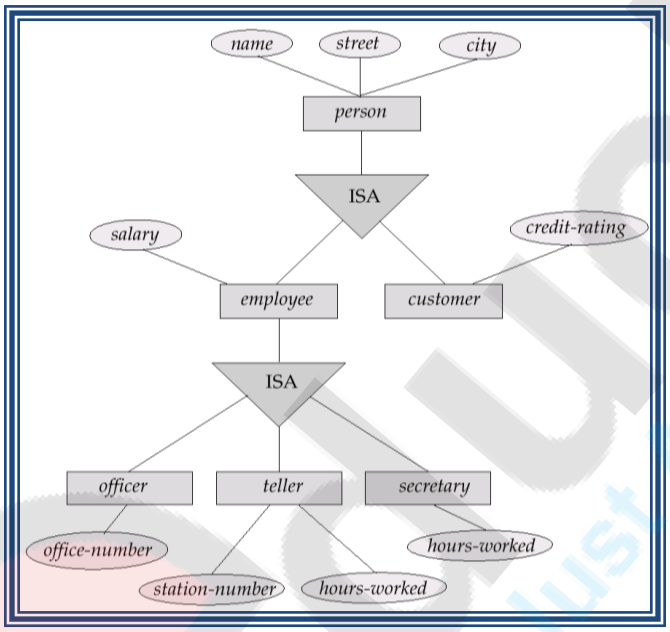
1. **Compulsory Question – ER Diagram for a given case study. Conversion to relational tables and normalization of these tables.**

2. **Differentiate between**

   i. **Generalization – Specialization**

Specialization

- Process of identifying subsets of an entity set that share some distinguishing characteristics
- Consider an entity set Person {name, street, city}. A Person may be – Customer, Employee
- Additional attributes – Customer {credit_rating}, Employee {salary}
- The ISA relationship is also referred to as a superclass-subclass relationship.
- It is a top-down design approach
- An entity of a subclass inherits all attributes of the superclass



Generalization

- Process of identifying some common characteristics of a collection of entity sets and creating a new entity set that contains entities possessing these common characteristics
- Used to express similarities between entity sets
- Subclasses are defined before the superclass
- Bottom up approach
- Specialization and generalization are simple inversions of each other

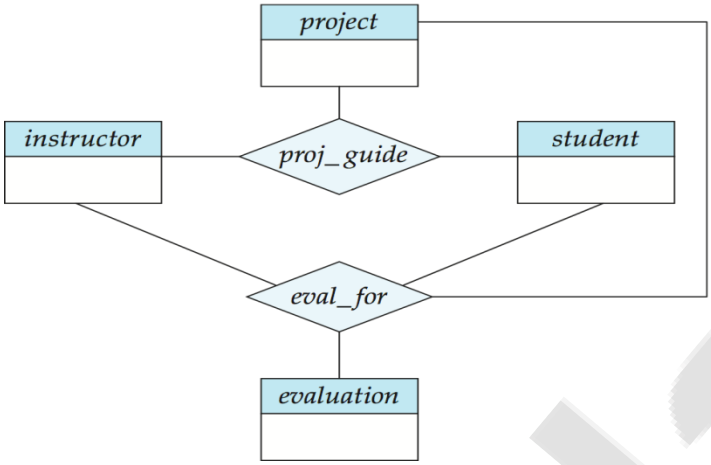
ii. ER model – relational model

iii. Aggregation – association

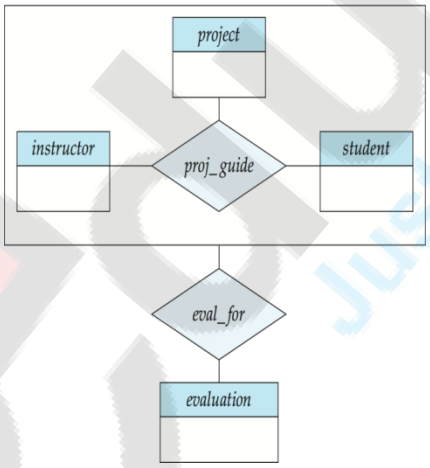
3. Write short notes on -

i. Aggregation

- The E-R model cannot express relationships among relationships.
- Aggregation allows us to treat a relationship set as an entity set for purpose of participation in other relationships
- Aggregation is an abstraction through which relationships are treated as higher-level entities



- A student is guided by a particular instructor on a particular project
- A student, instructor, project combination may have an associated evaluation
- Without introducing redundancy, the following diagram represents:
    i. A student is guided by a particular instructor on a particular project
    ii. A student, instructor, project combination may have an associated evaluation



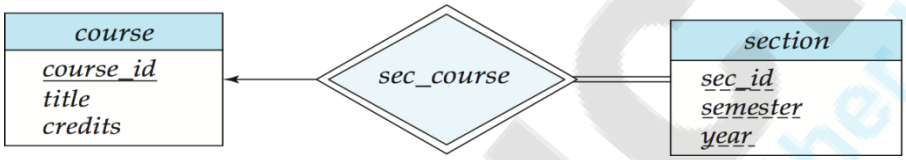
ii. Candidate key

- Minimal set of attributes necessary to uniquely identify an entity
- A superkey for which no subset is a superkey – minimal superkey
- Ex. {employeeID}

- An entity set may have more than one candidate key. Say if table has both employeeID and SSN, candidate keys will be
    i. {employeeID}
    ii. {ssn}

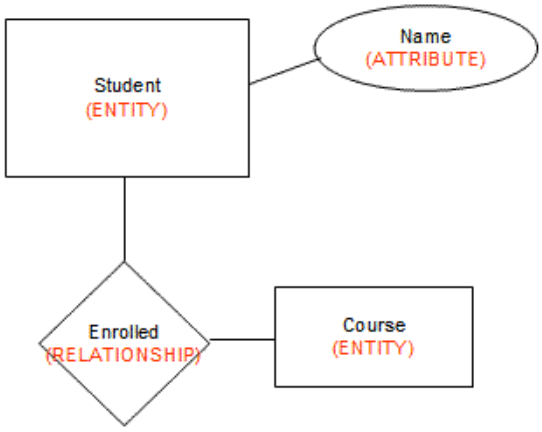
iii. Weak entity

- An entity set which does not have sufficient attributes to form a primary key
- An entity set that has a primary key is termed as strong entity set
- The existence of a weak entity set depends on the existence of a identifying(or owner) entity set
- It must relate to the identifying entity set via a total, one-to-many relationship set from the identifying → the weak
- The weak entity set is said to be existence dependent on the identifying entity set
- The identifying entity set is said to own the weak entity set that it identifies
- Discriminator(Partial key) of a weak entity set is a set of attributes that allows distinguishing entities of the weak entity set that depend on one particular strong entity. Ex. Payment number.
- The primary key of the weak entity is formed by the primary key of the identifying entity set plus weak entity set's discriminator. Ex. {account number, transaction number}



iv. ER model

- Describes data involved in a real-world enterprise in terms of objects and relationships among these objects
- Defines the conceptual view of a database.
- Used to develop an initial database design
- The basic things contained in an ER Model are-
    i. Entity
    ii. Attributes
    iii. Relationship

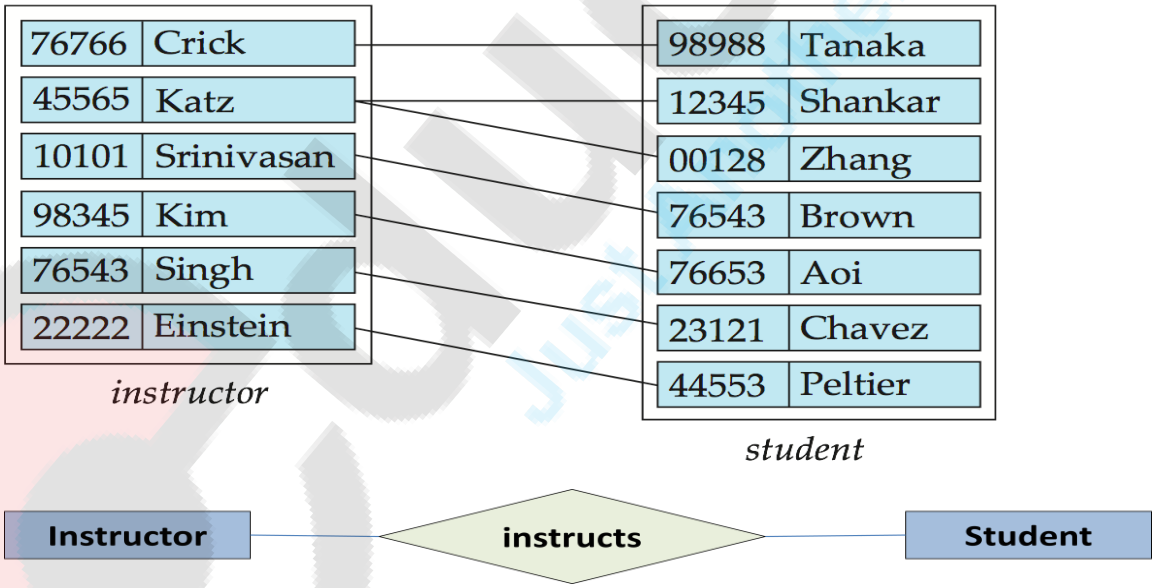


v. Association

- An association is a relationship among two or more entities.
- Ex. Instructor Crick instructs Tanaka. Relationship instructs has Student and Instructor as the participating entity sets.
- Set of relationships of the same type is known as relationship set
- If E1, E2, ……..En are entity sets, then a relationship set R is a subset of

  { (e1, e2,…….en) | e1 Є E1 , e2 Є E2 , …en Є En }

  where (e1, e2,…….en) is a relationship and entity sets E1, E2, ……..En participate in the relation



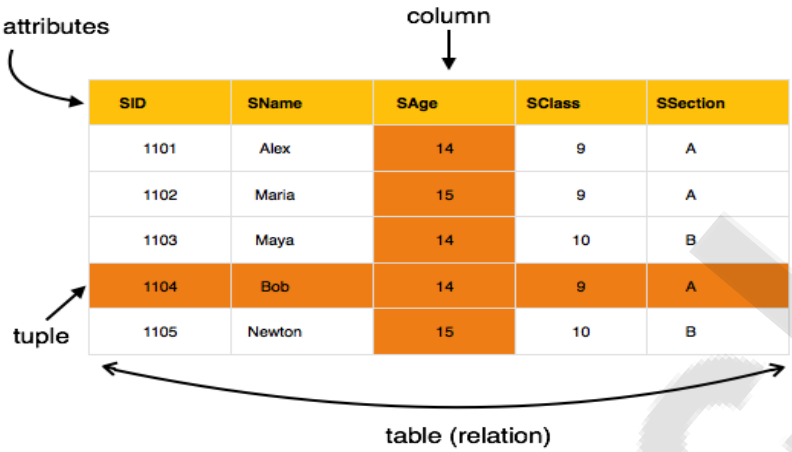
Module 3

1. Relational data model – advantages/ disadvantages

Data is organized in two-dimensional tables called relations. Table (collection of table) is used to represent data and relationships among those data. Each row in a relation contains a unique value. Each column in a relation contains values from a same domain.

Properties of a relational model

- Each relation (or table) in a database has a unique name
- Each row is unique
- Each relation is defined to be a set of unique tuples
- Each attribute within a table has a unique name
- For all relations, the domain of all attributes should be atomic
- The sequence of attributes (left to right) is insignificant
- The sequence of tuples is insignificant



- Advantages
  - Conceptual simplicity
  - Design, implementation, maintenance and usage ease
  - Flexibility
  - Complex query
  - Security
- Disadvantages
  - Hardware overheads
  - Ease of design leads to bad design

Module 4

1. Explain hash based indexing. Discuss use of hash function in identifying a bucket to search.

Hashing allow us to avoid accessing an index structure. Hashing provides a way of constructing indices.

Static hashing

A bucket is a unit of storage containing one or more records. Let K denote set of all search-key values and B denote set of all bucket addresses. Hash function h is a function from the set of all search-key values K to the set of all bucket addresses B. Hash function is used to locate records for access, insertion as well as deletion. To insert a record with search key value Ki, compute h(Ki), which gives the address of the bucket for that record. If there is space in that bucket to store the record, the record is stored in that bucket. Records with different search-key values may be mapped to the same bucket. Thus entire

bucket has to be searched sequentially to locate a record. Suppose two search-key value k5 and k7 have the same hash value; h(k5) = h(k7). In a hash file organization we obtain the bucket of a record directly from its search-key value using a hash function

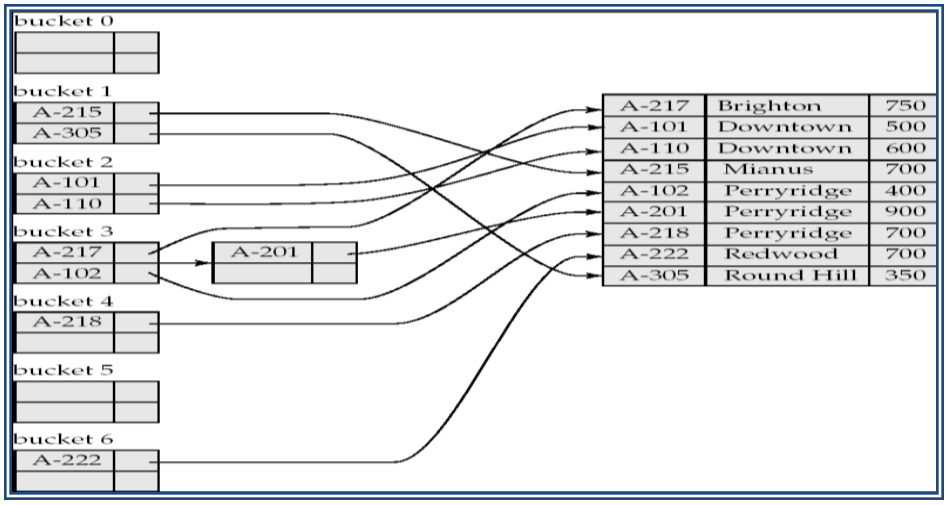
An ideal hash function distributes the stored keys uniformly across all buckets, so that every bucket has the same number of records. Worst hash function maps all search-key values to the same bucket. Choose a hash function that assigns search-key value to bucket with following qualities:

- An ideal hash function is uniform, i.e., each bucket is assigned the same number of search-key values from the set of all possible values
- Ideal hash function is random, so each bucket will have the same number of records assigned to it irrespective of the actual distribution of search-key values in the file

Example of Hash File Organization

- Hash Function :
  - Sum of binary representation of characters of key
  - Sum % number of buckets
- 10 buckets
- Binary representation of the i^{th} character is assumed to be integer i.
- Result of hash function. Ex
  - h(Perryridge) = 5
  - h(Round Hill) = 3 h(Brighton) = 3

- Hashing can be used not only for file organization, but also for index-structure creation
- A hash index organizes the search keys, with their associated record pointers, into a hash file structure
- Hash index is constructed as follows:
  - Apply a hash function on the search key to identify a bucket
  - Store the key and its associated pointers in the bucket



2. Differentiate-

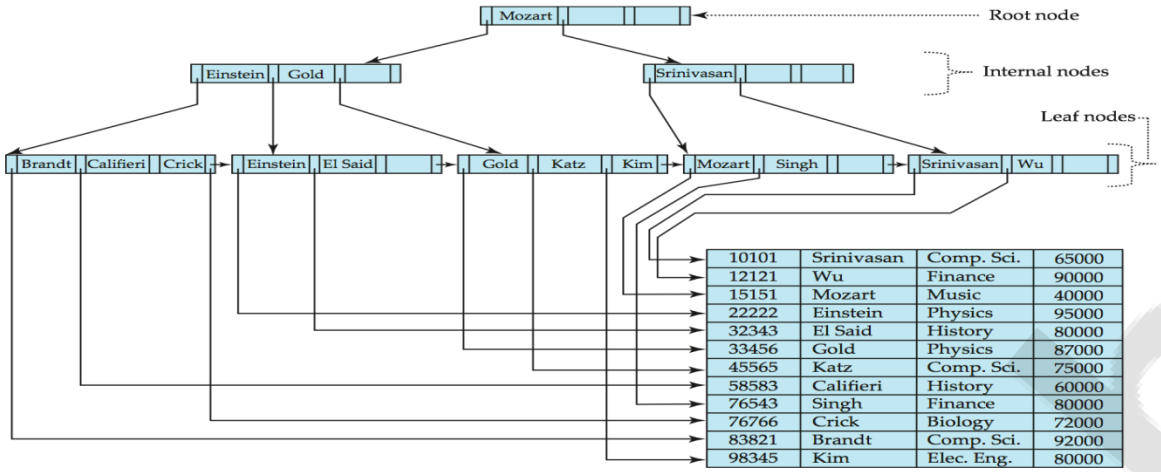
i. B & B+ tree

B+ tree

- A B^+-tree is a balanced rooted tree of order n if it satisfies following properties:
  - All paths from root to leaf are of the same length
  - Each node that is not a root or a leaf has between $\lceil n/2 \rceil$ and n children.
  - Each node, including leaf nodes (except root) has between $\lceil (n–1)/2 \rceil$ and n–1 values
  - Special cases:
    - If the root is not a leaf, it has at least 2 children.
    - If the root is a leaf (that is, there are no other nodes in the tree), it can have between 0 and (n–1) values.
- Contains less number of levels.
- B^+-Tree Node Structure

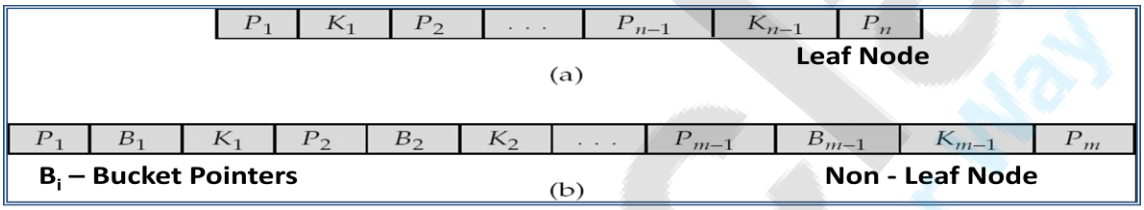
  - K_i are the search-key values
  - P_i are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes).
  - The search-keys in a node are ordered

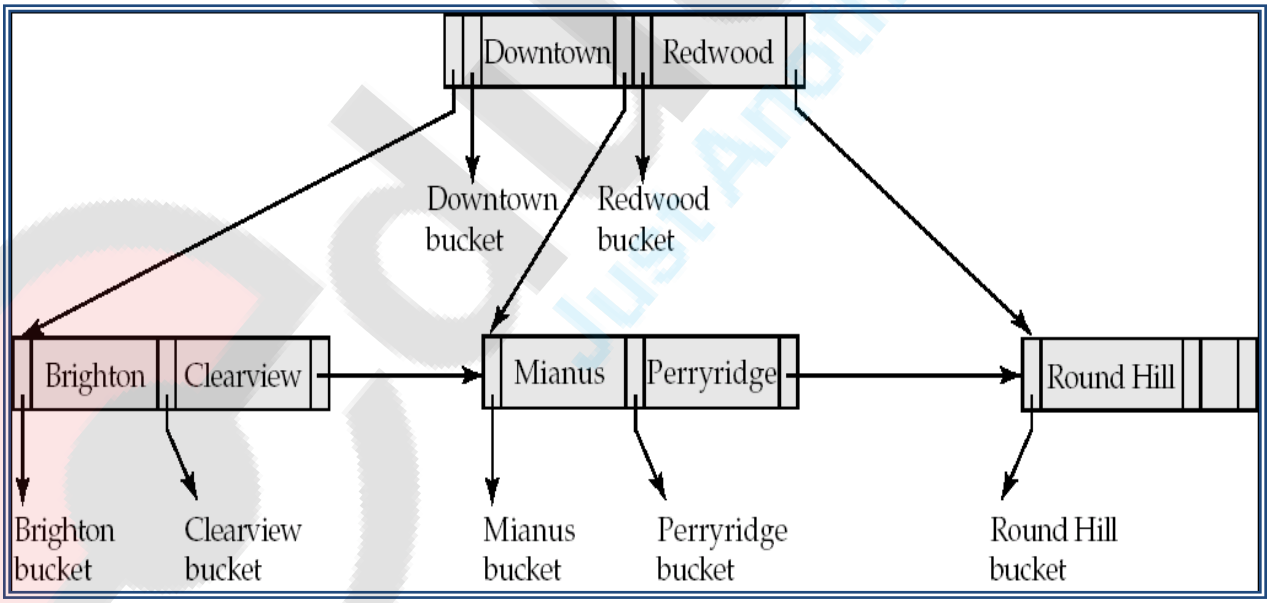
    K1 < K2 < K3 < . . . < Kn–1
  - P_n is used to chain together the leaf nodes in search key order



B Trees

- Similar to B^+-tree
- B-tree eliminates redundant storage of search keys
- A B-tree allows search-key values to appear only once. Search keys in nonleaf nodes do not appear again in leaf nodes
- An additional pointer field for each search key in a nonleaf node must be included





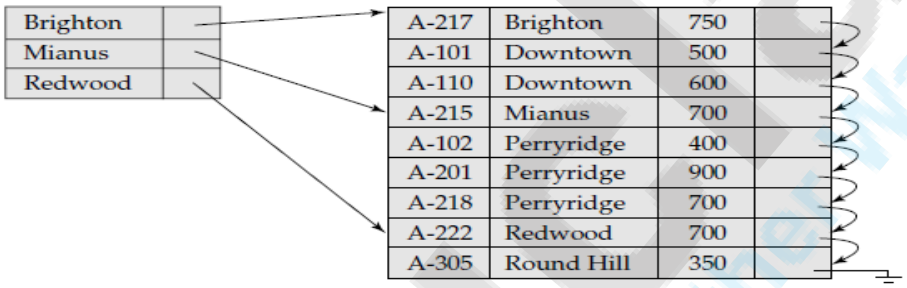
- Advantages of B-Tree indices:
  - May use less tree nodes than a corresponding B+-Tree.
  - Sometimes possible to find search-key value before reaching leaf node.
- Disadvantages of B-Tree indices:
  - Only small fraction of all search-key values are found early

- Insertion and deletion more complicated than in B+-Trees
- Implementation is harder than B+-Trees.

ii. Clustered vs. unclustered index

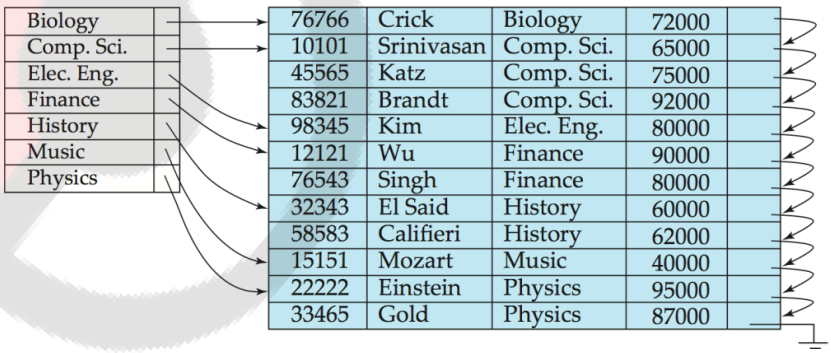
Clustered Index

- A clustered index determines the order in which the rows of a table are stored on disk i.e. search key of index specifies the sequential order of the file.
- If a table has a clustered index, then the rows of that table will be stored on disk in the same exact order as the clustered index.
- There can be only one clustered index per table, because the data rows themselves can be sorted in only one order.
- Can be sparse or dense.
- Also called Primary index(The search key of a primary index is usually but not necessarily the primary key)
- Index-sequential file: ordered sequential file with a primary index



Unclustered Index

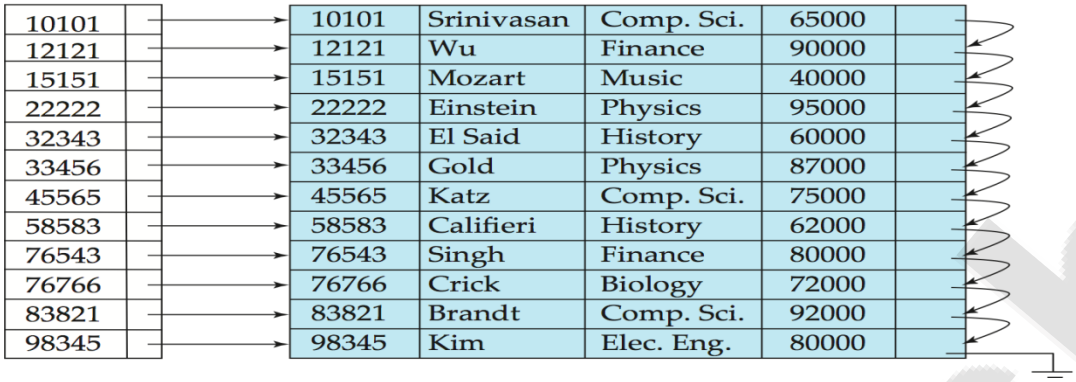
- An index whose search key specifies an order different from the sequential order of the file.
- A non-clustered index has no effect on which the order of the rows will be stored.
- A table can have multiple non-clustered indexes.
- Also called Secondary index.
- Has to be dense.



iii. Sparse vs. dense index

Dense index

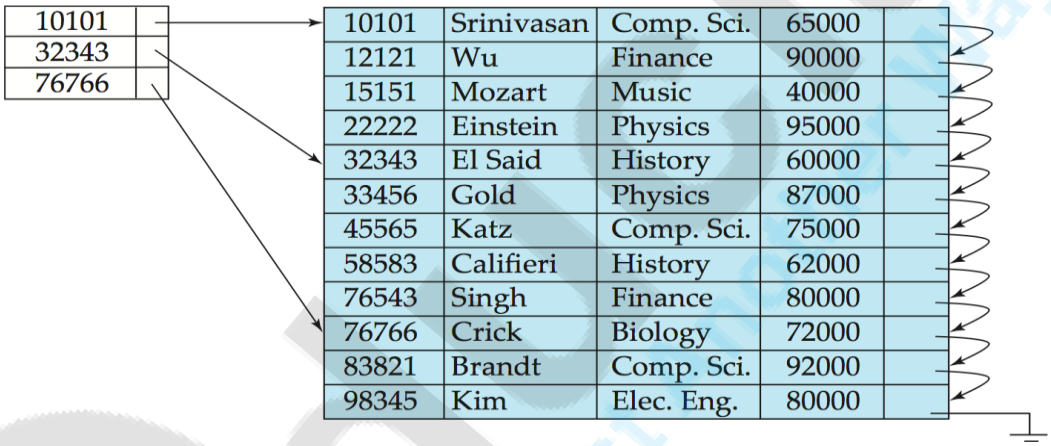
- An index record appears for every search-key value in the file



10101	10101	Srinivasan	Comp. Sci.	65000	
12121	12121	Wu	Finance	90000	
15151	15151	Mozart	Music	40000	
22222	22222	Einstein	Physics	95000	
32343	32343	El Said	History	60000	
33456	33456	Gold	Physics	87000	
45565	45565	Katz	Comp. Sci.	75000	
58583	58583	Califieri	History	62000	
76543	76543	Singh	Finance	80000	
76766	76766	Crick	Biology	72000	
83821	83821	Brandt	Comp. Sci.	92000	
98345	98345	Kim	Elec. Eng.	80000	

Sparse Index

- It contains index records for only some search-key values
- Used when records are sequentially ordered on search-key, i.e. with clustering index.
- 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



10101	10101	Srinivasan	Comp. Sci.	65000	
32343	12121	Wu	Finance	90000	
76766	15151	Mozart	Music	40000	
	22222	Einstein	Physics	95000	
	32343	El Said	History	60000	
	33456	Gold	Physics	87000	
	45565	Katz	Comp. Sci.	75000	
	58583	Califieri	History	62000	
	76543	Singh	Finance	80000	
	76766	Crick	Biology	72000	
	83821	Brandt	Comp. Sci.	92000	
	98345	Kim	Elec. Eng.	80000	

3. What is an index on a file of records? What is a search key for an index? Why do we need indexes?

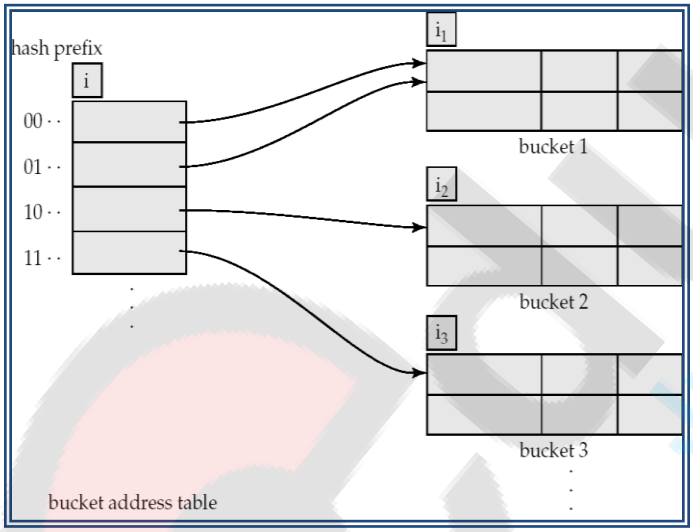
- Indexing mechanisms used to speed up random access to desired data
- An index record consists of a search-key value & pointers to one or more records with that value as their search-key value
- Search Key - attribute or set of attributes used to look up records in a file. (Not same as primary/ candidate key.)
- Pointer consists of identifier of a disk block & an offset within the disk block to identify record within the block
- An index file consists of records (called index entries) of the form

- Index files are typically much smaller than the original file
- Two basic kinds of indices:

- Ordered indices: search keys are stored in sorted order
- Hash indices: search keys are distributed uniformly across “buckets” using a “hash function”

4. Explain dynamic hashing with example.

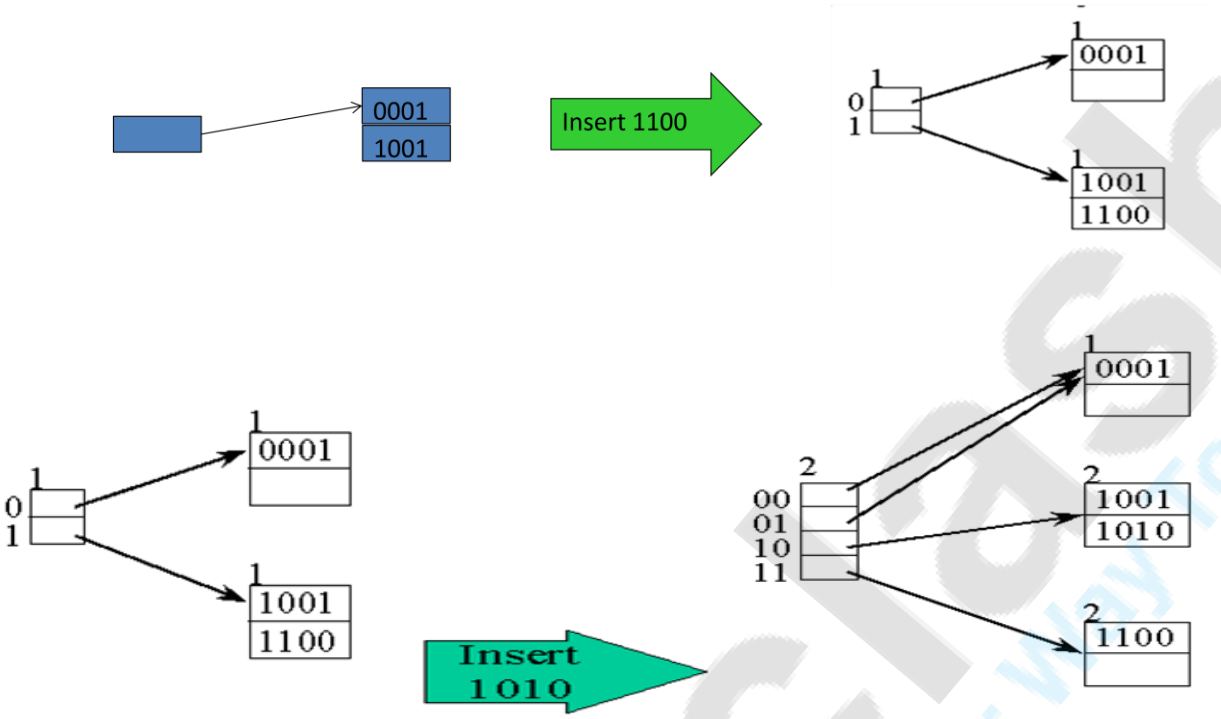
- Good for database that grows and shrinks in size. Allows the hash function to be modified dynamically
- Extendable hashing – one form of dynamic hashing
- Extendable hashing splits and combines buckets as the database grows and shrinks. Space efficiency is retained
- Reorganization is performed on only one bucket at a time, so resulting performance overhead is acceptably low
- Hash function generates values over a large range - b-bit integers, with b = 32
- Buckets are not created for each hash value. Buckets are created on demand, as records are inserted into the file
- Entire b is not used. At any point, i bits are used, where $0 \leq i \leq b$
- These i bits are used as an offset into an additional table of bucket addresses
- The value of i grows and shrinks with the size of the database
- Bucket address table size = 2^i. Initially i = 0
- Value of i grows and shrinks as the size of the database grows and shrinks
- Multiple entries in the bucket address table may point to a bucket
- The number of buckets also changes dynamically due to combining and splitting of buckets



Example

h(key) is 4 bits; 2 keys/bucket

Insert the record with hash value 0001 and 1001



Module V

- Normalisation is a set of data design standards.
- It is a process of decomposing unsatisfactory relations into smaller relations.
- Like entity–relationship modelling were developed as part of database theory.

Normalisation - Advantages

Reduction of data redundancy within tables:

- Reduce data storage space
- Reduce inconsistency of data
- Reduce update cost
- Remove many-to-many relationship
- Improve flexibility of the system

Normalisation - Disadvantages

- Reduction in efficiency of certain data retrieval as relations may be joined during retrieval.
- Increase join
- Increase use of indexes: storage (keys)

- Increase complexity of the system

First Normal Form - 1NF

A relation is in First Normal Form (1NF) if **ALL** its attributes are **ATOMIC**.

- If there are no repeating groups.
- If each attribute is a primitive.

  e.g. integer, real number, character string,

  but not lists or sets

- non-decomposable data item
- single-value

1NF - Actions Required

1) Examine for repeat groups of data
2) Remove repeat groups from relation
3) Create new relation(s) to include repeated data
4) Include key of the 0NF to the new relation(s)
5) Determine key of the new relation(s)

Second Normal Form - 2NF

A relation is in 2NF if it is in 1NF and every non-key attribute is dependent on the whole key

i.e. Is not dependent on part of the key only.

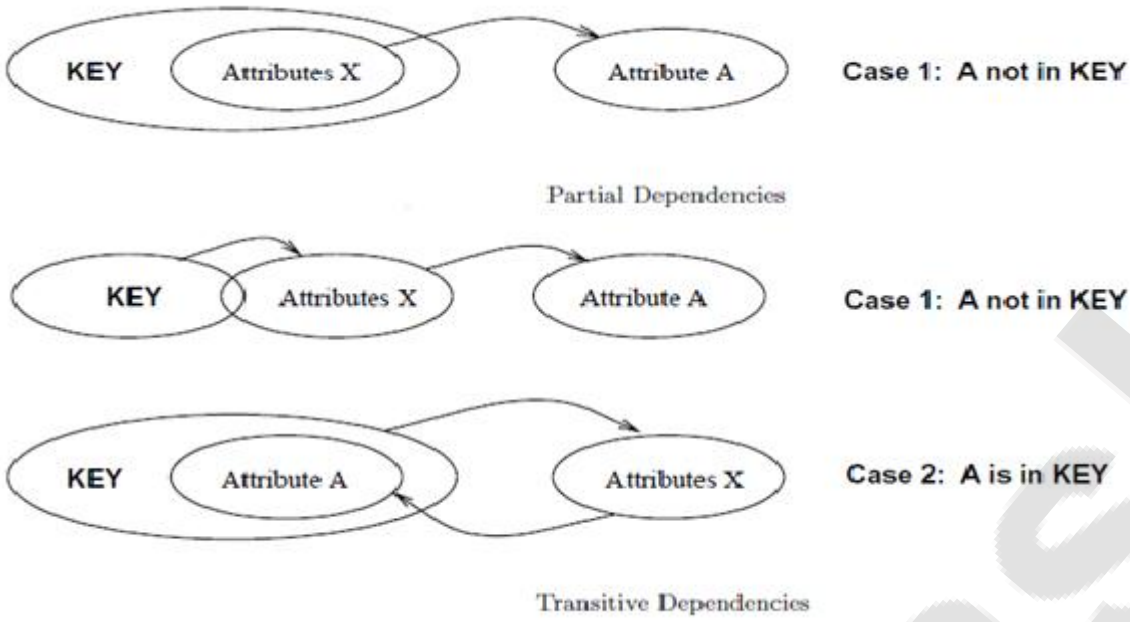
2NF - Actions Required

If entity has a concatenated key

1) Check each attribute against the whole key
2) Remove attribute and partial key to new relation
3) Optimise relations

Third Normal Form - 3NF

A relation is in 3NF if it is in 2NF and each non-key attribute is only dependent on the whole key, and not dependent on any non-key attribute. i.e. no transitive dependencies



3NF - Actions Required

1) Check each non-key attribute for dependency against other non-key fields

2) Remove attribute depended on another non-key attribute from relation

3) Create new relation comprising the attribute and non-key attribute which it depends on

4) Determine key of new relation

5) Optimise

Boyce-Codd Normal Form (BCNF)

A Table is said to be in BCNF if

a) It is in Third Normal Form

b) each determinant is a candidate key .

Fourth normal form

A Table is said to be in Fourth Normal Form if

a) It is in BCNF

b) It contains no more than one multi-valued dependency.

Fifth normal form

Also called as **Projection-Join Normal Form (PJNF**)

A Table is said to be in Fifth Normal Form if

a) It is in fourth Normal Form

b) Every join dependency in R is implied by candidate key if R.

Multivalued Dependencies

Suppose that we have a relation with attributes course, teacher, and book, which we denote as CTB.

There are no FDs; the key is CTB. Recommended texts for a course are independent of the instructor.

course	*teacher*	*book*
Physics101	Green	Mechanics
Physics101	Green	Optics
Physics101	Brown	Mechanics
Physics101	Brown	Optics
Math301	Green	Mechanics
Math301	Green	Vectors
Math301	Green	Geometry

There are three points to note here:

- The relation schema CTB is in BCNF; thus we would not consider decomposing it further if we looked only at the FDs that hold over CTB.
- There is redundancy. The fact that Green can teach Physics101 is recorded once per recommended text for the course. Similarly, the fact that Optics is a text for Physics101 is recorded once per potential teacher.
- The redundancy can be eliminated by decomposing CTB into CT and CB.
- The redundancy in this example is due to the constraint that the Books for a Course are independent of the Teachers, which cannot be expressed in terms of FDs.
- This constraint is an example of a multivalued dependency, or MVD.

Formal Definition Of MVD:

- Let R be a relation schema and let X and Y be subsets of the attributes of R.
    - **multivalued dependency** X→→Y is said to hold over R if, each X value is associated with a set of Y values and this set is independent of the values in the other attributes.

Fourth Normal Form

- Let R be a relation schema, X and Y be nonempty subsets of the attributes of R, and F be a set of dependencies that includes both FDs and MVDs.
- R is said to be in **fourth normal form (4NF)** if for every MVD X →→ Y that holds over R, one of the following statements is true:

- $Y \subseteq X$ or $XY = R$, or
- X is a superkey.

The relation CTB is not in 4NF because $C \twoheadrightarrow T$ is a nontrivial MVD and C is not a key. We can eliminate the resulting redundancy by decomposing CTB into CT and CB; each of these relations is then in 4NF.

In reading this definition, it is important to understand that the definition of a key has not changed—the key must uniquely determine all attributes through FDs alone. $X \twoheadrightarrow Y$ is a **trivial MVD** if $Y \subseteq X \subseteq R$ or $XY = R$; such MVDs always hold.

Join Dependencies

- A **join dependency** (JD) {R1, …. ,Rn} is said to hold over a relation R if R1, …. ,Rn is a lossless-join decomposition of R.
- Unlike FDs and MVDs, there is no set of sound and complete inference rules for JDs.

An MVD $X \twoheadrightarrow Y$ over a relation R can be expressed as the join dependency ⋈ {XY, X(R−Y)}. As an example, in the CTB relation, the MVD $C \twoheadrightarrow T$ can be expressed as the join dependency ⋈ {CT, CB}.

Fifth Normal Form

- A relation R is said to be in 5NF or PJNF if & only if, every join dependency that holds on R is implied by candidate keys of R

A relation schema R is said to be in **fifth normal form (5NF)** if for every JD ⋈ $\{R_1, \ldots, R_n\}$ that holds over R, one of the following statements is true:

- $R_i = R$ for some i, or
- The JD is implied by the set of those FDs over R in which the left side is a key for R.

The second condition deserves some explanation, since we have not presented inference rules for FDs and JDs taken together. Intuitively, we must be able to show that the decomposition of R into $\{R_1, \ldots, R_n\}$ is lossless-join whenever the **key dependencies** (FDs in which the left side is a key for R) hold. ⋈ $\{R_1, \ldots, R_n\}$ is a **trivial JD** if $R_i = R$ for some i; such a JD always holds.

DECOMPOSITION

- A decomposition of a relation schema R consists of replacing the relation schema by two (or more) relation schemas that each contain a subset of the attributes of R and together include all attributes in R
- Intuitively, we want to store the information in any given instance of R by storing projections of the instance

Lossless Join Decomposition

- Let R be a relation schema and let F be a set of FDs over R
- A decomposition of R into two schemas with attribute sets X and Y is said to be a lossless-join decomposition with respect to F if for every instance r of R that satisfies the dependencies in F,

$$\pi_X(r) \bowtie \pi_Y(r) = r$$

- All decompositions used to eliminate redundancy must be lossless
- The following simple test is very useful:

Let R be a relation and F be a set of FDs that hold over R

The decomposition of R into relations with attribute sets R1 and R2 is lossless if and only if F+ contains either the FD $R_1 \cap R_2 \rightarrow R_1$ or the FD $R_1 \cap R_2 \rightarrow R_2$ holds true.

Dependency-Preserving Decomposition

The decomposition of relation schema R with FDs F into schemas with attribute sets X and Y is **dependency-preserving** if $(F_X \cup F_Y)^+ = F^+$. That is, if we take the dependencies in F_X and F_Y and compute the closure of their union, we get back all dependencies in the closure of F. Therefore, we need to enforce only the dependencies in F_X and F_Y; all FDs in F^+ are then sure to be satisfied. To enforce F_X, we need to examine only relation X (on inserts to that relation). To enforce F_Y, we need to examine only relation Y.

Functional dependency: Attribute B has a functional dependency on attribute A if, for each value of attribute A, there is exactly one value of attribute B. For example, Employee Address has a functional dependency on Employee ID, because a particular Employee Address value corresponds to every Employee ID value. An attribute may be functionally dependent either on a single attribute or on a combination of attributes.

- Or in other words, An attribute B is said to be functionally dependent on attribute A if, every value of A uniquely determines the value of B. In this case, attribute A is determinant of B

SCP

S#	CITY	P#	QTY
S1	London	P1	100
S1	London	P2	100
S2	Paris	P1	200
S2	Paris	P2	200
S3	Paris	P2	300
S4	London	P2	400
S4	London	P4	400
S4	London	P5	400

Functional dependence. Case *a:* Let *r* be a relation, and let *X* and *Y* be arbitrary subsets of the set of attributes of *r*. Then we say that *Y* is functionally dependent on *X*—in symbols,

$X \rightarrow Y$

(read "*X* functionally determines *Y*," or simply "*X* arrow *Y*")—if and only if each *X* value in *r* has associated with it precisely one *Y* value in *r*. In other words, whenever two tuples of *r* agree on their *X* value, they also agree on their *Y* value.

For example, the relation shown in Fig. 11.1 satisfies the FD

{ S# } → { CITY }

Trivial functional dependency: A trivial functional dependency is a functional dependency of an attribute on a superset of itself. {Employee ID, Employee Address} → {Employee Address} is trivial, as is {Employee Address} → {Employee Address}.

Full functional dependency: An attribute is fully functionally dependent on a set of attributes X if it is

a) functionally dependent on X, and

b) not functionally dependent on any proper subset of X. {Employee Address} has a functional dependency on {Employee ID, Skill}, but not a full functional dependency, for it is also dependent on {Employee ID}.

Transitive dependency: A transitive dependency is an indirect functional dependency, one in which X→Z only by virtue of X→Y and Y→Z.

Multivalued dependency: A multivalued dependency is a constraint according to which the presence of certain rows in a table implies the presence of certain other rows. Or in other words, a multivalued dependency is said to be existing between A and B if for each value of attribute A, there is one or more associated value of attribute B.

Join dependency: A table T is subject to a join dependency if T can always be recreated by joining multiple tables each having a subset of the attributes of T.

Non-prime attribute: A non-prime attribute is an attribute that does not occur in any candidate key. Employee Address would be a non-prime attribute in the "Employees' Skills" table.

Closure of a Set of Functional Dependencies

- Given a set F set of functional dependencies, there are certain other functional dependencies that are logically implied by F.
    1. E.g. If $A \rightarrow B$ and $B \rightarrow C$, then we can infer that $A \rightarrow C$
- The set of all functional dependencies logically implied by F is the closure of F.
- We denote the closure of F by F^+.
- We can find all of F^+ by applying Armstrong's Axioms:
    1. if $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$ **(reflexivity)**
    2. if $\alpha \rightarrow \beta$, then $\gamma\ \alpha \rightarrow \gamma\ \beta$ **(augmentation)**
    3. if $\alpha \rightarrow \beta$, and $\beta \rightarrow \gamma$, then $\alpha \rightarrow \gamma$ **(transitivity)**

  These rules are
- sound (generate only functional dependencies that actually hold) and
- complete (generate all functional dependencies that hold).

4. **Self-determination: $A \rightarrow A$.**

5. **Decomposition: If $A \rightarrow BC$, then $A \rightarrow B$ and $A \rightarrow C$.**
6. **Union: If $A \rightarrow B$ and $A \rightarrow C$, then $A \rightarrow BC$.**
7. **Composition: If $A \rightarrow B$ and $C \rightarrow D$, then $AC \rightarrow BD$.**

8. Pseudo transitivity rule. If a→ ß holds and γß→ d holds, then aγ → d holds.

Minimal Cover for a Set of FDs

A **minimal cover** for a set F of FDs is a set G of FDs such that:

1. Every dependency in G is of the form X → A, where A is a single attribute.

2. The closure F+ is equal to the closure G+.

3. If we obtain a set H of dependencies from G by deleting one or more dependencies, or by deleting attributes from a dependency in G, then F+ ≠ H+.

i.e. a minimal cover for a set F of FDs is an equivalent set of dependencies that is minimal in two respects:

(1) Every dependency is as small as possible; that is, each attribute on the left side is necessary and the right side is a single attribute.

(2) Every dependency in it is required in order for the closure to be equal to F+.

- A general algorithm for obtaining a minimal cover of a set F of FDs:

1. **Put the FDs in a standard form:** Obtain a collection G of equivalent FDs with a single attribute on the right side (using the decomposition axiom).

2. **Minimize the left side of each FD:** For each FD in G, check each attribute in the left side to see if it can be deleted while preserving equivalence to F+.

3. **Delete redundant FDs:** Check each remaining FD in G to see if it can be deleted while preserving equivalence to F+.

- Note that the order in which we consider FDs while applying these steps could produce different minimal covers; there could be several minimal covers for a given set of FDs.
- More important, it is necessary to minimize the left sides of FDs before checking for redundant FDs.

Module VI

Overview of query optimization

- Query optimization is a function of many RDBMS in which multiple query plans for satisfying a query are examined and a good query plan is identified
- This may or not be the absolute best strategy because there are many ways of doing plans
- There is a trade-off between the amount of time spent figuring out the best plan and the amount running the plan
- Cost based query optimizers evaluate the resource footprint of various query plans and use this as the basis for plan selection
- Typically the resources which adds to cost are CPU path length, amount of disk buffer space, disk storage service time, and interconnect usage between units of parallelism
- The set of query plans examined is formed by examining possible access

paths (e.g., primary index access, secondary index access, full file scan) and various relational table join techniques (e.g., merge join, hash join, product join)

- The search space can become quite large depending on the complexity of the SQL query

- There are two types of optimization. These consist of logical optimization which generates a sequence of relational algebra to solve the query
- In addition there is physical optimization which is used to determine the means of carrying out each operation

Query Evaluation Plan

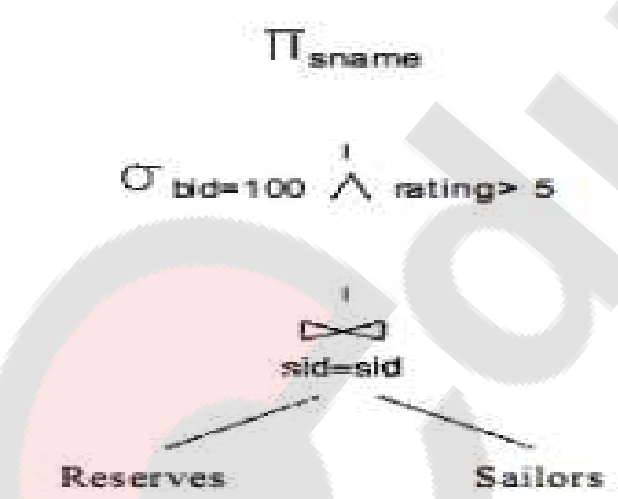
- A query evaluation plan (or simply plan) consists of an extended relational algebra tree, with additional annotations at each node indicating the access methods to use for each table and the implementation method to use for each relational operator
- Consider the following SQL query:

```
SELECT S.sname
FROM   Reserves R, Sailors S
WHERE  R.sid = S.sid
       AND R.bid = 100 AND S.rating > 5
```

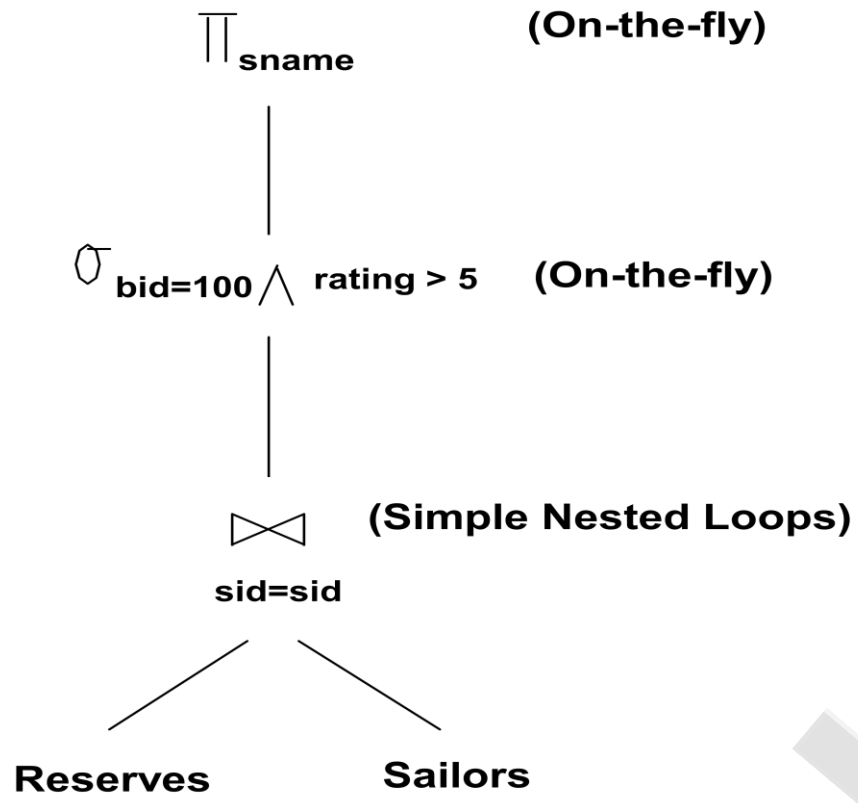
This query can be expressed in relational algebra as follows:

$$\pi_{sname}(\sigma_{bid=100 \wedge rating>5}(Reserves \bowtie_{sid=sid} Sailors))$$

- This expression is shown in the form of a tree
- The algebra expression partially specifies how to evaluate the query-first compute the natural join of Reserves and Sailors, then perform the selections, and finally project the sname

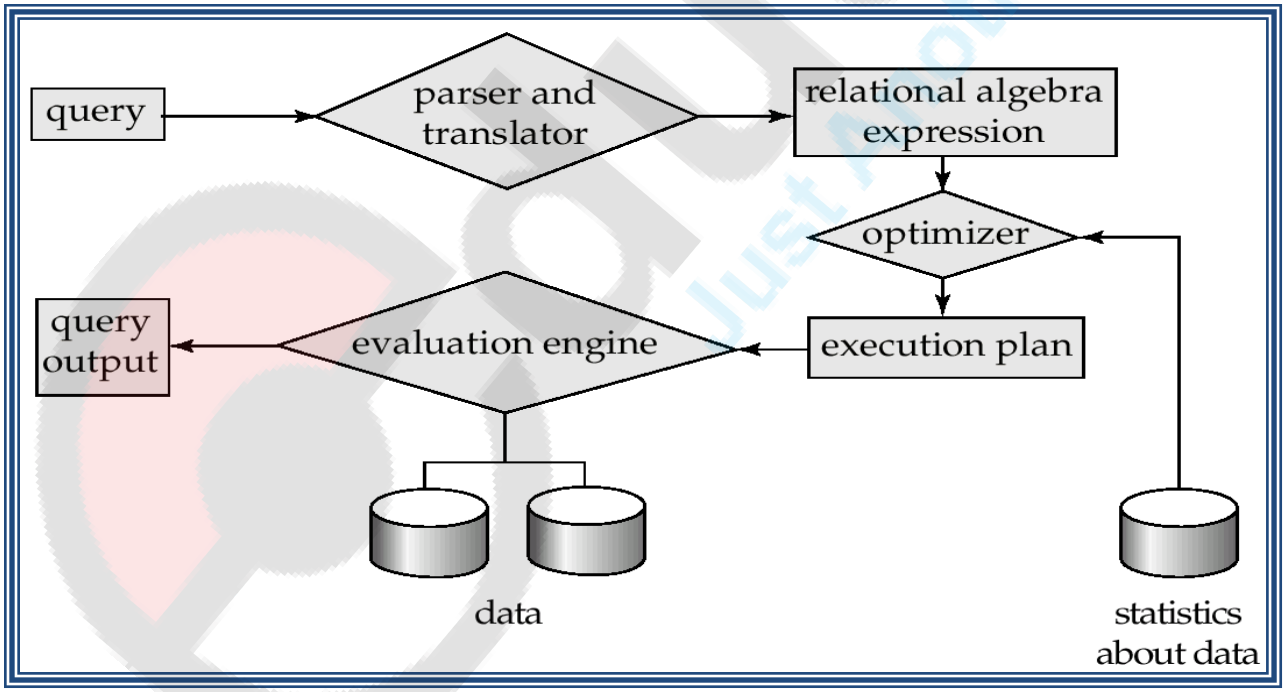


- To obtain a fully specified evaluation plan, we must decide on an implementation for each of the algebra operations involved
- We can use a page-oriented simple nested loops join with Reserves as the outer table and apply selections and projections to each tuple in the result of the join as it is produced
- The result of the join before the selections and projections is never stored in its entirety



Basic Steps in Query Processing: Evaluation of query

1. Parsing and translation
2. Optimization
3. Evaluation



Basic Steps in Query Processing

- Parsing and translation
  - translate the query into its internal form. This is then translated into relational algebra.
  - Parser checks syntax, verifies relations
- Evaluation

- The query-execution engine takes a query-evaluation plan, executes that plan, and returns the answers to the query.

- Before query processing can begin, the system must translate the query into a usable form
- A language such as SQL is suitable for human use, but is ill-suited to be the system's internal representation of a query
- A more useful internal representation is one based on the extended relational algebra
- Thus, the first action the system must take in query processing is to translate a given query into its internal form
- This translation process is similar to the work performed by the parser of a compiler
- In generating the internal form of the query, the parser checks the syntax of the user's query, verifies that the relation names appearing in the query are names of the relations in the database, and so on
- The system constructs a parse-tree representation of the query, which it then translates into a relational-algebra expression
- If the query was expressed in terms of a view, the translation phase also replaces all uses of the view by the relational-algebra expression that defines the view
- Most compiler texts cover parsing.

Basic Steps in Query Processing : Optimization

- A relational algebra expression may have many equivalent expressions
  - E.g., $\sigma_{balance<2500}(\Pi_{balance}(account))$ is equivalent to $\Pi_{balance}(\sigma_{balance<2500}(account))$
- Each relational algebra operation can be evaluated using one of several different algorithms
  - Correspondingly, a relational-algebra expression can be evaluated in many ways.
- Annotated expression specifying detailed evaluation strategy is called an **evaluation-plan**.
  - E.g., can use an index on *balance* to find accounts with balance < 2500,
  - or can perform complete relation scan and discard accounts with balance ≥ 2500
- **Query Optimization**: Amongst all equivalent evaluation plans choose the one with lowest cost.
  - Cost is estimated using statistical information from the database catalog
    - e.g. number of tuples in each relation, size of tuples, etc.

Measures of Query Cost

- Cost is generally measured as total elapsed time for answering query
  - Many factors contribute to time cost
    - *disk accesses, CPU,* or even network *communication*
- Typically disk access is the predominant cost, and is also relatively easy to estimate. Measured by taking into account
  - Number of seeks * average-seek-cost
  - Number of blocks read * average-block-read-cost
  - Number of blocks written * average-block-write-cost
    - Cost to write a block is greater than cost to read a block
      - data is read back after being written to ensure that the write was successful
- For simplicity we just use *number of block transfers from disk* as the cost measure
  - We ignore the difference in cost between sequential and random I/O for simplicity

- We also ignore CPU costs for simplicity
- Costs depends on the size of the buffer in main memory
  - Having more memory reduces need for disk access
  - Amount of real memory available to buffer depends on other concurrent OS processes, and hard to determine ahead of actual execution
  - We often use worst case estimates, assuming only the minimum amount of memory needed for the operation is available
- Real systems take CPU cost into account, differentiate between sequential and random I/O, and take buffer size into account
- We do not include cost to writing output to disk in our cost formulae

Relational Optimization

- For each enumerated plan, we have to estimate its cost
- There are two parts to estimating the cost of an evaluation plan for a query block:
  1. For each node in the tree, we must estimate the cost of performing the corresponding operation. Costs are affected significantly by whether pipelining is used or temporary relations are created to pass the output of an operator to its parent.
  2. For each node in the tree, we must estimate the size of the result and whether it is sorted. This result is the input for the operation that corresponds to the parent of the current node, and the size and sort order in turn affect the estimation of size, cost, and sort order for the parent
- As we see, estimating costs requires knowledge of various parameters of the input relations, such as the number of pages and available indexes
- Such statistics are maintained in the DBMS's system
- We use the number of page l/Os as the metric of cost and ignore issues such as blocked access, for the sake of simplicity

Module VII

Transaction Concept

- A **transaction** is a *unit* of program execution that accesses and possibly updates various data items.
- 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.
- E.g. transaction to transfer $50 from account A to account B:

read(A)

$A := A - 50$

write(A)

read(B)

$B := B + 50$

write(B)

ACID properties

- **Consistency requirement** in above money transfer example:
  - the sum of A and B is unchanged by the execution of the transaction
  - Without the consistency requirement, money could be created or destroyed by the transaction
- In general, consistency requirements include
  - primary keys and foreign keys
  - Implicit integrity constraints
    - e.g. sum of balances of all accounts, minus sum of loan amounts must equal value of cash-in-hand
- A transaction must see a consistent database.
- During transaction execution the database may be temporarily inconsistent.
- When the transaction completes successfully the database must be consistent
  - Erroneous transaction logic can lead to inconsistency
  - **Atomicity requirement**
  - if the transaction fails after step 3 and before step 6, money will be "lost" leading to an inconsistent database state
  - Thus, if the transaction never started or was guaranteed to complete, such an inconsistent state would not be visible except during the execution of the transaction
  - the system should ensure that updates of a partially executed transaction are not reflected in the database
  - **Durability requirement** — once the user has been notified that the transaction has completed (i.e., the transfer of the $50 has taken place), the updates to the database by the transaction must persist even if there are software or hardware failures.
  - We can guarantee durability by ensuring that either
  - 1. The updates carried out by the transaction have been written to disk before the transaction completes.
  - 2. Information about the updates carried out by the transaction and written to disk is sufficient to enable the database to reconstruct the updates when the database system is restarted after the failure.
- **Isolation requirement** — if between steps 3 and 6, another transaction T2 is allowed to access the partially updated database, it will see an inconsistent database (the sum $A + B$ will be less than it should be).

T1	**T2**
1. **read**(A)	
2. $A := A - 50$	

3. **write**(*A*)

read(A), read(B), print(A+B)

4. **read**(*B*)
5. *B* := *B* + 50
6. **write**(*B*

- Isolation can be ensured trivially by running transactions **serially**
  - that is, one after the other.
- However, executing multiple transactions concurrently has significant benefits, as we will see later.

Schedules

- Schedule – a sequences of instructions that specify the chronological order in which instructions of concurrent transactions are executed
  - a schedule for a set of transactions must consist of all instructions of those transactions
  - must preserve the order in which the instructions appear in each individual transaction.
- A transaction that successfully completes its execution will have a commit instructions as the last statement (will be omitted if it is obvious)
- A transaction that fails to successfully complete its execution will have an abort instructions as the last statement (will be omitted if it is obvious)
- Serial schedule
  - Each serial schedule consists of a sequence of instructions from various transactions, where the instructions belonging to one single transaction appear together in that schedule.
  - Thus, for a set of *n* transactions, there exist *n*! different valid serial schedules.
- Concurrent schedule
  - When the database system executes several transactions concurrently, the corresponding schedule no longer needs to be serial.
  - Several execution sequences are possible, since the various instructions from both transactions may now be interleaved. In general, it is not possible to predict exactly how many instructions of a transaction will be executed before the CPU switches to another transaction. Thus, the number of possible schedules for a set of *n* transactions is much larger than *n*!.

Serializability

- A serializable schedule over a set S of committed transactions is a schedule whose effect on any consistent database instance is identical to that of some complete serial schedule over S
- The database instance that results from executing the given schedule is identical to the database instance that results from executing the transactions in some serial order
- **Basic Assumption** – Each transaction preserves database consistency.

- Thus serial execution of a set of transactions preserves database consistency.

A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule. Different forms of schedule equivalence give rise to the notions of:

1. **conflict serializability**

2. **view serializability**

If S is a schedule in which there are 2 consecutive Instructions l_i and l_j of transactions T_i and T_j respectively, then l_i and l_j will **conflict** if and only if there exists some item Q accessed by both l_i and l_j, and at least one of these instructions is write (Q).

1. l_i = **read**(Q), l_j = **read**(Q). l_i and l_j don't conflict.
2. l_i = **read**(Q), l_j = **write**(Q). They conflict.
3. l_i = **write**(Q), l_j = **read**(Q). They conflict
4. l_i = **write**(Q), l_j = **write**(Q). They conflict

- Intuitively, a conflict between l_i and l_j forces a (logical) temporal order between them.
- If l_i and l_j are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule.

View Serializability

- Let S and S' be two schedules with the same set of transactions. S and S' are **view equivalent** if the following three conditions are met:

1. For each data item Q, if transaction T_i reads the initial value of Q in schedule S, then transaction T_i 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 as 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'.

As can be seen, view equivalence is also based purely on **reads** and **writes** alone.

- **Conditions 1 and 2 ensure that each transaction reads the same values in both schedules and, therefore, performs the same computation**
- **Condition 3, coupled with conditions 1 and 2, ensures that both schedules result in the same final system state**
- Schedule 1 is not view equivalent to schedule 2, since, in schedule 1, the value of account A read by transaction T2 was produced by T1,whereas this case does not hold in schedule 2
- However, schedule 1 is view equivalent to schedule 3, because the values of account A and B read by transaction T2 were produced by T1 in both schedules.

Concurrency Control

Lock-Based Protocols

- One way to ensure serializability is to require that data items be accessed in a mutually exclusive manner; that is, while one transaction is accessing a data item, no other transaction can modify that data item

- The most common method used to implement this requirement is to allow a transaction to access a data item only if it is currently holding a lock on that item
- A lock is a mechanism to control concurrent access to a data item
- Data items can be locked in two modes :

1. *exclusive (X) mode*. Data item can be both read as well as written. X-lock is requested using **lock-X** instruction.
2. *shared (S) mode*. Data item can only be read. S-lock is requested using **lock-S** instruction.

- Lock requests are made to concurrency-control manager. Transaction can proceed only after request is granted.
- We require that every transaction request a lock in an appropriate mode on data item Q, depending on the types of operations that it will perform on Q
- The transaction makes the request to the concurrency-control manager. The transaction can proceed with the operation only after the concurrency-control manager grants the lock to the transaction
- Given a set of lock modes, we can define a compatibility function on them as follows: Let A and B represent arbitrary lock modes
- Suppose that a transaction Ti requests a lock of mode A on item Q on which transaction Tj (Ti != Tj) currently holds a lock of mode B
- If transaction Ti can be granted a lock on Q immediately, in spite of the presence of the mode B lock, then we say mode A is compatible with mode B. Such a function can be represented conveniently by a matrix
- An element comp(A, B) of the matrix has the value true if and only if mode A is compatible with mode B

Lock-compatibility matrix

	S	X
S	true	false
X	false	false

- A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions
- Any number of transactions can hold shared locks on an item,
  - but if any transaction holds an exclusive on the item no other transaction may hold any lock on the item.
- If a lock cannot be granted, the requesting transaction is made to wait till all incompatible locks held by other transactions have been released. The lock is then granted.

The Two-Phase Locking Protocol

- This is a protocol which ensures conflict-serializable schedules.
- Phase 1: Growing Phase
  - transaction may obtain locks
  - transaction may not release locks
- Phase 2: Shrinking Phase
  - transaction may release locks
  - transaction may not obtain locks

Two-phase locking protocol ensures conflict serializability.

- Two-phase locking *does not* ensure freedom from deadlocks
- Cascading roll-back is possible under two-phase locking. To avoid this, follow a modified protocol called **strict two-phase locking**. Here a transaction must hold all its exclusive locks till it commits/aborts.
- 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
- **Rigorous two-phase locking** is even stricter: here *all* locks are held till commit/abort. In this protocol transactions can be serialized in the order in which they commit
- With rigorous two-phase locking, transactions can be serialized in the order in which they commit
- Most database systems implement either strict or rigorous two-phase locking

Lock Conversions

Two-phase locking with lock conversions:

– **First Phase:**

  - **can acquire a lock-S on item**
  - **can acquire a lock-X on item**
  - **can convert a lock-S to a lock-X (upgrade)**

– **Second Phase:**

  - **can release a lock-S**
  - **can release a lock-X**
  - **can convert a lock-X to a lock-S (downgrade)**

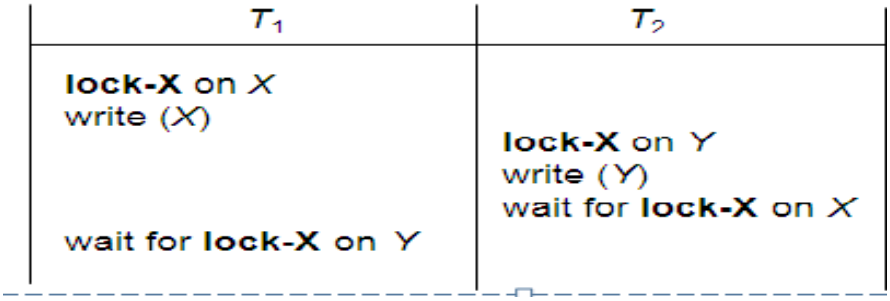
- **This protocol assures serializability. But still relies on the programmer to insert the various locking instructions.**

Deadlock Handling

- Consider the following two transactions:

T_1:	write (X)	T_2:	write(Y)
	write(Y)		write(X)

- Schedule with deadlock



- System is deadlocked if there is a set of transactions such that every transaction in the set is waiting for another transaction in the set.
- There are 2 methods to handle deadlock:
  - **Deadlock prevention** ensures that system will never enter a deadlock state.
  - **Deadlock detection** & recovery scheme allows the system to enter a deadlock state, & then try to recover by using detection & recovery scheme.

Deadlock prevention protocols ensure that the system will *never* enter into a deadlock state.

- Prevention is used if the probability that system enters a deadlock state is relatively high, otherwise detection & recovery are more efficient.
- Some prevention strategies :
  - Require that each transaction locks all its data items before it begins execution (predeclaration). i.e. either all data items are locked in one step or none are locked.
    - Hard to predict
    - Data item utilization may be very low, as locked data items may be unused for a long time
  - Impose ordering of all data items and require that a transaction can lock data items only in a sequence consistent with the ordering. (Graph based protocols)
  - A variation of the above approach is to use a total order of data items, in conjunction with two-phase locking. Once a transaction has locked a data item, it cannot request locks on items that precede that item in the ordering.
    - Easy to implement, if ordering is known
  - Another approach is to use preemption & transaction rollbacks. It uses transaction timestamps to control preemption. Two different deadlock prevention schemes using timestamps have been proposed:
    - Wait-die
    - Wound-wait

Holds lock on a data item

Waiting for T_J to release lock hold on the data item

	wait-die scheme (non-preemptive)	wound-wait scheme (preemptive)
If $TS(T_i)<TS(T_J)$	T_i waits	T_J wounded by T_i (i.e. T_J preempted or forced to rollback)
If $TS(T_i)>TS(T_J)$	T_i dies (i.e. rolled back)	T_i waits

1. **wait-die** scheme — non-preemptive
   – older transaction may wait for younger one to release data item. Younger transactions never wait for older ones; they are rolled back instead.
   – a transaction may die several times before acquiring needed data item
2. **wound-wait** scheme — preemptive
   – older transaction *wounds* (forces rollback) of younger transaction instead of waiting for it. Younger transactions may wait for older ones.
   – may be fewer rollbacks than *wait-die* scheme.

NOTE : Both in *wait-die* and in *wound-wait* schemes, a rolled back transactions is restarted with its original timestamp. **Older transactions thus have precedence over newer ones, and starvation is hence avoided.**

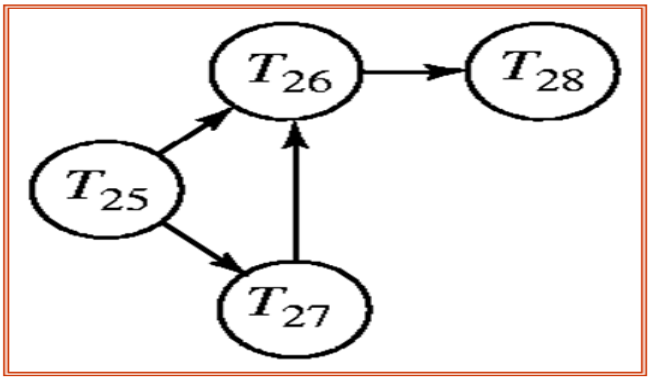
Deadlock Detection

- If a system does not employ some protocol that ensures deadlock freedom, then a detection and recovery scheme must be used
- An algorithm that examines the state of the system is invoked periodically to determine whether a deadlock has occurred
- If one has, then the system must attempt to recover from the deadlock
- To do so, the system must:

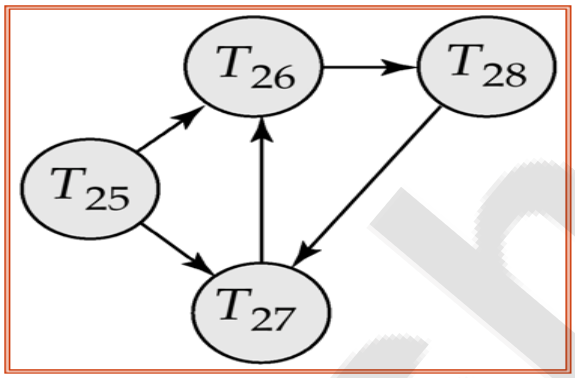
• Maintain information about the current allocation of data items to transactions, as well as any outstanding data item requests.
• Provide an algorithm that uses this information to determine whether the system has entered a deadlock state.
• Recover from the deadlock when the detection algorithm determines that a deadlock exists

- Deadlocks can be described as a *wait-for graph*, which consists of a pair $G = (V,E)$,
  - V is a set of vertices (all the transactions in the system)
  - E is a set of edges; each element is an ordered pair $T_i \rightarrow T_j$.
- If $T_i \rightarrow T_j$ is in E, then there is a directed edge from T_i to T_j, implying that T_i is waiting for T_j to release a data item.
- When T_i requests a data item currently being held by T_j, then the edge $T_i \rightarrow T_j$ is inserted in the wait-for graph. This edge is removed only when T_j is no longer holding a data item needed by T_i.
- The system is in a deadlock state if and only if the wait-for graph has a cycle.

- To detect a deadlock system must maintain wait-for graph & must invoke a deadlock-detection algorithm periodically to look for cycles in the graph.



Wait-for graph without a cycle



Wait-for graph with a cycle

Recovery and Atomicity

- To ensure atomicity despite failures, we first output information describing the modifications to stable storage without modifying the database itself.
- We study two approaches:
  - **log-based recovery**, and
  - **shadow-paging**
- We assume (initially) that transactions run serially, that is, one after the other.
- A **log** is kept on stable storage.
  - The log is a sequence of **log records**, and maintains a record of update activities on the database.
- When transaction T_i starts, it registers itself by writing a <T_i **start**>log record
- *Before* T_i executes **write**(X), a log record <T_i, X, V_1, V_2> is written, where V_1 is the value of X before the write, and V_2 is the value to be written to X.
  - Log record notes that T_i has performed a write on data item X_j X_j had value V_1 before the write, and will have value V_2 after the write.
- When T_i finishes it last statement, the log record <T_i **commi**t> is written
- *<Ti abort>. Transaction Ti has aborted*
- Whenever a transaction performs a write, it is essential that the log record for that write be created before the database is modified. Once a log record exists, we can output the modification to the database if that is desirable. Also, we have the ability to *undo a modification that has already been output to the database. We undo it by* using the old-value field in log records Two approaches using logs
  - Deferred database modification
  - Immediate database modification

Deferred Database Modification

- The **deferred database modification** scheme records all modifications to the log, but defers all the **write**s to after partial commit.
- Assume that transactions execute serially
- Transaction starts by writing <T_i ***start***> record to log.
- A **write**(X) operation results in a log record <T_i, X, V> being written, where V is the new value for X
  - Note: old value is not needed for this scheme
- The write is not performed on X at this time, but is deferred.
- When T_i partially commits, <T_i **commit**> is written to the log
- Finally, the log records are read and used to actually execute the previously deferred writes.
- Below we show the log as it appears at three instances of time.

<T_0 start>	<T_0 start>	<T_0 start>
<T_0, A, 1000, 950>	<T_0, A, 1000, 950>	<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>
	<T_0 commit>	<T_0 commit>
	<T_1 start>	<T_1 start>
	<T_1, C, 700, 600>	<T_1, C, 700, 600>
		<T_1 commit>
(a)	(b)	(c)

- redo(Ti) sets the value of all data items updated by transaction Ti to the new

values .The set of data items updated by Ti and their respective new values can be found in the log

- The redo operation must be **idempotent; that is, executing it several times must be** equivalent to executing it once
- This characteristic is required if we are to guarantee correct behavior even if a failure occurs during the recovery process
- After a failure, the recovery subsystem consults the log to determine which transactions need to be redone. Transaction *Ti needs to be redone if and only if the log* contains both the record *<Ti start> and the record <Ti commit>*
- If log on stable storage at time of crash is as in case:

  (a) No redo actions need to be taken

  (b) redo(T_0) must be performed since <T_0 **commi**t> is present

  (c) **redo**(T_0) must be performed followed by redo(T_1) since

  <T_0 **commit**> and <T_i commit> are present

Immediate Database Modification

- The immediate-modification technique allows database modifications to be output to the database while the transaction is still in the active state
- Data modifications written by active transactions are called **uncommitted modifications**
- In the event of a crash or a transaction failure, the system must use the old-value field of the log records
- Before a transaction Ti starts its execution, the system writes the record <Ti start> to the log
- During its execution, any write(X) operation by Ti is preceded by the writing of the appropriate new update record to the log
- When Ti partially commits, the system writes the record <Ti commit> to the log
- As the information in the log is used in reconstructing the state of the database, we cannot allow the actual update to the database to take place before the corresponding log record is written out to stable storage
- So it is required that, before execution of an output(B) operation, the log records corresponding to B be written onto stable storage
- Recovery procedure has two operations instead of one:
  - **Undo** (T_i) restores the value of all data items updated by T_i to their old values, going backwards from the last log record for T_i
  - **Redo** (T_i) sets the value of all data items updated by T_i to the new values, going forward from the first log record for T_i
- Both operations must be **idempotent**
  - That is, even if the operation is executed multiple times the effect is the same as if it is executed once
    - Needed since operations may get re-executed during recovery
- When recovering after failure:
  - Transaction T_i needs to be undone if the log contains the record $<T_i$ **start**$>$, but does not contain the record $<T_i$ **commit**$>$.
  - Transaction T_i needs to be redone if the log contains both the record $<T_i$ **start**$>$ and the record $<T_i$ **commit**$>$.
- Undo operations are performed first, then redo operations.