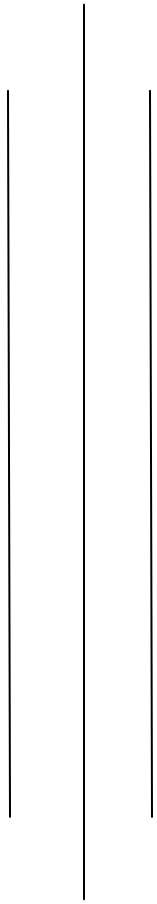
A

Course Manual

On

Database Management System



Prepared By: Bal Krishna Nyaupane
balkrishnanyaupane@gmail.com

Sagarmatha Engineering College

Sanepa, Lalitpur,

G.P.O Box: 19910

01-5527274

Chapter -1 Introduction to DBMS

1.1 Introduction to DBMS

- ✓ A database is a collection of related data necessary to manage an organization. It includes transient data such as input documents, reports and intermediate results obtained during processing.
- ✓ Database: collection of logically interrelated data and description of this data, designed to meet the information needs for organization.
- ✓ Database System: (Database + DBMS Software) It is an integrated collections of related files along with the detail about their definition, interpretation, manipulation and maintenance
- ✓ A DBMS is a set of procedures that manage the database and provide the access to the database in a form required by any application program. It effectively ensures that necessary data in the desired form is available for diverse applications of different organizations.

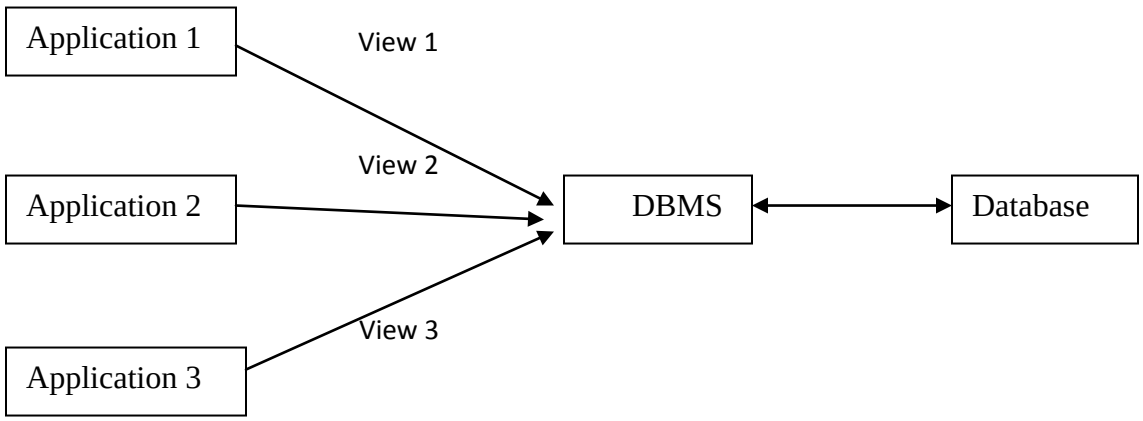


Fig: - Distinction between a DBMS and database

1.2 History of Database Systems

- ❖ 1950s and early 1960s:
  - ✓ Data processing using magnetic tapes for storage
    - ➢ Tapes provided only sequential access
  - ✓ Punched cards for input
- ❖ Late 1960s and 1970s:
  - ✓ Hard disks allowed direct access to data
  - ✓ Network and hierarchical data models in widespread use
  - ✓ Ted Codd defines the relational data model
    - ➢ Would win the ACM Turing Award for this work

- IBM Research begins System R prototype
- UC Berkeley begins Ingres prototype

✓ High-performance (for the era) transaction processing

❖ 1980s:

✓ Research relational prototypes evolve into commercial systems

- SQL becomes industrial standard

✓ Parallel and distributed database systems

✓ Object-oriented database systems

❖ 1990s:

✓ Large decision support and data-mining applications

✓ Large multi-terabyte data warehouses

✓ Emergence of Web commerce

❖ Early 2000s:

✓ XML and XQuery standards

✓ Automated database administration

❖ Later 2000s:

✓ Giant data storage systems

- Google BigTable, Yahoo PNuts, Amazon,

1.3 Objectives

- ✓ A database should provide for efficient storage, update, and retrieval of data.
- ✓ A database should be reliable - the stored data should have high integrity and promote user trust in that data.
- ✓ A database should be adaptable and scalable to new and unforeseen requirements and applications.
- ✓ A database should identify the existence of common data and avoid duplicate recording. Selective redundancy is sometimes allowed to improve performance or for better reliability.

1.4 Characteristics of a Database

- ✓ Structure
  - data types
  - data behavior
- ✓ Persistence
  - store data on secondary storage
- ✓ Retrieval
  - a declarative query language
  - a procedural database programming language

- ✓ Performance
  - retrieve and store data quickly Correctness
- ✓ Sharing
  - concurrency
- ✓ Reliability and resilience
- ✓ Large volumes
- ✓ **Characteristics of Database Approach**
  - Self –describing nature of a database system
  - Insulation between programs ,data and data abstraction
  - Support of multiple views of data
  - Sharing of data and Multi-user Transaction processing
- ✓ **Data in the database has some characteristics**
  - **Shared**: Data in a database is shared among different users and application.
  - **Persistence**: Data in the database exists permanently.
  - **Integrity:** Data should be correct with respect to the real world entity that they represent.
  - **Security:** Data need to be protected from unauthorized access
  - **Consistency:** Data must be consistent to the data type that stores it
- ✓ **Database Management System Approach**
  - Controlled redundancy
    - consistency of data & integrity constraints
  - Integration of data
    - self-contained & represents semantics of application
  - Data and operation sharing
    - multiple interfaces
  - Services & Controls
    - security & privacy controls
    - backup & recovery
    - enforcement of standards
  - Flexibility
    - data independence
    - data accessibility
    - reduced program maintenance
  - Ease of application development

1.5 Database Applications

- ✓ Banking: transactions
- ✓ Airlines: reservations, schedules
- ✓ Universities: registration, grades
- ✓ Sales: customers, products, purchases
- ✓ Online retailers: order tracking, customized recommendations
- ✓ Manufacturing: production, inventory, orders, supply chain
- ✓ Human resources: employee records, salaries, tax deductions

1.6 Database System Vs File System

- ✓ Data redundancy and inconsistency
  - ➢ Multiple file formats, duplication of information in different files
- ✓ Difficulty in accessing data
  - ➢ Need to write a new program to carry out each new task
- ✓ Data isolation
  - ➢ multiple files and formats
- ✓ Integrity problems
  - ➢ Integrity constraints (e.g., account balance > 0) become "buried" in program code rather than being stated explicitly
  - ➢ Hard to add new constraints or change existing ones
- ✓ Atomicity problems
  - ➢ 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 anomalies
  - ➢ Concurrent access needed for performance
  - ➢ Uncontrolled concurrent accesses can lead to inconsistencies
  - ➢ Example: Two people reading a balance (say 100) and updating it by withdrawing money (say 50 each) at the same time
- ✓ Security problems
  - ➢ Hard to provide user access to some, but not all, data

Note: Database systems offer solutions to the entire above problem

1.7 Levels of Abstraction

- ✓ A major purpose of database system is to provide users with an abstract view of data.

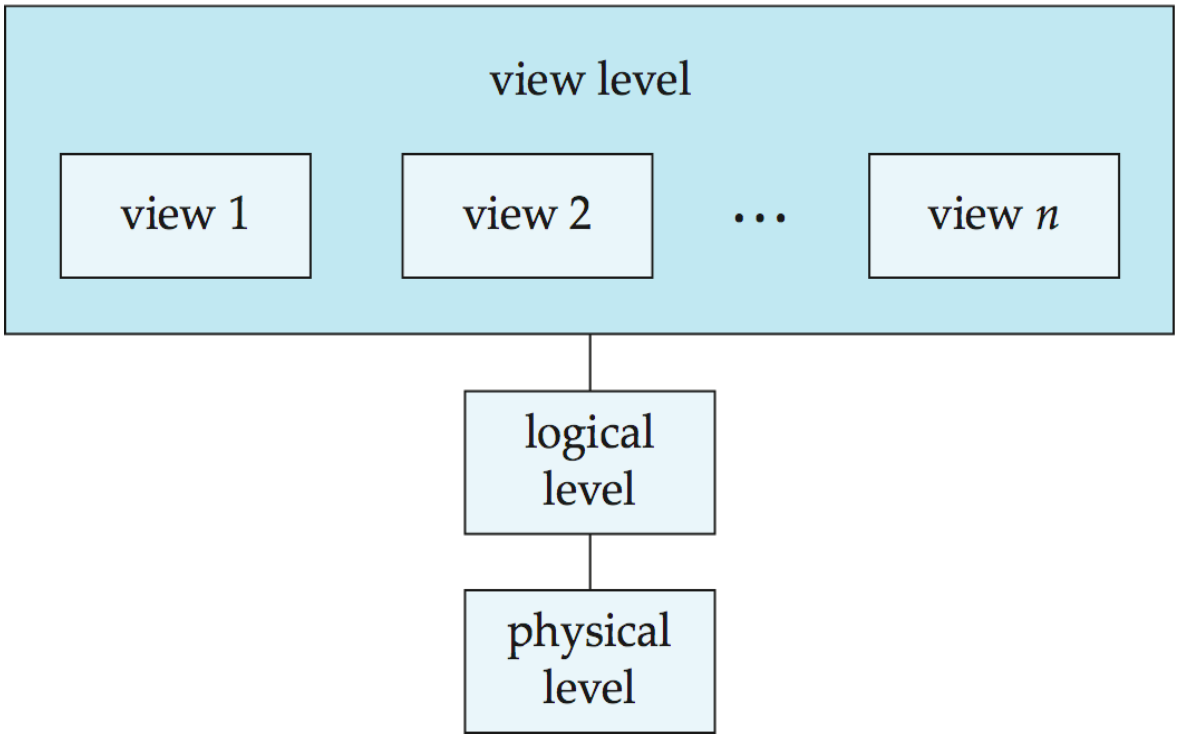


Fig: Three level of data abstraction

- ✓ Physical level:
  - Lowest level of abstraction.
  - Describes how the data are actually stored

- ✓ Logical Level
  - Describes what data stored in database, and what relationships exist among the data.
  - Describes the entire database in terms of small number of relatively simple structure.

```
type instructor = record
        ID : string;
        name : string;
        dept_name : string;
        salary : integer;
end;
```

- ✓ View Level:
  - Application programs hide details of data types. Views can also hide information (such as an employee's salary) for security purposes.

- ✓ **Schema and Instance**
  - Schema:
    - ➢ The logical structure of the database
    - ➢ Example: The database consists of information about a set of customers and accounts and the relationship between them
    - ➢ Analogous to type information of a variable in a program
    - ➢ Physical schema: database design at the physical level
    - ➢ Logical schema: database design at the logical level
  - Instance:
    - ➢ the actual content of the database at a particular point in time
    - ➢ Analogous to the value of a variable

1.8 Data Independence

- ✓ Data independence which means that upper levels are unaffected by changes in lower levels.
- ✓ In general, the interfaces between the various levels and components should be well defined so that changes in some parts do not seriously influence others.
- ✓ When a schema at a lower level is changed, only the mappings between this schema and higher-lever schemas need to be changed in a DBMS that fully supports data independence.
- ✓ The higher-level schemas themselves are unchanged. Hence, the application programs need not be changed since they refer to the external schemas.
- ✓ Two types of independence
  1. Physical Data Independence
  2. Logical Data Independence
- ✓ **Physical Data Independence**
  - It indicates that the physical storages structures or devices could be changed without affecting the conceptual schema.
  - The ability to modify the physical schema without changing the logical schema.
  - Applications depend on the logical schema
- ✓ **Logical Data Independence**
  - The capacity to change the conceptual schema without having to change the external schemas and their application programs.
  - The conceptual schema can be changed without affecting the existing external schemas.

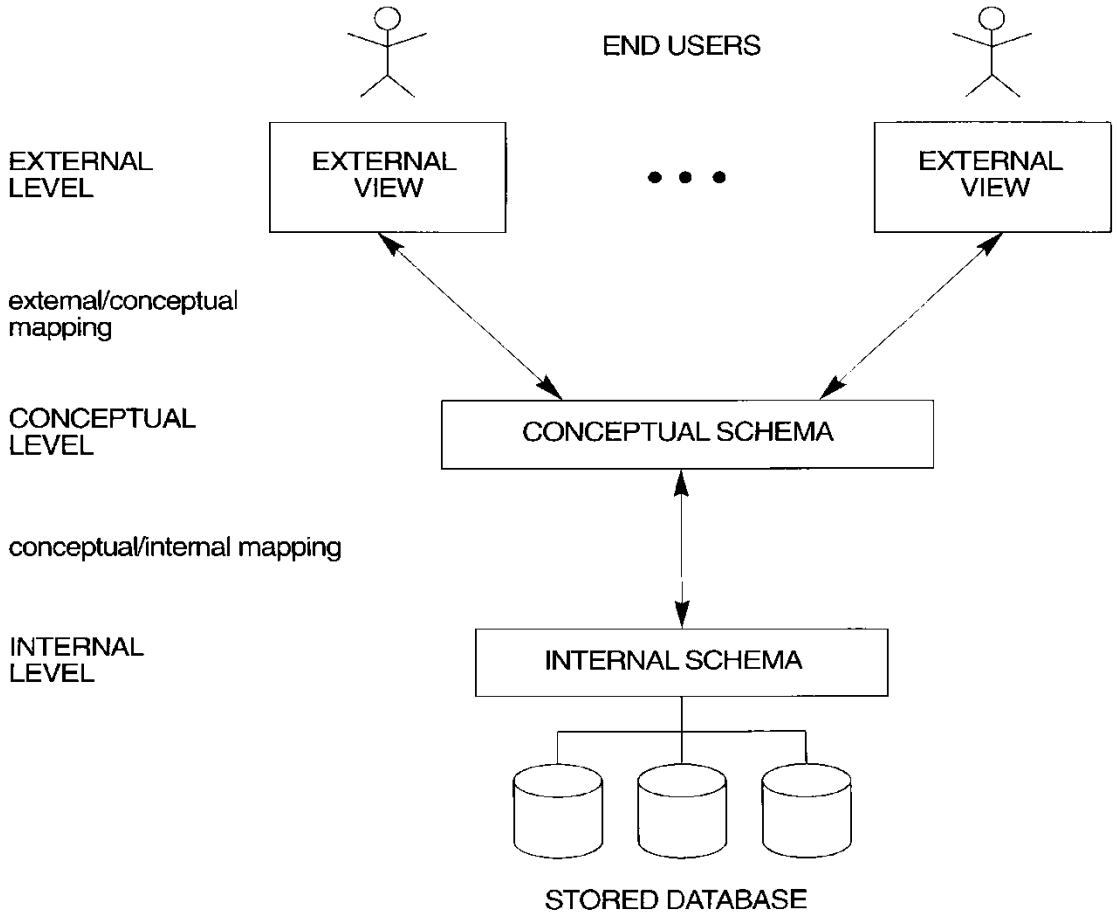
1.9 Concepts of DDL, DML, DCL

- ✓ **Data-Definition language (DDL)**
  - Specification notation for defining the database schema
  - It is used to create and modify the structure of database objects in database.
    - ♦ CREATE – Creates objects in the database
    - ♦ ALTER – Alters objects of the database
    - ♦ DROP – Deletes objects of the database
  - DDL compiler generates a set of tables stored in a data dictionary Special set of tables called data dictionary.
  - Data dictionary contains metadata (i.e., data about data that describes the object in the database)
  - Database schema
  - Integrity constraints
    - Primary key (ID uniquely identifies instructors)
    - Referential integrity (references constraint in SQL)

- ✓ **Data-manipulation Languages (DML)**
  - It is used to retrieve, store, modify, delete, insert and update data in database.
    - ♦ SELECT–Retrieves data from a table
    - ♦ INSERT - Inserts data into a table
    - ♦ UPDATE – Updates existing data into a table
    - ♦ DELETE – Deletes all records from a table
  - DML also known as query language
  - Two classes of languages
  - Procedural – user specifies what data is required and how to get those data
  - Declarative (nonprocedural) – user specifies what data is required without specifying how to get those data
- ✓ **Data Control Language (DCL)**
  - It is used to create roles, permissions, and referential integrity as well it is used to control access to database by securing it.
    - ♦ GRANT – Gives user's access privileges to database
    - ♦ REVOKE – Withdraws user's access privileges to database given with the GRANT command
- ✓ **Transactional Control Language (TCL**
  - It is used to manage different transactions occurring within a database.
    - ♦ COMMIT – Saves work done in transactions
    - ♦ ROLLBACK – Restores database to original state since the last COMMIT command in transactions
      SAVE TRANSACTION – Sets a save point within a transaction

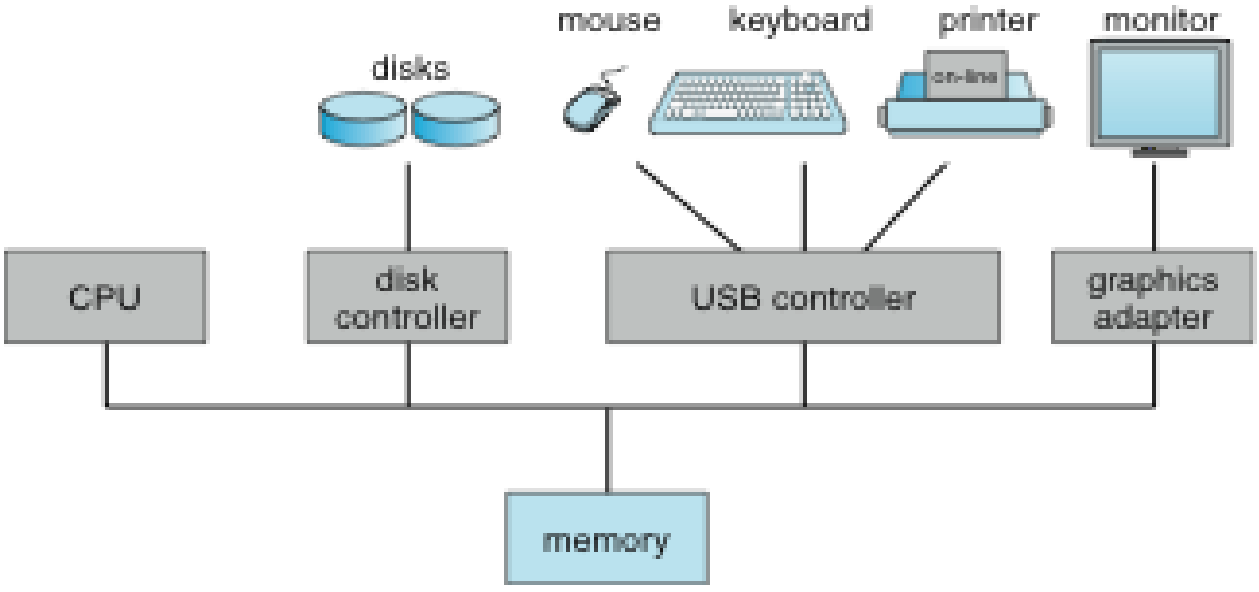
1.10 DBMS Three-Schema Architecture

- ✓ Proposed to support DBMS characteristics of
  - Insulation of programs and data/program and operations (program-data and program-operation independence)
  - Support of multiple views of the data.
  - Use of catalog (database description)
- ✓ **Three-Schema Architecture**
  - **Internal schema** at the internal level to describe data storage structures and access paths. Typically uses a physical data model.
  - **Conceptual schema** at the conceptual level to describe the structure and constraints for the whole database. Uses a conceptual or an implementation data model.
  - **External schema** at the external level to describe the various user views. Usually uses the same data model as the conceptual level or high-level data model

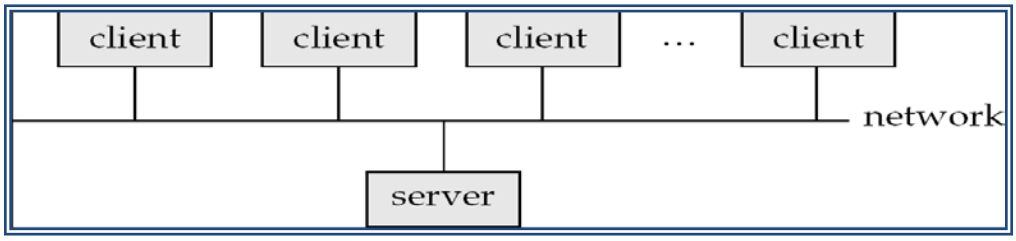


1.11 Classification of DBMSs

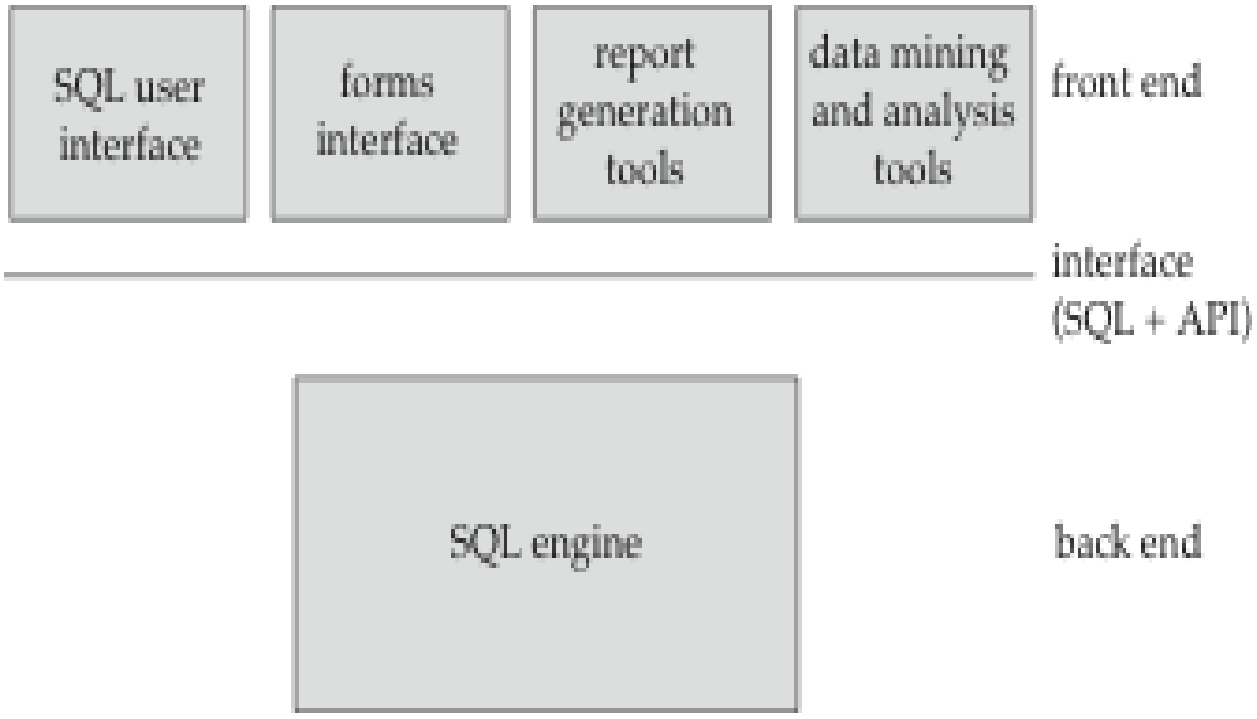
- ✓ Based on the data model used:
  - Traditional: Relational, Network, Hierarchical.
  - Emerging: Object-oriented, Object-relational.
- ✓ Other classifications:
  - Single-user (typically used with micro- computers) vs. multi-user (most DBMSs).
  - Centralized (uses a single computer with one database) vs. distributed (uses multiple computers, multiple databases)
- ✓ Single-user system
  - Single-user system (e.g., personal computer or workstation): desk-top unit, single user, usually has only one CPU and one or two hard disks; the OS may support only one user.
- ✓ Multi-user system
  - Multi-user system: more disks, more memory, multiple CPUs, and a multi-user OS. Serve a large number of users who are connected to the system vie terminals. Often called server systems.
- ✓ Centralized Systems
  - Combines everything into single system including- DBMS software, hardware, application programs and user interface processing software
  - Run on a single computer system and do not interact with other computer systems.
  - General-purpose computer system: one to a few CPUs and a number of device controllers that are connected through a common bus that provide access to shared memory.



- ✓ **Client-Server DBMS**
  - Server systems satisfy requests generated at *m* client systems, whose general structure is shown below:



- ✓ Database functionality can be divided into:
  - **Back-end**: manages access structures, query evaluation and optimization, concurrency control and recovery.
  - **Front-end**: consists of tools such as forms, report-writers, and graphical user interface facilities.
- ✓ The interface between the front-end and the back-end is through SQL or through an application program interface.



- ✓ **Two Tier Client-Server Architecture**
  - User Interface Programs and Application Programs run on the client side
  - Interface called ODBC (Open Database Connectivity) provides an Application program interface (API) allow client side programs to call the DBMS. Most DBMS vendors provide ODBC drivers.
  - A client program may connect to several DBMSs.
  - Other variations of clients are possible: e.g., in some DBMSs, more functionality is transferred to clients including data dictionary functions, optimization and recovery across multiple servers, etc. In such situations the server may be called the Data Server.
- ✓ **Three Tier Client-Server Architecture**
  - Common for Web applications
  - Intermediate Layer called Application Server or Web Server:
    - ♦ stores the web connectivity software and the rules and business logic (constraints) part of the application used to access the right amount of data from the database server
    - ♦ Acts like a conduit for sending partially processed data between the database server and the client.
  - Additional Features- Security:
    - ♦ encrypt the data at the server before transmission
    - ♦ decrypt data at the client

Chapter -2 Data models

2.1 Introduction to Data Model

- ✓ A set of concepts to describe the structure of a database, and certain constraints that the database should obey.
- ✓ **Categories of Data Models:**
  - **Conceptual** (**high-level**, **semantic**) data models: Provide concepts that are close to the way many users perceive data. (Also called entity-based or object-based data models.)
    - ♦ entity
    - ♦ attribute
    - ♦ relationship
  - **Physical** (**low-level**, **internal**) data models: Provide concepts that describe details of how data is stored in the computer.
    - ♦ record formats
    - ♦ record ordering
    - ♦ access paths
  - **Implementation** (**representational**) data models: Provide concepts that fall between the above two, balancing user views with some computer storage details.
    - ♦ relational
    - ♦ network
    - ♦ hierarchical

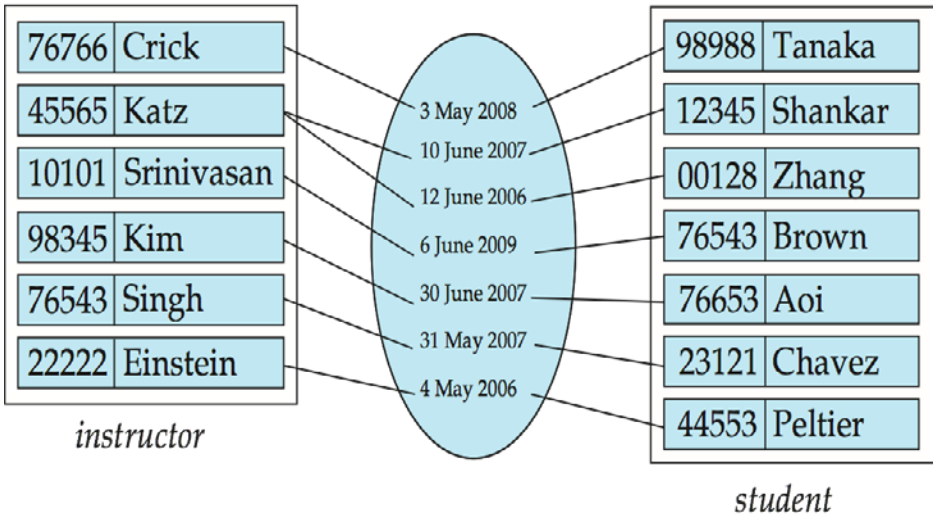
2.2 E-R Model

- ✓ A database can be modeled as:
  - a collection of entities,
  - Relationship among entities.
- ✓ **Entity:**
  - An entity is an object that exists and is distinguishable from other objects.
  - Real-world object distinguishable from other objects.
  - Example: specific person, company, event, plant
- ✓ **Entity set :**
  - An entity set is a set of entities of the same type that share the same properties.
  - All entities in an entity set have the same set of attributes.
  - Each entity set has a key.
  - Each attribute has a domain
  - Example: set of all Departments, Professors, Students, Administrators
- ✓ **Relationship**
  - The connections among two or more entity Sets
  - A relationship is an association among several entities

- Example:
  - Ram works in Pharmacy department.
  - Students and Professors are under a certain department
  - Admin manage the campus/ departments

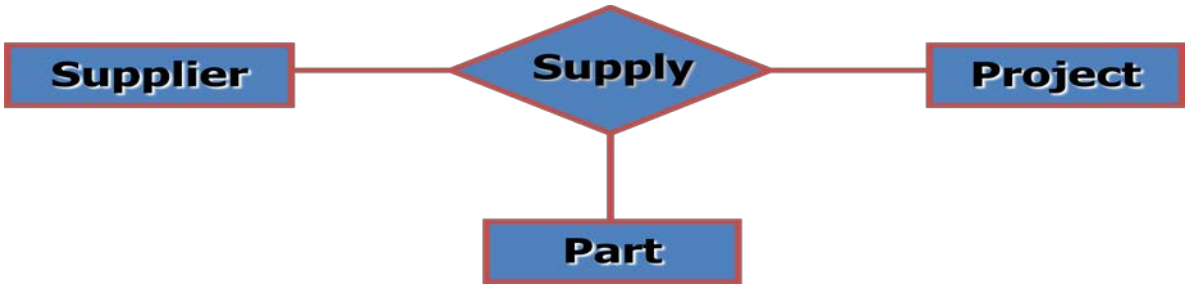
✓ **Relationship set**

- A relationship set is a mathematical relation among $n \geq 2$ entities, each taken from entity sets $\{(e_1, e_2, \ldots e_n) \mid e_1 \in E_1, e_2 \in E_2, \ldots, e_n \in E_n\}$ where $(e_1, e_2, \ldots, e_n)$ is a relationship
- An attribute can also be property of a relationship set.
- A relationship may also have attributes called **descriptive attributes**.
- For instance, the advisor relationship set between entity sets instructor and student may have the attribute date which tracks when the student started being associated with the advisor



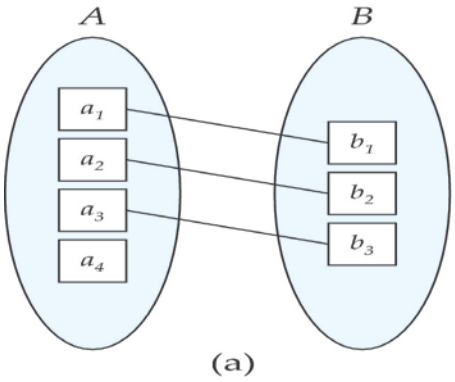
✓ **Degree of a Relationship Set**

- The degree of a relationship type is the number of participating entity
- Binary Relationship :
  - Involve two entity sets (or degree two).
  - Most relationship sets in a database system are binary.
- Ternary Relationship: involves 3 entities
- N-array Relationship: involves n entities
- Example: students work on research projects under the guidance of an instructor is a binary relationship.
- Relationship supply is a ternary relationship between supplier, part and project

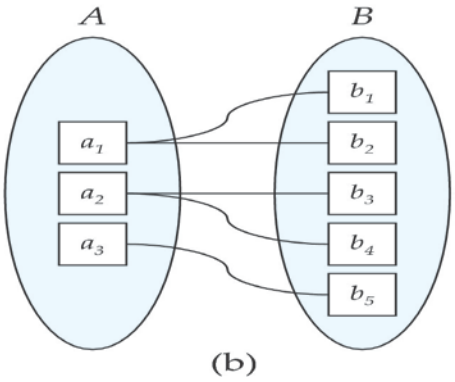


2.2.1 Kinds of Constraints

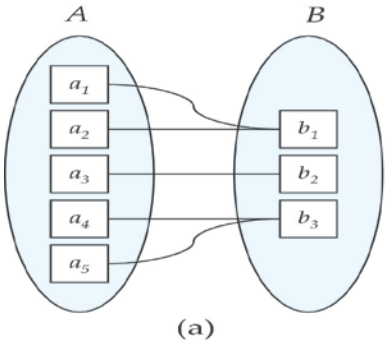
- ✓ What kind of constraints can be defined in the ER Model?
  - Cardinality Constraints
  - Participation Constraints
- ✓ Together called "Structural Constraints"
- ✓ **Cardinality Constraints**
  - Also called mapping Cardinality Constraints
  - Express the number of entities to which another entity can be associated via a relationship set.
  - Most useful in describing binary relationship sets.
  - For a binary relationship set the mapping cardinality must be one of the following types:
    - ♦ **One to one:** An entity in A is associated with at most one entity in B, and an entity in B is associated with at most one entity in A. Example: One department can have only one manager
    - ♦ **One to many:** An entity in A is associated with any number (zero or more) of entity in B .An entity in B, however, can be associated with at most one entity in A. Example: One department can have Many employees
    - ♦ **Many to one:** An entity in A is associated with at most one entity in B .An entity in B, however, can be associated with any number (zero or more) of entity in A. Example: Many employees works in One department
    - ♦ **Many to many:** An entity in A is associated with any number (zero or more) of entities in B and an entity in B is associated with any number (zero or more) of entities in A. Example: Many employees works in many department



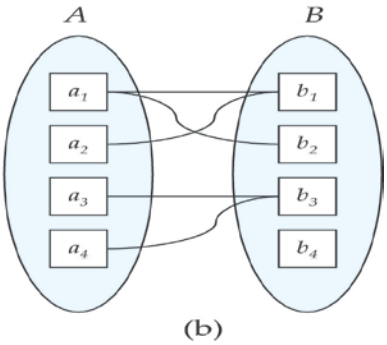
One to One



One to Many

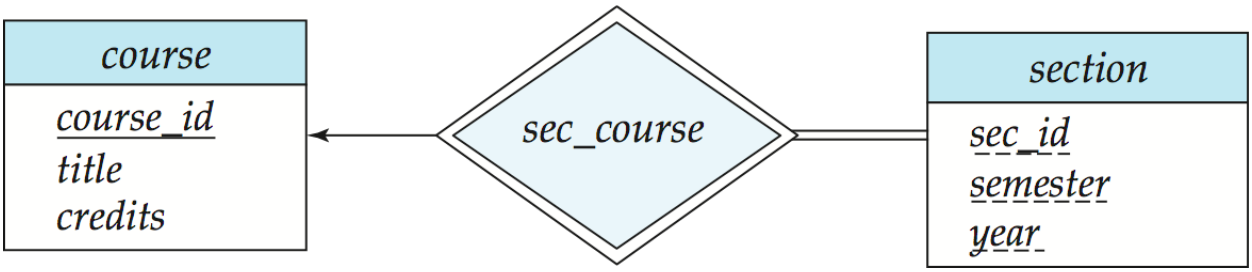


Many to One



Many to Many

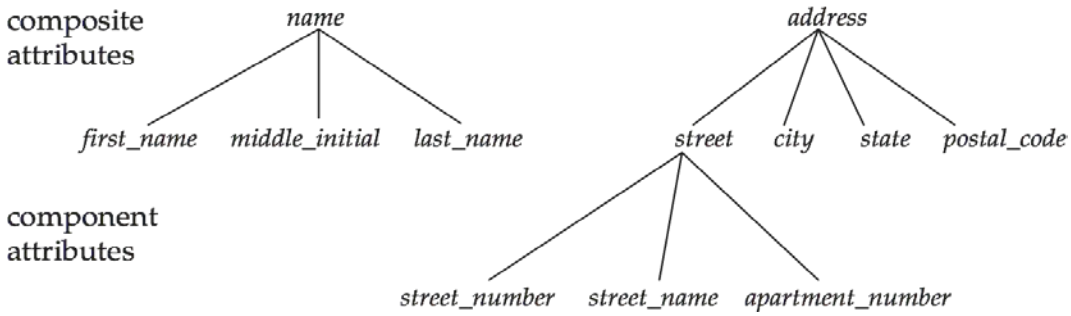
- ✓ **Note: Some elements in A and B may not be mapped to any elements in the other set**
- ✓ **Participation Constraints**
  - **Total participation** (indicated by double line): every entity in the entity set participates in at least one relationship in the relationship set
    - ♦ Example: participation of section in sec_course is total
    - ♦ every section must have an associated course
  - **Partial participation**: some entities may not participate in any relationship in the relationship set
    - ♦ Example: participation of instructor in advisor is partial



2.2.2**Attributes:**

- ✓ The properties of the entities in the set.
- ✓ Descriptive properties possessed by all members of an entity set.
- ✓ **Domain** – the set of permitted values for each attribute
- ✓ Example: Instructor = (ID, name, street, city, salary)
  course= (course_id, title, credits)
- ✓ **Type of Attribute :**

1. **Simple and composite attributes**
   - A simple (atomic) attribute is one that cannot be broken down into smaller components
   - Composite attributes can be divided into smaller parts which represent simple attributes with independent meaning
   - Composite attributes can be divided into subparts



2. **Single-valued and multivalued attributes**
   - A multivalued attribute is one that may have more than one value for a given instance.
   - Example: a person may have more than mobile number; employee may have more than one Skill.
3. **Derived attributes**
   - A derived attribute is one whose value can be calculated from related attribute values
   - Can be computed from other attributes
   - Example: age, given date_of_birth
4. **Descriptive attributes**
   - A relationship may also have attributes called descriptive attributes.

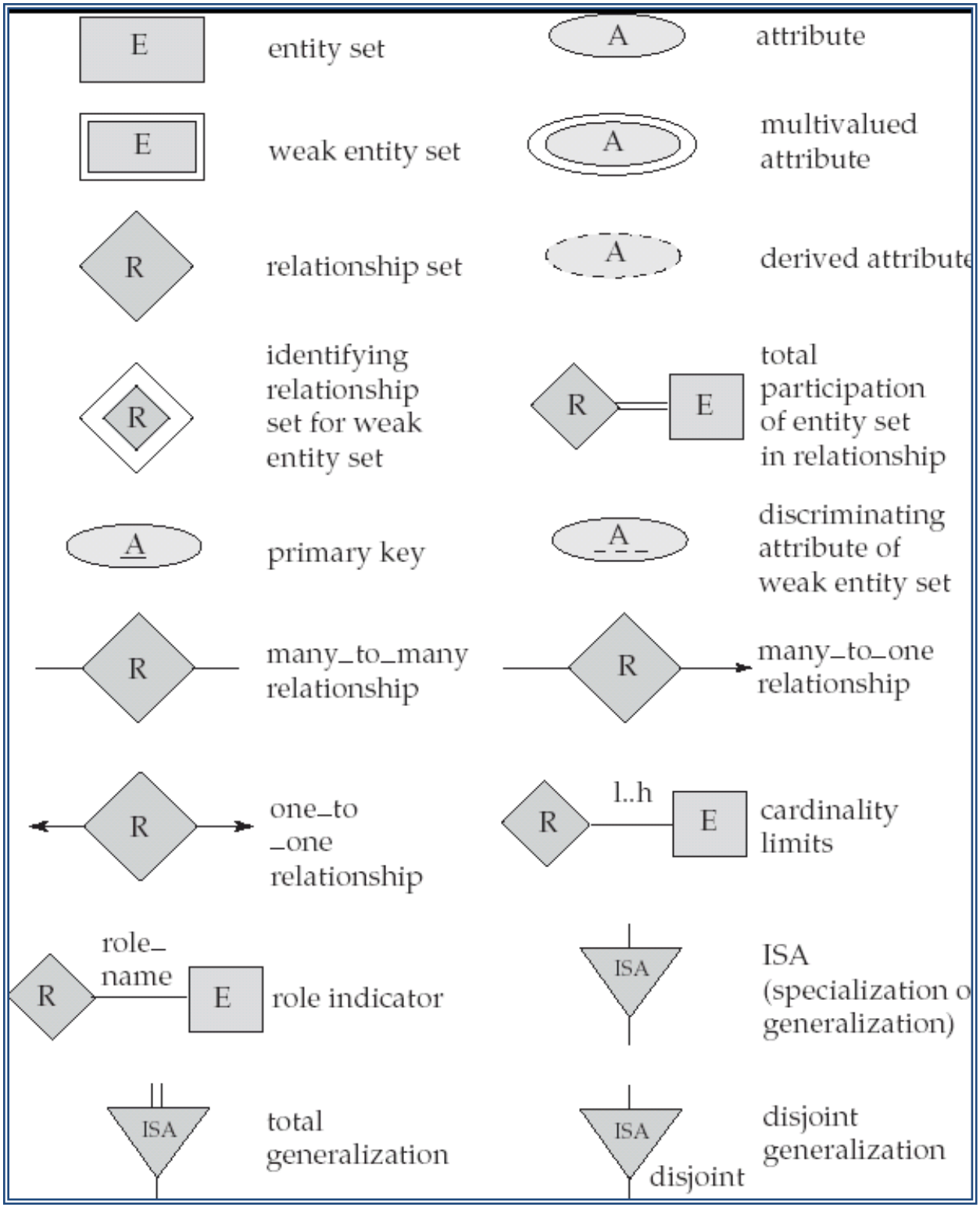
2.2.3 Keys:

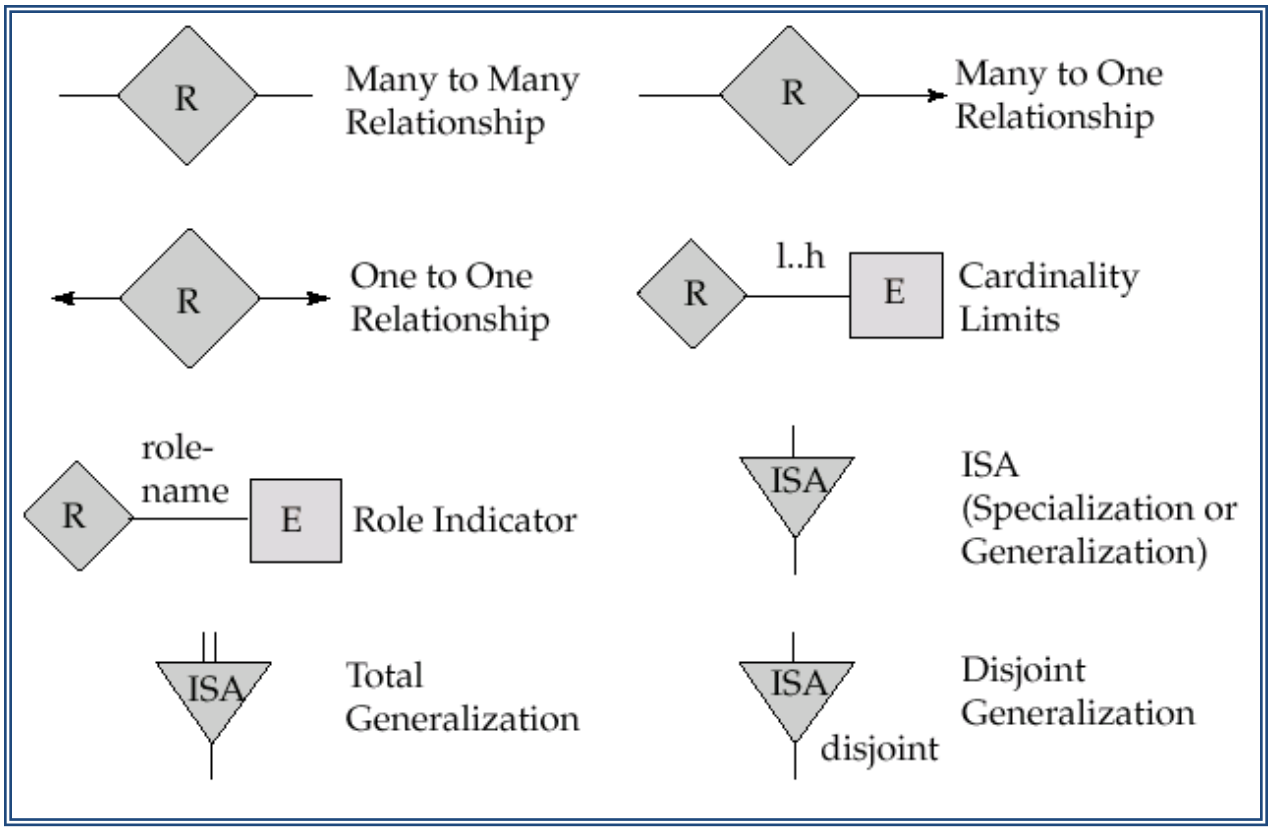
- ✓ A group of one or more attributes that uniquely identify an entity in the entity set
- ✓ **Types of Keys**
    1. **Super Key:**
        - a set of attributes that allows to identify and entity uniquely in the entity set
    2. **Candidate Key:**
        - Is a minimal super key that uniquely identifies either an entity or a relationship
    3. **Primary Key:**
        - Is a candidate key that is chosen by the database designer to identify the entities of an entity set

2.2.4 Strong and Weak Entity Sets

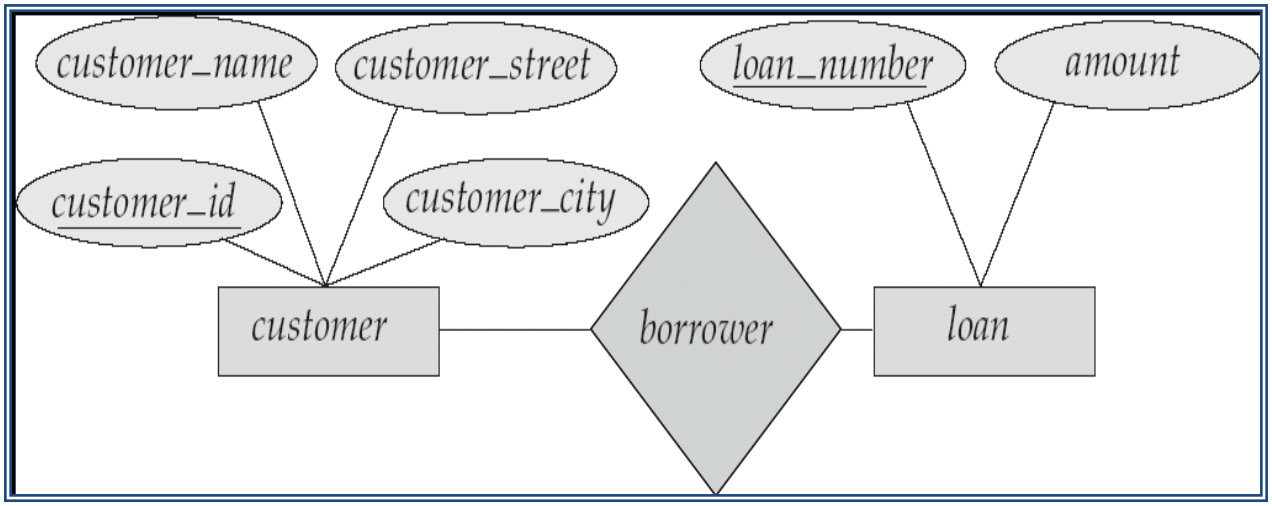
- ✓ **Strong entity set:**
    - One that exists independently from other entity types
    - Always have a unique characteristic (identifier) – an attribute or combination of attributes that uniquely distinguish each occurrence of that identity
- ✓ **Weak Entity Sets:**
    - An entity set that does 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
    - 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.
    - participate fully in the identifying relationship

2.3 Symbols Used in E-R Notation





Examples on ER-Diagram



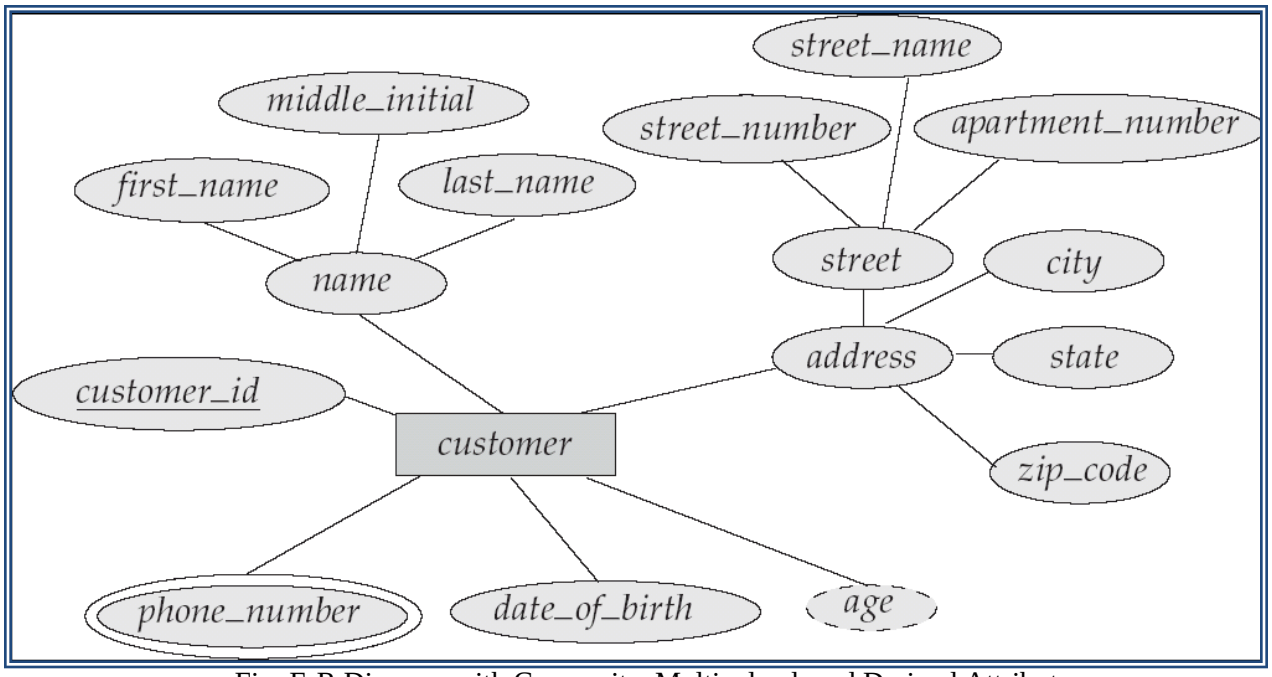
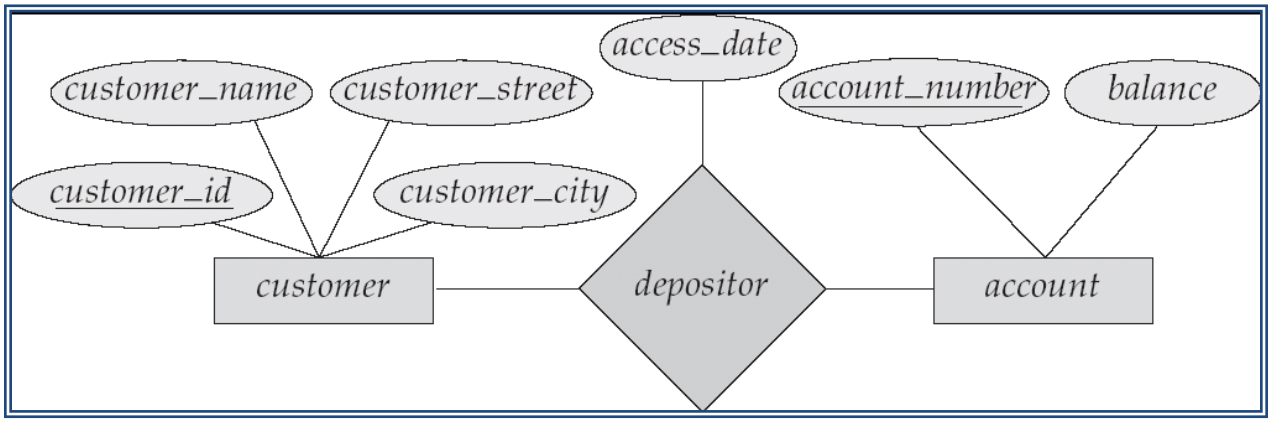


Fig: E-R Diagram with Composite, Multivalued, and Derived Attributes



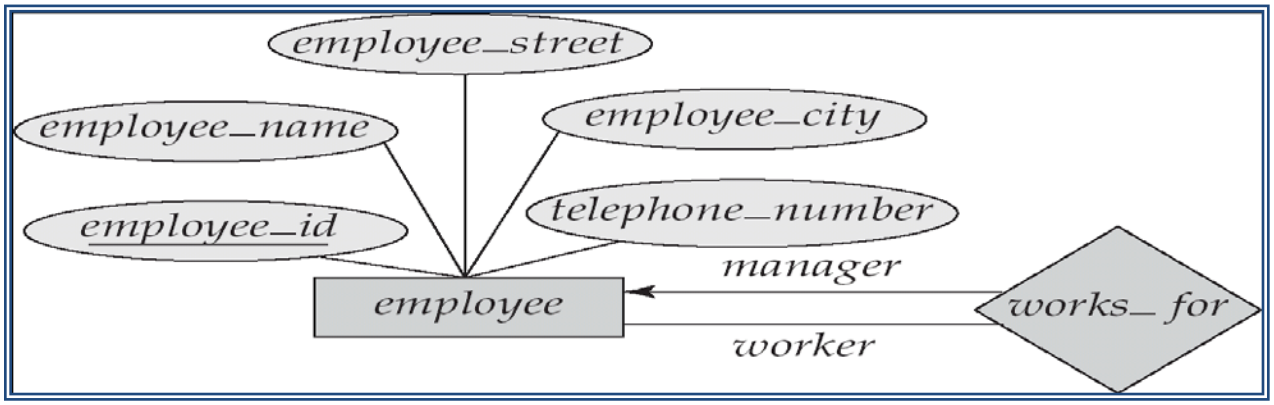


Fig: The labels “manager” and “worker” are called roles; they specify how employee entities interact via the works for relationship set. Roles are indicated in E-R diagrams by labeling the lines that connect diamonds to rectangles.

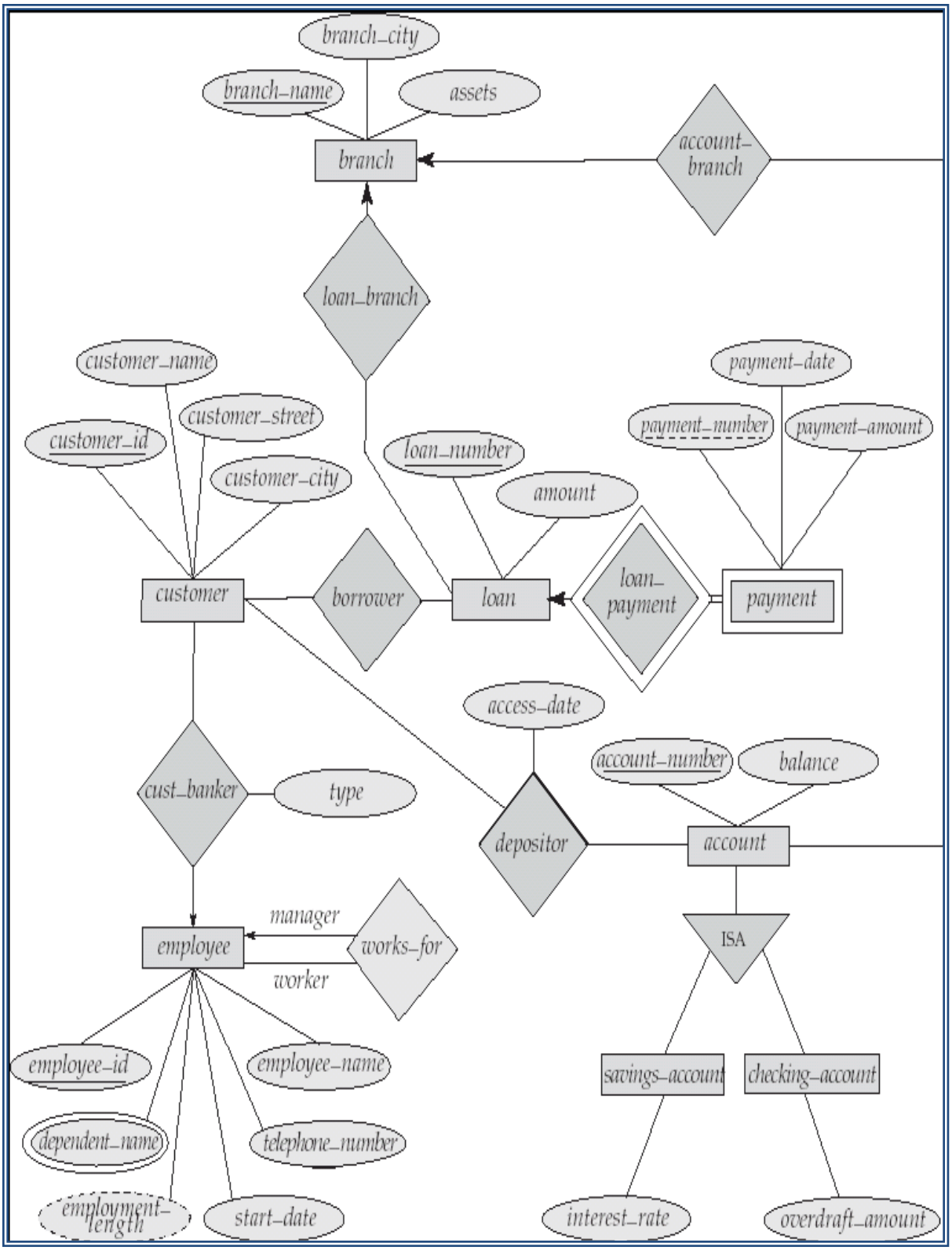
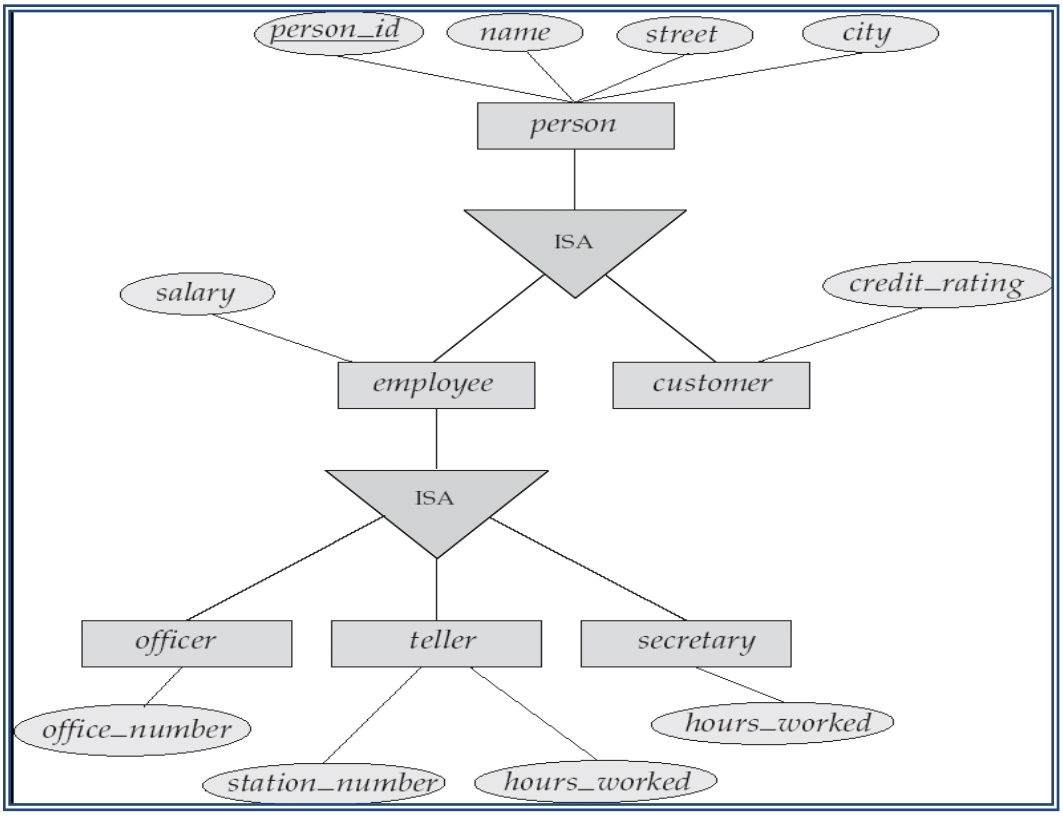


Fig: E-R Diagram for a Banking Enterprise

2.5 Extended ER Features: Specialization and Generalization

- ✓ **Specialization**
  - The process of designating sub grouping within an entity set is called specialization.
  - Top-down design process; designate sub groupings within an entity set that are distinctive from other entities in the set.
  - These sub groupings become lower-level entity sets that have attributes or participate in relationships that do not apply to the higher-level entity set.
  - Depicted by a triangle component labeled ISA (E.g., instructor "is a" person).
  - **Attribute inheritance** – a lower-level entity set inherits all the attributes and relationship participation of the higher-level entity set to which it is linked.
- ✓ **Generalization**
  - A bottom-up design process – combine a number of entity sets that share the same features into a higher-level entity set
  - Generalization is a containment relationship that exists between a higher level entity set and one or lower level entity sets.
  - Specialization and generalization are simple inversions of each other; they are represented in an E-R diagram in the same way
  - The ISA relationship also referred to as **superclass - subclass** relationship



2.5.1 Design Constraints on a Specialization/Generalization

- ✓ Constraint on which entities can be members of a given lower-level entity set.

1. Condition-defined

- If we can determine exactly those entities that will become members of each subclass by a condition, the subclasses are called predicate-defined (or condition-defined) subclasses
  - ♦ Condition is a constraint that determines subclass members
  - ♦ Display a predicate-defined subclass by writing the predicate condition next to the line attaching the subclass to its super class
  - ♦ Example: all customers over 65 years are members of senior-citizen entity set; senior-citizen ISA person.

2. user-defined

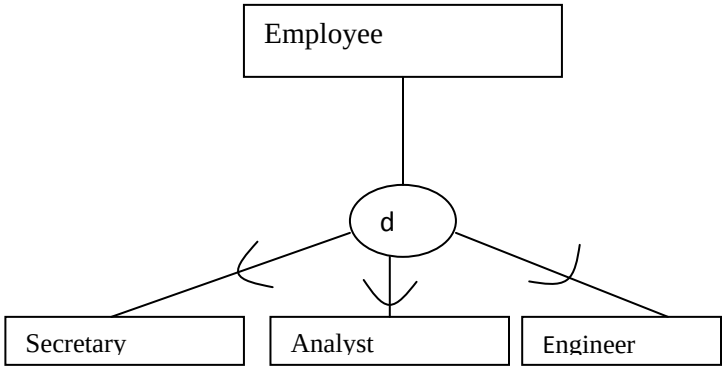
- If no condition determines membership, the subclass is called user-defined
  - ♦ Membership in a subclass is determined by the database users by applying an operation to add an entity to the subclass
  - ♦ Membership in the subclass is specified individually for each entity in the super class by the user
- Constraint on whether or not entities may belong to more than one lower-level entity set within a single generalization.

1. **Disjoint**
   - an entity can belong to only one lower-level entity set
   - Noted in E-R diagram by having multiple lower-level entity sets link to the same triangle
   - An account entity can satisfy only one condition for the account type attribute ; an entity can be saving or checking ;but cannot be both
   - The d in the specialization circle stands for disjoint.

2. **overlap**
   - That is the same entity may be a member of more than one subclass of the specialization
   - Overlap constraint is shown by placing an o in the specialization circle.
   - Example: Person is a employee and customer. Employee can be customer.

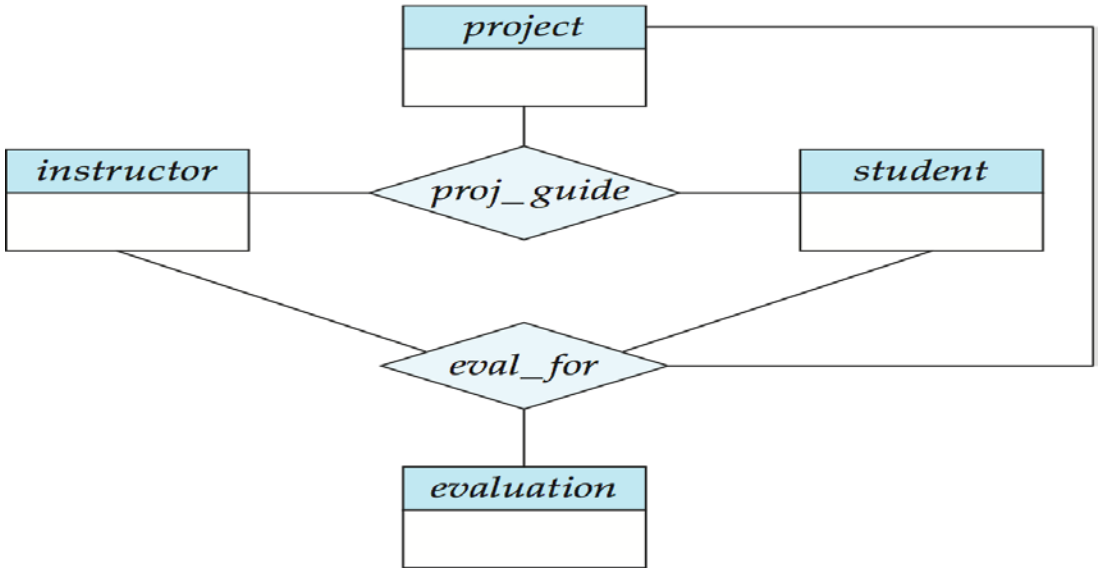
3. **Completeness constraint**
   - The completeness constraint may be either total or partial.
   - A total specialization constraint specifies that every entity in the super class must be a member of at least one subclass of the specialization.
   - Total specialization is shown by using a double line to connect the super class to the circle.
   - A single line is used to display a partial specialization, meaning that an entity does not have to belong to any of the subclasses.

Disjoint, partial



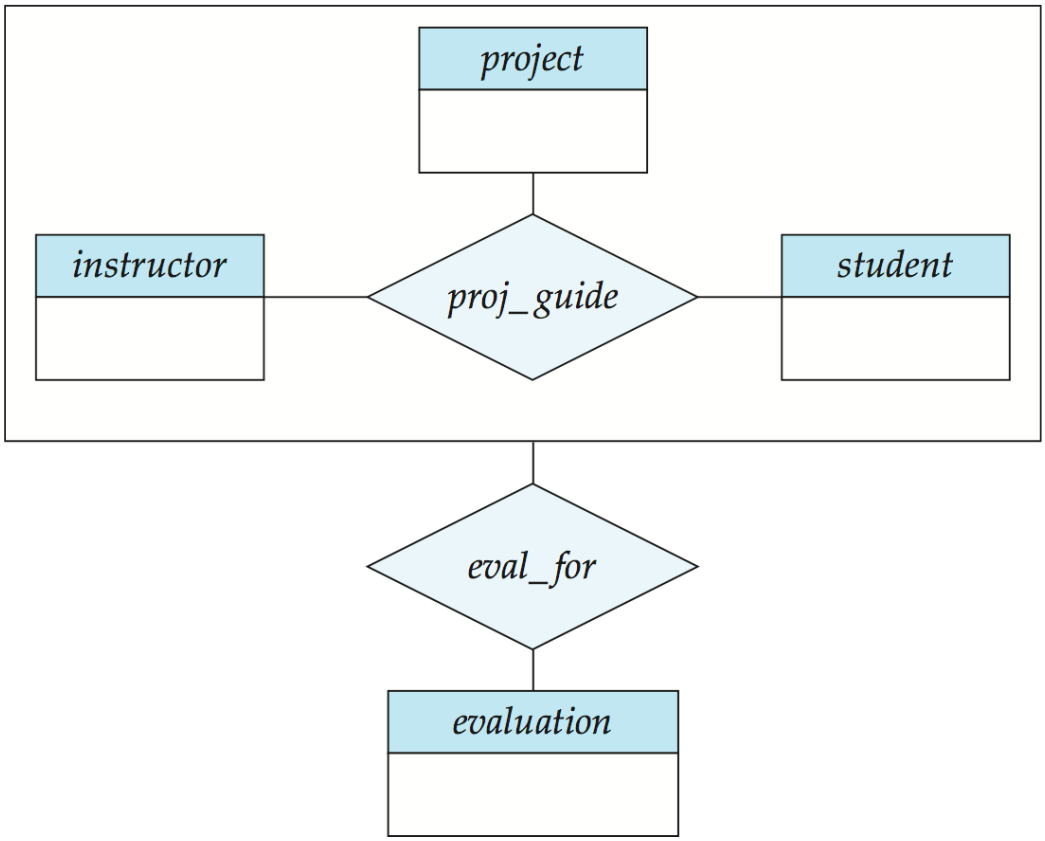
2.6 Aggregation

- ✓ Consider the ternary relationship proj_guide, which we saw earlier
- ✓ Suppose we want to record evaluations of a student by a guide on a project



- ✓ Relationship sets eval_for and proj_guide represent overlapping information
  - Every eval_for relationship corresponds to a proj_guide relationship
  - However, some proj_guide relationships may not correspond to any eval_for relationships .So we can't discard the proj_guide relationship
- ✓ Eliminate this redundancy via aggregation
  - Treat relationship as an abstract entity
  - Allows relationships between relationships
  - Abstraction of relationship into new entity

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



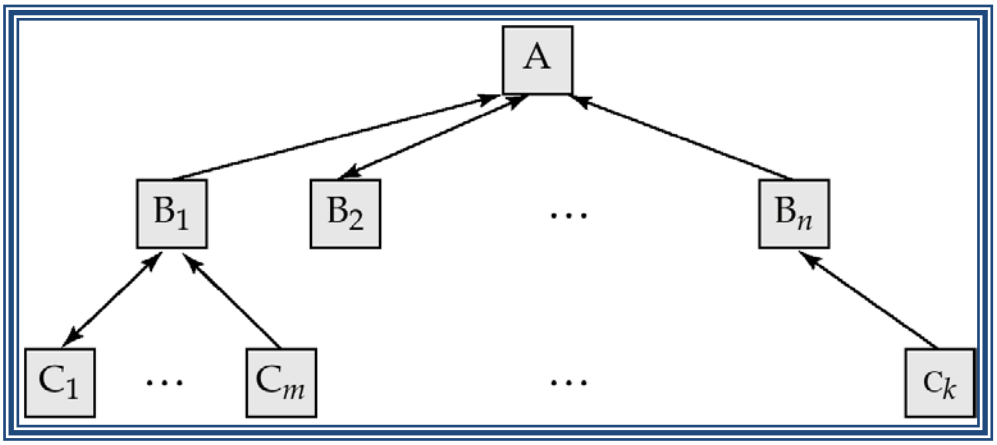
2.7 Relational model

- ✓ The model was first proposed by Dr. E.F. Codd of IBM in 1970 in the following paper: "A Relational Model for Large Shared Data Banks," Communications of the ACM, June 1970.
- ✓ The relational Model of Data is based on the concept of a Relation.
- ✓ A Relation is a mathematical concept based on the ideas of sets.
- ✓ The strength of the relational approach to data management comes from the formal foundation provided by the theory of relations.
- ✓ Relational model is most widely used data model for commercial data-processing. The reason it's used so much is, because it's simple and easy to maintain.

- ✓ The model is based on a collection of tables. Users of the database can create tables, insert new tables or modify existing tables.
- ✓ **Definition:**
  - The relation is formed over the Cartesian product of the sets; each set has values from a domain; that domain is used in a specific role which is conveyed by the attribute name.
  - Formally, Given R(A1, A2, ..., An) Where A1, A2, …, An are attributes
  - R = (A1, A2, …, An) is a relation schema
    - ♦ r(R) denotes a relation r on the relation schema R
    - ♦ Example:
    - ♦ Customer_schema = (F_name,L_name,Mobile,District,VDC,Ward_Num)
    - ♦ customer (Customer_schema)
  - The name of a relation and the set of attributes for a relation is called a Schema. The schema for the relation with the relation name followed by a parenthesized list of its attributes.
  - Relational database schema = collection of relation schemas.
  - The rows of a relation, other than the header row containing the attribute names are called Tuples. A tuple has one component for each attribute of the relation.
  - Number of tupels present in the relation is called Cardinality of relation. If there is four tuples in the relation, then cardinality of this relation is 4.
  - Number of attribute/fields present in the relation is called Degree/Arity of relation. If there is four attributes in the relation, then degree of this relation is 4.
- ✓ **Characteristics of Relations**
  - Ordering of tuples in a relation r(R): The tuples are not considered to be ordered, even though they appear to be in the tabular form.
  - Ordering of attributes in a relation schema R (and of values within each tuple): We will consider the attributes in R(A1, A2, ..., An) and the values in t=<v1, v2, ..., vn> to be ordered .
  - Values in a tuple: All values are considered atomic (indivisible). A special null value is used to represent values that are unknown or inapplicable to certain tuples.
  - Notation: We refer to component values of a tuple t by t[Ai] = vi (the value of attribute Ai for tuple t).Similarly, t[Au, Av, ..., Aw] refers to the sub tuple of t containing the values of attributes Au, Av, ..., Aw, respectively.

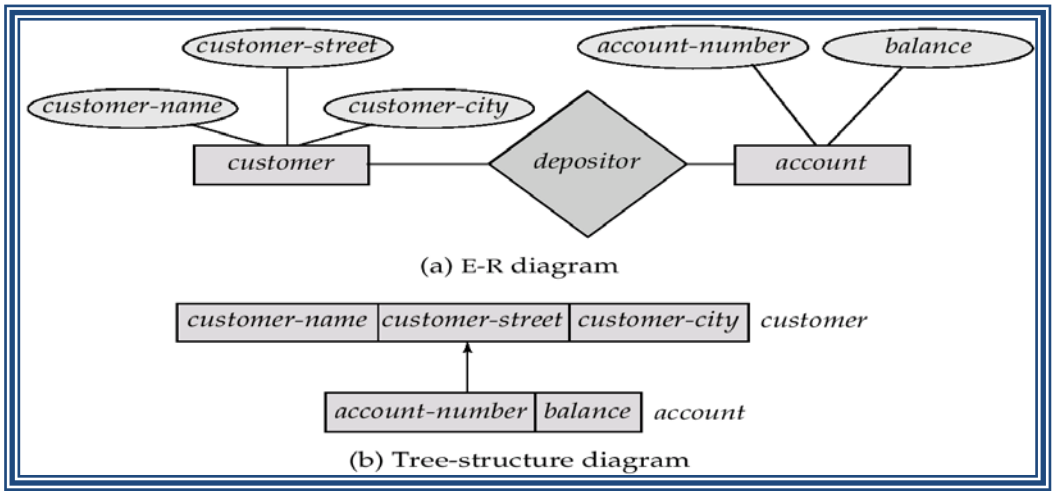
2.8 Hierarchical database

- ✓ A hierarchical database consists of a collection of records which are connected to one another through links.
- ✓ A record is a collection of fields, each of which contains only one data value.
- ✓ A link is an association between precisely two records.
- ✓ The schema for a hierarchical database consists of
    - boxes, which correspond to record types
    - lines, which correspond to links
- ✓ Record types are organized in the form of a rooted tree.
    - No cycles in the underlying graph.
    - Relationships formed in the graph must be such that only one-to-many or one-to-one relationships exist between a parent and a child.
    - A segment with no parent is called the "root"
    - A segment with no children is called a "leaf"
- ✓ A parent may have an arrow pointing to a child, but a child must have an arrow pointing to its parent.
- ✓ Database schema is represented as a collection of tree-structure diagrams.
    - single instance of a database tree
    - The root of this tree is a dummy node
    - The children of that node are actual instances of the appropriate record type
- ✓ Segment instance – particular data corresponding to the segment type
- ✓ A parent-child relationship – 1: N relationship between two segment types.



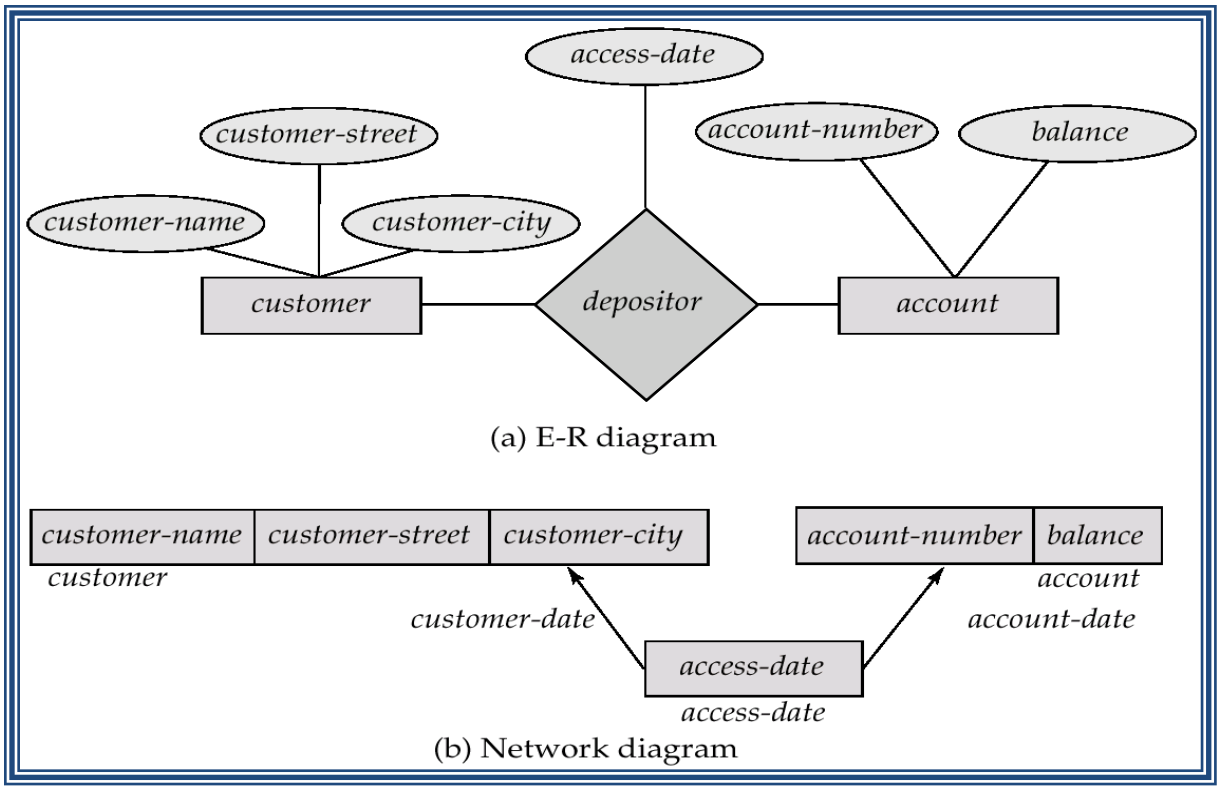
- ✓ When transforming E-R diagrams to corresponding tree-structure diagrams, we must ensure that the resulting diagrams are in the form of rooted trees
- ✓ Example E-R diagram with two entity sets, customer and account, related through a binary, one-to-many relationship depositor.
- ✓ Corresponding tree-structure diagram has

- The record type customer with three fields: customer-name, customer-street, and customer-city.
- the record type account with two fields: account-number and balance
- the link depositor, with an arrow pointing to customer



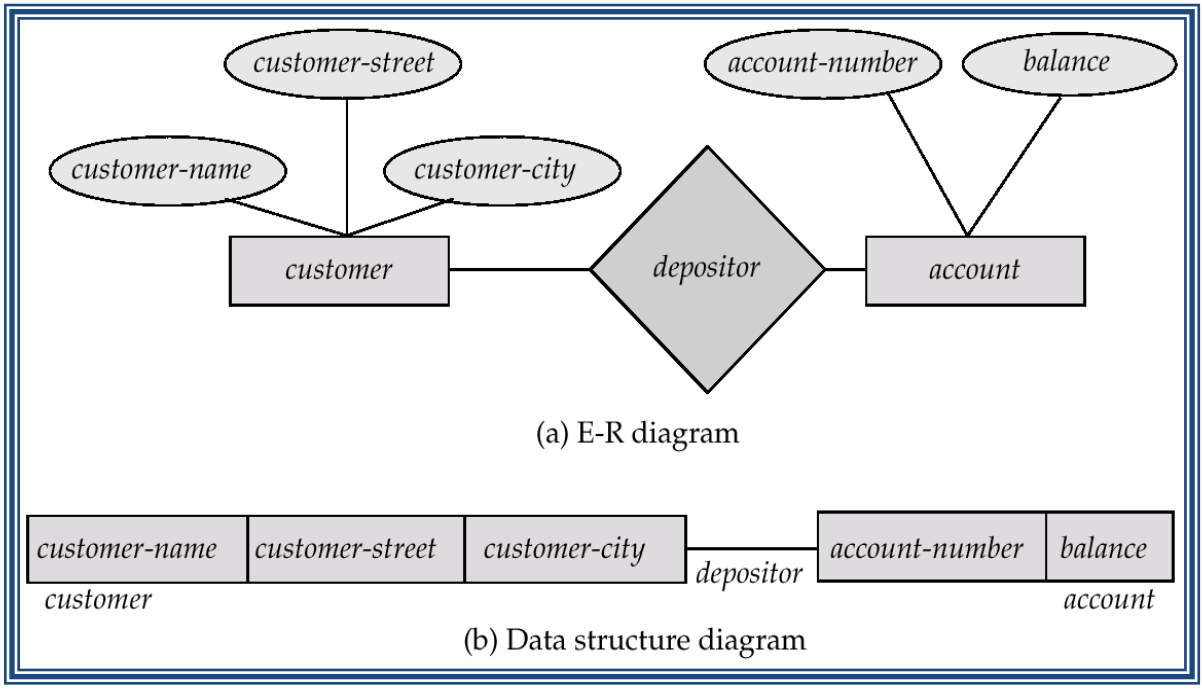
2.9 Network Model

- ✓ A network database is an extension of hierarchical database. In this model also, data elements of a database are organized to have parent child relationships, and all types of relationships among data elements must be determined when a database is first designed.
- ✓ In a network database, however, a child data element can have more than one parent element or no parent at all. Moreover, in this type of database, database management system permits extraction of needed information by beginning from any data element in database structure, instead of starting from root data element.



(a) E-R diagram

(b) Network diagram



(a) E-R diagram

(b) Data structure diagram

Chapter 3: Relational languages

3.1 Introduction to SQL

- ✓ SQL is an ANSI (American National Standards Institute) standard computer language for accessing and manipulating database systems.
- ✓ SQL works with database programs like MS Access, DB2, Informix, MS SQL Server, Oracle, Sybase, etc.
- ✓ Most of the SQL database programs also have their own proprietary extensions in addition to the SQL standard.

3.2 History of SQL

- ✓ SQL is based on the relational tuple calculus
- ✓ SEQUEL: Structured English query Language; part of SYSTEM R, 1974
- ✓ SQL/86: ANSI & ISO standard
- ✓ SQL/89: ANSI & ISO standard
- ✓ SQL/92 or SQL2: ANSI & ISO standard
- ✓ SQL3: in the works...
- ✓ SQL2 supported by ORACLE, SYBASE, INFORMIX, IBM DB2, SQL SERVER, OPENINGRES

3.3 Several parts of SQL language

- ✓ **Data Definition Language:** The most important DDL statements in SQL are:
  - CREATE - creates a new database table
  - ALTER - alters (changes) a database table
  - DROP - deletes a database table
- ✓ **Data Manipulation Language:** The most important DML statements in SQL are:
  - SELECT - extracts data from a database table
  - UPDATE - updates data in a database table
  - DELETE - deletes data from a database table
  - INSERT INTO - inserts new data into a database table
- ✓ **Data Control Language (DCL):** The most important DDL statements in SQL are:
  - GRANT – Gives user's access privileges to database
  - REVOKE – Withdraws user's access privileges to database given with the GRANT command

- ✓ **View Definition**
- ✓ **Transition Control**
- ✓ **Embedded SQL and Dynamic SQL**

3.4 Domain Types in SQL

- ✓ **Char (n).** Fixed length character string, with user-specified length n.
- ✓ **Varchar(n).** Variable length character strings, with user-specified maximum length n.
- ✓ **int.** Integer (a finite subset of the integers that is machine-dependent).
- ✓ **smallint.** Small integer (a machine-dependent subset of the integer domain type).
- ✓ **Numeric (p,d).** Fixed point number, with user-specified precision of p digits, with n digits to the right of decimal point.
- ✓ **Real, double precision.** Floating point and double-precision floating point numbers, with machine-dependent precision.
- ✓ **Float (n).** Floating point number, with user-specified precision of at least n digits.
- ✓ **date:** Dates, containing a (4 digit) year, month and date
  - ➢ Example: **date** '2005-7-27'
- ✓ **Time:** Time of day, in hours, minutes and seconds.
  - ➢ Example: **time** '09:00:30' **time** '09:00:30.75'
- ✓ **timestamp**: date plus time of day
  - ➢ Example: **timestamp** '2005-7-27 09:00:30.75'
- ✓ **interval:** period of time
  - ➢ Example: interval '1' day
  - ➢ Subtracting a date/time/timestamp value from another gives an interval value
  - ➢ Interval values can be added to date/time/timestamp values
- ✓ **Operations on complex types:**
  - ➢ Subtracting a date/time/timestamp value from another gives an interval value
  - ➢ Interval values can be added to date/time/timestamp values
  - ➢ Values of individual fields can be extracted from date/time/timestamp: **extract** (**year from** r.starttime)
  - ➢ String types can typically be cast to date/time/timestamp: **cast** <string-valued-expression> **as date**

3.5 Integrity constraints

- ✓ Integrity constraints ensure that changes made to the database by authorized users do not result in loss of data consistency.

Domain Constraints

- ✓ A domain of possible values must be associated with every attribute in the database.
- ✓ Declaring an attribute of a particular domain acts as a restraint on the values it can take.
- ✓ They are easily tested by the system. EX1: cannot set an integer variable to "cat".

Data Integrity

- ✓ Databases are used to store data

- ✓ The data is used to create information which is needed for making decisions. Therefore, we need to make sure that the data which is stored in the database is correct and consistent. This is known as **data integrity**.
- ✓ For relational databases, there are **entity integrity** and **referential integrity** rules which help to make sure that we have data integrity

Attribute Integrity

- ✓ Attribute integrity is not part of the relational model. It is used by database software to help with data integrity. The software makes sure that data for particular fields is of the correct type (eg letters or numbers) or the correct length

Entity Integrity

- ✓ The entity integrity rule applies to Primary Keys. The entity integrity rule says that the value of a Primary Key in a table must be unique and it can never have no value (null)
- ✓ Operations on the database which insert new data, update existing data, or delete data must follow this rule

Referential Integrity

- ✓ The referential integrity key applies to Foreign Keys. A relation schema may have an attribute that corresponds to the primary key of another relation. The attribute is called a **foreign key.**
- ✓ The referential integrity key says that the value of a Foreign key must either be null (ie have no value) or be equal to the value in the linked table where the Foreign Key is the Primary Key
- ✓ Ensuring that a value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation.
- ✓ EX1: In a banking system, the attribute branch-name in Account-Schema is a foreign key referencing the primary key of Branch-Schema.

Database Modification

- Inserting, deleting and updating can cause violations of referential integrity.
- Therefore, the system must check that referential integrity is maintained when you perform these operations.
- If referential integrity is violated during these operations, the default action is to reject the operation.
  - Referential Integrity in SQL: Foreign Keys

- Foreign Keys are specified as part of the SQL 'create table' statement by using the 'foreign key' clause.
- By default, foreign key references the primary key attributes of the referenced table.

Cascading

- When referential integrity is violated during a modification, instead of just rejecting the modification, you can cascade:
  - Delete cascade
  - Update cascade
- Delete Cascade
  - In a delete cascade, anything that has references to the deleted item is also deleted.
- Update Cascade
  - In an update cascade, when the updated item results in a violation of referential integrity, the system will update accordingly to fix the problem.

3.6 SQL (Structure query language)

3.6.1 Integrity Constraints in Create Table

✓ Constraint restricts the values that the table can store. We can declare integrity constraints at the table level or column level. In column-level constraints the constraint type is specified after specifying the column data type i.e., before the delimiting comma. Where as in the table-level constraints the constraint type is going to be specified as separate comma-delimited clauses after defining the columns.

There are six constraints

1. Not Null
2. Unique Key
3. Check
4. Primary Key
5. Foreign Key
6. Default

1. Not Null

✓ If a column in a table is specified as Not Null, then it's not possible to insert a null in such a column. It can be implemented with create and alter commands. When we implement the Not Null constraint with alter command there should not be any null values in the existing table.

2. Unique Key

- ✓ The unique constraint doesn't allow duplicate values in a column. If the unique constraint encompasses two or more columns, no two equal combinations are allowed. Note: There is a different behavior in the following Relational Databases.
- ✓ MS SQL Server: In this, we can insert one Row with a Null value against the Unique Key constraint column.
- ✓ Oracle: In this, we can insert any number of Rows with a Null value against the Unique Key constraint column. Please keep it in mind that two Null values are not equal.

3. Check

- ✓ Check constraint is used to restrict the values before inserting into a table.

4. Primary Key

- ✓ The key column with which we can identify the entire Table is called as a primary key column. A primary key is a combination of a Unique and a Not Null constraint; it will not allow null and duplicate values. A table can have only one primary key.
- ✓ A primary key can be declared on two or more columns as a Composite Primary Key.

5. Foreign Key

- ✓ Columns defined as foreign keys refer the Primary Key of other tables. The Foreign Key "points" to a primary key of another table, guaranteeing that you can t enter data into a table unless the referenced table has the data already which enforces the REFERENTIAL INTEGRITY. This column will take Null values.

6. Default

- ✓ The DEFAULT constraint is used to insert a default value into a column.
- ✓ The default value will be added to all new records, if no other value is specified.

3.6.2 Data Definition Langugage (DDL)

- ✓ TABLE CREATION Syntax

```
CREATE TABLE <table name> (
<column_name1> <data type>[(<width>)]
[constraint <constraint name> <constraint type>],
<column_name2> <data type>[(<width>)],
<column_name3> <data type>[(<width>)],
<column_name4> <data type>[(<width>)],
.....
```

<column_nameN> <data type>[(<width>)]);

Example

```
Create table section (     course_id varchar (8),
                           sec_id varchar (8) ,
                           Semester varchar (6),
                           Year numeric (4,0),
                           Building varchar (15),
                           room_number varchar (7),
                           Time slot id varchar (4),
                           Primary key (course_id, sec_id, semester, year),
                           Check (semester in ('Fall', 'Winter', 'Spring', 'Summer'))

Create table person (    ID char (10),
                         name char(40),
                         mother char(10),
                         father  char(10),
                         primary key ( ID),
                         foreign key father references person,
                         foreign key mother references  person)

Create table course (    course_id   char(5) primary key,
                         title        varchar(20),
                         dept_name varchar(20)
                         foreign key (dept_name) references department
                          on delete cascade
                          on update cascade  )

Create table Employee (empno number (4)  constraint  pk_emp primary  key,
                         ename varchar2(50),
                         salary  number(10,2),
                         hire_date date,
                         gender char(1)  constraint  chk_gen  check(gender in  ('M', 'F', 'm', 'f')),
                         email varchar2(50)  unique   );
CREATE table  Persons ( P_Id int NOT NULL,
                        LastName varchar(255) NOT NULL,
                        FirstName varchar(255),
```

```
Address varchar(255),
City varchar(255) DEFAULT 'Sandnes' )
```

3.6.3 Drop and Alter Table Constructs

- ✓ The drop table command deletes all information about the dropped relation from the database.
- ✓ The alter table command is used to alter attributes/domain/constraint to an existing relation:

- ✓ **To Add attributes in table**

```
Alter table < table_name>
 add   <attribute_name>    < datatype(size)>
```

Example

```
Alter table employee
Add salary numeric(5.2)
```

- ✓ **To drop attributes from table**

```
Alter table < table_name>
 Drop   column <attribute_name>
```

- ✓ **To change the data type of a column in a table**

```
ALTER TABLE < table_name>
ALTER COLUMN <column_name>   datatype(size)
```

- ✓ **To modify the length of data type in a table**

```
ALTER TABLE < table_name>
ALTER COLUMN <column_name>   datatype(size)
```

- ✓ **To Add NOT NULL constraint**

```
ALTER TABLE < table_name>
ALTER COLUMN <column_name>   datatype(size) NOT NULL
```

Example

```
ALTER TABLE employee
ALTER COLUMN Emp_Id  nvarchar(50)  NOT  NULL
```

- ✓ **To Add NULL Constraint**

```
ALTER TABLE < table_name>
ALTER COLUMN <column_name>   datatype(size)  NULL
```

- ✓ **To Add Default constraint**

```
ALTER TABLE < table_name>
Add   constraint <constraint name> Default <default_value> For < column_name>
```

- ✓ **To Add constraint in table**

Alter table < table name>
Add constraint <constraint name> < constraint type> <column_name>

Example

Alter table employee
Add constraint pk_emp PRIMARY KEY (Emp_Id)

Alter table employee
Add constraint check_salary CHECK (salary>=20000)

Alter table employee
Add constraint check_degree CHECK (degree in ('BE',' M .sc '))

Alter table employee
Add constraint fk_emp FOREIGN KEY (dept_num) References Department

Alter table employee
Add constraint unique_emp UNIQUE (email)

- ✓ **To drop constraint from table**

Alter table < table_name>
Drop constraint <constraint_name>

Example

Alter table employee
Drop constraint pk_emp

Alter table employee
Drop constraint fk_emp

Alter table employee
Drop constraint check_salary

Alter table employee
Drop constraint unique_emp

- ✓ **To drop a default constraint**

ALTER TABLE < table_name>
Drop constraint <constraint_name>

3.6.4 Data Manipulation language (DML)

3.6.4 .1 Select Statement

- ✓ The SELECT statement is used to select data from a table. The tabular result is stored in a result table (called the result-set).
- ✓ To select all columns from table, use a * symbol instead of column names.
- ✓ The DISTINCT keyword is used to return only distinct (different) values.
- ✓ Syntax
  - ❖ SELECT "column_name" FROM "table_name"
  - ❖ SELECT * FROM “table_name"
  - ❖ SELECT DISTINCT "column_name" FROM "table_name"

Where

- ✓ To conditionally select data from a table, a WHERE clause can be added to the SELECT statement.
- ✓ Comparison results can be combined using the logical connectives and, or, and not.
- ✓ With the WHERE clause, the following operators can be used

Operator	Description
=	Equal
<>	Not equal
>	Greater than
<	Less than
>=	Greater than or equal
<=	Less than or equal
BETWEEN	Between an inclusive range
LIKE	Search for a pattern

- ✓ Syntax
  - ❖ SELECT "column_name" FROM "table_name”
    WHERE "condition"
  - ❖ SELECT "column_name" FROM "table_name"
    WHERE "simple condition" [AND|OR] "simple condition"
  - ❖ SELECT "column_name" FROM "table_name"
    WHERE "column_name" BETWEEN 'value1' AND 'value2'
  - ❖ SELECT "column_name" FROM "table_name"
    WHERE "column_name" IN ('value1', 'value2', ...)

String Operations

- ✓ SQL includes a string-matching operator for comparisons on character strings.
- ✓ The operator "like" uses patterns that are described using two special characters:
  - ❖ Percent (%). The % character matches any substring.
  - ❖ Underscore (_). The _ character matches any character.
  - ❖ concatenation (using "||")
- ✓ Syntax
  - ❖ SELECT "column_name" FROM "table_name" WHERE "column_name" LIKE {PATTERN}

3.6.4.2 Ordering tuple

- ✓ Used to sort the tuple either ascending or descending order in the result of query
- ✓ Specify **desc** for descending order or **asc** for ascending order, for each attribute; ascending order is the default.
- ✓ syntax
  - ❖ SELECT "column_name" FROM "table_name"[WHERE "condition"] ORDER BY "column_name" [ASC, DESC]
- ✓ Suppose that we wish to list the entire Student relation in ascending order of amount. If several students have the same age, then order them in descending order by first name.
- ✓ Syntax
  - ❖ SELECT * FROM Student ORDER BY age asc, first_name desc

3.6.4.3 Tuple Variables

- ✓ Tuple variables are defined in the **from** clause via the use of the **as** clause.
- ✓ Find the names of all branches that have greater assets than some branch located in KTM
- ✓ **select distinct** T.branch_name **from** branch **as** T, branch **as** S
  where T.assets > S.assets **and** S.branch_city = 'KTM'
- ✓ Keyword **as** is optional and may be omitted
  borrower **as** T ≡ borrower T

3.6.4.4 The Rename Operation

- ✓ The SQL allows renaming relations and attributes using the **as** clause:
  old-name **as** new-name
- ✓ Find the name, loan number and loan amount of all customers; rename the column name loan_number as loan_id.

  Select customer_name, borrower.loan_number **as** loan_id, amount
  from borrower, loan
  where borrower.loan_number = loan.loan_number

3.6.4.5 Set Operations

- ✓ The set operations **union, intersect,** and **except** operate on relations and correspond to the relational algebra operations $\cup$, $\cap$, $-$.
- ✓ Each of the above operations automatically eliminates duplicates; to retain all duplicates write **union all, intersect all** and **except all** in place of **union, intersect** and **except** respectively **.**
- ✓ Suppose a tuple occurs m times in r and n times in s, then, it occurs:
  - ❖ m + n times in r **union all** s
  - ❖ min(m,n) times in r **intersect all** s
  - ❖ max(0, m – n) times in r **except all** s
- ✓ Syntax (Sql statement) Set operation (Sql statement)

3.6.4.6 Aggregate Functions

- ✓ An aggregate function summarizes the results of an expression over a number of rows, returning a single value.
- ✓ Some of the commonly used aggregate functions are
  - ❖ **avg:** average value
  - ❖ **min:** minimum value
  - ❖ **max:** maximum value
  - ❖ **sum:** sum of values
  - ❖ **count:** number of values
- ✓ Remember: COUNT (*) is the only function which won't ignore Nulls. Other functions like SUM, AVG, MIN, MAX they ignore Nulls.
- ✓ Each subgroup of tuples consists of the set of tuples that have the same value for the grouping attribute(s).
- ✓ **Group by** clause used to group a set of tuples having same value on given attribute.
- ✓ The attribute or attributes given in the **group by** clause are placed in one group.
- ✓ Syntax

```
SELECT column_name, aggregate_function(column_name)
FROM  table_name
WHERE  column_name condition
GROUP BY column_name
```

- ✓ The HAVING clause is used for specifying a selection condition on groups (rather than on individual tuples)
- ✓ Predicates in the **having** clause are applied after the formation of groups whereas predicates in the **where** clause are applied before forming groups

- ✓ Syntax

```
SELECT column_name, aggregate_function(column_name)
FROM table_name
WHERE column_name condition
GROUP BY column_name
HAVING aggregate_function(column_name) condition
```

3.6.4.7 Null Values

- ✓ It is possible for tuples to have a null value, denoted by null, for some of their attributes
- ✓ Null signifies an unknown value or that a value does not exist.
- ✓ The predicate is **null** can be used to check for null values.
- ✓ Example: Find all loan number which appear in the loan relation with null values for amount. **select** loan_number
  from loan
  where amount **is null**
- ✓ The result of any arithmetic expression involving null is null
- ✓ Example: 5 + null returns null
- ✓ Any comparison with null returns unknown
- ✓ Example: 5 < null or null <> null or null = null
- ✓ Three-valued logic using the truth value unknown:
- ✓ OR: (unknown **or** true) = true,
  (unknown **or** false) = unknown
  (unknown **or** unknown) = unknown
- ✓ AND: (true **and** unknown) = unknown,
  (false **and** unknown) = false,
  (unknown **and** unknown) = unknown
- ✓ NOT: (**not** unknown) = unknown
- ✓ "P **is unknown**" evaluates to true if predicate P evaluates to unknown
- ✓ Result of **where** clause predicate is treated as false if it evaluates to unknown
- ✓ Total all loan amounts
  select sum (amount)
  from loan
- ✓ Above statement ignores null amounts
- ✓ Result is null if there is no non-null amount
- ✓ All aggregate operations except **count (*)** ignore tuples with null values on the aggregated attributes.

3.6.5 Nested Queries (Sub Queries)

- ✓ A nested query is a form of a SELECT command that appears inside another SQL statement. It is also termed as subquery.
- ✓ The SELECT commands containing a subquery are referred as parent statement. The rows returned by the subquery are used by the parent statement.
- ✓ Sub queries are SELECT statements embedded within another SELECT statement
  - ➢ the results of the **inner** SELECT statement (or **sub select**) are used in the **outer** statement to help determine the contents of the final result
    - ❖ **Inner** to **outer** means evaluating statements from right to left
    - ❖ A **subselect** can be used in the **WHERE** and **HAVING** clauses of an outer SELECT statement
- ✓ Sub queries can be used with a number of operators:
  - ➢ IN, NOT IN
  - ➢ ALL
  - ➢ SOME, ANY
  - ➢ EXISTS, NOT EXISTS
- ✓ The ALL operator may be used with subqueries that produce a single column of numbers.
- ✓ If the subquery is preceded by the keyword ALL, the condition will only be TRUE if it is satisfied by all the values produced by the subquery
- ✓ The SOME operator may be used with subqueries that produce a single column of numbers. SOME and ANY can be used interchangeably.
- ✓ If the subquery is preceded by the keyword SOME, the condition will only be TRUE if it is satisfied by any (one or more) values produced by the subquery.
- ✓ EXISTS and NOT EXISTS produce a simple TRUE/FALSE result.
- ✓ EXISTS is TRUE if and only if there exists at least one row in the result table returned by the subquery; it is FALSE if the subquery returns an empty result table. NOT EXISTS is the opposite of EXISTS.
- ✓ (= **some**) ≡ **in** However, (≠ **some**) ≠ **not in**
- ✓ (≠ **all**) ≡ **not in** However, (= **all**) ≠ **in**
- ✓ Sql Example: Find all customers who have both an account and a loan at the Perryridge branch

```
Select distinct customer_name
 from borrower, loan
 where borrower.loan_number = loan.loan_number and
            branch_name = 'Perryridge'  and
            (branch_name, customer_name )
             in (select branch_name, customer_name
                 from depositor, account
                 where depositor.account_number = account.account_number )
```

- ✓ Find all branches that have greater assets than some branch located in KTM.

```
Select branch_name
        from branch
        where assets > some
                (select assets
                 from branch
                 where branch_city = 'KTM')
```

- ✓ Find all customers who have a loan at the bank but do not have an account at the bank

```
Select distinct customer_name
from borrower
where customer_name not in (select customer_name
                            from depositor)
```

3.6.6 Modification of the Database

- ✓ Modification of the database includes the
  - Insert : to insert data into the table
  - Delete : to delete data from the table
  - Update : to update existing data on the table
- ✓ **Insert Into Statement**
  - ➢ INSERT INTO table_name VALUES (value1, value2…)
  - ➢ INSERT INTO "table_name" ("column1", "column2"...)
    VALUES ("value1", "value2" ...)
  - ➢ To copy the part of information from one table to another table, we can write the sql following form
    - If we have two tables having same domain of a1 &b1, a2 &b2 in T1 (a1,a2,a3,a4,a5) and T2(b1,b2),then sql to insert the information of T1 into T2 is

```
Insert into T2
select a1, a2
from T1
```

- ✓ **Delete From Statement**
  - ➢ DELETE FROM "table_name" WHERE {condition}
  - ➢ Example: Delete the record of all accounts with balances below the average at the bank.

```
Delete from account
        where balance < (select avg (balance)
                         from account)
```

- **Update Statement**
  - UPDATE "table_name" SET "column_1" = [new value] WHERE {condition}
  - UPDATE tbLNAME

```
set column =  case
                When predicate1 then result1
                When predicate2 then result2
                …….
                When predicate n then result n
                Else  result
                end
```

- Example: Increase all accounts with balances over 20,000 by 10%, all other accounts receive 15%.

```
Update account
        set balance = case
        when balance <= 20000 then balance *1.1
        else  balance * 1.15
        end
```

3.7 Joined Relation

- Join operations take two relations and return as a result another relation.
- These additional operations are typically used as subquery expressions in the from clause
- In relational databases, a join operation matches records in two tables. The two tables must be joined by at least one common field. That is, the join field is a member of both tables.
- **Equi-join**–a join in which the joining condition is based on equality between values in the common columns; common columns appear redundantly in the result table
- **Natural join**–an equi-join in which one of the duplicate columns is eliminated in the result table
- An **inner join** is a join in which the DBMS selects records from two tables only when the records have the same value in the common field that links the tables
- An **outer join** returns all rows that satisfy the join condition and those rows from one table for which no rows from the other satisfy the join condition. Such rows are not returned by a simple join.
- There are three types of an OUTER JOIN: LEFT, RIGHT, and FULL. The LEFT OUTER JOIN keeps the stray rows from the "left" table (the one listed first in your query statement). In the result set, columns from the other table that have no corresponding data are filled with NULL values. Similarly, the RIGHT OUTER JOIN keeps stray rows from

the right table, filling columns from the left table with NULL values. The FULL OUTER JOIN keeps all stray rows as part of the result set.

subject

sub_code	sub_name	lecture_hour	practical_hour
201CT	DBMS	6	3
335CT	DSA	6	3
346CT	TOC	4	0
445CT	CG	4	3

author

author_ID	sub_name
A03	SE
B315	DSA
KP05	TOC
SP35	MP

Subject Natural Join Author

sub_code	sub_name	lecture_hour	practical_hour	author_ID
335CT	DSA	6	3	B315
346CT	TOC	4	0	KP05

Subject Inner Join Author on S.sub_name = A.S_name:

sub_code	sub_name	lecture_hour	practical_hour	author_ID	sub_name
335CT	DSA	6	3	B315	DSA
346CT	TOC	4	0	KP05	TOC

Select * from Subject Left Outer Join Author on S.sub_name= A.s_name:

sub_code	sub_name	lecture_hour	practical_hour	author_ID	sub_name
335CT	DSA	6	3	B315	DSA
346CT	TOC	4	0	KP05	TOC
201CT	DBMS	6	6	null	null
445CT	CG	4	3	null	null

Select * from Subject Right Outer Join Author on S.sub_name= A.s_name:

sub_code	sub_name	lecture_hour	practical_hour	author_ID	sub_name
335CT	DSA	6	3	B315	DSA
346CT	TOC	4	0	KP05	TOC
null	null	null	null	A03	SE
null	null	null	null	SP35	MP

Select * from Subject Full Outer Join Author on S.sub_name= A.s_name:

sub_code	sub_name	lecture_hour	practical_hour	author_ID	sub_name
335CT	DSA	6	3	B315	DSA
346CT	TOC	4	0	KP05	TOC
201CT	DBMS	6	6	null	null
445CT	CG	4	3	null	null
null	null	null	null	A03	SE
null	null	null	null	SP35	MP

3.8 Embedded SQL

- ✓ The SQL standard defines embeddings of SQL in a variety of programming languages such as C, Java, and C#.
- ✓ A language to which SQL queries are embedded is referred to as a host language, and the SQL structures permitted in the host language comprise embedded SQL.
- ✓ Approach: Embed SQL in the host language.
  - ➢ A preprocessor converts the SQL statements into special API calls.
  - ➢ Then a regular compiler is used to compile the code.
- ✓ Language constructs:
  - ➢ Connecting to a database:
    EXEC SQL CONNECT
  - ➢ Declaring variables:
    EXEC SQL BEGIN (END) DECLARE SECTION
  - ➢ Statements:
    EXEC SQL Statement

3.9 View

- ✓ A view is a logical or virtual table, which does not, exists physically in the database. Views are also called as Dictionary Objects.
- ✓ Advantages
  - Security – The confidential columns can be suppressed from viewing and manipulation.
  - Complexity can be avoided.
  - Logical columns can be created.
  - Views can represent a subset of the data contained in a table
  - Views can join and simplify multiple tables into a single virtual table
  - Views can act as aggregated tables, where the database engine aggregates data (sum, average etc.) and presents the calculated results as part of the data
  - Views take very little space to store; the database contains only the definition of a view, not a copy of all the data it presents
  - Views can limit the degree of exposure of a table or tables to the outer world
- ✓ Views can be classified into two categories on the way they are created.
  - Simple View
    If the view is created from a single table, it is called as a simple view.We can do DML Operation, if it is Simple View.
  - Complex View
    - If the view is created on multiple tables, it is called as a complex view.
- ✓ To create view
  - Create View < View Name> as <Query>

- ✓ Example
  - A view of instructors without their salary

    create view faculty **as**
    select ID, name, dept_name
    from instructor
  - Create a view of department salary totals

    create view departments_total_salary(dept_name, total_salary) **as**
    select dept_name, **sum** (salary)
    from instructor
    group by dept_name;

3.10 Database Security and Authorization

- ✓ The information in your database is important. Therefore, we need a way to protect it against unauthorized access, malicious destruction or alteration, and accidental introduction of data inconsistency.

Database Security

- ✓ Database Security refers to protection from malicious access.
- ✓ Absolute protection is impossible
- ✓ Therefore, make the cost to the perpetrator so high it will deter most attempts.
- ✓ Malicious Access
- ✓ Some forms of malicious access:
  - Unauthorized reading (theft) of data
  - Unauthorized modification of data
  - Unauthorized destruction of data

Security Levels

- ✓ To protect a database, we must take security measures at several levels.
  - Database System: Since some users may modify data while some may only query, it is the job of the system to enforce authorization rules.
  - Operating System: No matter how secure the database system is, the operating system may serve as another means of unauthorized access.
  - Network: Since most databases allow remote access, hardware and software security is crucial.
  - Physical: Sites with computer systems must be physically secured against entry by intruders or terrorists.
  - Human: Users must be authorized carefully to reduce the chance of a user giving access to an intruder.

Authorization

- ✓ For security purposes, we may assign a user several forms of authorization on parts of the databases which allow:
    - Read: read tuples.
    - Insert: insert new tuple, not modify existing tuples.
    - Update: modification, not deletion, of tuples.
    - Delete: deletion of tuples.
- ✓ We may assign the user all, none, or a combination of these. In addition to the previously mentioned, we may also assign a user rights to modify the database schema:
    - Index: allows creation and modification of indices.
    - Resource: allows creation of new relations.
    - Alteration: addition or deletion of attributes in a tuple.
    - Drop: allows the deletion of relations.

Authorization in SQL

- ✓ The SQL language offers a fairly powerful mechanism for defining authorizations by using privileges.
- ✓ Privileges in SQL
- ✓ SQL standard includes the privileges:
    - Delete
    - Insert
    - Select
    - Update
    - References: permits declaration of foreign keys.
- ✓ SQL includes commands to grant and revoke privileges.
- ✓ GRANT Command Syntax
    - grant <privilege list> on <relation or view name> to <user>
- ✓ The following privileges can be specified:
    - SELECT: Can read all columns (including those added later via ALTER TABLE command).
    - INSERT (col-name): Can insert tuples with non-null or non-default values in this column.
    - INSERT means same right with respect to all columns.
    - DELETE: Can delete tuples.
    - REFERENCES (col-name): Can define foreign keys (in other tables) that refer to this column.
- ✓ If a user has a privilege with the GRANT OPTION, can pass privilege on to other users (with or without passing on the GRANT OPTION). Examples
    - GRANT INSERT, SELECT ON Student TO Ram
        - Ram can query Student or insert tuples into it.

- GRANT DELETE ON Student TO Hari WITH GRANT OPTION
  - Hari can delete tuples, and also authorize others to do so.
- GRANT UPDATE (age) ON Student TO Dinesh
  - Dinesh can update (only) the age field of Student tuples.
- GRANT SELECT ON ActiveSailors TO Guppy, Yuppy
  - This does NOT allow the 'uppies to query Sailors directly!

✓ By default, a user granted privileges is not allowed to grant those privileges to other users. To allow this, we append the term "with grant option" clause to the appropriate grant command.

- grant select on branch to U1 with grant option

✓ To revoke a privilege we use the 'revoke' clause, which is used very much like 'grant'.

✓ Revoke Command syntax

- revoke <privilege list> on <relation or view name> from <user list>

✓ Example: Revoke INSERT, SELECT ON Sailors from Horatio

Roles

✓ Roles are a collection of privileges or access rights. When there are many users in a database it becomes difficult to grant or revoke privileges to users. Therefore, if you define roles, you can grant or revoke privileges to users, thereby automatically granting or revoking privileges.

✓ The Syntax to create a role is:

- CREATE ROLE role_name

✓ To grant CREATE TABLE privilege to a user by creating a testing role:

- First, create a manager Role
  - Creating role manager
- Second, grant a CREATE TABLE privilege to the ROLE Manager. You can add more privileges to the ROLE.
  - Grant create table to manager
- Third, grant the role to a Ram.
  - Grant Manager to Ram

✓ To revoke a CREATE TABLE privilege from Manager ROLE, you can write:

- Revoke Create table from manager

3.11 Triggers and Assertion

Triggers

- ✓ A trigger is a statement that the system executes automatically as a side effect of a modification to the database.
- ✓ To design a trigger we must meet two requirements:
  - Specify when a trigger is to be executed. This is broken up into an event that causes the trigger to be checked and a condition that must be satisfied for trigger execution to proceed.
  - Specify the actions to be taken when the trigger executes.
- ✓ This is referred to as the event-condition-action model of triggers. The database stores triggers just as if they were regular data. This way they are persistent and are accessible to all database operations. Once a trigger is entered into the database, the database system takes on the responsibility of executing it whenever the event occurs and the condition is satisfied.
- ✓ Need for Triggers: A good use for a trigger would be, for instance, if you own a warehouse and you sell out of a particular item, to automatically re-order that item and automatically generate the order invoice. So, triggers are very useful for automating things in your database.
- ✓ Three parts of a trigger:
  - Event (activates the trigger) insert, delete or update of the database.
  - Condition (tests whether the trigger should run) a Boolean statement or a query
  - Action (what happens if the trigger runs) wide variety of options.

Assertions

- ✓ An **assertion** is a statement in SQL that ensures a certain condition will always exist in the database. Assertions are like column and table constraints, except that they are specified separately from table definitions. An example of a column constraint is NOT NULL, and an example of a table constraint is a compound foreign key, which, because it's compound, cannot be declared with column constraints.
- ✓ Defined independently from any table.
- ✓ Activated on any modification of any table mentioned in the assertion.
- ✓ Components include:
  - a constraint name,
  - followed by CHECK,
  - followed by a condition
- ✓ Query result must be empty
  - If the query result is not empty, the assertion has been violated

- ✓ Example : The salary of an employee must not be greater than the salary of the manager of the department that the employee works for

```
CREAT ASSERTION SALARY_CONSTRAINT
CHECK (NOT EXISTS (SELECT *
                   FROM EMPLOYEE E, EMPLOYEE M,
        DEPARTMENT D
                   WHERE E.SALARY > M.SALARY AND
              E.DNO=D.NUMBER AND
        D.MGRSSN=M.SSN))
```

- ✓ Number of boats plus number of sailors is < 100.

```
CREATE ASSERTION smallClub
CHECK
( (SELECT COUNT (S.sid) FROM Sailors S)
+ (SELECT COUNT (B.bid) FROM Boats B) < 100);
```

Example on SQL

STUDENT (CRN, FNAME, LNAME, DOB, AGE, DISTRICT, WARD_NO, VDC/MUNICIPALITY, PROGRAM, BATCH, PHONE)
ACCOUNT (CRN, ACCOUNT_ID, FEE, PAID_MONEY)

1. Find Name (combine FNAME and LNAME) of students who study in either BCT or BEX.

```
SELECT FNAME+"+LNAME as NAME
FROM STUDENT
WHERE PROGRAM IN ('BCT','BEX')
```

Alternate method:

```
SELECT FNAME+"+LNAME AS NAME
FROM STUDENT
WHERE PROGRAM ='BCT' or PROGRAM ='BEX'
```

2. Find FNAME of students between age 20 to 25.

```
SELECT FNAME
FROM STUDENT
WHERE AGE BETWEEN 20 AND 25
```

3. Find ACCOUNT_ID of the student whose age is 20

```
SELECT ACCOUNT_ID,FNAME,LNAME
FROM STUDENT, TBL_ACCOUNT
WHERE STUDENT.CRN=TBL_ACCOUNT.CRN
AND AGE=20
```

4. Find FNAME of the students whose last name begin from ‘S’.

```
SELECT FNAME
FROM STUDENT
WHERE LNAME LIKE 'S%'
```

5. Find FNAME of student who is the youngest of the college.

```
SELECT FNAME
FROM STUDENT
WHERE AGE IN(SELECT MIN(AGE) FROM STUDENT)
```

6. Find CRN and fname of the students whose lname has at least five characters.

```
SELECT CRN,FNAME
FROM STUDENT
WHERE LNAME LIKE '_____%'
```

7. Sort the list of students according to age in ascending order. If there are number of students having same age then sort them in descending order according to first name.

```
SELECT *
FROM STUDENT
ORDER BY AGE ASC,FNAME DESC
```

8. Find the name of the students who live in KTM district and paid money is 20000.

```
SELECT FNAME+' '+LNAME AS NAME
FROM STUDENT, TBL_ACCOUNT
WHERE STUDENT.CRN=TBL_ACCOUNT.CRN
AND TBL_ACCOUNT.FEE =20000
AND STUDENT.DISTRICT='KATHMANDU'
```

9. Count the total number of students in each ward of Lalitpur metropolitan city.

```
SELECT WARD_NO,COUNT(CRN) AS
NUMBER_OF_STUDENT_IN_WARD
FROM STUDENT
WHERE DISTRICT='LALITPUR'
GROUP BY WARD_NO
```

10. Find the eldest person name in each batch.

```
SELECT FNAME,AGE,BATCH
 FROM STUDENT,( SELECT MAX(AGE)AS MAX_AGE , PROGRAM FROM
STUDENT GROUP BY BATCH)  AS T(A,B)
WHERE Student.Age=T.A=Student.Porgramme=T.B
```

11. Find the name of students whose age and batch is same as of Rita.

```
SELECT FNAME,AGE,BATCH
FROM STUDENT
WHERE AGE IN  (SELECT AGE
FROM STUDENT
WHERE FNAME='RITA')AND
BATCH IN(SELECT BATCH FROM STUDENT
WHERE FNAME='RITA')
```

12. Find the program and average age of student in each program that average age greater than 20.

```
SELECT  PRORAM, AVG(AGE) AS AVG_AGE
FROM STUDENT
GROUP BY  PROGRAM
HAVING AVG(AGE)>20
```

13. Display NAME and ACCOUNT_ID of student who live in Chitwan district.

```
SELECT FNAME+' '+LNAME AS NAME, ACCOUNT_ID
FROM  STUDENT, TBL_ACCOUNT
WHERE STUDENT.CRN= TBL_ACCOUNT.CRN
AND  STUDENT.DISTRICT='CHITWAN'
```

14. Find the total number of computer students in each batch.

```
SELECT COUNT (CRN), batch FROM student
WHERE program = 'BCT'
GROUP BY batch
```

15. Find the name of students whose age is same AS age of Ram.

```
SELECTFNAME, LNAME FROM student
WHERE age in (SELECT age FROM student WHEREFNAME = 'Ram')
```

16. Find the name of student who are elder than the some student live in KTM district.

```
SELECTFNAME, LNAME FROM student
WHERE age > SOME (SELECT age FROM student WHERE branch_city = 'KTM')
```

17. Delete a record of students who live in KTM metropolitan.

```
DELETE FROM student
WHERE metropolitian_city = 'KTM'
```

18. Delete the record of student whose district is same AS that of Hari.

```
DELETE FROM student
WHERE district in (SELECT district FROM student WHEREFNAME = 'Hari')
```

19. Change the batch 2066 to 2068 of all students whose program is 'BCT'

```
UPDATE student
SET batch = 2068
WHERE program = 'BCT' and batch = 2066
```

20. Update the batch of all student from 2066 to 2068, 2067 to 2069 AND rest batch as it is of all student

```
UPDATE student
SET batch = case
When batch = 2066 then 2068
When batch = 2067 then 2069
Else batch
End
```

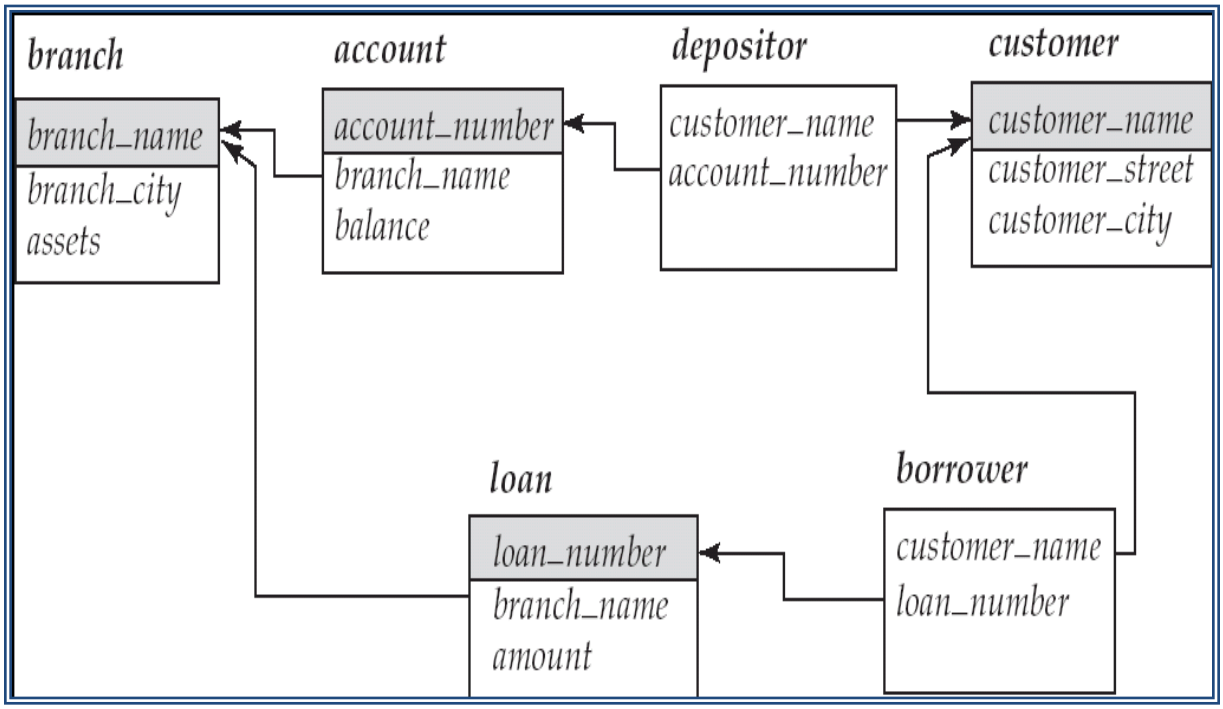
3.12 Relational Algebra

- ✓ Relational algebra is the basic set of operations for the relational model
- ✓ Relational Algebra is algebra whose operands are relations and operators are designed to do the most commons things that we need to do with relations**.**
- ✓ A relation schema is given by R(A1,…,Ak), the name of the relation and the list of the attributes in the relation
- ✓ A relation is a set of tuples that are valid instances of its schema
- ✓ Relational algebra expressions take as input relations and produce as output new relations.
- ✓ After each operation, the attributes of the participating relations are carried to the new relation. The attributes may be renamed, but their domain remains the same.

Basic Relational Algebra Operations

- ✓ Select
- ✓ Project
- ✓ Union
- ✓ Set Difference (or Subtract or minus)
- ✓ Cartesian product
- ✓ Rename

Bank Schema



Select Operation

- ✓ Unary operations
- ✓ The SELECT operation (denoted by σ (sigma)) is used to select a subset of the tuples from a relation based on a **selection condition**
- ✓ Notation: $\sigma_p(r)$
  - *p* is called the selection predicate
- ✓ Defined as:

  $\sigma_p(r) = \{t \mid t \in r \text{ and } p(t)\}$
- ✓ Where *p* is a formula in propositional calculus consisting of terms connected by : ∧ (and), ∨ (or), ¬ (not)
- ✓ Each term is one of:

  <Attribute> *op* <attribute> or <constant>

  Where *op* is one of: =,≠, >, ≥. <. ≤
- ✓ The SELECT operation $\sigma_{<\text{selection condition}>}(R)$ produces a relation S that has the same schema (same attributes) as R
- ✓ SELECT s is commutative:
  - $\sigma_{<\text{condition1}>}(\sigma_{<\text{condition2}>}(R)) = \sigma_{<\text{condition2}>}(\sigma_{<\text{condition1}>}(R))$
- ✓ Because of commutativity property, a cascade (sequence) of SELECT operations may be applied in any order:
  - $\sigma_{<\text{cond1}>}(\sigma_{<\text{cond2}>}(\sigma_{<\text{cond3}>}(R)) = \sigma_{<\text{cond2}>}(\sigma_{<\text{cond3}>}(\sigma_{<\text{cond1}>}(R))))$

- ✓ A cascade of SELECT operations may be replaced by a single selection with a conjunction of all the conditions:
  - $\sigma_{<cond1>}(\sigma_{<cond2>}(\sigma_{<cond3>}(R)) = \sigma_{<cond1> \text{ AND } <cond2> \text{ AND } <cond3>}(R))$
- ✓ The number of tuples in the result of a SELECT is less than (or equal to) the number of tuples in the input relation R
- ✓ Example of selection:

  σ branch_name="Perryridge"(account)

Project Operation

- ✓ Unary operations
- ✓ Notation: $\prod_{A_1, A_2, \ldots, A_k}(r)$

  Where A_1, A_2 are attributes names and r is a relation name.
- ✓ The result is defined as the relation of k columns obtained by erasing the columns that are not listed
- ✓ Duplicate rows removed from result, since relations are sets
- ✓ PROJECT is not commutative
- ✓ Example: To eliminate the branch_name attribute of account

  $\Pi_{\text{account\_number, balance}}$ (account)

Union Operation

- ✓ Binary operations
- ✓ Notation: $r \cup s$
- ✓ Defined as:
  - $r \cup s = \{t \mid t \in r \text{ or } t \in s\}$
- ✓ Duplicate tuples are eliminated
- ✓ For $r \cup s$ to be valid.
  - r, s must have the same Arity (same number of attributes)
  - The attribute domains must be compatible (example: 2nd column of r deals with the same type of values as does the 2nd column of s)
- ✓ Example: to find all customers with either an account or a loan

  $\Pi_{\text{customer\_name}}$ (depositor) $\cup$ $\Pi_{\text{customer\_name}}$ (borrower)

Set Difference Operation

- ✓ Notation $r - s$
- ✓ Defined as:
  - $r - s = \{t \mid t \in r \textbf{ and } t \notin s\}$
- ✓ The result of R – S, is a relation that includes all tuples that are in R but not in S
- ✓ Set differences must be taken between **compatible** relations.
  - *r* and *s* must have the same Arity
  - attribute domains of *r* and *s* must be compatible

Cartesian-Product Operation

- ✓ Notation r x s
- ✓ Defined as:
  - r x s = {t q | t ∈ r **and** q ∈ s}
- ✓ If attributes of r(R) and s(S) are not disjoint, then renaming must be used.
- ✓ This operation is used to combine tuples from two relations in a combinatorial fashion.
- ✓ Denoted by R(A1, A2, . . ., An) x S(B1, B2, . . ., Bm)
- ✓ Result is a relation Q with degree n + m attributes: Q(A1, A2, . . ., An, B1, B2, . . ., Bm), in that order.
- ✓ The resulting relation state has one tuple for each combination of tuples—one from R and one from S. Hence, if R has n_R tuples (denoted as $|R| = n_R$), and S has n_S tuples, then R x S will have $n_R * n_S$ tuples.

Rename Operation

- ✓ In some cases, we may want to rename the attributes of a relation or the relation name or both
- ✓ The general RENAME operation ρ can be expressed by any of the following forms:
  - $\rho_{S(B1, B2, \ldots, Bn)}(R)$ changes both:
    - The relation name to S, and the column names to B1, B1,…,Bn
  - $\rho_S(R)$ changes:
    - the relation name only to S
- ✓ If a relational-algebra expression E has Arity n, then

$$\rho_{x(A_1, A_2, \ldots, A_n)}(E)$$

  - Returns the result of expression E under the name X, and with the attributes renamed to A_1 , A_2 , …., A_n .
- ✓ Find the largest account balance
  - $\Pi_{balance}(account) - \Pi_{account.balance} (\sigma_{account.balance < d.balance} (account \times \rho_d (account)))$

Set-Intersection Operation

- ✓ Notation: r ∩ s
- ✓ Defined as:
- ✓ r ∩ s = { t | t ∈ r **and** t ∈ s }
- ✓ Assume:
  - r, s have the same Arity
  - Attributes of r and s are compatible
- ✓ Note: r ∩ s = r – (r – s)

Aggregate Functions and Operations

- ✓ Aggregation function takes a collection of values and returns a single value as a result.
  - **avg**: average value
  - **min**: minimum value
  - **max**: maximum value
  - **sum**: sum of values
  - **count**: number of values
- ✓ Aggregate operation in relational algebra

$$G_1, G_2, \ldots, G_n \, \vartheta_{F_1(A_1), F_2(A_2), \ldots, F_n(A_n)}(E)$$

  - Where E is any relational-algebra expression and $G_1, G_2 \ldots, G_n$ is a list of attributes on which to group (can be empty)
  - Each F_i is an aggregate function
  - Each A_i is an attribute name

- ✓ **Example**

branch_name	*account_number*	*balance*
Perryridge	A-102	400
Perryridge	A-201	900
Brighton	A-217	750
Brighton	A-215	750
Redwood	A-222	700

Relation account grouped by branch-name

branch_name g **sum**(balance) (account)

branch_name	**sum**(*balance*)
Perryridge	1300
Brighton	1500
Redwood	700

Delete, Insert and Update Operation

Delete

- ✓ A delete request is expressed similarly to a query, except instead of displaying tuples to the user; the selected tuples are removed from the database.
- ✓ Can delete only whole tuples; cannot delete values on only particular attributes
- ✓ A deletion is expressed in relational algebra by:
  - $r \leftarrow r - E$

  Where r is a relation and E is a relational algebra query.
- ✓ Delete all account records in the New road branch

$$Account \leftarrow account - \sigma_{branch\_name = \text{“New Road”}} (account)$$

- ✓ Delete all accounts at branches located in KTM

$$r_1 \leftarrow \sigma_{branch\_city = \text{“Needham”}} (account \bowtie branch)$$

$$r_2 \leftarrow \prod_{account\_number,\ branch\_name,\ balance} (r_1)$$

$$r_3 \leftarrow \prod_{customer\_name,\ account\_number} (r_2 \bowtie depositor)$$

$$Account \leftarrow account - r_2$$

$$Depositor \leftarrow depositor - r_3$$

Insertion

- ✓ To insert data into a relation, we either:
- ✓ specify a tuple to be inserted
- ✓ write a query whose result is a set of tuples to be inserted
- ✓ in relational algebra, an insertion is expressed by:
  - $r \leftarrow r \cup E$

  Where *r* is a relation and *E* is a relational algebra expression.
- ✓ The insertion of a single tuple is expressed by letting *E* be a constant relation containing one tuple.
- ✓ Insert information in the database specifying that Smith hasRs. 1200 in account A-973 at the Patan branch.

$$Account \leftarrow account \cup \{(\text{“A-973”}, \text{“Patan”}, 1200)\}$$

$$Depositor \leftarrow depositor \cup \{(\text{“Smith”}, \text{“A-973”})\}$$

- ✓ Provide as a gift for all loan customers in the Patan branch, Rs.200 savings account. Let the loan number serve as the account number for the new savings account.

$$r1 \leftarrow (\sigma_{branch\_name = \text{“Patan”}} (borrower \bowtie loan))$$

$$Account \leftarrow account \cup \prod_{loan\_number,\ branch\_name,\ 200} (r1)$$

$$Depositor \leftarrow depositor \cup \prod_{customer\_name,\ loan\_number} (r1)$$

Updating

- ✓ A mechanism to change a value in a tuple without charging *all* values in the tuple
- ✓ Use the generalized projection operator to do this task

$$r \leftarrow \prod_{F_1, F_2, \ldots, F_l,} (r)$$

- ✓ Each F_i is either
  - the I th attribute of r, if the I th attribute is not updated, or,
  - if the attribute is to be updated F_i is an expression, involving only constants and the attributes of r, which gives the new value for the attribute
- ✓ Pay all accounts with balances over $10,000 6 percent interest and pay all others 5 percent

$$Account \leftarrow \prod_{account\_number,\ branch\_name,\ balance} * 1.06\ (\sigma_{BAL > 10000}\ (account))$$
$$\cup \prod_{account\_number,\ branch\_name,\ balance} * 1.05\ (\sigma_{BAL \le 10000}\ (account))$$

Binary Relational Operations: JOIN

- ✓ The general form of a join operation on two relations R(A1, A2, . . ., An) and S(B1, B2, . . ., Bm) is: R $\bowtie_{\text{<join condition>}}$ S
  - Where R and S can be any relations that result from general relational algebra expressions.
- ✓ The join condition is called theta
- ✓ Theta can be any general Boolean expression on the attributes of R and S;
- ✓ Most join conditions involve one or more equality conditions "AND"ed together.

Natural-Join Operation

- ✓ Notation: r $\bowtie$ s
- ✓ Let *r* and *s* be relations on schemas *R* and *S* respectively. Then, r $\bowtie$ s is a relation on schema $R \cup S$ obtained as follows:
  - Consider each pair of tuples t_r from *r* and t_s from *s*.
  - If t_r and t_s have the same value on each of the attributes in $R \cap S$, add a tuple *t* to the result, where
    - *t* has the same value as t_r on *r*
    - *t* has the same value as t_s on *s*
- ✓ Example:
- ✓ $R = (A, B, C, D)$
- ✓ $S = (E, B, D)$
- ✓ Result schema = (A, B, C, D, E)
- ✓ $r \bowtie s$ is defined as:

$$\prod_{r.A,\ r.B,\ r.C,\ r.D,\ s.E} (\sigma_{r.B = s.B \wedge r.D = s.D}\ (r \times s))$$

Outer Join

1. An extension of the join operation that avoids loss of information.
2. Computes the join and then adds tuples form one relation that does not match tuples in the other relation to the result of the join.
3. Uses null values: null signifies that the value is unknown or does not exist
4. Outer join are three types: Left outer join , right outer join, full outer join.

Example On Relational Algebra

1. Find the first name and last name of all students

$$\pi_{f\_name, l\_name}(student)$$

2. Find the first name of student whose age is 20 and live in Kathmandu district

$$\pi_{f\_name}(\sigma_{age=20 \wedge district="KTM"}(student))$$

3. Find the first name of student who live in Kathmandu or Lalitpur district

$$\pi_{f\_name}(\sigma_{district="KTM" \vee district="LPR"}(student))$$

$$\pi_{f\_name}(\sigma_{district="KTM"}(student)) \cup \pi_{f\_name}(\sigma_{district="LPR"}(student))$$

4. Find the first name of student who do not live in Kathmandu district

$$\pi_{f\_name}(\sigma_{district \neq "KTM"}(student))$$

$$\pi_{f\_name}(student) - \pi_{f\_name}(\sigma_{district="KTM"}(student))$$

5. Find the name of customer who have an account at Putalisadak Branch

$$\pi_{customer\_Name}(\sigma_{Account.Account\_Number=Depositor.Account\_Number \wedge Branch\_Name="Putalisadak"}(Account \times Depositor))$$

6. Find the customer name who have made a load at the bank located in Kathmandu city

$$\pi_{customer\_Name}\left(\begin{array}{l}\sigma_{Borrower.Loan\_number=loan.loan\_number \wedge loan.branch\_name=branch.branch\_name \wedge branch\_city="KTM"} \\ \\ (Borrower \times Loan \times Branch)\end{array}\right)$$

7. Find the first name of youngest student name of the college

$$\pi_{f\_name}(student) - \pi_{student.f\_name}(\sigma_{student.age>d.age}(student \times \delta_d(student)))$$

8. Find the first name of student whose batch and programme are same as Ram's batch and programme.

$$\pi_{f\_name}(\sigma_{student.batch=d.Rbatch \wedge student.programme=d.Rprogramme}(student \times \delta_{d(Rbatch,Rprogramme)}(\pi_{batch,programme}(\sigma_{f\_name="RAM"}(student)))))$$

9. Find the name of customer who have an account and a loan at the bank

$$\Pi_{customer\_name}(borrower) \cap \Pi_{customer\_name}(depositor)$$

10. Count the total number of students in each batch

$$Batch\zeta_{count(CRN) \text{ as total\_Student}}(student)$$

11. Find the average age of students of a collegessss

$$\zeta_{avg(age) as \text{ average\_age}}(student)$$

12. Find the min age of student in each programme

$$\zeta_{\min(age)\,as\ \text{minimum\_age}}(student)$$

13. Find the name of customer whose balance is 10000 at the bank.

$$\Pi_{customer\_name}(\sigma_{balance=10000}(account \text{N} depositor))$$

14. Delete the record of students who live in KTM district.

$$student \leftarrow student - \sigma_{district="KTM"}(student)$$

15. Delete the record of student whose fee is less than 10000.

$$r1 \leftarrow \sigma_{fee<10000}(student N account)$$

$$r2 \leftarrow \Pi_{CRN,f\_name,l\_name,age,batch,programme}(r1)$$

$$r3 \leftarrow \Pi_{account\_id,fee,due\_amount,CRN}(r1)$$

$$student \leftarrow student - r2$$

$$student \leftarrow account - r3$$

16. Delete the account records of customer who live in Kathmandu city

$$r1 \leftarrow \sigma_{customer\_city="KTM"}(account N depositor N customer)$$

$$r2 \leftarrow \Pi_{account\_number,branch\_name,balance}(r1)$$

$$account \leftarrow account - r2$$

$$r3 \leftarrow \Pi_{account\_number,customer\_name}(r1)$$

$$depositor \leftarrow depositor - r3$$

17. Increase the balance of all account by 10%

$$account \leftarrow \Pi_{account\_number,branch\_name,balance*1.1}(account)$$

18. Increase the balance by 10% if the balance is greater than 10000 if below don't change.

$$account \leftarrow \Pi_{account\_number,branch\_name,balance*1.1}(\sigma_{b>10000}(account))$$
$$\cup (account - \sigma_{balance>10000}(account))$$

Chapter 4

4.1 Introduction

- ✓ Normalization is formal process for determining which field belongs in which tables in a relational database.
- ✓ Database normalization is the process of removing redundant data from your tables in to improve storage efficiency, data integrity, and scalability.
- ✓ Normalization is the process of efficiently organizing data in a database with two goals in mind
    - First goal: eliminate redundant data
    - Second Goal: ensure data dependencies make sense
- ✓ Bad database design may have
    - Repetition of information
    - Inability to represent certain information
- ✓ In order to comply with the relational model it is necessary to
    - Remove repeating groups and
    - Avoid redundancy and data anomalies by remoting partial and transitive functional dependencies.
- ✓ **Relational Database Design:** All attributes in a table must be atomic, and solely dependent upon the fully primary key of that table.
- ✓ **Redundant** data is where we have stored the same 'information' more than once. i.e., the redundant data could be removed without the loss of information. Such redundancy could lead to the Insert, delete and update anomalies

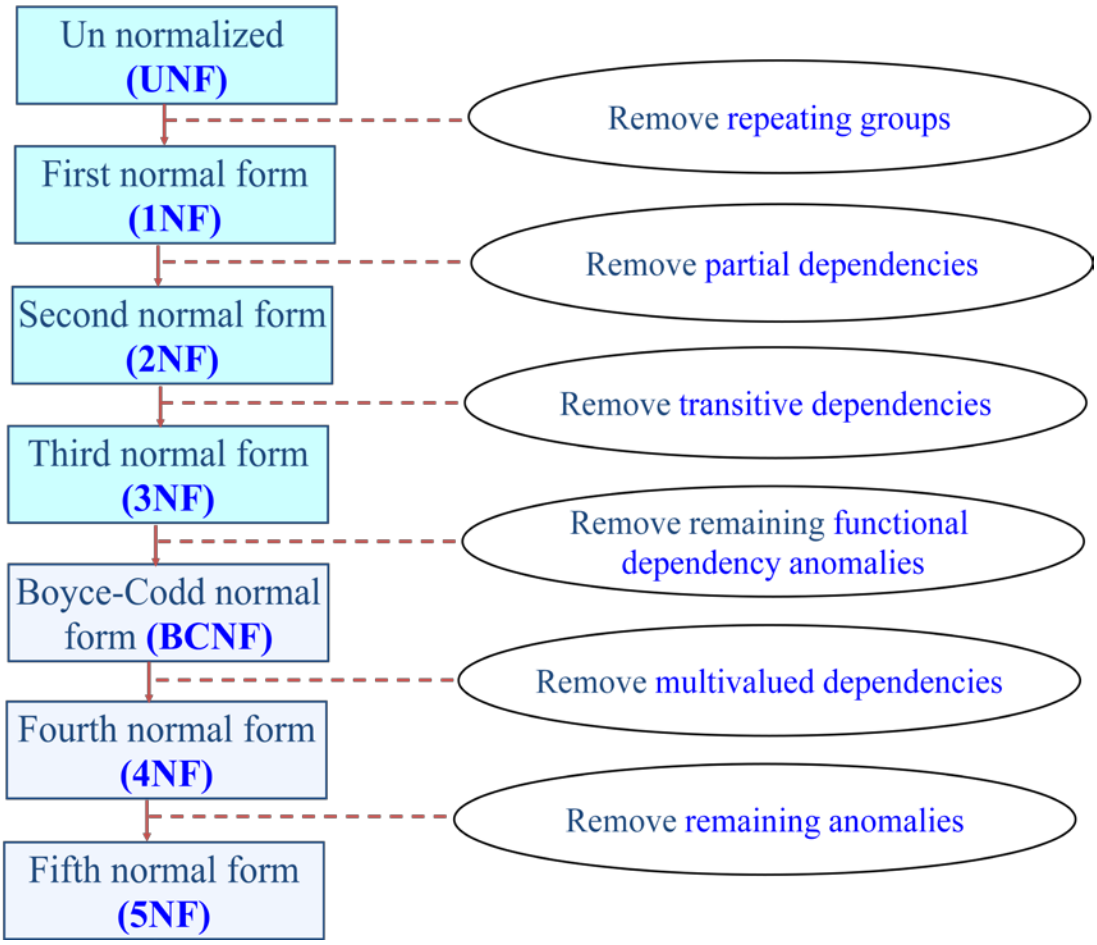
Purpose of Normalization

- ✓ To avoid redundancy by storing each 'fact' within the database only once.
- ✓ To put data into a form that conforms to relational principles (e.g., single valued attributes, each relation represents one entity) - no repeating groups.
- ✓ To put the data into a form that is more able to accurately accommodate change.
- ✓ To avoid certain updating 'anomalies'.
- ✓ To facilitate the enforcement of data constraints.

Advantages of Normalization

- ✓ Elimination of redundant data storage.
- ✓ Closed modeling of real world entities, processes and their relationship.
- ✓ Structuring of data so that model is flexible.
- ✓ Less storage space
- ✓ Easier to add data
- ✓ Flexible Structure

Stages of Normalisation



4.2 Functional dependency

- ✓ **Formal Definition**: Attribute B is functionally dependent upon attribute A (or a collection of attributes) if a value of A determines a single value of attribute B at any one time.
- ✓ **Formal Notation**: **A → B** this should be read as **'A determines B'** or **'B is functionally dependant on A'**. A is called the determinant and B is called the object of the determinant.
- ✓ **Mathematical Definition** : Let *R* be a relation schema , $\alpha \subseteq R$ *and* $\beta \subseteq R$

  The functional dependency

  $$\alpha \rightarrow \beta$$

  Holds on *R* if and only if for any legal relations *r*(R), whenever any two tuples t_1 and t_2 of *r* agree on the attributesα, they also agree on the attributesβ. That is,

  $$t_1[\alpha] = t_2[\alpha] \Rightarrow t_1[\beta] = t_2[\beta]$$

- ✓ Example: Consider *r* (A, *B)* with the following instance of *r*.

A1	B1
A1	B2
A2	B3

- ✓ On this instance, $A \rightarrow B$ does **NOT** hold, but $B \rightarrow A$ does hold.
- ✓ Example
  - Loan_Number→ Amount
  - CRN→F_Name
  - (Degree, Experience) → Salary
- ✓ A functional dependency is **trivial** if, the consequent is a subset of the determinant. In other words, it is impossible for it not to be satisfied.
- ✓ In general, *A* functional dependency of the form $\alpha \rightarrow \beta$ is trivial if $\beta \subseteq \alpha$.
- ✓ Example
  - AB →A
  - A →A
  - customer_nam , loan_number → customer_name
  - customer_name → customer_name
- ✓ A functional dependency is **non- trivial,** if dependency is not trivial.
- ✓ Example
  - A→B
  - (Degree, Experience) → Salary
- ✓ **Compound Determinants**: If more than one attribute is necessary to determine another attribute in an entity, then such a determinant is termed a composite determinant.
- ✓ **Full Functional Dependency**: Only of relevance with composite determinants. This is the situation when it is necessary to use all the attributes of the composite determinant to identify its object uniquely.
- ✓ **Partial Functional Dependency**: This is the situation that exists if it is necessary to only use a subset of the attributes of the composite determinant to identify its object uniquely.

Example:

order#	line#	qty	price
A001	001	10	200
A002	001	20	400
A002	002	20	800
A004	001	15	300

Full Functional Dependencies

(Order#, line#) → qty

(Order#, line#) → price

Example:

student#	unit#	room	grade
9900100	A01	TH224	2
9900010	A01	TH224	14
9901011	A02	JS075	3
9900001	A01	TH224	16

Repetition of data!

Full Functional Dependencies

(student#, unit#) → grade

Partial Functional Dependencies

unit# → room

4.2.1 Closure of a Set of Functional Dependencies

- ✓ The set of all Functional dependencies implied by a set of functional dependencies F is called the closures of F.
- ✓ Given a set F of functional dependencies, there are certain other functional dependencies that are logically implied by F.
- ✓ For example: 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.
- ✓ Denoted the closure of F by F^+.
- ✓ F^+ is a superset of F.
- ✓ Finding all of F^+ by applying Armstrong's Axioms:
  - if $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$ **(reflexivity)**
  - if $\alpha \rightarrow \beta$, then $\gamma\ \alpha \rightarrow \gamma\ \beta$ **(augmentation)**
  - if $\alpha \rightarrow \beta$, and $\beta \rightarrow \gamma$, then $\alpha \rightarrow \gamma$ **(transitivity)**
- ✓ Additional rules are
  - If $\alpha \rightarrow \beta$ holds *a*nd $\alpha \rightarrow \gamma$ holds, then $\alpha \rightarrow \beta\gamma$ holds **(union)**
  - If $\alpha \rightarrow \beta\gamma$ holds, then $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds **(decomposition)**
  - If $\alpha \rightarrow \beta$ holds *a*nd $\gamma\ \beta \rightarrow \delta$ holds, then $\alpha\ \gamma \rightarrow \delta$ holds **(pseudotransitivity)**
  - **Note:** The above rules can be inferred from Armstrong's axioms.
- ✓ Example : For given FD, find all possible F^+

  R = (A, B, C, G, H, I)

  F = { $A \rightarrow B$, $A \rightarrow C$,$CG \rightarrow H$,$G \rightarrow I$, $B \rightarrow H$}
  - Some members of F^+
    - $A \rightarrow H$ by transitivity from $A \rightarrow B$ and $B \rightarrow H$

- $AG \rightarrow I$ by augmenting $A \rightarrow C$ with G, to get $AG \rightarrow CG$ and then transitivity with $CG \rightarrow I$
- $CG \rightarrow HI$ by augmenting $CG \rightarrow I$ to infer $CG \rightarrow CGI$, and augmenting of $CG \rightarrow H$ to infer $CGI \rightarrow HI$, and then transitivity

✓ Problem 1: For given FD

R = (A, B, C, D,E,G,H,I,J)

F = { $AB \rightarrow E$, $AG \rightarrow J$,$BE \rightarrow I$,$E \rightarrow G$, $GI \rightarrow H$}

I. List all possible F^+

II. Does $AB \rightarrow GH$?

✓ Problem 2: For given FD

R = (A, B, C, D, E, F, G)

F = {$A \rightarrow B$, $ABCD \rightarrow E$, $EF \rightarrow G$}

I. List all possible F^+

II. Does $ACDF \rightarrow G$, implied by the set of FD's?

✓ Problem 3: For given two set of F1 & F2 for a relation

F1 = {$A \rightarrow B$, $AB \rightarrow C$, $D \rightarrow AC$, $D \rightarrow E$}

F2 = {$A \rightarrow BC$, $D \rightarrow AE$}

F1 & F2 are equivalent or not?

✓ Problem 4: Let R = (A, B, C, D, E, F)

F = {$A \rightarrow BC$, $B \rightarrow E$, $CD \rightarrow EF$}

I. List all possible F^+

II. Does $AD \rightarrow F$, implied by the set of FD's?

4.2.2 Closure of Attribute Sets

✓ Given a set of attributes A, define the closure of A under F (denoted by A^+) as the set of attributes that are functionally determined by a under F

✓ Algorithm to compute A^+, the closure of a under F

```
result := A;
while (changes to result) do
    for each β → γ in F do
        begin
            if β ⊆ result then result := result ∪ γ
        end
```

✓ **Uses of Attribute Closure**

- Testing for super key- To test, if α is a super key, we compute α^+, and check if α^+ contains all attributes of *R*. Then α is a super key of R.
- Testing functional dependencies -To check if a functional dependency $\alpha \rightarrow \beta$ holds (or, in other words, is in F^+), just check if $\beta \subseteq \alpha^+$.

- ✓ Example : For given FD, find closure of $(AG)^+$

  R = (A, B, C, G, H, I)

  F = { $A \rightarrow B$, $A \rightarrow C$,$CG \rightarrow H$,$G \rightarrow I$, $B \rightarrow H$}

  - Solution
    1. result = AG
    2. result = ABCG ($A \rightarrow C$ and $A \rightarrow B$)
    3. result = ABCGH ($CG \rightarrow H$ and $CG \subseteq AGBC$)
    4. result = ABCGHI ($CG \rightarrow I$ and $CG \subseteq AGBCH$)
  - Here AG is the candidate key because closure of AG contains all attributes of R.
- ✓ Problem1: Let R = (A, B, C, D,E)

  F = { $B \rightarrow CD$, $B \rightarrow A$, $E \rightarrow C$, $AD \rightarrow B$,$D \rightarrow E$}

  Is $B \rightarrow E$ in F^+ ?
- ✓ List out all possible super key and candidate key of above given five problems

4.2.3 Canonical Cover

- ✓ Sets of functional dependencies may have redundant dependencies that can be inferred from the others
  - For example: $A \rightarrow C$ is redundant in: {$A \rightarrow B$, $B \rightarrow C$, $A \rightarrow C$}
  - Parts of a functional dependency may be redundant
  - E.g.: on RHS: {$A \rightarrow B$, $B \rightarrow C$, $A \rightarrow CD$} can be simplified to

    {$A \rightarrow B$, $B \rightarrow C$, $A \rightarrow D$}
  - E.g.: on LHS: {$A \rightarrow B$, $B \rightarrow C$, $AC \rightarrow D$} can be simplified to

    {$A \rightarrow B$, $B \rightarrow C$, $A \rightarrow D$}
- ✓ Intuitively, a canonical cover of F is a "minimal" set of functional dependencies equivalent to F, having no redundant dependencies or redundant parts of dependencies

4.2.3.1 Extraneous Attributes

- ✓ Consider a set F of functional dependencies and the functional dependency $\alpha \rightarrow \beta$ in F.
  - Attribute A is extraneous in α if $A \in \alpha$ and F logically implies

    $(F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\}$.
  - Attribute A is extraneous in β if $A \in \beta$ and the set of functional dependencies

    $(F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$ logically implies F.

- ✓ Example 1: Given F = {$A \rightarrow C$, $AB \rightarrow C$ }

  B is extraneous in $AB \rightarrow C$ because {$A \rightarrow C$, $AB \rightarrow C$} logically implies $A \rightarrow C$ (I.e. the result of dropping B from $AB \rightarrow C$).

- ✓ Example 2: Given F = {A → C, AB → CD}
  C is extraneous in AB → CD since AB → C can be inferred even after deleting C
- ✓ Example 3: Computing a Canonical Cover
  R = (A, B, C)
  F = {A → BC, B → C, A → B, AB → C}
  - Combine A → BC and A → B into A → BC
    - Set is now {A → BC, B → C, AB → C}
  - A is extraneous in AB → C
    - Check if the result of deleting A from AB → C is implied by the other dependencies
      - Yes: in fact, B → C is already present!
    - Set is now {A → BC, B → C}
  - C is extraneous in A → BC
    - Check if A → C is logically implied by A → B and the other dependencies
      - Yes: using transitivity on A → B and B → C.
        - Can use attribute closure of A in more complex cases
  - The canonical cover is: A → B, B → C

4.3 Normalization Stage

Unnormalised Normal Form (UNF)

- ✓ A relation is Unnormalised when it has not had any normalization rules applied to it, and it suffers from various anomalies.
- ✓ A relation has repeating group, so it has more than one value for a given key.
- ✓ A repeating group is an attribute (or set of attributes) that can have more than one value for a primary key value.
- ✓ Repeating Groups are not allowed in a relational design, since all attributes have to be 'atomic' - i.e., there can only be one value per cell in a table.

First Normal Form(1NF)

- ✓ A relational schema R is in first normal form if the domains of all attributes of R are atomic. Domain is atomic if its elements are considered to be indivisible units.
- ✓ Ensure that each table has a primary key: minimal set of attributes which can uniquely identify a record.
- ✓ Eliminate repeating groups (categories of data which would seem to be required a different number of times on different records) by defining keyed and non-keyed attributes appropriately.
- ✓ Atomicity: Each attribute must contain a single value, not a set of values.

Second Normal Form(2NF)

- ✓ A relation is in 2NF if, and only if, it is in 1NF and every non-key attribute is fully dependent on the primary key.
- ✓ Remove partial functional dependencies into a new relation
- ✓ A relation R is in 2NF if
  - R is 1NF , and
  - All non-prime attributes are fully dependent on the candidate keys.
- ✓ Converting from 1NF to 2NF:
  - Identify the primary key for the 1NF relation.
  - Identify the functional dependencies in the relation.
  - If partial dependencies exist on the primary key remove them by placing then in a new relation along with a copy of their determinant

Third Normal Form(3NF)

- ✓ A relation is in 3NF if, and only if, it is in 2NF and every non-key attribute is non-transitively dependent on the primary key.
- ✓ Remove transitive dependencies into a new relation
- ✓ A relation schema *R* is in third normal form (3NF) if for all:

  $\alpha \rightarrow \beta$ in F^+

  At least one of the following holds:
  - $\alpha \rightarrow \beta$ is trivial (i.e., $\beta \in \alpha$)
  - α is a super key for *R*
  - Each attribute *A* in $\beta - \alpha$ is contained in a candidate key for *R*.
  - (**NOTE***:* each attribute may be in a different candidate key)
- ✓ Converting from 2NF to 3NF:
  - Identify the primary key in the 2NF relation.
  - Identify functional dependencies in the relation.
  - If transitive dependencies exist on the primary key remove them by placing them in a new relation along with a copy of their dominant.

Boyce-Codd Normal Form(BCNF)

- ✓ BCNF refers to decompositions involving Relations with more than one candidate key, where the candidate keys are composite and overlapping
- ✓ A relation is in BCNF if and only if every determinant is a candidate key.
- ✓ A determinant is any attribute whose value determines other values with a row.
- ✓ If a table contains only one candidate key, the 3NF and the BCNF are equivalent. BCNF is a special case of 3NF.

- ✓ 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

  $\alpha \rightarrow \beta$

  Where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds:
  - $\alpha \rightarrow \beta$ is trivial (i.e., $\beta \subseteq \alpha$)
  - α is a super key for *R*

- ✓ **Example 1 :**

  In UNF

  ORDER (order-no, order-date, cust-no, cust-name, cust-add, prod-no, prod-desc, unit-price, ord-qty, line-total, order-total)

  In 1NF

  ORDER (order-no, order-date, cust-no, cust-name, cust-add, order-total)
  PRODUCT_DES (order-no, prod-no, prod-desc, unit-price, ord-qty, line-total)

  In 2NF

  ORDER (order-no, order-date, cust-no, cust-name, cust-add, order-total)
  PRODUCT-2 (prod-no, prod-desc, unit-price)
  ORDER-LINE-2 (order-no, prod-no, ord-qty, line-total)

  In 3NF

  PRODUCT-2 (prod-no, prod-desc, unit-price)
  ORDER-LINE-2 (order-no, prod-no, ord-qty, line-total)
  CUSTOMER-3 (cust-no, cust-name, cust-add)
  ORDER-3 (order-no, order-date, cust-no, order-total)

- ✓ **Example 2:**

  In UNF

  T(Patient_Num,F_name,L_name,Ward_num,ward_Name,Prescription_date,Drug_Code, Drug_Name,Dosage,length_Of_Treatment)

  In 1NF

  T1 (Patient_Num,F_name,L_name,Ward_num,ward_Name)
  T2 (Patient_Num, Prescription_date, Drug_Code, Drug_Name, Dosage, length_Of_Treatment)

  In2NF

  T1 (Patient_Num,F_name,L_name,Ward_num,ward_Name)
  T2 (Patient_Num, Prescription_date, Drug_Code, Dosage, length_Of_Treatment)
  T3 **(** Drug_Code, Drug_Name**)**

  In3NF

  T1 (Patient_Num,F_name,L_name,Ward_num)
  T2 (Patient_Num, Prescription_date, Drug_Code, Dosage, length_Of_Treatment)
  T3 **(** Drug_Code, Drug_Name**)**
  T4 (Ward_num, ward_Name)

✓ **Example1: BCNF**

Relation R and functional dependency F
R = (branch_name, branch_city, assets,customer_name, loan_number, amount)
F = {branch_name → assets branch_city , loan_number → amountbranch_name
}
Key = {loan_number, customer_name}
Decomposition

R_1 = (branch_name, branch_city, assets)
R_2 = (branch_name, customer_name, loan_number, amount)
R_3 = (branch_name, loan_number, amount)
R_4 = (customer_name, loan_number)

Final decomposition: R_1, R_3, R_4

✓ **Example2: BCNF**

- Let us assume the following reality
  - For each subject, each student is taught by one Instructor
  - Each Instructor teaches only one subject
  - Each Subject is taught by several Instructors

Student	Course	Instructor
Ram	DBMS	AD
Ram	DSA	PS
Sita	DBMS	AS
Rita	DSA	PS
Shyam	OS	KS
Shyam	DBMS	AS

This relation is in 3NF but **NOT** in BCN F, so we should decompose so to meet BCNF property. Learning (**Student, Instructor) , Teaching (Instructor, Course)**

Comparison of BCNF and 3NF

✓ Relations in BCNF and 3NF
  - Relations in BCNF: no repetition of information
  - Relations in 3NF: problem of repetition of information

✓ Decomposition in BCNF and in 3NF
  - It is always possible to decompose a relation into relations in 3NF and
    - the decomposition is lossless
    - dependencies are preserved
  - It is always possible to decompose a relation into relations in BCNF and

- the decomposition is lossless
- dependencies may not preserved

4.4 Decompositions

- ✓ The process of decomposition of a relation R into set of relations R1, R2, .Rn is based on identifying different components and using that as a basis of decomposition. The decomposed relation R1, R2…..Rn are projections of R and are of course not disjointing otherwise the glue holding the information together would be lost.
- ✓ Desirable properties of decomposition are:
    - Attributes Preservations
    - Lossless-join Decomposition
    - Dependency preservation
    - Lack of Redundancy

4.4.1 Lossless-join Decomposition

- ✓ Decomposition of R = (A, B, C)

A	B	C
1	a	x
2	b	y
3	c	z

R_1 = (A, B)

A	B
1	a
2	b
3	c

$\prod_{A\,B}(R)$

R_2 = (B, C)

B	C
a	x
b	y
c	z

$\prod_{B,\,C}(R)$

A	B	C
1	a	x
2	b	y
3	c	z

$\prod_{A,\,B,\,C}(R1 \bowtie R2)$

- ✓ For the case of R = (R_1, R_2), we require that for all possible relations r on schema R

  $r = \prod_{R1}(r) \bowtie \prod_{R2}(r)$
- ✓ A decomposition of R into R_1 and R_2 is lossless join if at least one of the following dependencies is in F^+:
    - $R_1 \cap R_2 \rightarrow R_1$
    - $R_1 \cap R_2 \rightarrow R_2$
- ✓ The above functional dependencies are a sufficient condition for lossless join decomposition; the dependencies are a necessary condition only if all constraints are functional dependencies.

- ✓ Example : Given R(A,B,C) F = {A → B, B → C)
  - Can be decomposed in two different ways
    - R_1 = (A, B), R_2 = (B, C)
      - ➢ Lossless-join decomposition: $R_1 \cap R_2$ = {B} and B → BC
    - R_1 = (A, B), R_2 = (A, C)
      - ➢ Lossless-join decomposition: $R_1 \cap R_2$ = {A} and A → AB
- ✓ Example1: R= (A B C D E) is decompose into two relations R_1= (AB), R_2= (ACDE) with given FD = {A→ B, BC → E, ED → A}.It is loss less or not?
- ✓ Example 2: R (A B C D E) is decompose into two relations R_1= (BCD), R_2= (ACE) with given FD= {AB → C, C→ E, B→D, E→A}. Is this decomposition lossless?

4.4.2 Dependency preservation

- ✓ A decomposition D = {R_1, R_2, ..., R_n} of R is dependency-preserving with respect to F if the union of the projections of F on each R_i in D is equivalent to F; that is if $(F_1 \cup F_2 \cup \ldots \cup F_n)^+ = F^+$
- ✓ Let F_i be the set of dependencies F^+ that include only attributes in R_i.
  - A decomposition is dependency preserving, if $(F_1 \cup F_2 \cup \ldots \cup F_n)^+ = F^+$
  - If it is not, then checking updates for violation of functional dependencies may require computing joins, which is expensive.
- ✓ Example : Given R(A,B,C) F = {A → B, B → C)
  - Can be decomposed in two different ways
    - R_1 = (A, B), R_2 = (B, C)
      - ➢ Here FD1={ A → B },FD2={ B → C }
      - ➢ So, {FD1 ∪ FD2 }$^+$= F^+
      - ➢ Dependency preserving
    - R_1 = (A, B), R_2 = (A, C)
      - ➢ Here FD1={ A → B },FD2={ A → C }
      - ➢ So, {FD1 ∪ FD2 }$^+ \neq F^+$
      - ➢ Not Dependency preserving
- ✓ Example 1: R (A B C D) with FD: {A ➔ B, B ➔ C, C ➔ D}. R is Decompose into two relations: R_1 (A B C) and R_2(C D). Show that decomposition is dependency preserving.
- ✓ Example 2: R (A B C D) with FD: {A ➔ B, B ➔ C, C ➔ D}. R is Decompose into two relations: R_1 (A CD) and R_2 (BC). Show that decomposition is not dependency preserving.

4.5 Multi-valued dependencies

- ✓ In contrast to the functional independency, the multivalued dependency requires that certain tuples be present in a relation. Therefore, a multivalued dependency is also referred as a tuple-generating dependency.
- ✓ Multi-valued dependencies (MVDs) express a condition among tuples of a relation that exists when the table (relation) is trying to represent
  - More than one many-many relationship.
  - Then certain columns (attributes) become independent of one another
  - And their values must appear in all combinations.
- ✓ **Formal Definition**: Let R be a relation schema and let $\alpha \subseteq R$ and $\beta \subseteq R$. The multivalued dependency

  $$\alpha \rightarrow\rightarrow \beta$$

  - holds on R if in any legal relation r(R), for all pairs for tuples t_1 and t_2 in r such that $t_1[\alpha] = t_2[\alpha]$, there exist tuples t_3 and t_4 in r such that:

    $t_1[\alpha]=t_2[\alpha]=t_3[\alpha]=t_4[\alpha]$
    $t_3[\beta]=t_1[\beta]$
    $t_3[R-\beta]=t_2[R-\beta]$
    $t_4[\beta]=t_2[\beta]$
    $t_4[R-\beta] = t_1[R-\beta]$
- ✓ Tabular representation of $\alpha \rightarrow\rightarrow \beta$

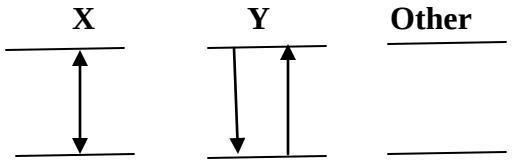
	α	β	$R-\alpha-\beta$
t_1	$a_1 \ldots a_i$	$a_{i+1} \ldots a_j$	$a_{j+1} \ldots a_n$
t_2	$a_1 \ldots a_i$	$b_{i+1} \ldots b_j$	$b_{j+1} \ldots b_n$
t_3	$a_1 \ldots a_i$	$a_{i+1} \ldots a_j$	$b_{j+1} \ldots b_n$
t_4	$a_1 \ldots a_i$	$b_{i+1} \ldots b_j$	$a_{j+1} \ldots a_n$

- ✓ Example: Customers(name, addr, phones, sodasLiked)
  - A customer's phones are independent of the sodas they like.
  - Many-many relations:
    - Customers ⇔ Phones Customers ⇔ Sodas
  - Each phone appears with each soda in all combinations.
  - E.g., For 3 phones (Home, Work, Cell) and 10 sodas, we need 30 tuples
  - There is only one FD: name → addr

- ✓ Tuples Implied by name->->phones
  - If we have tuples:

Name	Add	Phone	Soda Liked
ram	a	P1	S1
ram	a	P2	S2
ram	a	P1	S2
ram	a	P2	S1

- ✓ If we have first two tuples (1 & 2), then the last two tuples (3 & 4) must also be in the relation. Name-phone and Name-soda relations are independent.
- ✓ Representation of $X \rightarrow\rightarrow Y$



4.6 Fourth Normal Form

- ✓ The redundancy caused by MVDs can't be removed by transforming the database schema to BCNF.
- ✓ There is a stronger normal form, called 4th Normal for (4NF), that (intuitively):
  - Treats MVDs as FDs when it comes to decomposition
  - But not when determining keys of the relation.
- ✓ A relation schema R is in 4NF with respect to a set D of functional and multivalued dependencies if for all multivalued dependencies in D^+ of the form $\alpha \rightarrow\rightarrow \beta$, where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following hold:
  - $\alpha \rightarrow\rightarrow \beta$ is trivial (i.e., $\beta \subseteq \alpha$ or $\alpha \cup \beta = R$)
  - α is a super key for schema R
- ✓ If a relation is in 4NF it is in BCNF
- ✓ A relation R is in 4NF if and only if, for every one of its non-trivial multivalued dependencies $X \rightarrow\rightarrow Y$, X is a super key—that is, X is either a candidate key or a superset thereof.
  - Nontrivial MVD means that:
    - Y is not a subset of X, and
    - X and Y are not, together, all the attributes.
  - Trivial MVDs means that:
    - $Y \subseteq X$ or
    - $X \cup Y = R$

- ✓ **Decomposition into 4NF Method: Example**
  - Customers(<u>name</u>, addr, <u>phones</u>, <u>sodasLiked</u>)

    FD: name → addr

    MVDs: name →→ phones

    name →→ sodasLiked

    Key is {<u>name</u>, <u>phones</u>, <u>sodasLiked</u>}.
    - Decompose using name→addr:
      - Drinkers1(<u>name</u>, addr)
        - In 4NF; only dependency is name→addr.
    - Drinkers2(<u>name</u>, <u>phones</u>, <u>beersLiked</u>)
      - Not in 4NF. MVD's name→→phones and name→→beersLiked apply. No FD's, so all three attributes form the key.
    - Decompose Drinkers2
      - Either MVD name ->-> phones or name ->-> beersLiked tells us to decompose to:
        - Drinkers3(<u>name</u>, <u>phones</u>)
        - Drinkers4(<u>name</u>, <u>beersLiked</u>)
- ✓ **Example**

  R = (A, B, C, G, H, I)

  F = {A →→ B, B →→ HI, CG →→ H}
  - R is not in 4NF since A →→ B and A is not a super key for R
  - Decomposition
    - R_1 = (A, B) (R_1 is in 4NF)
    - R_2 = (A, C, G, H, I) (R_2 is not in 4NF)
    - R_3 = (C, G, H) (R_3 is in 4NF)
    - R_4 = (A, C, G, I) (R_4 is not in 4NF)
  - Since A →→ B and B →→ HI, A →→ HI, A →→ I
    - R_5 = (A, I) (R_5 is in 4NF)
    - R_6 = (A, C, G) (R_6 is in 4NF)

- ✓ **Normalization BCNF to 4NF Relations**

Branch_Staff_Client relation

Branch_No	*SName*	*CName*
B3	Ann Beech	Aline Stewart
B3	David Ford	Aline Stewart
B3	Ann Beech	Mike Richie
B3	David Ford	Mike Richie

Branch_Staff relation

Branch_No	*SName*
B3	Ann Beech
B3	David Ford

Branch_Client relation

Branch_No	*CName*
B3	Aline Stewart
B3	Mike Richie

4.7 Join dependency and 5NF

- ✓ A join dependency (JD) denoted by JD (R1, R2, …, Rn) on relational schema R specifies a constraint on the states r of R. The constraint states that every legal state r of R is equal to the join of its projections on R1, R2, …, Rn. That is for every such r we have:

  $\Pi R1(r) * \Pi R2(r) * \ldots * \Pi Rn(r) = r$
- ✓ The **lossless-join property** refers to the fact that whenever we decompose relations using normalization we can rejoin the relations to produce the original relation.
- ✓ A **lossless-join dependency** is a property of decomposition which ensures that no spurious tuples are generated when relations are natural joined.
- ✓ There are cases where it is necessary to decompose a relation into more than two relations to guarantee a lossless-join.
- ✓ Fifth normal form (5NF) is based on join dependencies. A relation is in fifth normal form (5NF) if and only if every nontrivial join dependency is implied by the super keys of R.
- ✓ A table is said to be in the 5NF if and only if it is in 4NF and every join dependency in it is implied by the candidate keys. 5NF is always achievable
- ✓ Example
  - consider R (S_id, S_name, Status, City) with S_id and S_name candidate keys
  - ({S_id, S_name, Status}, {S_id, City}) is a JD because S_id is a candidate key in R
  - ({S_id, S_name}, {S_id, Status}, {S_name, City}) is a JD because S_id and S_name are both candidate keys in R
- ✓ Example: Consider a relation Supply (sname, partName, projName).

sname	partName	projName
Smith	Bolt	X
Smith	Nut	Y
Adam	Bolt	Y
Walton	Nut	Z
Adam	Nail	X
Adam	Bolt	X
Smith	Bolt	Y

sname	partName
Smith	Bolt
Smith	Nut
Adam	Bolt
Walton	Nut
Adam	Nail

sname	projName
Smith	X
Smith	Y
Adam	Y
Walton	Z
Adam	X

partName	projName
Bolt	X
Nut	Y
Bolt	Y
Nut	Z
Nail	X

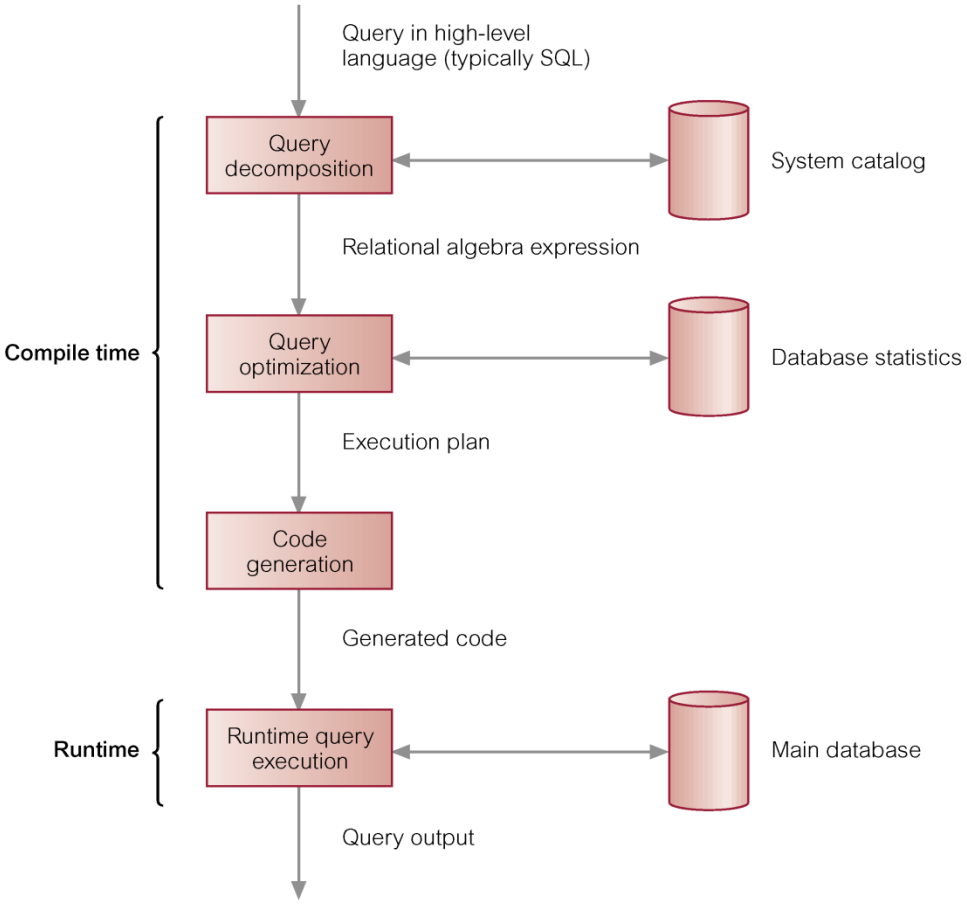
4.8 Domain-key normal form(DKNF)

- ✓ R is in DK/NF if and only if every constraint of R is a logical consequence of domain constraints and (candidate) key constraints.
- ✓ Ronald Fagin (1981) proved that if a Relation is in DKNF then it is free from any anomalies (redundancies). Including the ones caused by FDs, MVDs, JDs.
- ✓ DKNF not always achievable, and there is no formal definition to verify if a relation schema is in DKNF
- ✓ DKNF Example
  - Accounts whose account-number begins with the digit 9 are special high-interest accounts with a minimum balance of 2500.
  - General constraint: ``If the first digit of t [account-number] is 9, then t [balance] ≥ 2500."
  - DKNF design:
    - Regular-acct-schema = (branch-name, account-number, balance)
    - Special-acct-schema = (branch-name, account-number, balance)
  - Domain constraints for {Special-acct-schema} require that for each account:
    - The account number begins with 9.
    - The balance is greater than 2500.

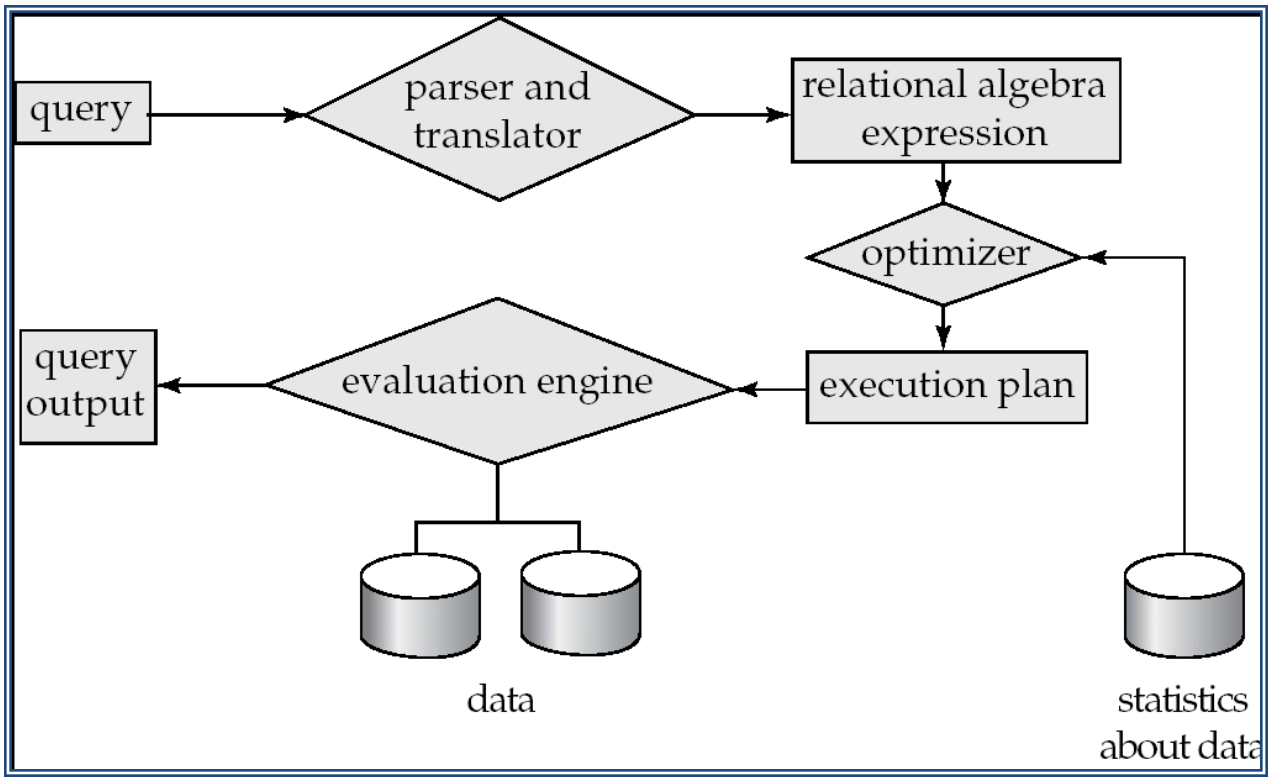
Chapter 5: Query Processing and optimization

5.1 Introduction

- ✓ Query Processing – activities involved in retrieving data from the database:
  - SQL query translation into low-level language implementing relational algebra.
  - Query execution
- ✓ Query Optimization – selection of an efficient query execution plan
- ✓ Phases of Query Processing



- ✓ **Basic Steps in Query Processing**
  - Parsing and translation
  - Optimization
  - Evaluation
- 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.
- 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.
  - Given relational algebra expression may have many equivalent expressions
    - E.g. $\sigma_{balance<2500}(\Pi_{balance}(account)$ is equivalent to $\Pi_{balance}(\sigma_{balance<2500}(account))$
  - Any 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
  - Amongst all equivalent expressions, try to choose the one with cheapest possible evaluation-plan. Cost estimate of a plan based on statistical information in the DBMS catalog.

5.2 Query cost Estimation

- ✓ We can choose a strategy based on reliable information, database systems may store statistics for each relation r. These statistics includes
  - The number of tuples in a relation
  - The size of a tuple in a relation
  - The number of distinct values that appear in the relation r for a particular attribute.
- ✓ 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 the number of block transfers from disk and the number of seeks as the cost measures
  - t_T – time to transfer one block
  - t_S – time for one seek
  - Cost for b block transfers plus S seeks

$$b * t_T + S * t_S$$

5.3 Query Operation

5.3.1Slection operation

- ✓ File scan – search algorithms that locate and retrieve records that fulfill a selection condition.
- ✓ Algorithm A1 (linear search). Scan each file block and test all records to see whether they satisfy the selection condition.
  - Cost estimate = b_r block transfers + 1 seek
    - ♦ b_r denotes number of blocks containing records from relation r
  - If selection is on a key attribute, can stop on finding record
    - ♦ cost = (b_r /2) block transfers + 1 seek
  - Linear search can be applied regardless of
    - ♦ selection condition or
    - ♦ ordering of records in the file, or
    - ♦ availability of indices

- ✓ **A2 (binary search)**. Applicable if selection is an equality comparison on the attribute on which file is ordered.

- Assume that the blocks of a relation are stored contiguously
- Cost estimate (number of disk blocks to be scanned):
  - cost of locating the first tuple by a binary search on the blocks
    $\lceil \log_2(b_r) \rceil * (t_T + t_S)$
  - If there are multiple records satisfying selection
    - Add transfer cost of the number of blocks containing records that satisfy selection condition
    - Will see how to estimate this cost in Chapter 14

✓ **Index scan** – search algorithms that use an index
- Selection condition must be on search-key of index.

✓ **A3 (primary index on candidate key, equality).** Retrieve a single record that satisfies the corresponding equality condition
- Cost = $(h_i + 1) * (t_T + t_S)$

✓ **A4 (primary index on nonkey, equality)** Retrieve multiple records.
- Records will be on consecutive blocks
  - Let b = number of blocks containing matching records
- $= h_i * (t_T + t_S) + t_S + t_T * b$

✓ **A5 (equality on search-key of secondary index).**
- Retrieve a single record if the search-key is a candidate key
  - Cost = $(h_i + 1) * (t_T + t_S)$
- Retrieve multiple records if search-key is not a candidate key
  - each of n matching records may be on a different block
  - Cost = $(h_i + n) * (t_T + t_S)$
    - Can be very expensive!
- Can implement selections of the form $\sigma_{A \le V}(r)$ or $\sigma_{A \ge V}(r)$ by using
  - a linear file scan or binary search,
  - or by using indices in the following ways:

✓ **A6 (primary index, comparison).** (Relation is sorted on A)
- For $\sigma_{A \ge V}(r)$ use index to find first tuple ≥ v and scan relation sequentially from there
- For $\sigma_{A \le V}(r)$ just scan relation sequentially till first tuple > v; do not use index

✓ **A7 (secondary index, comparison).**
- For $\sigma_{A \ge V}(r)$ use index to find first index entry ≥ v and scan index sequentially from there, to find pointers to records.
- For $\sigma_{A \le V}(r)$ just scan leaf pages of index finding pointers to records, till first entry > v
- In either case, retrieve records that are pointed to
  - requires an I/O for each record
  - Linear file scan may be cheaper

5.3.2 Sorting

- ✓ Sorting may be requested by the query (e.g., order by), or is an important preprocessing step for other queries, such as those that involve a join.
- ✓ If records are completely in main memory, standard sorting algorithms such as quick sort apply
- ✓ Otherwise, some records are still on disk, resulting in what is called external sorting
- ✓ Common external sorting technique: external sort- merge, which cumulatively sorts multiple runs of the data based on amount that fits in memory at one time

5.3.3 Join Operation

- ✓ Several different algorithms to implement joins
  - Nested-loop join
  - Merge-join
  - Hash-join
- ✓ Choice based on cost estimate
- ✓ Examples use the following information
- ✓ Number of records of customer: 10,000 depositor: 5000
- ✓ Number of blocks of customer: 400 depositor: 100

Nested-Loop Join

- ✓ To compute the theta join $r \bowtie_\theta s$

```
for each tuple t_r in r do begin
    for each tuple t_s in s do begin
        test pair (t_r,t_s) to see if they satisfy the join condition θ
        if they do, add t_r • t_s to the result.
    end
end
```

- ✓ r is called the outer relation and s the inner relation of the join.
- ✓ Requires no indices and can be used with any kind of join condition.
- ✓ Expensive since it examines every pair of tuples in the two relations.
- ✓ In the worst case, if there is enough memory only to hold one block of each relation, the estimated cost is
  $n_r * b_s + b_r$ block transfers, plus $n_r + b_r$ seeks
- ✓ If the smaller relation fits entirely in memory, use that as the inner relation.
  - Reduces cost to $b_r + b_s$ block transfers and 2 seeks
- ✓ Assuming worst case memory availability cost estimate is
  - with depositor as outer relation:
    - ♦ $5000 * 400 + 100 = 2{,}000{,}100$ block transfers,
    - ♦ $5000 + 100 = 5100$ seeks

- with customer as the outer relation
  - 10000 ∗ 100 + 400 = 1,000,400 block transfers and 10,400 seeks

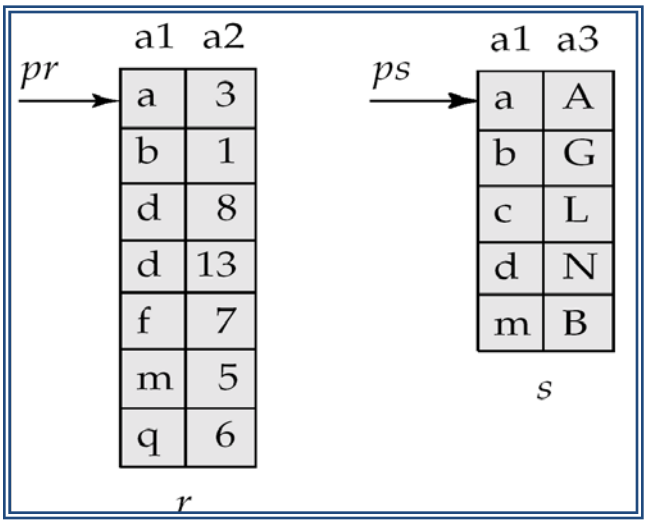
✓ If smaller relation (depositor) fits entirely in memory, the cost estimate will be 500 block transfers.

✓ **Example of Nested-Loop Join Costs**

- Compute depositor customer, with depositor as the outer relation.
- Let customer have a primary B^+-tree index on the join attribute customer-name, which contains 20 entries in each index node.
- Since customer has 10,000 tuples, the height of the tree is 4, and one more access is needed to find the actual data
- depositor has 5000 tuples
- Cost of block nested loops join
  - 400*100 + 100 = 40,100 block transfers + 2 * 100 = 200 seeks
    - worst case memory
    - be significantly less with more memory
- Cost of indexed nested loops join
  - + 5000 * 5 = 25,100 block transfers and seeks.
  - CPU cost likely to be less than that for block nested loops join

Merge-Join

- Sort both relations on their join attribute (if not already sorted on the join attributes).
- Merge the sorted relations to join them
  - Join step is similar to the merge stage of the sort-merge algorithm.
  - Main difference is handling of duplicate values in join attribute — every pair with same value on join attribute must be matched
  - Detailed algorithm in book



- Can be used only for equi-joins and natural joins
- Each block needs to be read only once (assuming all tuples for any given value of the join attributes fit in memory
- Thus the cost of merge join is:

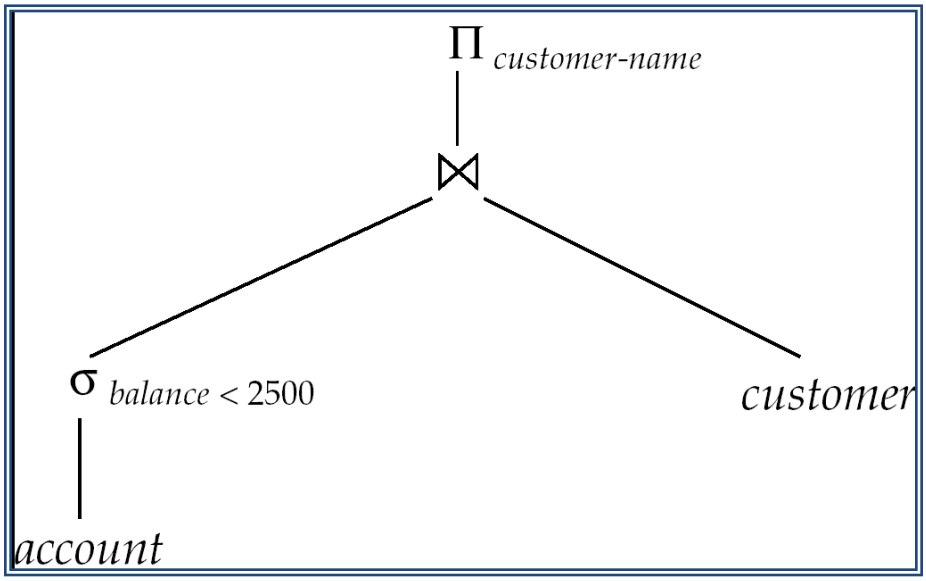
  $b_r + b_s$ block transfers $+\lceil b_r / b_b \rceil + \lceil b_s / b_b \rceil$ seeks
  - \+ The cost of sorting if relations are unsorted.
- **hybrid merge-join:** If one relation is sorted, and the other has a secondary B^+-tree index on the join attribute
  - Merge the sorted relation with the leaf entries of the B^+-tree.
  - Sort the result on the addresses of the unsorted relation's tuples
  - Scan the unsorted relation in physical address order and merge with previous result, to replace addresses by the actual tuples
    - Sequential scan more efficient than random lookup

Hash-Join

- ✓ Applicable for equi-joins and natural joins.
- ✓ A hash function h is used to partition tuples of both relations
- ✓ h maps JoinAttrs values to {0, 1, ..., n}, where JoinAttrs denotes the common attributes of r and s used in the natural join.
  - $r_0, r_1, \ldots, r_n$ denote partitions of r tuples
    - Each tuple $t_r \in r$ is put in partition r_i where $i = h(t_r[\text{JoinAttrs}])$.
  - $r_0,, r_1. \ldots, r_n$ denotes partitions of s tuples
    - Each tuple $t_s \in s$ is put in partition s_i, where $i = h(t_s[\text{JoinAttrs}])$.
- ✓ Note: In book, r_i is denoted as $H_{ri,}$ s_i is denoted as H_{si} and n is denoted as $n_{h.}$
- ✓ *r* tuples in r_i need only to be compared with *s* tuples in s_i Need not be compared with *s* tuples in any other partition, since:
  - an *r* tuple and an *s* tuple that satisfy the join condition will have the same value for the join attributes.
  - If that value is hashed to some value *i*, the *r* tuple has to be in r_i and the *s* tuple in s_i.
- ✓ **Example of Cost of Hash-Join: customer $\bowtie$ depositor**
  - Assume that memory size is 20 blocks
  - $b_{depositor}$= 100 and $b_{customer}$ = 400.
  - Depositor is to be used as build input. Partition it into five partitions, each of size 20 blocks. This partitioning can be done in one pass.
  - Similarly, partition customer into five partitions, each of size 80. This is also done in one pass.
  - Therefore total cost, ignoring cost of writing partially filled blocks:
    - 3(100 + 400) = 1500 block transfers $+2(\lceil 100/3 \rceil + \lceil 400/3 \rceil) = 336$ seeks

5.4 Evaluation of Expressions

- ✓ The algorithms shown so far pertain to individual relational algebra operations
- ✓ In reality, a typical query combines them — for example, select customer_name from account natural inner join customer where balance < 2500 consists of a selection, a natural join, and a projection
- ✓ Two general approaches for this:
  - Materialization
  - Pipelining
- ✓ As with the other algorithms, one is general but expensive, while the other is more efficient but doesn't apply to all cases.
- ✓ **Materialization**
  - Execute a single operation at a time which generates a temporary file that will be used as an input to the next operation.
  - Time consuming approach because it will generate and sort many temporary files.
  - Materialized evaluation walks the parse or expression tree of the relational algebra operation, and performs the innermost or leaf-level operations first
  - The intermediate result of each operation is materialized — an actual, but temporary, relation —and becomes input for subsequent operations
  - The cost of materialization is the sum of the individual operations plus the cost of writing the intermediate results to disk — a function of the blocking factor (number of records per block) of the temporaries
  - E.g., in figure below, compute and store $\sigma_{balance<2500}$(account); then compute and store its join with customer, and finally compute the projection on customer-name.



- Materialized evaluation is always applicable
- Cost of writing results to disk and reading them back can be quite high
    - Our cost formulas for operations ignore cost of writing results to disk, so
        - Overall cost = Sum of costs of individual operations + cost of writing intermediate results to disk
- Double buffering: use two output buffers for each operation, when one is full write it to disk while the other is getting filled
    - Allows overlap of disk writes with computation and reduces execution time

✓ **Pipelining**

- The problem with materialization is just that — lots of temporary files, lots of I/O
- With pipelined evaluation, operations form a queue, and results are passed from one operation to another as they are calculated, hence the technique's name
- Avoids write-outs of entire intermediate relations
- General approach: restructure the individual operation algorithms so that they take streams of tuples as both input and output
- As the result tuples from one operation are produced, they are provided as input for parent operations. So no need to store temporary files to disk.
- E.g., in previous expression tree, don't store result of instead, pass tuples directly to the join. Similarly, don't store result of join, pass tuples directly to projection.
- Much cheaper than materialization: no need to store a temporary relation to disk.
- Pipelining may not always be possible – e.g., sort, hash-join.
- For pipelining to be effective, use evaluation algorithms that generate output tuples even as tuples are received for inputs to the operation. Pipelines can be executed in two ways: **demand driven** and **producer driven**
- In **demand driven** or **lazy** evaluation
    - system repeatedly requests next tuple from top level operation
    - Each operation requests next tuple from children operations as required, in order to output its next tuple
    - In between calls, operation has to maintain "**state**" so it knows what to return next
- In **producer-driven** or **eager** pipelining
    - Operators produce tuples eagerly and pass them up to their parents
        - Buffer maintained between operators, child puts tuples in buffer, parent removes tuples from buffer
        - if buffer is full, child waits till there is space in the buffer, and then generates more tuples
    - System schedules operations that have space in output buffer and can process more input tuples
- Alternative name: **pull** and **push** models of pipelining

5.4 Transformation of Relational Expressions

- ✓ Generation of query-evaluation plans for an expression involves several steps:
  - Generating logically equivalent expressions
  - Use equivalence rules to transform an expression into an equivalent one.
  - Annotating resultant expressions to get alternative query plans
  - Choosing the cheapest plan based on estimated cost
- ✓ The overall process is called cost based optimization.
- ✓ **Equivalence of Expressions**: Relations generated by two equivalent expressions have the same set of attributes and contain the same set of tuples, although their attributes may be ordered differently.
- ✓ **Equivalence Rules:** An equivalence rule says that expressions of two forms are equivalent

1 Conjunctive selection operations can be deconstructed into a sequence of individual selections.

$$\sigma_{\theta_1 \wedge \theta_2}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$$

2 Selection operations are commutative.

$$\sigma_{\theta_1}(\sigma_{\theta_2}(E)) = \sigma_{\theta_2}(\sigma_{\theta_1}(E))$$

3 Only the last in a sequence of projection operations is needed, the others can be omitted.

$$\Pi_{t_1}(\Pi_{t_2}(\ldots(\Pi_{tn}(E))\ldots)) = \Pi_{t_1}(E)$$

4 Selections can be combined with Cartesian products and theta joins.

- $\sigma_\theta(E_1 \text{ X } E_2) = E_1 \bowtie_\theta E_2$
- $\sigma_{\theta 1}(E_1 \bowtie_{\theta 2} E_2) = E_1 \bowtie_{\theta 1 \wedge \theta 2} E_2$

5 Theta-join operations (and natural joins) are commutative.

$$E_1 \bowtie_\theta E_2 = E_2 \bowtie_\theta E_1$$

6 a) Natural join operations are associative:

$$(E_1 \bowtie E_2) \bowtie E_3 = E_1 \bowtie (E_2 \bowtie E_3)$$

b) Theta joins are associative in the following manner:

$$(E_1 \bowtie_{\theta 1} E_2) \bowtie_{\theta 2 \wedge \theta 3} E_3 = E_1 \bowtie_{\theta 2 \wedge \theta 3} (E_2 \bowtie_{\theta 2} E_3)$$

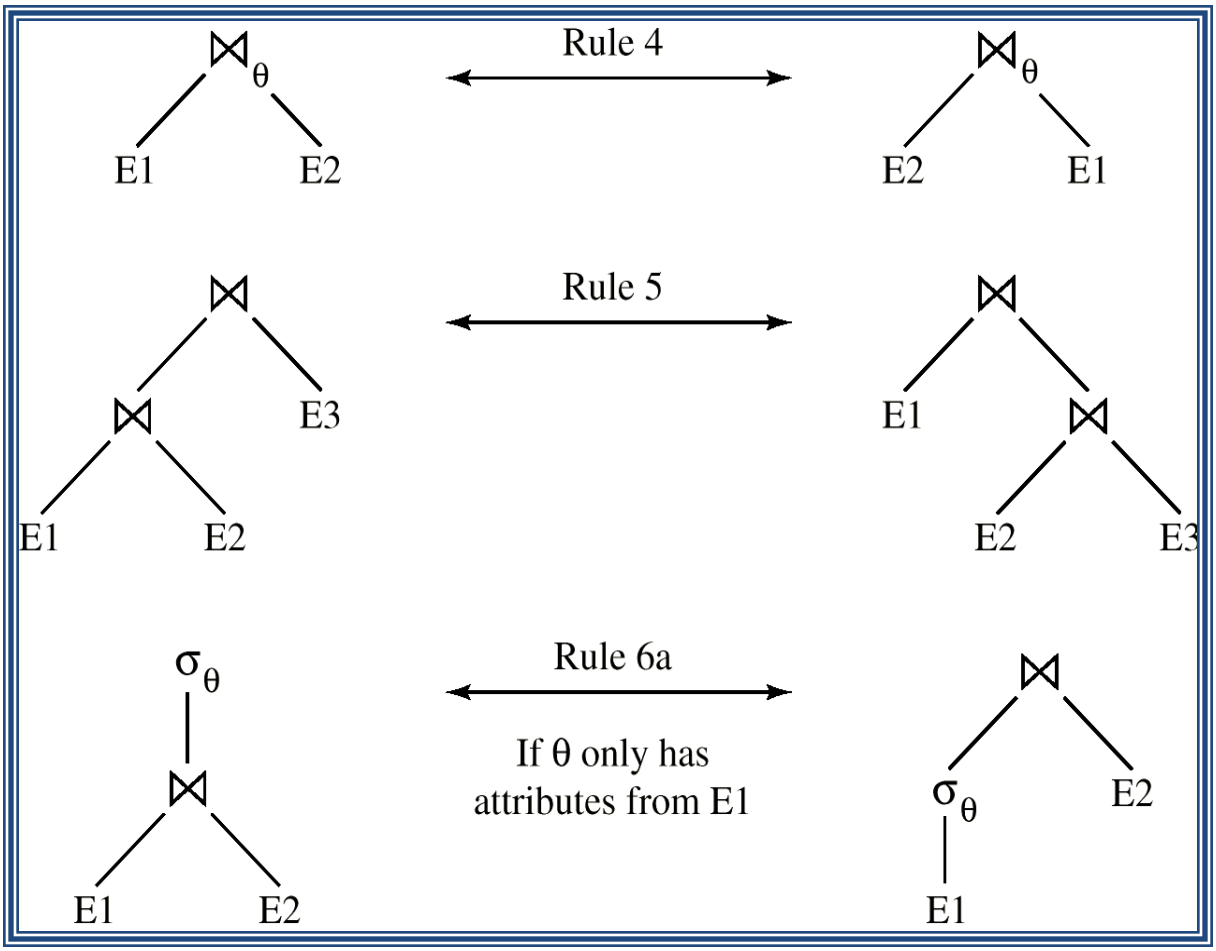
where θ_2 involve attributes from only E_2 and E_3.

7 The selection operation distributes over the theta join operation under the Following two conditions:

(a) When all the attributes in θ_0 involve only the attributes of one of the expressions (E_1) being joined.

$$\sigma_{\theta 0}(E_1 \bowtie_\theta E_2) = (\sigma_{\theta 0}(E_1)) \bowtie_\theta E_2$$

b) When θ_1 involves only the attributes of E_1 and θ_2 involves only the attributes of E_2. $\sigma_{\theta 1 \wedge \theta 2}(E_1 \bowtie_\theta E_2) = (\sigma_{\theta 1}(E_1)) \bowtie_\theta (\sigma_{\theta 2}(E_2))$



8 The projections operation distributes over the theta join operation as follows:

a) if P involves only attributes from $L_1 \cup L_2$:

$$\prod_{L_1 \cup L_2}(E_{1 \ldots\ldots \theta} E_2) = (\prod_{L_1}(E_1))_{\ldots\ldots\theta}(\prod_{L_2}(E_2))$$

b) Consider a join $E_1 \bowtie_\theta E_2$.

- Let L_1 and L_2 be sets of attributes from E_1 and E_2, respectively.
- Let L_3 be attributes of E_1 that are involved in join condition θ, but are not in $L_1 \cup L_2$, and
- Let L_4 be attributes of E_2 that are involved in join conditionθ, but are not in $L_1 \cup L_2$.

$$\prod_{L_1 \cup L_2}(E_1 \ldots\ldots_\theta E_2) = \prod_{L_1 \cup L_2}((\prod_{L_1 \cup L_3}(E_1))_{\ldots\ldots\theta}(\prod_{L_2 \cup L_4}(E_2)))$$

9. The set operations union and intersection are commutative

$E_1 \cup E_2 = E_2 \cup E_1$

$E_1 \cap E_2 = E_2 \cap E_1$

(Set difference is not commutative).

10. Set union and intersection are associative.

$$(E_1 \cup E_2) \cup E_3 = E_1 \cup (E_2 \cup E_3)$$

$$(E_1 \cap E_2) \cap E_3 = E_1 \cap (E_2 \cap E_3)$$

11. The selection operation distributes over $\cup$, $\cap$ and $-$.

$$\sigma_\theta (E_1 - E_2) = \sigma_\theta (E_1) - \sigma_\theta(E_2)$$

and similarly for $\cup$ and $\cap$ in place of $-$

Also: $\sigma_\theta (E_1 - E_2) = \sigma_\theta(E_1) - E_2$

and similarly for $\cap$ in place of $-$, but not for $\cup$

12. The projection operation distributes over union

$$\Pi_L(E_1 \cup E_2) = (\Pi_L(E_1)) \cup (\Pi_L(E_2))$$

Transformation Example

- ✓ Query: Find the names of all customers who have an account at some branch located in KTM. $\Pi_{\text{customer-name}}(\sigma_{\text{branch-city = "KTM"}}(\text{branch} \bowtie (\text{account} \bowtie \text{depositor})))$

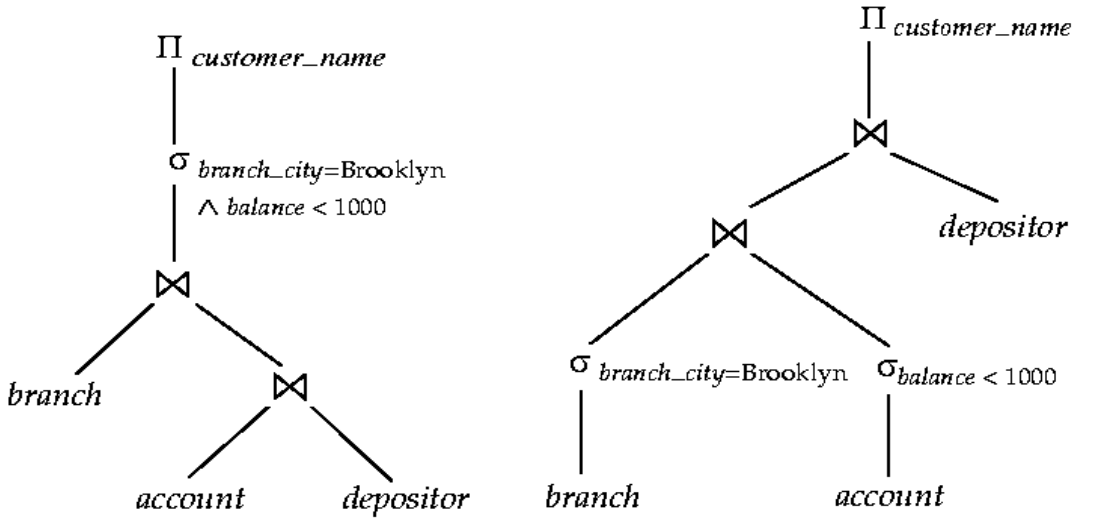
  Transformation using rule 7a.

  $\Pi_{\text{customer-name}} ((\sigma_{\text{branch-city = "KTM"}} (\text{branch})) \bowtie (\text{account} \bowtie \text{depositor}))$
- ✓ Find the names of all customers with an account at a KTM city whose account balance is over 1000.

  $\Pi_{\text{customer-name}}((\sigma_{\text{branch-city = "KTM"} \wedge \text{ balance > 1000}} (\text{branch} \bowtie (\text{account} \bowtie \text{depositor})))$

  Transformation using join associatively (Rule 6a)

  $\Pi_{\text{customer-name}}((\sigma_{\text{branch-city = "KTM"} \wedge \text{ balance > 1000}} (\text{branch} \bowtie (\text{account})) \bowtie \text{depositor})$
- ✓ Second form provides an opportunity to apply the "perform selections early" rule, resulting in the sub expression
  - $\sigma_{\text{branch-city = "KTM"}} (\text{branch}) \bowtie \sigma_{\text{balance > 1000}} (\text{account}$



(a) Initial expression tree (b) Tree after multiple transformations

5.5 Choice of Evaluation Plans

- ✓ Must consider the interaction of evaluation techniques when choosing evaluation plans
  - Choosing the cheapest algorithm for each operation independently may not yield best overall algorithm. E.g.
    - ♦ Merge-join may be costlier than hash-join, but may provide a sorted output which reduces the cost for an outer level aggregation.
    - ♦ Nested-loop join may provide opportunity for pipelining
- ✓ Practical query optimizers incorporate elements of the following two broad approaches:
  - Search all the plans and choose the best plan in a cost-based fashion.
  - Uses heuristics to choose a plan.
- ✓ **Cost-Based Optimization**
  - Consider finding the best join-order for $r_1 \bowtie r_2 \bowtie \ldots r_n$.
  - There are $(2(n-1))!/(n-1)!$ Different join orders for above expression. With $n = 7$, the number is 665280, with $n = 10$, the number is greater than 176 billion!
  - No need to generate all the join orders. Using dynamic programming, the least-cost join order for any subset of $\{r_1, r_2 \ldots r_n\}$ is computed only once and stored for future use.
- ✓ **Cost of Optimization**
  - With dynamic programming time complexity of optimization with bushy trees is $O(3^n)$.
    - ♦ With $n = 10$, this number is 59000 instead of 176 billion!
  - Space complexity is $O(2^n)$
  - To find best left-deep join tree for a set of n relations:
    - ♦ Consider n alternatives with one relation as right-hand side input and the other relations as left-hand side input.
    - ♦ Modify optimization algorithm:
      - Replace "**for each** non-empty subset $S1$ of S such that $S1 \neq S$"
      - By: **for each** relation r in S
        let S1 = S – r .
  - If only left-deep trees are considered, time complexity of finding best join order is $O(n\, 2^n)$
    - ♦ Space complexity remains at $O(2^n)$
  - Cost-based optimization is expensive, but worthwhile for queries on large datasets (typical queries have small n, generally < 10)

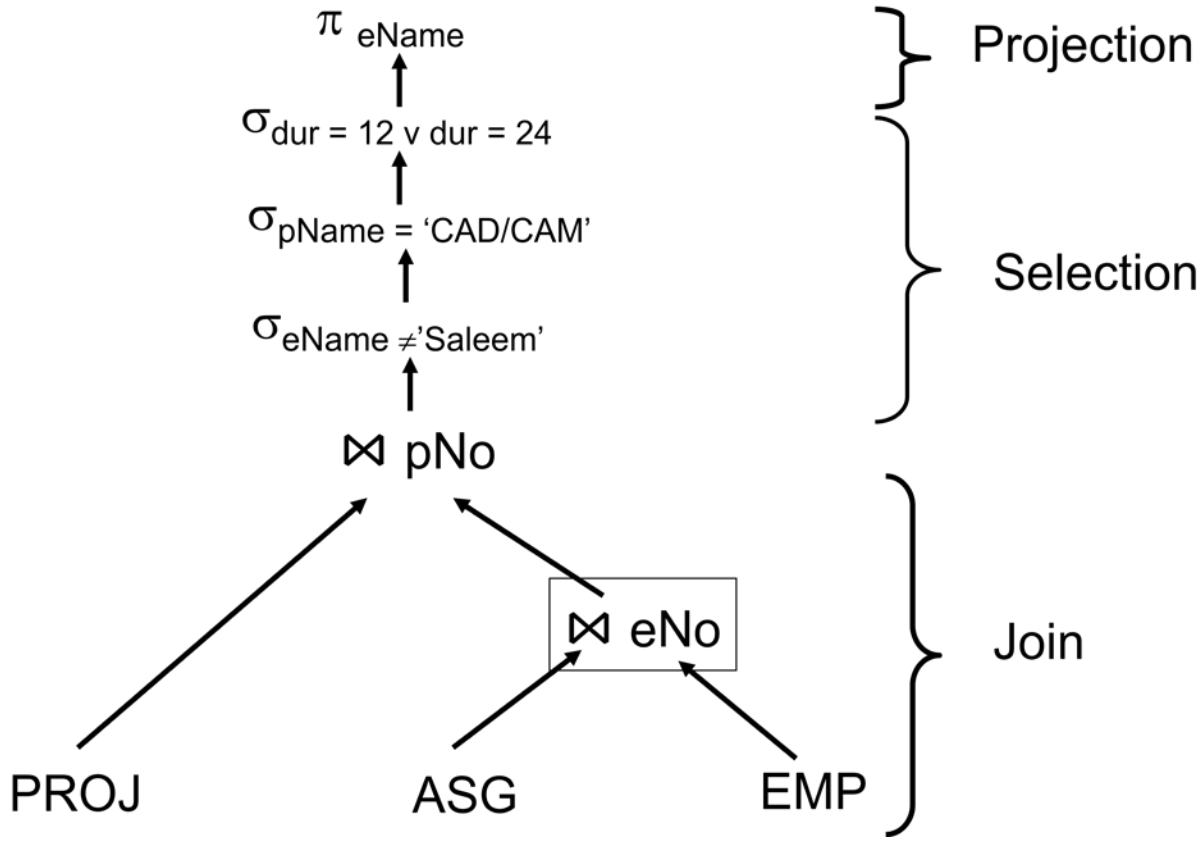
- ✓ **Heuristic Optimization**
  - Cost-based optimization is expensive, even with dynamic programming.
  - Systems may use *heuristics* to reduce the number of choices that must be made in a cost-based fashion.
  - Heuristic optimization transforms the query-tree by using a set of rules that typically (but not in all cases) improve execution performance:
    - ♦ Perform selection early (reduces the number of tuples)
    - ♦ Perform projection early (reduces the number of attributes)
    - ♦ Perform most restrictive selection and join operations (i.e. with smallest result size) before other similar operations.
    - ♦ Some systems use only heuristics; others combine heuristics with partial cost-based optimization.

5.6 Query Decomposition

- ✓ Transforms an SQL (relational calculus) query into relational algebra query
- ✓ Both are on global relations in DDBS-
- ✓ Steps in QD
  - Normalization
  - Analysis
  - Elimination of Redundancy
  - Rewriting
- ✓ Normalization
  - Input Query can be complex
  - Lexical and syntactic Analysis (like compilers)
  - Treatment of WHERE Clause
  - Two possible forms-
    - ♦ Conjunctive NF-
      - $(p_{11} v P_{12} v \ldots . v p_{1n}) \wedge \ldots \wedge (p_{m1} v p_{m2} v \ldots . v p_{mn})$
    - ♦ Disjunctive NF
      - $(p_{11} \wedge P_{12} \wedge \ldots . \wedge p_{1n})$ v…v$(p_{m1} \wedge p_{m2} \wedge \ldots . \wedge p_{mn})$
  - Transformation is based on equivalence rules
  - Example:
    - ♦ SELECT eName FROM EMP, ASG
      WHERE EMP.eNo = ASG.eNo AND ASG.pNo = ‘P1’AND
      dur = 12 OR dur = 24
    - ♦ Qualification in Con NF
      - EMP.eNo = ASG.eNo ^ ASG.pNo = ‘P1’ ^(dur = 12 v dur = 24)
    - ♦ Qualification in Dis NF
      - (EMP.eNo = ASG.eNo ^ASG.pNo = ‘P1’ ^ dur = 12)
        v
      - (EMP.eNo = ASG.eNo ^ASG.pNo = ‘P1’ ^ dur = 24)

- ✓ Analysis
  - Reject incorrect ones
  - Incorrect type
    - ♦ Relations/attributes not exist
    - ♦ Wrong operations.
  - Semantically Incorrect
    - ♦ Components do not contribute in result
    - ♦ Detection possible in certain cases; not contain disjunction or negation
    - ♦ Query graph and join graph
- ✓ Elimination of Redundancy
  - Expression replacement already used in views
  - User mistake or this replacement may contain redundant predicates
  - Simplification on idempotency rules
    - ♦ $p \wedge p \Leftrightarrow p$ $p \vee p \Leftrightarrow p$
    - ♦ $p \wedge true \Leftrightarrow p$ $p \vee false \Leftrightarrow p$
    - ♦ $p \wedge false \Leftrightarrow false$
    - ♦ $p \vee true \Leftrightarrow true$ $p \vee \neg p \Leftrightarrow true$
- ✓ Rewriting
  - Transforming SQL to Relational Algebra straightaway
  - Restructuring operators to improve efficiency
  - Operator Tree is used
    - ♦ Leaf nodes are operand relations
    - ♦ Non leaf are intermediate tables produced as a result of some relational operators
    - ♦ Example Select eName

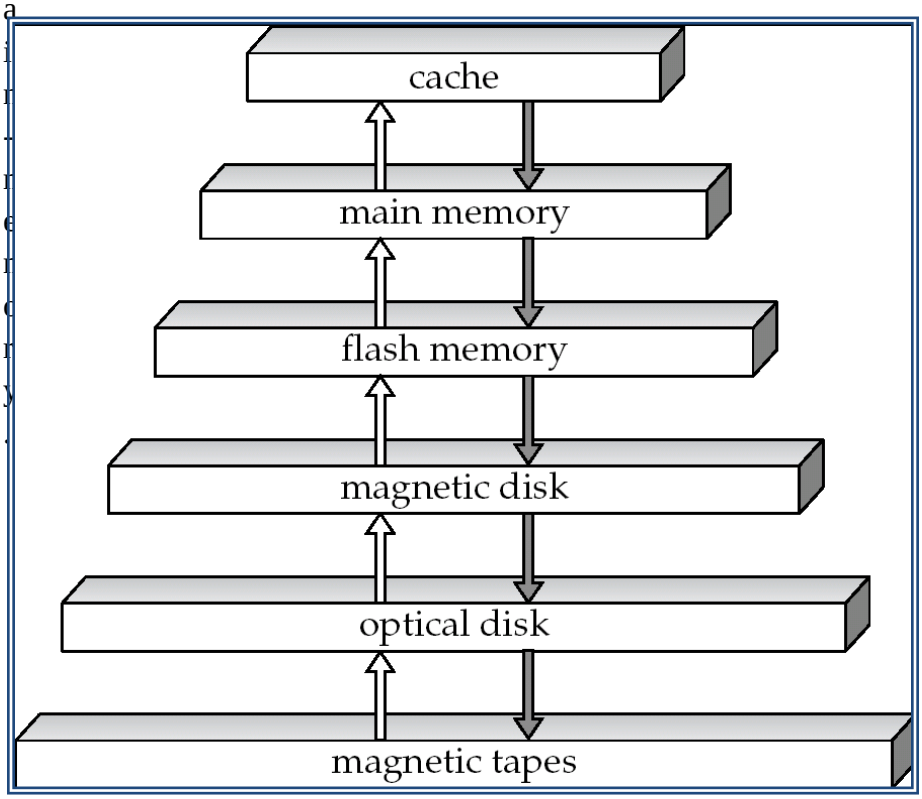
```
FROM    EMP, ASG, PROJ
WHERE   EMP.eNo = ASG.eNo
AND     ASG.pNo = PROJ.pNo
AND     eName ≠ 'Saleem'
AND     pName = 'CAD/CAM'
AND     (dur = 36 or dur = 24)
```


π eName
σdur = 12 v dur = 24
σpName = 'CAD/CAM'
σeName ≠'Saleem'
⋈ pNo
⋈ eNo
PROJ
ASG
EMP
Projection
Selection
Join

Chapter 6: File Structure and Hashing

6.1 Physical Storage Media

- ✓ Differentiate storage into two category:
  - volatile storage: loses contents when power is switched off
  - non-volatile storage:
    - ♦ Contents persist even when power is switched off.
    - ♦ Includes secondary and tertiary storage, as well as batter- backed up m a i n - m e m o r y



Storage Hierarchy

Storage Type

- ✓ **Primary storage:**
  - Fastest media but volatile (cache, main memory).
- ✓ **Secondary storage:**
  - next level in hierarchy, non-volatile, moderately fast access time
  - also called on-line storage
  - E.g. flash memory, magnetic disks
- ✓ **Tertiary storage:**
  - lowest level in hierarchy, non-volatile, slow access time
  - also called off-line storage
  - E.g. magnetic tape, optical storage

Cache :

- ✓ Fastest and most costly form of storage; volatile; managed by the computer system hardware.

Main memory:

- ✓ Fast access (10s to 100s of nanoseconds; 1 nanosecond = 10^{-9} seconds)
- ✓ Generally too small (or too expensive) to store the entire database
- ✓ Capacities of up to a few Gigabytes widely used currently
- ✓ Capacities have gone up and per-byte costs have decreased steadily and rapidly (roughly factor of 2 every 2 to 3 years)
- ✓ Volatile — contents of main memory are usually lost if a power failure or system crash occurs.

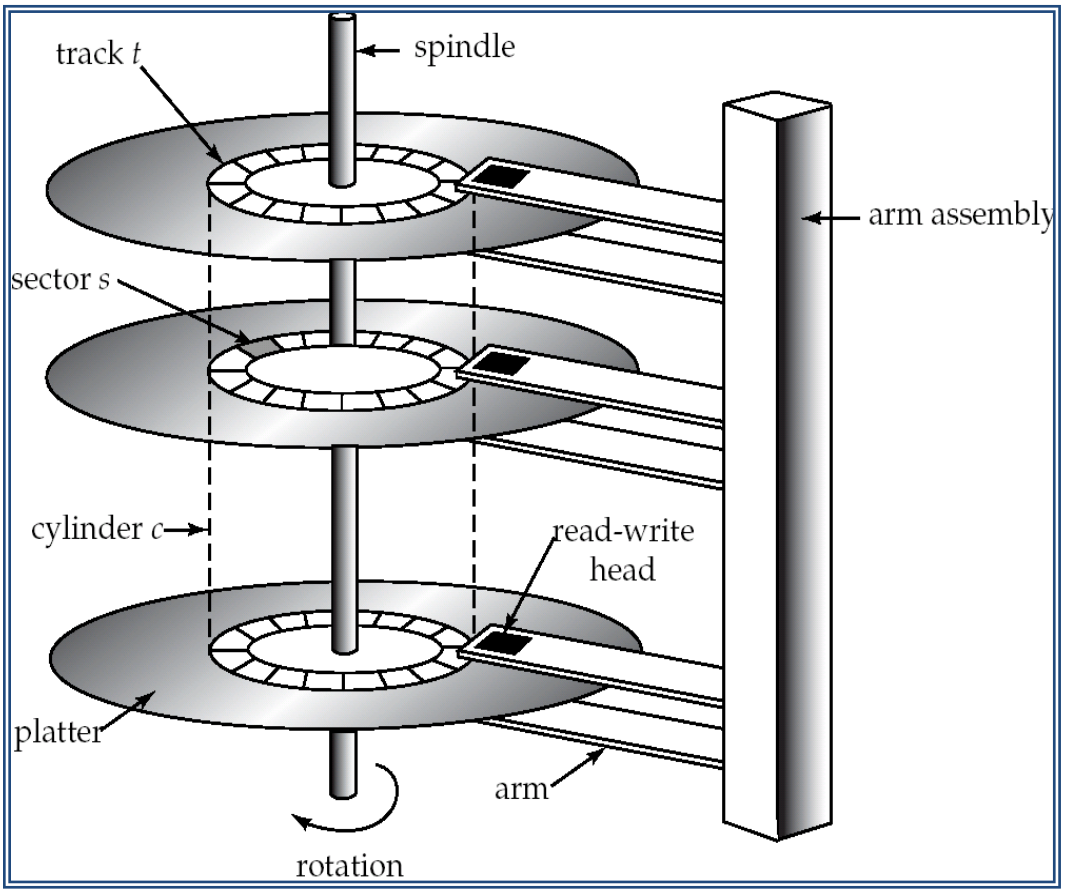
Flash memory:

- ✓ Data survives power failure
- ✓ Data can be written at a location only once, but location can be erased and written to again
- ✓ Can support only a limited number (10K – 1M) of write/erase cycles.
- ✓ Erasing of memory has to be done to an entire bank of memory
- ✓ Reads are roughly as fast as main memory
- ✓ But writes are slow (few microseconds), erase is slower
- ✓ Cost per unit of storage roughly similar to main memory
- ✓ Widely used in embedded devices such as digital cameras
- ✓ Is a type of EEPROM (Electrically Erasable Programmable Read-Only Memory)

Magnetic-disk

- ✓ Data is stored on spinning disk, and read/written magnetically
- ✓ Primary medium for the long-term storage of data; typically stores entire database.
- ✓ Data must be moved from disk to main memory for access, and written back for storage
- ✓ Much slower access than main memory
- ✓ **direct-access** – possible to read data on disk in any order, unlike magnetic tape
- ✓ Capacities range up to roughly 1000 GB currently
  - Much larger capacity and cost/byte than main memory/flash memory
  - Growing constantly and rapidly with technology improvements (factor of 2 to 3 every 2 years)
- ✓ Survives power failures and system crashes
  - disk failure can destroy data, but is rare
- ✓ **Read-write head**
  - Positioned very close to the platter surface (almost touching it)
  - Reads or writes magnetically encoded information.

- ✓ Surface of platter divided into circular **tracks**
  - Over 50K-100K tracks per platter on typical hard disks



- ✓ Each track is divided into **sectors.**
  - A sector is the smallest unit of data that can be read or written.
  - Sector size typically 512 bytes
  - Typical sectors per track: 500 (on inner tracks) to 1000 (on outer tracks)
- ✓ To read/write a sector
  - disk arm swings to position head on right track
  - platter spins continually; data is read/written as sector passes under head
- ✓ Head-disk assemblies
  - multiple disk platters on a single spindle (1 to 5 usually)
  - One head per platter, mounted on a common arm.
- ✓ **Cylinder** i consists of i^{th} track of all the platters
- ✓ **Disk controller** – interfaces between the computer system and the disk drive hardware.
  - accepts high-level commands to read or write a sector
  - initiates actions such as moving the disk arm to the right track and actually reading or writing the data
  - Computes and attaches **checksums** to each sector to verify that data is read back correctly

        - If data is corrupted, with very high probability stored checksum won't match recomputed checksum
    - Ensures successful writing by reading back sector after writing it
    - Performs remapping of bad sectors

- ✓ **Performance Measures of Disks**
    - **Access time** – the time it takes from when a read or write request is issued to when data transfer begins. Consists of:
    - **Seek time** – time it takes to reposition the arm over the correct track.
        - Average seek time is 1/2 the worst case seek time.
            - Would be 1/3 if all tracks had the same number of sectors, and we ignore the time to start and stop arm movement
        - 4 to 10 milliseconds on typical disks
    - **Rotational latency** – time it takes for the sector to be accessed to appear under the head.
        - Average latency is 1/2 of the worst case latency.
        - 4 to 11 milliseconds on typical disks (5400 to 15000 r.p.m.)
    - **Data-transfer rate** – the rate at which data can be retrieved from or stored to the disk.
        - 25 to 100 MB per second max rate, lower for inner tracks
        - Multiple disks may share a controller, so rate that controller can handle is also important
            - E.g. ATA-5: 66 MB/sec, SATA: 150 MB/sec, Ultra 320 SCSI: 320 MB/s
            - Fiber Channel (FC2Gb): 256 MB/s

Optical storage

- ✓ non-volatile, data is read optically from a spinning disk using a laser
- ✓ CD-ROM (640 MB) and DVD (4.7 to 17 GB) most popular forms
- ✓ Write-one, read-many (WORM) optical disks used for archival storage (CD-R, DVD-R, DVD+R)
- ✓ Multiple write versions also available (CD-RW, DVD-RW, DVD+RW, and DVD-RAM)
- ✓ Reads and writes are slower than with magnetic disk
- ✓ Juke-box systems, with large numbers of removable disks, a few drives, and a mechanism for automatic loading/unloading of disks available for storing large volumes of data

Tape storage

- ✓ non-volatile, used primarily for backup (to recover from disk failure), and for archival data
- ✓ sequential-access – much slower than disk
- ✓ very high capacity (40 to 300 GB tapes available)
- ✓ tape can be removed from drive ⇒ storage costs much cheaper than disk, but drives are expensive
- ✓ Tape jukeboxes available for storing massive amounts of data
    - hundreds of terabytes (1 terabyte = 109 bytes) to even a petabyte (1 petabyte = 1012 bytes)

6.2 File Organization

- ✓ The database is stored as a collection of files. Each file is a sequence of records. A record is a sequence of fields.
- ✓ Mainly two type of Records
  - Fixed –Length Records
  - Variable –Length Records

Fixed –Length Records

- ✓ assume record size is fixed
- ✓ each file has records of one particular type only
- ✓ different files are used for different relations
- ✓ This case is easiest to implement
- ✓ Store record *i* starting from byte $n * (i - 1)$, where *n* is the size of each record.
- ✓ Record access is simple but records may cross blocks

record 0	A-102	Perryridge	400
record 1	A-305	Round Hill	350
record 2	A-215	Mianus	700
record 3	A-101	Downtown	500
record 4	A-222	Redwood	700
record 5	A-201	Perryridge	900
record 6	A-217	Brighton	750
record 7	A-110	Downtown	600
record 8	A-218	Perryridge	700

- ✓ Two problems:
  - Difficult to delete a record from the structure
    - ♦ Deleted space must be filled by another record or
    - ♦ Marking deleted records so that they can be ignored
  - Block size must be a multiple of n(file size).Some record cross the boundaries. So it requires two block access for a record
- ✓ Approaches for deletion :
  - Move records i + 1, . . ., n to i, . . . , n – 1 (Requires more access to move the record)
  - Don't move records, but link all free records on a **free list**
  - Store the address of the first deleted record in the **file header**.
  - this first record to store the address of the second deleted record, and so on

record 0	A-102	Perryridge	400
record 1	A-305	Round Hill	350
record 3	A-101	Downtown	500
record 4	A-222	Redwood	700
record 5	A-201	Perryridge	900
record 6	A-217	Brighton	750
record 7	A-110	Downtown	600
record 8	A-218	Perryridge	700

Record 2 Deleted and All Records Moved

header				
record 0	A-102	Perryridge	400	
record 1				
record 2	A-215	Mianus	700	
record 3	A-101	Downtown	500	
record 4				
record 5	A-201	Perryridge	900	
record 6				
record 7	A-110	Downtown	600	
record 8	A-218	Perryridge	700	

- ✓ Can think of these stored addresses as pointers since they "point" to the location of a record.
- ✓ More space efficient representation: reuse space for normal attributes of free records to store pointers. (No pointers stored in in-use records.)

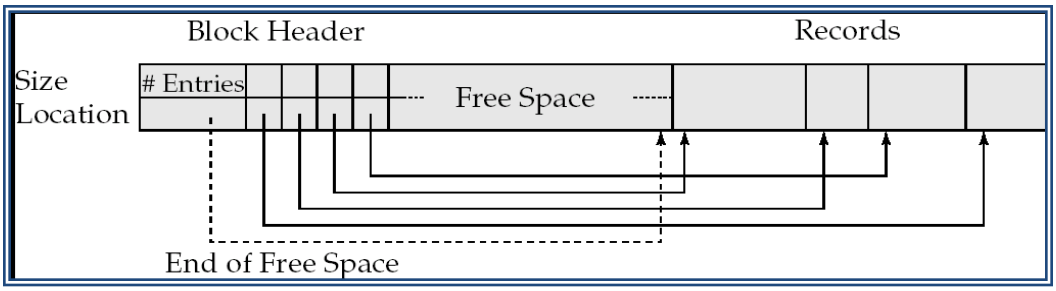
- ✓ On new insertion of record we use the record pointed by file header and change the header pointer to point the next available record. If no space available, add new record at the end of file.

Variable-Length Records

- ✓ Variable-length records arise in database systems in several ways:
  - Storage of multiple record types in a file.
  - Record types that allow variable lengths for one or more fields.
  - Record types that allow repeating fields (used in some older data models).
- ✓ Mainly two approaches
  - Byte-String Representation
  - Fixed-Length Representation
- ✓ **Byte-String Representation**
  - Attach a special end-of-record(⊥) symbol to the end of record
  - Disadvantage:
    - ♦ Not easy to reuse space occupied by deleted record
    - ♦ No space for records to grow longer. If the records becomes longer it must be moved
  - Modified form of byte-string representation called slotted page structure.
  - Slotted page header contains:
    - ♦ Number of record entries
    - ♦ End of free space in the block
    - ♦ Location and size of each record

0	Perryridge	A-102	400	A-201	900	A-218	700	⊥
1	Round Hill	A-305	350	⊥				
2	Mianus	A-215	700	⊥				
3	Downtown	A-101	500	A-110	600	⊥		
4	Redwood	A-222	700	⊥				
5	Brighton	A-217	750	⊥				

Byte-String Representation



Slotted page structure

- Records can be moved around within a page to keep them contiguous with no empty space between them; entry in the header must be updated.
- Pointers should not point directly to record — instead they should point to the entry for the record in header.

✓ **Fixed-Length Representation**

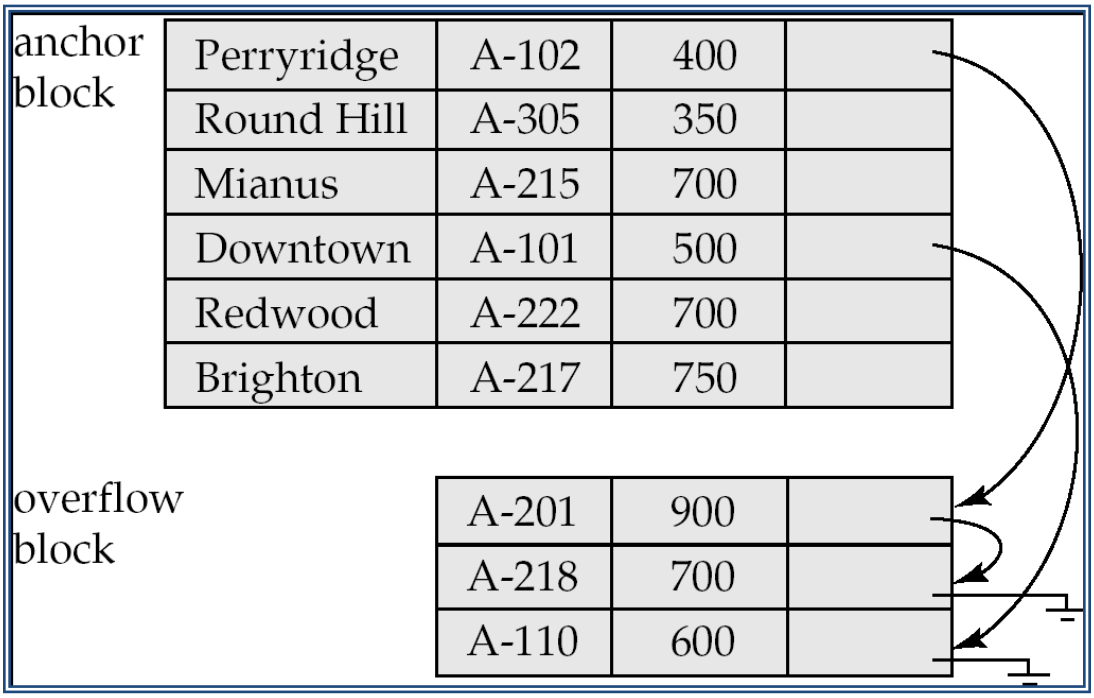
- Use one or more fixed length records:
  - **Reserved space** : can use fixed-length records of a known maximum length; unused space in shorter records filled with a null
  - **List Representation** : can use fixed-length records of a known maximum length; unused space in shorter records filled with a null

0	Perryridge	A-102	400	A-201	900	A-218	700
1	Round Hill	A-305	350	⊥	⊥	⊥	⊥
2	Mianus	A-215	700	⊥	⊥	⊥	⊥
3	Downtown	A-101	500	A-110	600	⊥	⊥
4	Redwood	A-222	700	⊥	⊥	⊥	⊥
5	Brighton	A-217	750	⊥	⊥	⊥	⊥

Reserved space method

0	Perryridge	A-102	400	
1	Round Hill	A-305	350	
2	Mianus	A-215	700	
3	Downtown	A-101	500	
4	Redwood	A-222	700	
5		A-201	900	
6	Brighton	A-217	750	
7		A-110	600	
8		A-218	700	

- ✓ Disadvantage to pointer structure; space is wasted in all records except the first in a chain.
- ✓ Solution is to allow two kinds of block in file:
  - Anchor block – contains the first records of chain
  - Overflow block – contains records other than those that are the first records of chairs.

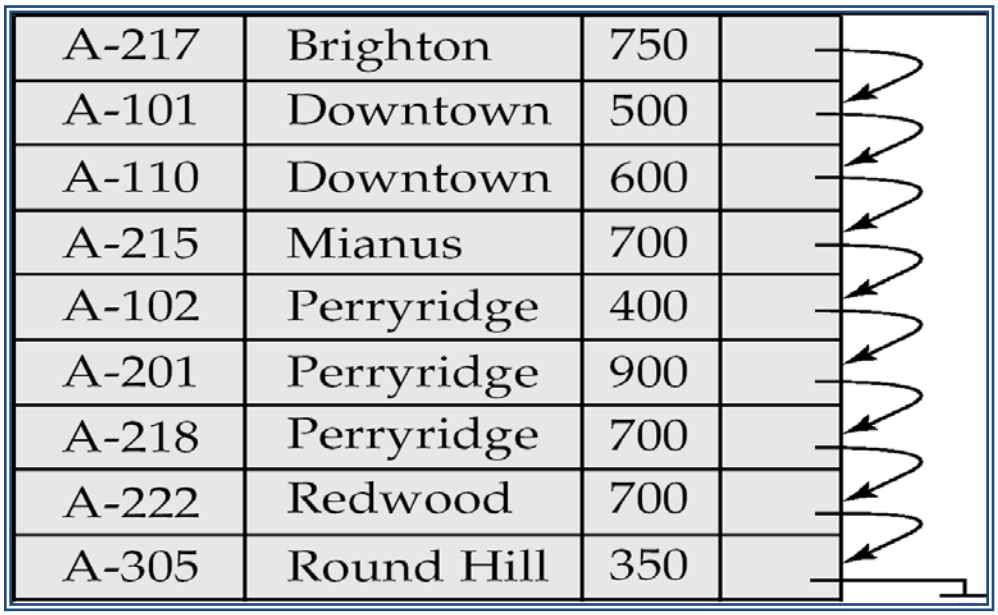


Anchor block & Overflow block

6.3 Organization of Records in Files

- ✓ **Heap file Organization**
  - A record can be placed anywhere in the file where there is space
  - No ordering of records
  - Single relation for each relation
- ✓ **Hashing file Organization**
  - A hash function computed on some attribute of each record
  - The result specifies in which block of the file the record should be placed
- ✓ **Sequential file Organization**
  - Store records in sequential order, based on the value of the search key of each record

- ✓ **Sequential File Organization**
  - Designed for efficient processing of records in sorted order based on search key.
  - Suitable for applications that require sequential processing of the entire file
  - The records in the file are ordered by a search-key. A search key is any attributes or set of attributes.
  - For fast retrieval of records in search key order: records are linked together by pointer.



A-217	Brighton	750	
A-101	Downtown	500	
A-110	Downtown	600	
A-215	Mianus	700	
A-102	Perryridge	400	
A-201	Perryridge	900	
A-218	Perryridge	700	
A-222	Redwood	700	
A-305	Round Hill	350	

- ✓ For Insertion follow the following rule: –locate the position where the record is to be inserted
  - If there is free space insert there
  - If no free space, insert the record in an overflow block
  - In either case, pointer chain must be updated
- ✓ For Deletion – use pointer chains
- ✓ Need to reorganize the file from time to time to restore sequential order

A-217	Brighton	750	
A-101	Downtown	500	
A-110	Downtown	600	
A-215	Mianus	700	
A-102	Perryridge	400	
A-201	Perryridge	900	
A-218	Perryridge	700	
A-222	Redwood	700	
A-305	Round Hill	350	
A-888	North Town	800	

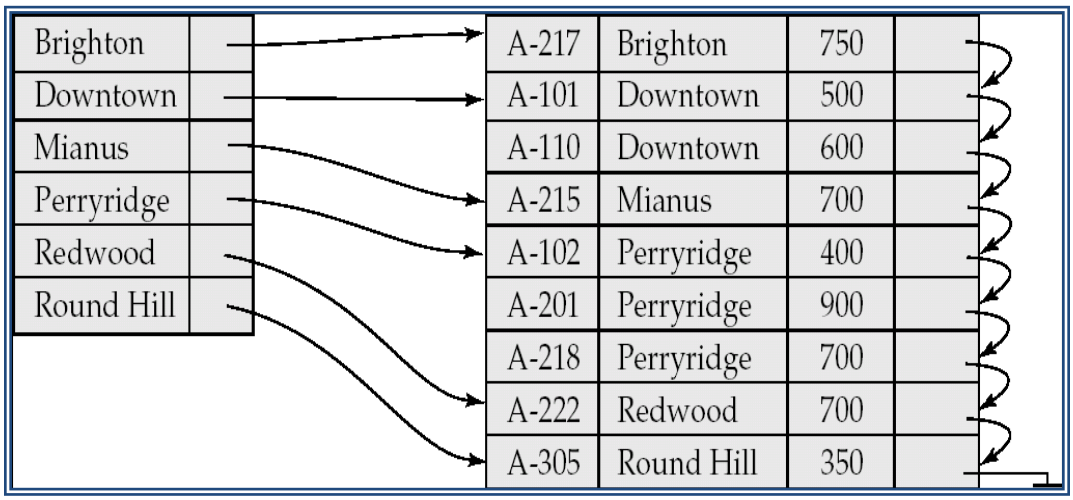
6.4 Indexing and Hashing

- ✓ Indexing mechanisms used to speed up access to desired data.
  - E.g., author catalog in library
- ✓ **Search Key** - attribute to set of attributes used to look up records in a file.
- ✓ An **index file** consists of records (called **index entries**) of the form

Search-key	Pointer

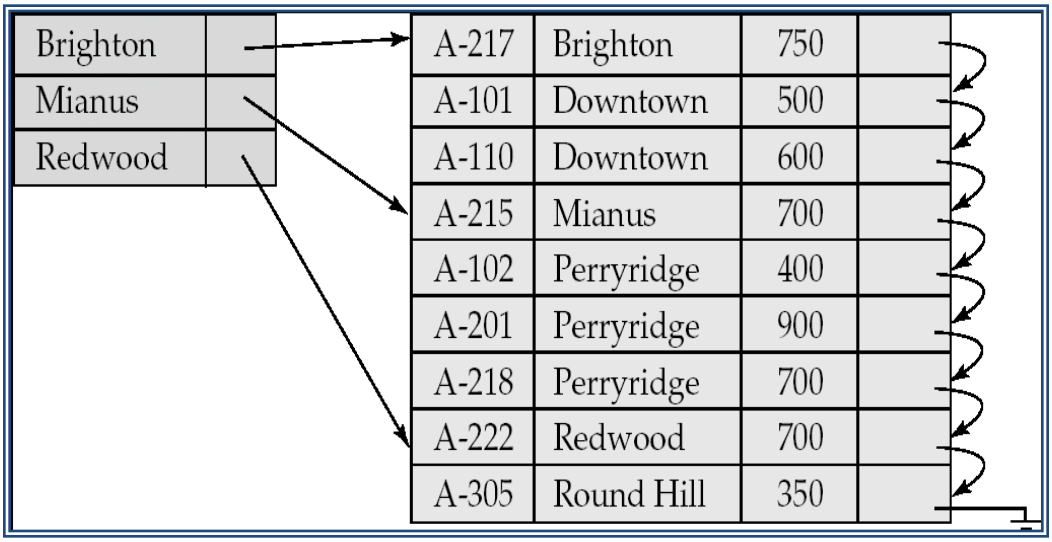
- ✓ Index files are typically much smaller than the original file
- ✓ Two basic kinds of indices:
  - **Ordered indices:** search keys are stored in sorted order
  - **Hash indices:** search keys are distributed uniformly across "buckets" using a "hash function".
- ✓ **Index Evaluation Metrics**
  - **Access type**
    - ♦ Finding records with a specified attribute value
    - ♦ Finding records whose attribute values fall in a specified range
  - **Access time**
    - ♦ Time to find a particular value
  - **Insertion time**
    - ♦ Time to insert new value as well as to update the index structure
  - **Deletion time**
    - ♦ Time to delete data item as well as to update the index structure
  - **Space overhead**
    - ♦ Additional space occupied by an index structure
- ✓ **Ordered Indices**
  - In an **ordered index,** index entries are stored sorted on the search key value. E.g., author catalog in library.
  - Two types of order index
    - ♦ **Primary index**
    - ♦ **Secondary index**
  - **Primary index:** in a sequentially ordered file, the index whose search key specifies the sequential order of the file.
  - Also called **clustering index**
  - The search key of a primary index is usually but not necessarily the primary key.
  - Primary index are two types:
    - ♦ Dense Index
    - ♦ Sparse Index

- **Dense index :**
  - Index record appears for every search-key value in the file.
  - The index record contains the search key value and a pointer to the first data record with that search value.
  - Rest of the records with the same search key value would be store sequentially after the first record.



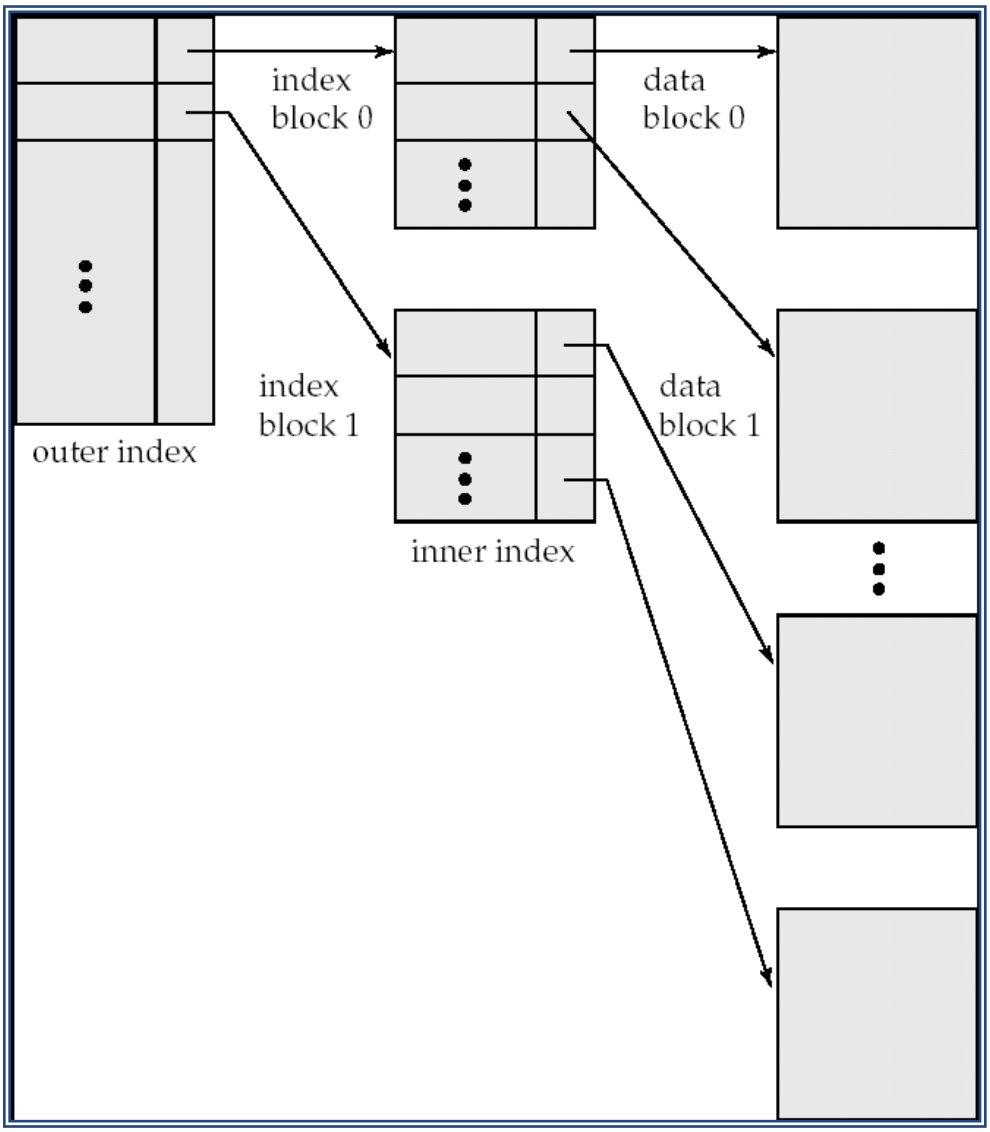
Dense Index

- **Sparse Index**
  - Sparse Index: contains index records for only some search-key values.
  - 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



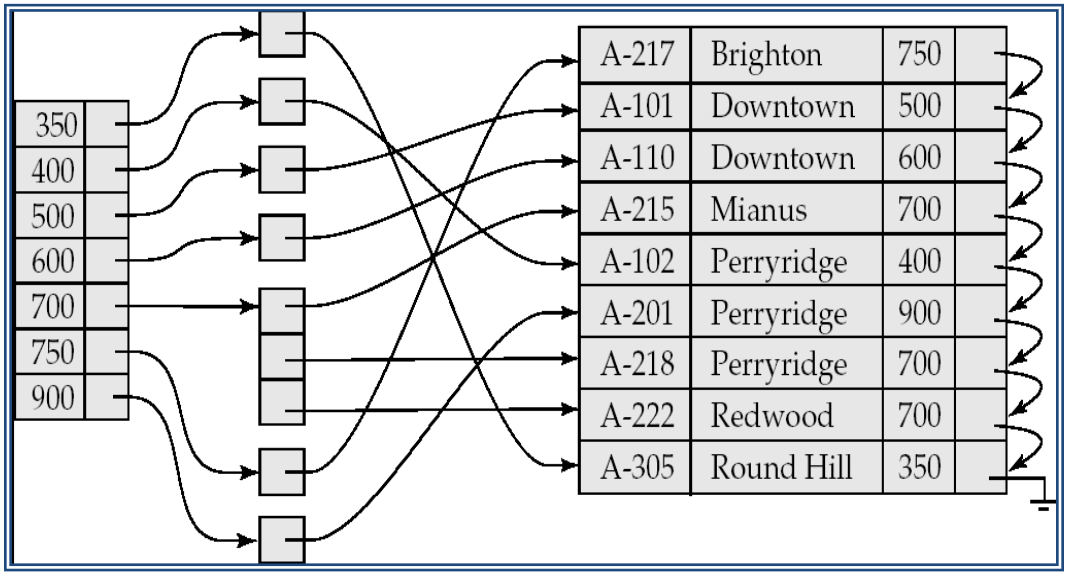
Sparse Index

- Compared to dense indices:
  - Less space and less maintenance overhead for insertions and deletions.
  - Generally slower than dense index for locating records.
- **Multilevel Index**
  - If primary index does not fit in memory, access becomes expensive.
  - Solution: treat primary index kept on disk as a sequential file and construct a sparse index on it.
    - outer index – a sparse index of primary index
    - inner index – the primary index file
  - If even outer index is too large to fit in main memory, yet another level of index can be created, and so on.
  - Indices at all levels must be updated on insertion or deletion from the file.



Multilevel Index

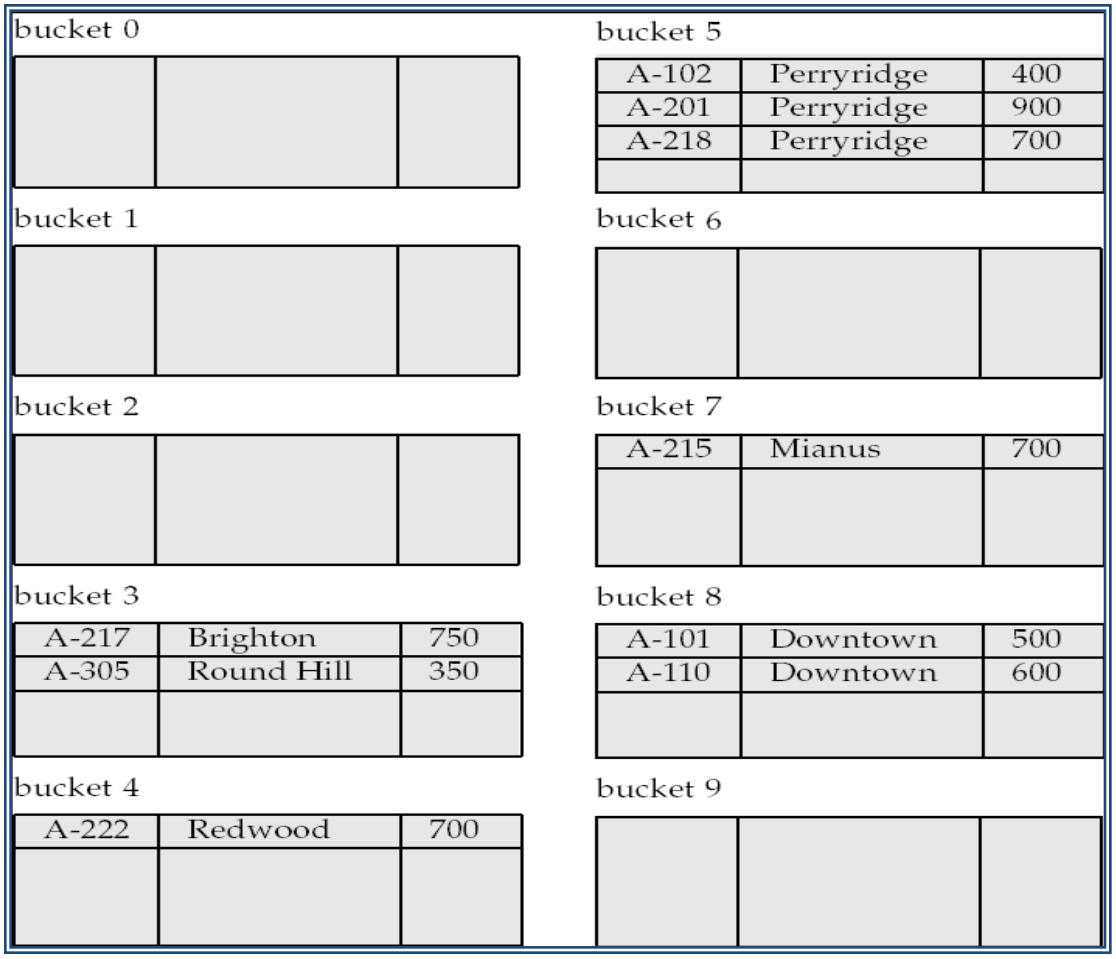
- **Secondary Indices**
  - Secondary index: an index whose search key specifies an order different from the sequential order of the file. Also called non-clustering index.
  - Frequently, one wants to find all the records whose values in a certain field (which is not the search-key of the primary index) satisfy some condition.
    - Example 1: In the account relation stored sequentially by account number, we may want to find all accounts in a particular branch
    - Example 2: as above, but where we want to find all accounts with a specified balance or range of balances
  - We can have a secondary index with an index record for each search-key value
  - A secondary index must contain pointers of all the records.
  - Index record points to a bucket that contains pointers to all the actual records with that particular search-key value.
  - Secondary indices have to be dense



Secondary Index

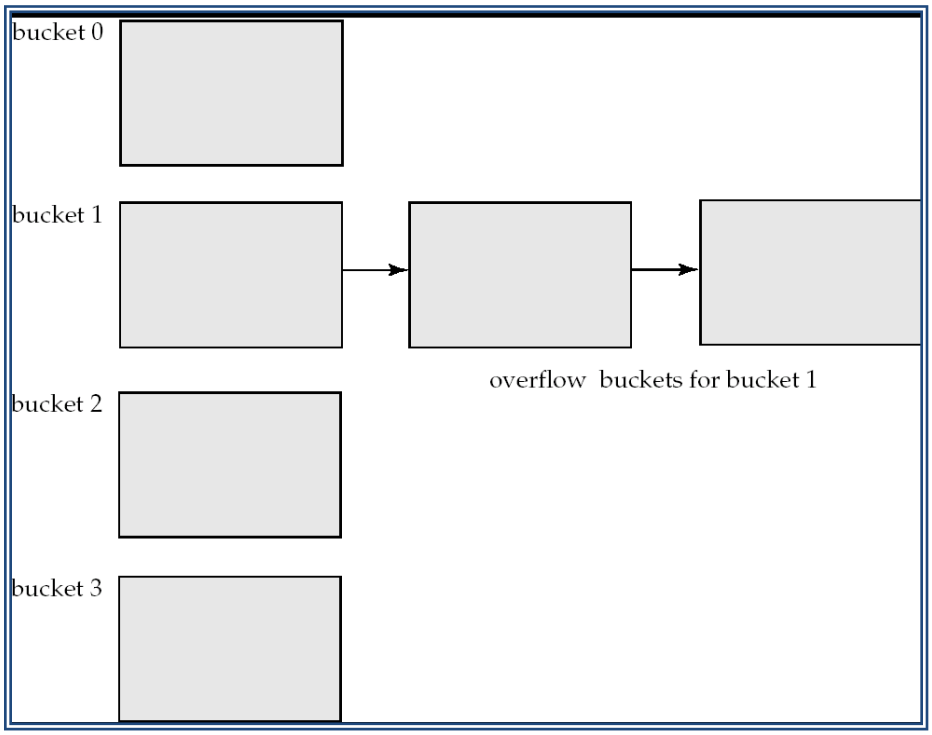
Hashing

- ✓ A bucket is a unit of storage containing one or more records (a bucket is typically a disk block).
- ✓ In a hash file organization we obtain the bucket of a record directly from its search-key value using a hash function.
- ✓ 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.
- ✓ 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.
- ✓ Hash file organization of account file, using branch_name as key
  - There are 10 buckets,
  - The binary representation of the i^{th} character is assumed to be the integer i.
  - The hash function returns the sum of the binary representations of the characters modulo 10
    - ♦ E.g. h(Perryridge) = 5 h(Round Hill) = 3 h(Brighton) = 3

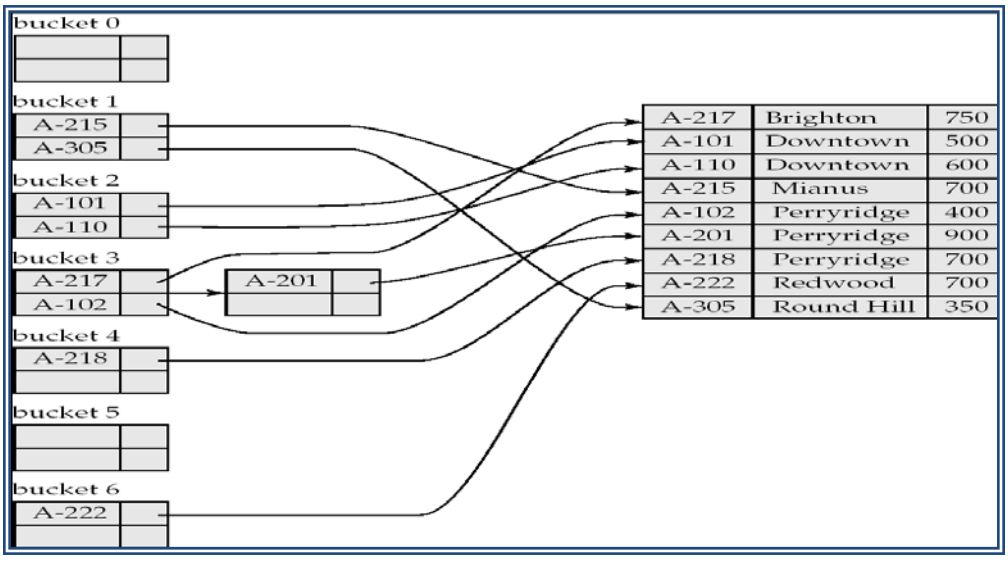


- ✓ Worst hash function maps all search-key values to the same bucket; this makes access time proportional to the number of search-key values in the file.
- ✓ An ideal hash function is **uniform**, i.e., each bucket is assigned the same number of search-key values from the set of all possible values.
- ✓ 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.
- ✓ Typical hash functions perform computation on the internal binary representation of the search-key.
    - For example, for a string search-key, the binary representations of all the characters in the string could be added and the sum modulo the number of buckets could be returned.
- ✓ **Bucket overflow can occur because of**
    - **Insufficient buckets**
        - ♦ nB must be chosen such that nB> nr/fr; where nB=number of bucket, nr= number of record, fr=number of records that will fit in a bucket.
    - **Skew**
        - ♦ A bucket may overflow even when buckets still have space,this situation is called skew.
        - ♦ Skew can occur due to two reasons:
            - multiple records have same search-key value
            - chosen hash function produces non-uniform distribution of key values
- ✓ Although the probability of bucket overflow can be reduced, it cannot be eliminated; it is handled by using **overflow buckets**.
- ✓ **Overflow chaining** – the overflow buckets of a given bucket are chained together in a linked list called closed hashing.
- ✓ **closed hashing**
    - If a record must be inserted into a bucket b, b is already full, and then system provides an overflow bucket for b and inserts the record into the overflow bucket. If the overflow is also full, the system provides another overflow bucket, and so on.
- ✓ An alternative, called open hashing, which does not use overflow buckets, is not suitable for database applications.
- ✓ Open addressing are two types:
    - Linear probing
    - Quadratic probing
- ✓ **Hash Indices**
    - 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.
- Strictly speaking, hash indices are always secondary indices
  - If the file itself is organized using hashing, a separate primary hash index on it using the same search-key is unnecessary.
  - However, we use the term hash index to refer to both secondary index structures and hash organized files.



Handling of Bucket Overflow

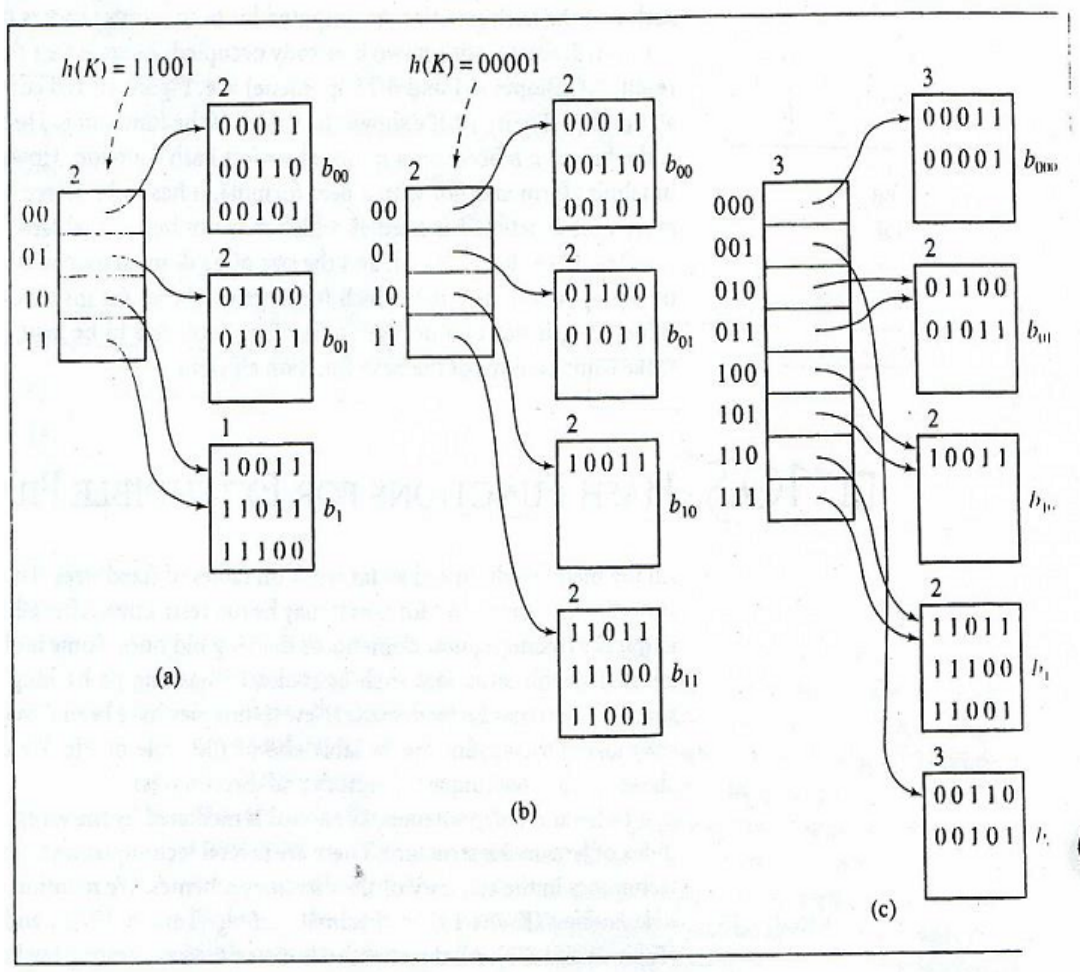


Example of hash Index

- ✓ **Deficiencies of Static Hashing**
  - In static hashing, function h maps search-key values to a fixed set of B of bucket addresses. Databases grow or shrink with time.
    - ♦ If initial number of buckets is too small, and file grows, performance will degrade due to too much overflows.
    - ♦ If space is allocated for anticipated growth, a significant amount of space will be wasted initially (and buckets will be under full).
    - ♦ If database shrinks, again space will be wasted.
  - One solution: periodic re-organization of the file with a new hash function
    - ♦ Expensive, disrupts normal operations
  - Better solution: allow the number of buckets to be modified dynamically.

- ✓ **Extendable hashing**
  - In this hashing scheme the set of keys can be varied, and the address space is allocated dynamically
  - Good for database that grows and shrinks in size
  - Allows the hash function to be modified dynamically
  - Extendable hashing – one form of dynamic hashing
  - Keys stored in buckets.
  - Each bucket can only hold a fixed size of items.
  - Index is an extendible table; h(x) hashes a key value x to a bit map; only a portion of a bit map is used to build a directory.
  - Once the bucket is full – split the bucket into two
  - Two situation will be possible:
    - ♦ Directory remains of the same size adjust pointer to a bucket
    - ♦ Size grows from 2^k to 2^{k+1} i.e. directory size can be 1, 2, 4, 8, 16 etc.
      - Number of buckets will remain the same, i.e. some references will point to the same bucket.
  - Finally, one can use bitmap to build the index but store an actual key in the bucket!
  - Assume that a hashing technique is applied to a dynamically changing file composed of buckets, and each bucket can hold only a fixed number of items.
  - Extendible hashing accesses the data stored in buckets indirectly through an index that is dynamically adjusted to reflect changes in the file.
  - The characteristic feature of extendible hashing is the organization of the index, which is an expandable table.
  - A hash function applied to a certain key indicates a position in the index and not in the file (or table or keys). Values returned by such a hash function are called pseudo keys.

- The file requires no reorganization when data are added to or deleted from it, since these changes are indicated in the index. Only one hash function h can be used, but depending on the size of the index, only a portion of the added h(K) is utilized.
- A simple way to achieve this effect is by looking at the address into the string of bits from which only the i leftmost bits can be used.



- ✓ **Comparison of Ordered Indexing and Hashing**
    - Benefits of extendable hashing:
        - ♦ Hash performance does not degrade with growth of file
        - ♦ Minimal space overhead
    - Disadvantages of extendable hashing
        - ♦ Extra level of indirection to find desired record
        - ♦ Bucket address table may itself become very big (larger than memory)
            - Cannot allocate very large contiguous areas on disk either
            - Solution: B^+-tree structure to locate desired record in bucket address table
        - ♦ Changing size of bucket address table is an expensive operation
    - Cost of periodic re-organization
    - Relative frequency of insertions and deletions
    - Expected type of queries:
        - ♦ Hashing is generally better at retrieving records having a specified value of the key.
        - ♦ If range queries are common, ordered indices are to be preferred
    - In practice:
        - ♦ PostgreSQL supports hash indices, but discourages use due to poor performance
        - ♦ Oracle supports static hash organization, but not hash indices
        - ♦ SQLServer supports only B+-trees

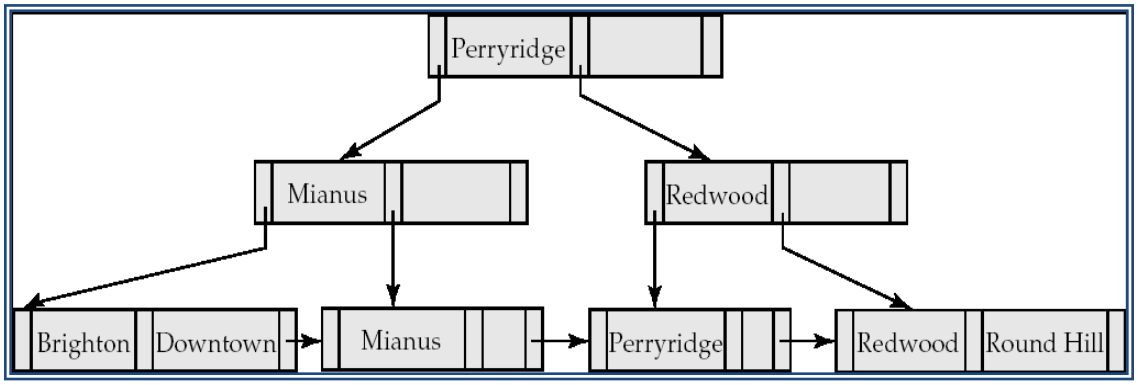
6.4 B+ Tree

- ✓ B^+-tree indices are an alternative to indexed-sequential files.
- ✓ Disadvantage of indexed-sequential files
    - Performance degrades as file grows, since many overflow blocks get created.
    - Periodic reorganization of entire file is required.
- ✓ Advantage of B^+-tree index files:
    - Automatically reorganizes itself with small, local, changes, in the face of insertions and deletions.
    - Reorganization of entire file is not required to maintain performance.
- ✓ (Minor) disadvantage of B^+-trees:
    - Extra insertion and deletion overhead, space overhead.
- ✓ Advantages of B^+-trees outweigh disadvantages
    - B^+-trees are used extensively

- ✓ A B^+-tree is a rooted tree satisfying the 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 (n/2)and n children.
  - A leaf node has between ((n–1)/2)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.
- ✓ Typical node

  - 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

    $K_1 < K_2 < K_3 < \ldots < K_{n-1}$



Example of B+ Tree

Chapter 7 Transactions and concurrency control

7.1 Introduction

- ✓ A transaction is a unit of program execution that accesses and possibly updates various data items.
- ✓ A transaction is an action, or a series of actions, carried out by a single user or an application program, which reads or updates the contents of a database.
- ✓ A transaction must see a consistent database.
- ✓ During transaction execution the database may be temporarily inconsistent. When the transaction completes successfully (is committed), the database must be consistent. After a transaction commits, the changes it has made to the database persist, even if there are system failures.
- ✓ Multiple transactions can execute in parallel.
- ✓ Two main issues to deal with:
    - Failures of various kinds, such as hardware failures and system crashes
    - Concurrent execution of multiple transactions

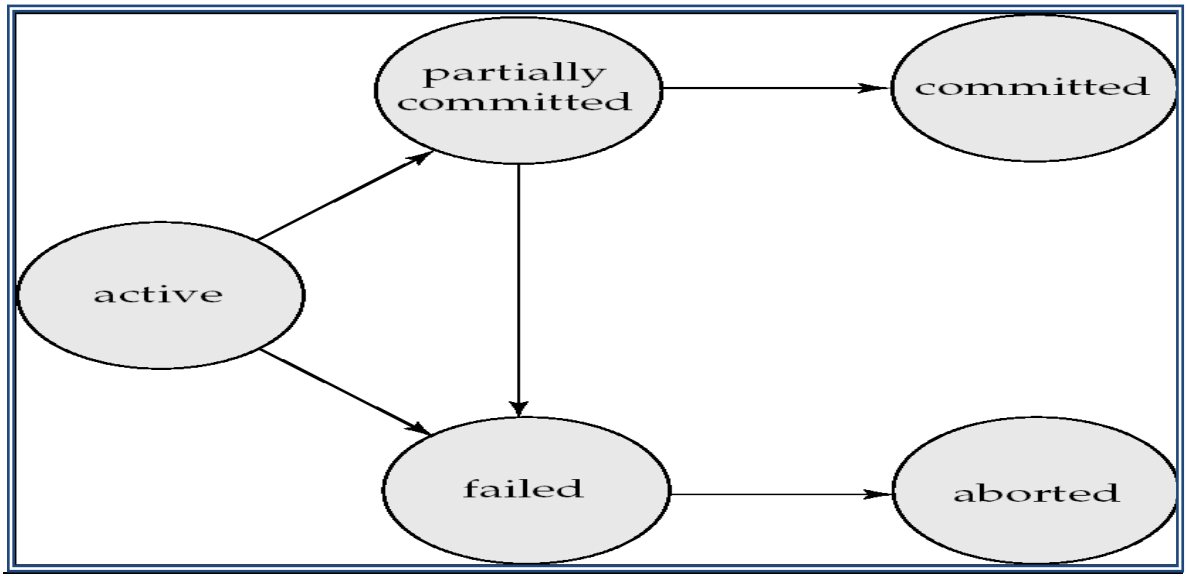
7.2 ACID Properties

- ✓ A transaction is a unit of program execution that accesses and possibly updates various data items. To preserve the integrity of data the database system must ensure:
    - **Atomicity**
        - ♦ Either all operations of the transaction are properly reflected in the database or none are.
        - ♦ Transaction is indivisible – it completes entirely or not at all, despite failures.
    - **Consistency.** Execution of a transaction in isolation preserves the consistency of the database. Consistency transfers the database from one consistent state to another consistent state.
    - **Isolation:**
        - ♦ Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions. That is, for every pair of transactions $_{Ti}$ and $_{Tj}$, it appears to $_{Ti}$ that either $_{Tj}$, finished execution before $_{Ti}$ started, or $_{Tj}$ started execution after $_{Ti}$ finished.
        - ♦ The effects of a transaction are not visible to other transactions until it has completed
    - **Durability**. After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

- ✓ It is the responsibility of the applications and DBMS to maintain the constraints.
- ✓ Transaction to transfer 50 from account A to account B:
    1. **read**(*A*)
    2. *A* := *A* – 50
    3. **write**(*A*)
    4. **read**(*B*)
    5. *B* := *B* + 50
    6. **write**(*B)*

- ✓ **Atomicity requirement** — if the transaction fails after step 3 and before step 6, the system should ensure that its updates are not reflected in the database, else an inconsistency will result.
- ✓ **Consistency requirement** – the sum of A and B is unchanged by the execution of the transaction.
- ✓ **Isolation requirement** — if between steps 3 and 6, another transaction 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).
    - 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.
- ✓ **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 despite failures.

7.3 Transaction State



- ✓ **Active** – the initial state; the transaction stays in this state while it is executing
- ✓ **Partially committed** – after the final statement has been executed.
- ✓ **Failed --** after the discovery that normal execution can no longer proceed.
- ✓ **Aborted** – after the transaction has been rolled back and the database restored to its state prior to the start of the transaction. Two options after it has been aborted:
  - restart the transaction; can be done only if no internal logical error
  - kill the transaction
- ✓ **Committed** – after successful completion.

7.4 Concurrent Executions

- ✓ Multiple transactions are allowed to run concurrently in the system. Advantages are:
  - **Increased processor and disk utilization**, leading to better transaction throughput: one transaction can be using the CPU while another is reading from or writing to the disk
  - **Reduced average response time** for transactions: short transactions need not wait behind long ones.
- ✓ **Concurrency control schemes** – mechanisms to achieve isolation; that is, to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database
- ✓ **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)
  - A serial schedule in which T_1 is followed by T_2: Schedule 1

T_1	T_2
read(A) $A := A - 50$ write (A) read(B) $B := B + 50$ write(B)	read(A) $temp := A * 0.1$ $A := A - temp$ write(A) read(B) $B := B + temp$ write(B)

- A serial schedule where T_2 is followed by T_1. Schedule 2

T_1	T_2
	read(A) $temp := A * 0.1$ $A := A - temp$ write(A) read(B) $B := B + temp$ write(B)
read(A) $A := A - 50$ write(A) read(B) $B := B + 50$ write(B)	

- Let T_1 and T_2 be the transactions defined previously. The following schedule is not a serial schedule, but it is equivalent to Schedule 1. In Schedules 1, 2 and 3, the sum A + B is preserved

T_1	T_2
read(A) $A := A - 50$ write(A)	
	read(A) $temp := A * 0.1$ $A := A - temp$ write(A)
read(B) $B := B + 50$ write(B)	
	read(B) $B := B + temp$ write(B)

- The following concurrent schedule does not preserve the value of (*A + B*).

T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
	$B := B + temp$
	write(B)

7.5 Serializability

- ✓ A schedule S is **serial** if, for every transaction T participating in the schedule, all the operations of T is executed consecutively in the schedule.
- ✓ No interleaving occurs in serial schedule
- ✓ A schedule S of n transactions is **serializable** if it is equivalent to some serial schedule of the same n transactions.
- ✓ Two schedules are called **result equivalent** if they produce the same final state of the database.
- ✓ A schedule is serializable if it is equivalent to a serial schedule. Different forms of schedule equivalence give rise to the notions of:
  - Conflict serializability
  - View serializability
- ✓ **Conflict serializability**
  - Two actions Ai and Aj executed on the same data object by Ti and Tj conflicts if either one of them is a write operation.
  - Let Ai and Aj are **consecutive non-conflicting actions** that belong to different transactions. We can swap Ai and Aj without changing the result.
  - Two schedules are conflict equivalent if they can be turned one into the other by a sequence of non-conflicting swaps of adjacent actions.

- If a schedule *S* can be transformed into a schedule *S´* by a series of swaps of non-conflicting instructions, we say that *S* and *S´* are **conflict equivalent**.
- We say that a schedule *S* is **conflict serializable** if it is conflict equivalent to a serial schedule.
- Actions l_i and l_j of transactions T_i and T_j respectively, **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 wrote Q.
  - l_i = **read**(Q), l_j = **read**(Q). l_i and l_j don't conflict.
  - l_i = **read**(Q), l_j = **write**(Q). They conflict.
  - l_i = **write**(Q), l_j = **read**(Q). They conflict
  - 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. That is, replacing their order will change the result!
- 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.
- **Example:** Schedule 3 can be transformed into Schedule 4, a serial schedule where T_2 follows T_1, by series of swaps of non-conflicting instructions.
  - Therefore Schedule 3 is conflict serializable.

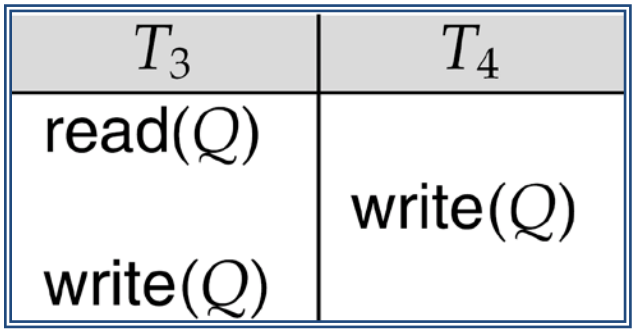
T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Schedule 3

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

Schedule 4

- Example of a schedule that is not conflict serializable. We are unable to swap instructions in the above schedule to obtain either the serial schedule < T_3, T_4 >, or the serial schedule < T_4, T_3 >.



- ✓ **Test for Conflict Serializability of a Schedule**
  - For each transaction T_i participating in schedule S, create a node labeled T_i in the precedence graph
  - For each case in S where T_j executes a read_item(X) after T_i executes a write_item(X),create an edge ($T_i \rightarrow T_j$)
  - For each case in S where T_j executes a write_item(X) after T_i executes a read_item(X) create an edge ($T_i \rightarrow T_j$)
  - For each case in S where T_j executes a write_item(X) after T_i executes a write_item(X) create an edge ($T_i \rightarrow T_j$)
  - The schedule S is serializable if and only if the precedence graph has no cycles.

- ✓ **Example**

  Assume we have these three transactions:

  T1: $r_1(x)$; $w_1(x)$; $r_1(y)$; $w_1(y)$

  T2: $r_2(z)$; $r_2(y)$; $w_2(y)$; $r_2(x)$; $w_2(x)$

  T3: $r_3(y)$; $r_3(z)$;$w_3(y)$;$w_3(z)$

  Assume we have these schedules:

  S1: $r_2(z)$;$r_2(y)$;$w_2(y)$; $r_3(y)$;$r_3(z)$; $r_1(x)$;$w_1(x)$; $w_3(y)$;$w_3(z)$;$r_2(x)$; $r_1(y)$;$w_1(y)$; $w_2(x)$

  No equivalent serial schedule

  (Cycle x ($T_1 \rightarrow T_2$), y ($T_2 \rightarrow T_1$))

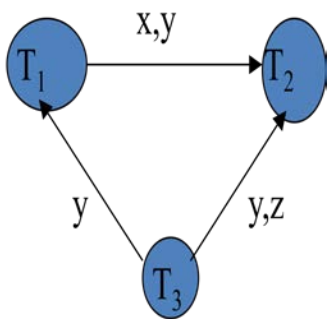
  (Cycle x ($T_1 \rightarrow T_2$), yz($T_2 \rightarrow T_3$),y($T_3 \rightarrow T_1$))

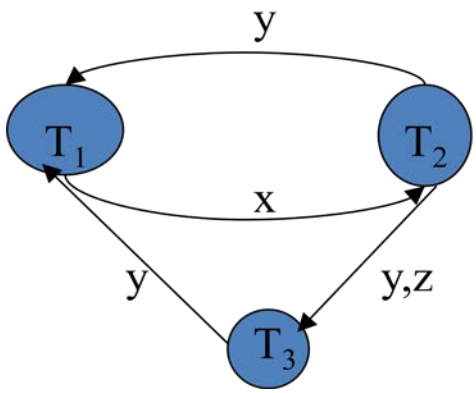
  Assume we have another schedule for the same transactions:

  S2: $r_3(y)$;$r_3(z)$;$r_1(x)$; $w_1(x)$;$w_3(y)$;$w_3(z)$;$r_2(z)$; $r_1(y)$;$w_1(y)$;$r_2(y)$; $w_2(y)$;$r_2(x)$;$w_2(x)$

  Equivalent serial schedule

  $T_3 \rightarrow T_1 \rightarrow T_2$





✓ **View serializability**

- ♦ A schedule S is view serializable it is view equivalent to a serial schedule.
- ♦ Every conflict serializable schedule is also view serializable.
- ♦ Schedules S1 and S2 are view equivalent if:
  - If Ti reads initial value of A in S1, then Ti also reads initial value of A in S2
  - If Ti reads value of A written by Tj in S1, then Ti also reads value of A written by Tj in S2
  - If Ti writes final value of A in S1, then Ti also writes final value of A in S2

♦ **Example**

- Consider the following schedule of three transactions
  T1: r1(X), w1(X); T2: w2(X); and T3: w3(X):
  Schedule S: r1(X); w2(X); w1(X); w3(X); c1; c2; c3;
- In S, the operations w2(X) and w3(X) are blind writes, since T1 and T3 do not read the value of X.
- S is view serializable, since it is view equivalent to the serial schedule T1, T2, T3.
- However, S is not conflict serializable, since it is not conflict equivalent to any serial schedule.

7.6 Recoverable Schedules

- ✓ **Recoverable schedule** — if a transaction T_j reads a data items previously written by a transaction T_i, the commit operation of T_i should appear before the commit operation of T_j.
- ✓ The following schedule is not recoverable if T_2 commits immediately after the read

T1	T2
Read(A)	
Write(A)	
	Read(A)
Read(B)	

- ✓ If T_1 should abort, T_2 would have read (and possibly shown to the user) an inconsistent database state. Hence database must ensure that schedules are recoverable.
- ✓ **Cascading rollback** – a single transaction failure leads to a series of transaction rollbacks. Consider the following schedule where none of the transactions has yet committed (so the schedule is recoverable)

T1	T2	T3
Read(A)		
Read(B)		
Write(A)		
	Read(A)	
	Write(A)	
		Read(A)

  - If T_1 fails, T_2 and T_3 must also be rolled back.
  - Can lead to the undoing of a significant amount of work
- ✓ **Cascade less schedules** — cascading rollbacks cannot occur; for each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i, the commit operation of T_i appears before the read operation of T_j.
- ✓ Every cascade less schedule is also recoverable.

7.7 Lock-Based Protocols

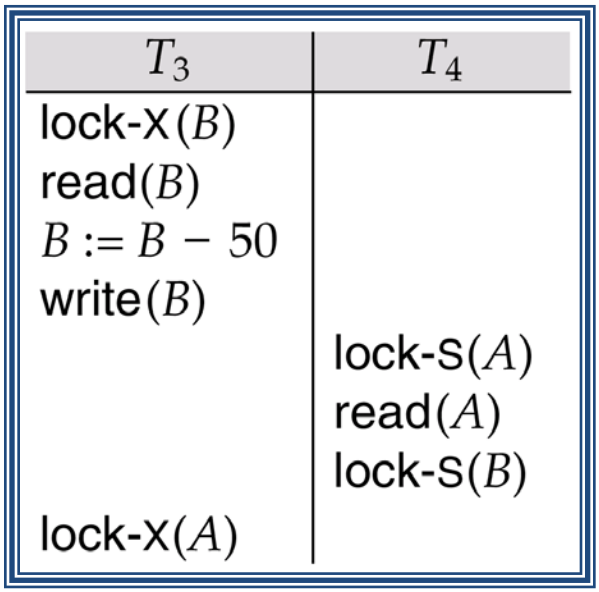
- ✓ A lock is a mechanism to control concurrent access to a data item.
- ✓ Concurrency control in DBMS ensures that database transactions are performed concurrently without violating the data integrity of a database. The main goal of concurrency control is to allow several transactions to be executing simultaneously such that collection of manipulated data item is left in a consistent state.
- ✓ A locking **protocol** is a set of rules followed by all transactions while requesting and releasing locks.
- ✓ Locking is procedure used to control concurrent access to data, when one transaction is accessing the database; a lock may be deny access to other transactions to prevent incorrect results.
- ✓ 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.
- ✓ 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.
- ✓ If a transaction has a shared lock on a data item, it can be read the item but not update it.
- ✓ If a transaction has a shared lock on a data items, other transaction can obtain a shared lock on the data item, but not exclusive locks.
- ✓ If a transaction has a exclusive lock on a data item, it can be read and update it.
- ✓ If a transaction has a exclusive lock on a data items, other transaction can't obtain either a shared lock or a exclusive lock on the data item.
- ✓ 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.

7.7.1 Pitfalls of Lock-Based Protocols

- Consider the partial schedule



T_3	T_4
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

- Neither T_3 nor T_4 can make progress — executing lock-**S** (B) causes T_4 to wait for T_3 to release its lock on B, while executing lock-**X** (A) causes T_3 to wait for T_4 to release its lock on A.
- Such a situation is called a **deadlock**.
  - To handle a deadlock one of T_3 or T_4 must be rolled back and its locks released.
- **Starvation** is also possible if concurrency control manager is badly designed. For example:
  - Suppose a transaction T2 has a shared mode lock on a data item, and another transaction T1 requests an exclusive mode lock on the data item. Clearly, T1 has to wait for T2to release the shared mode lock.
  - Meanwhile, a transaction T3 may request a shared mode lock on same data item. At this time T2 may release the lock, but still T1 has to wait for T3 to finish.
  - But again, there may be a new transaction T4 that request a shared mode lock on the same data item, and is granted the lock before T3 release it.
  - It is possible that there is a sequence of transactions that each requests a shared mode lock on the same data item, and each transaction release the lock a short while after it is granted, but T1 never gets the exclusive mode lock on the data item. The transaction T1 may never make progress, and is said to be **starved.**
- Concurrency control manager can be designed to prevent starvation.

- To avoid transaction by granting locks in the following manner: When a transaction Ti requests a lock on a data item Q in a particular mode M, the concurrency-control manager grants the lock provided that
  - ♦ There is no other transaction holding a lock on Q in a mode that conflicts with M
  - ♦ There is no other transaction that is waiting for a lock on Q, and that made its lock requests before Ti.

7.7.2 The Two-Phase Locking Protocol

- Example of a transaction performing not two phase locking:

  T_2: **lock-S***(A)*;
  read *(A)*;
  unlock*(A)*;
  lock-S*(B)*;
  read *(B)*;
  unlock*(B)*;
  display*(A+B)*
- Locking as above is not sufficient to guarantee serializability — if *A* and *B* get updated in-between the read of *A* and *B*, the displayed sum would be wrong.
- 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
- The protocol assures serializability. It can be proved that the transactions can be serialized in the order of their **lock points** (i.e. the point where a transaction acquired its final lock).
- 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.
- **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.
- There can be conflict serializable schedules that cannot be obtained if two-phase locking is used.

- However, in the absence of extra information (e.g., ordering of access to data), two-phase locking is needed for conflict serializability in the following sense:

  Given a transaction T_i that does not follow two-phase locking, we can find a transaction T_j that uses two-phase locking, and a schedule for T_i and T_j that is not conflict serializable.

T1	T2	T3
Lock-X(A)		
Read(A)		
Lock-S(A)		
Read(B)		
Write(A)		
Unlock(A)		
	Lock-X(A)	
	Read(A)	
	Write(A)	
	Unlock(A)	
		Lock-S(A)
		Read(A)

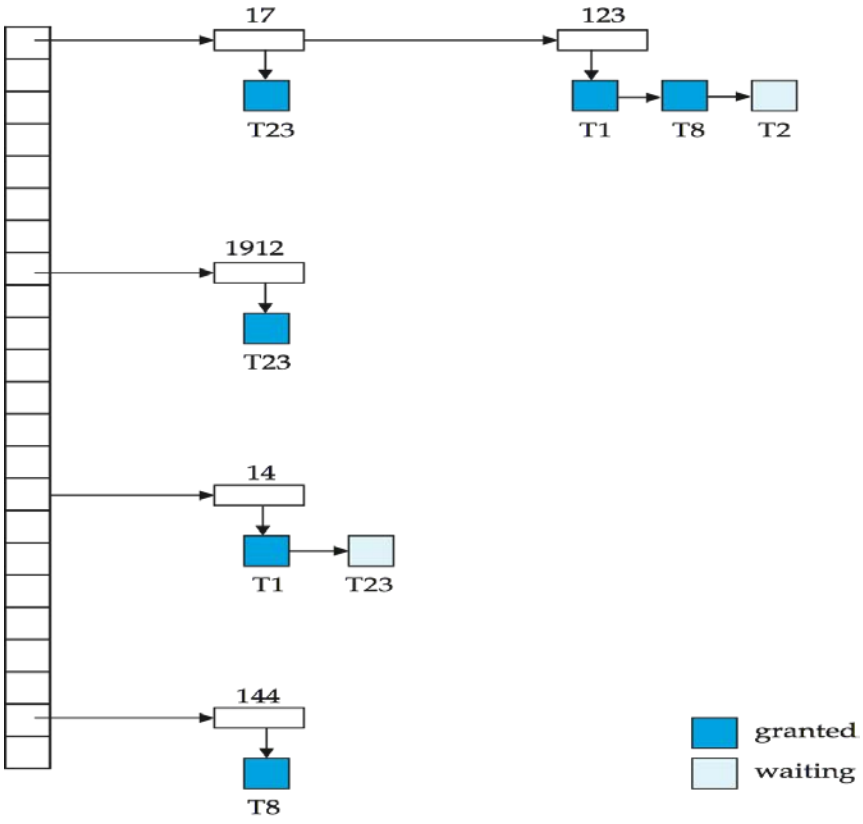
Fig: Partial Schedule under Two Phase locking

7.7.3 Lock Conversions

- ✓ Lock conversion provide a mechanism for upgrading a shared lock to exclusive lock, and downgrading an exclusive lock to a shared lock.
- ✓ Upgrading can take place in only the growing phase, whereas downgrading can take places in only the shrinking phase.
- ✓ 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.

7.7.4 Implementation of Locking

- ✓ A **lock manager** can be implemented as a separate process to which transactions send lock and unlock requests.
- ✓ The lock manager replies to a lock request by sending a lock grant messages (or a message asking the transaction to roll back, in case of a deadlock).
- ✓ The requesting transaction waits until its request is answered.
- ✓ The lock manager maintains a data-structure called a **lock table** to record granted locks and pending requests.
- ✓ The lock table is usually implemented as an in-memory hash table indexed on the name of the data item being locked.
- ✓ Black rectangles indicate granted locks, white ones indicate waiting requests
- ✓ Lock table also records the type of lock granted or requested
- ✓ New request is added to the end of the queue of requests for the data item, and granted if it is compatible with all earlier locks
- ✓ Unlock requests result in the request being deleted, and later requests are checked to see if they can now be granted
- ✓ If transaction aborts, all waiting or granted requests of the transaction are deleted
    - lock manager may keep a list of locks held by each transaction, to implement this efficiently



7.8 Deadlock Handling

- ✓ Consider the following two transactions:

  T_1: Write(X) T_2: Write(Y)
  Write(Y) Write(X)

- ✓ Schedule with deadlock:

T1	T2
Lock-X on X	
Write(X)	
	lock-X on *Y*
	write (*Y*)
	wait for **lock-X** on *X*
wait for **lock-X** on *Y*	

- ✓ 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.
- ✓ **Deadlock prevention** protocols ensure that the system will never enter into a deadlock state. Some prevention strategies:
  - Require that each transaction locks all its data items before it begins execution (pre declaration).
  - Impose partial ordering of all data items and require that a transaction can lock data items only in the order specified by the partial order (graph-based protocol).

7.8.1 Deadlock prevention

- ✓ Two approach to deadlock prevention
  - One approach ensures that no cyclic waits can occur by ordering the requests for locks, requiring all locks to be acquired together.
  - Another approach performs transaction rolled back instead of waiting for a lock, whenever the wait could potentially result in a deadlock.
- ✓ The first approach requires that each transaction locks all its data items before execution. Moreover, either all are locked in one step or none are locked. There are two main disadvantages:
  - It is often hard to predict, before the transaction begins, what data items need to be locked.
  - Data item utilization may be very low, since many of the data items may be locked but unused for a long time.
- ✓ The second approach for preventing deadlocks is to use preemption and transaction rollback. These schemes use timestamps just for deadlock prevention.

- **wait-die** scheme — non-preemptive
  - ♦ When transaction Ti requests a data item currently held by Tj, Ti is allowed to wait only if it has a timestamp smaller than that of Tj. Otherwise, Ti rolled back (dies).
  - ♦ For example, suppose that transactions T1, T2, and T3 have timestamps 20, 30, and 40 respectively. If T1 request a data item held by T2, then T2 will wait. If T3 requests a data item held by T2, then T3 will be rolled back.
- **wound-wait** scheme — preemptive
  - ♦ When transaction Ti requests a data item currently held by Tj, Ti is allowed to wait only if it has a timestamp larger than that of Tj. Otherwise, Ti rolled back.
  - ♦ For example, suppose that transactions T1, T2, and T3 have timestamps 20, 30, and 40 respectively. If T1 request a data item held by T2, then the data item will be preempted from T2 and T2 will be rolled back. If T3 requests a data item held by T2, then T3 will wait.

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

✓ **Timeout-Based Schemes :**
- A transaction waits for a lock only for a specified amount of time. After that, the wait times out and the transaction is rolled back.
  Thus deadlocks are not possible
- Simple to implement; but starvation is possible. Also difficult to determine good value of the timeout interval.

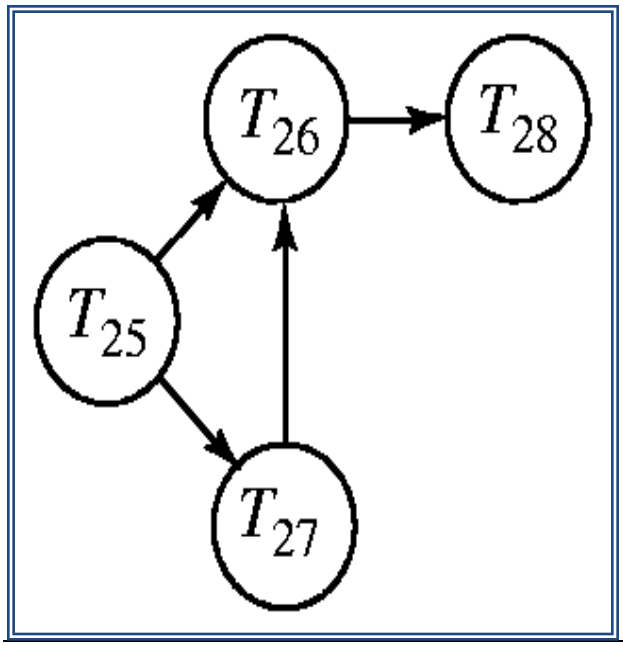
7.8.2 Deadlock Detection

✓ 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 $T_i \rightarrow T_j$.

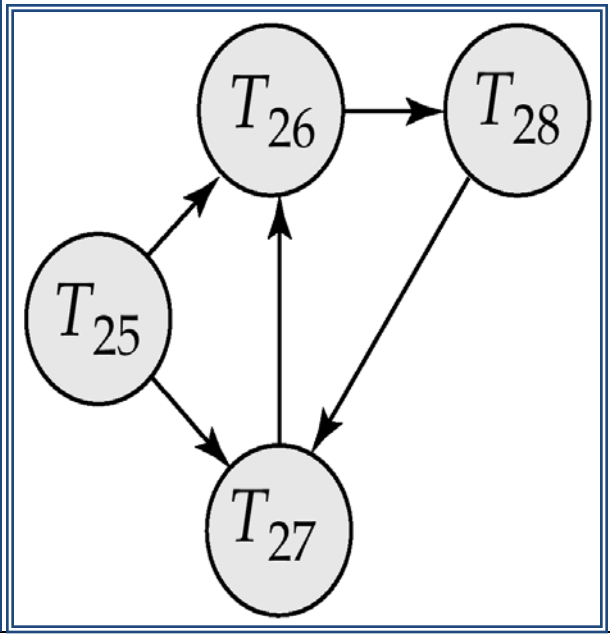
✓ If $T_i \rightarrow T_j$ is in E, then there is a directed edge from T_i to T_j, implying that T_i is waiting for T_j to release a data item.

✓ When T_i requests a data item currently being held by T_j, then the edge T_i 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. Must invoke a deadlock-detection algorithm periodically to look for cycles.



Wait-for graph without a cycle



Wait-for graph with a cycle

7.8.3 Deadlock Recovery

- ✓ When deadlock is detected :
  - Some transaction will have to rolled back (made a victim) to break deadlock. Select that transaction as victim that will incur minimum cost.
  - Rollback -- determine how far to roll back transaction
    - Total rollback: Abort the transaction and then restart it.
    - More effective to roll back transaction only as far as necessary to break deadlock.
- ✓ Starvation happens if same transaction is always chosen as victim. Include the number of rollbacks in the cost factor to avoid starvation

Chapter 8 : Crash Recovery

8.1 Failure Classification

- ✓ **Transaction failure** :
  - **Logical errors**: transaction cannot complete due to some internal error condition. Ex Bad input ,Data not found, Overflow
  - **System errors**: the database system must terminate an active transaction due to an error condition (e.g., deadlock)
- ✓ **System crash**: a power failure or other hardware or software failure causes the system to crash.
  - **Fail-stop assumption:** non-volatile storage contents are assumed to not be corrupted by system crash
    - ♦ Database systems have numerous integrity checks to prevent corruption of disk data
- ✓ **Disk failure**: a head crash or similar disk failure destroys all or part of disk storage
  - Destruction is assumed to be detectable: disk drives use **checksums to detect failures**
- ✓ **Recovery Algorithms**
  - Recovery algorithms are techniques to ensure database consistency and transaction atomicity and durability despite failures
  - Recovery algorithms have two parts
    - ♦ Actions taken during normal transaction processing to ensure enough information exists to recover from failures
    - ♦ Actions taken after a failure to recover the database contents to a state that ensures atomicity, consistency and durability

8.2 Storage Structure

- ✓ **Storage Types**
  - **Volatile storage**:
    - ♦ does not survive system crashes
    - ♦ examples: main memory, cache memory
  - **Nonvolatile storage**:
    - ♦ survives system crashes
    - ♦ examples: disk, tape, flash memory, non-volatile RAM
  - **Stable storage**
    - ♦ Information Residing in it never lost
    - ♦ a mythical form of storage that survives all failures
    - ♦ approximated by maintaining multiple copies on distinct nonvolatile media

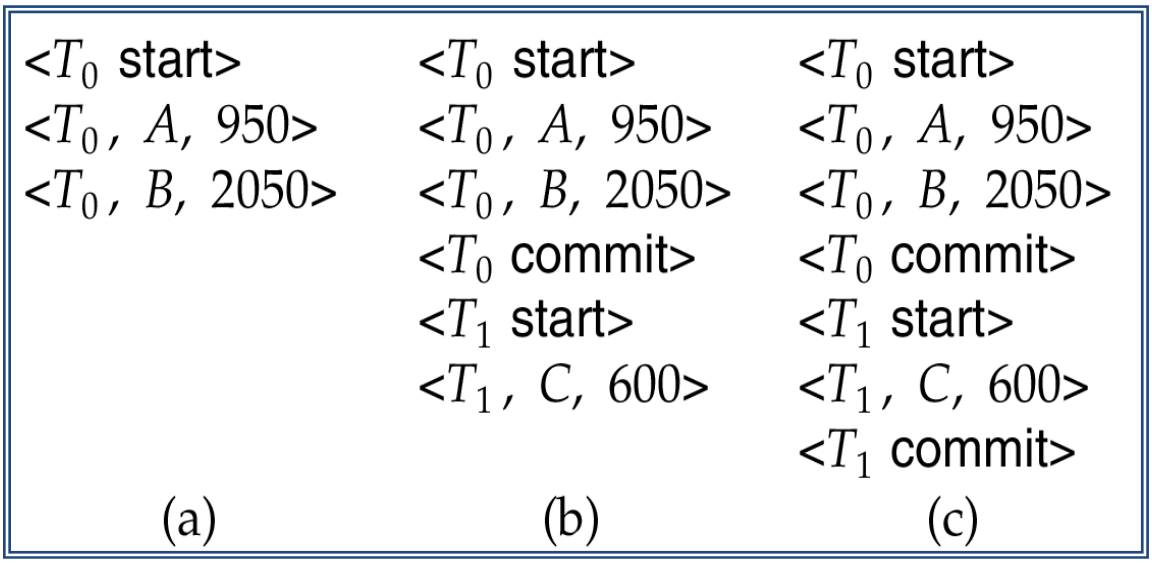
8.3 Log-Based Recovery

- ✓ 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, V1, V2> 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 $<T_i$ **commit**> is written.
- ✓ We assume for now that log records are written directly to stable storage (that is, they are not buffered)
- ✓ Two approaches using logs
  - Deferred database modification
  - Immediate database modification

8.3.1Deferred Database Modification

- ✓ The deferred database modification scheme records all modifications to the log, but defers all the writes to after partial commit.
- ✓ If the system crashes before the transaction completes its execution, or if the transaction aborts, then the information on the log is simply ignored.
- ✓ 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.
- ✓ Using log, the system handle any failure that results in the information loss. The Recovery schemes uses the following recovery procurers:
  - Redo (Ti) sets the value of all data items updated by transaction Ti to the new values. New value can be found in log.
  - The redo() operation must be idempotent:
    - Executing it several times must be equivalent to executing it once
    - This is required if we are to generate correct behavior even if a failure occurs during the recovery process.
- ✓ During recovery after a crash, a transaction needs to be redone if and only if both $<T_i$ start> and$<T_i$ commit> are there in the log.

- ✓ Redoing a transaction T_i (redo T_i) sets the value of all data items updated by the transaction to the new values.
- ✓ Crashes can occur while the transaction is executing the original updates, or while recovery action is being taken.



- ✓ If log on stable storage at time of crash is as in case:
  - No redo actions need to be taken
  - redo(T_0) must be performed since <T_0 **commi**t> is present
  - **redo**(T_0) must be performed followed by redo(T_1) since <T_0 **commit**> and <T_i commit> are present

8.3.2Immediate Database Modification

- ✓ Allows database modification to be output to the database while the transaction is still active.
- ✓ Database modifications written by active transactions are called uncommitted modifications.
- ✓ since undoing may be needed, update logs must have both old value and new value
- ✓ Update log record must be written before database item is written
  - We assume that the log record is output directly to stable storage
  - Can be extended to postpone log record output, so long as prior to execution of an **output**(B) operation for a data block B, all log records corresponding to items B must be flushed to stable storage
- ✓ Output of updated blocks can take place at any time before or after transaction commit
- ✓ Order in which blocks are output can be different from the order in which they are written.

Log	Write	Output
<T_0 **start**>		
<T_0, A, 1000, 950>		
T_o, B, 2000, 2050		
	A = 950	
	B = 2050	
<T_0 **commit**>		
<T_1 **start**>		
<T_1, C, 700, 600>		
	C = 600	
		B_B, B_C
<T_1 **commit**>		
		B_A

- Note: B_X denotes block containing X.

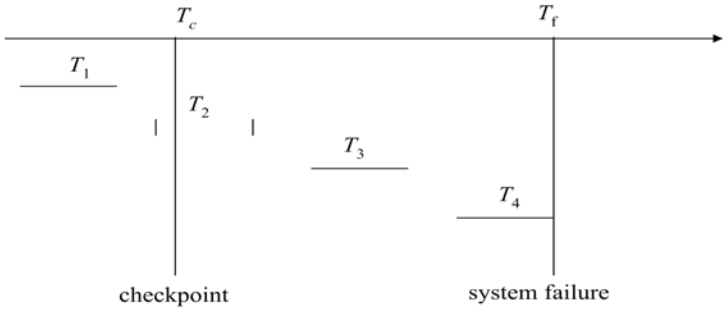
- ✓ Recovery procedure has two operations instead of one:
- ✓ **undo**(T_i) restores the value of all data items updated by T_i to their old values, going backwards from the last log record for T_i
- ✓ **redo**(T_i) sets the value of all data items updated by T_i to the new values, going forward from 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.
- ✓ 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)

- ✓ Recovery actions in each case above are:
  - Undo (T_0): B is restored to 2000 and A to 1000.
  - Undo (T_1) and redo (T_0): C is restored to 700, and then A and B are set to 950 and 2050 respectively.
  - Redo (T_0) and redo (T_1): A and B is set to 950 and 2050 respectively. Then C is set to 600

8.4 Checkpoints

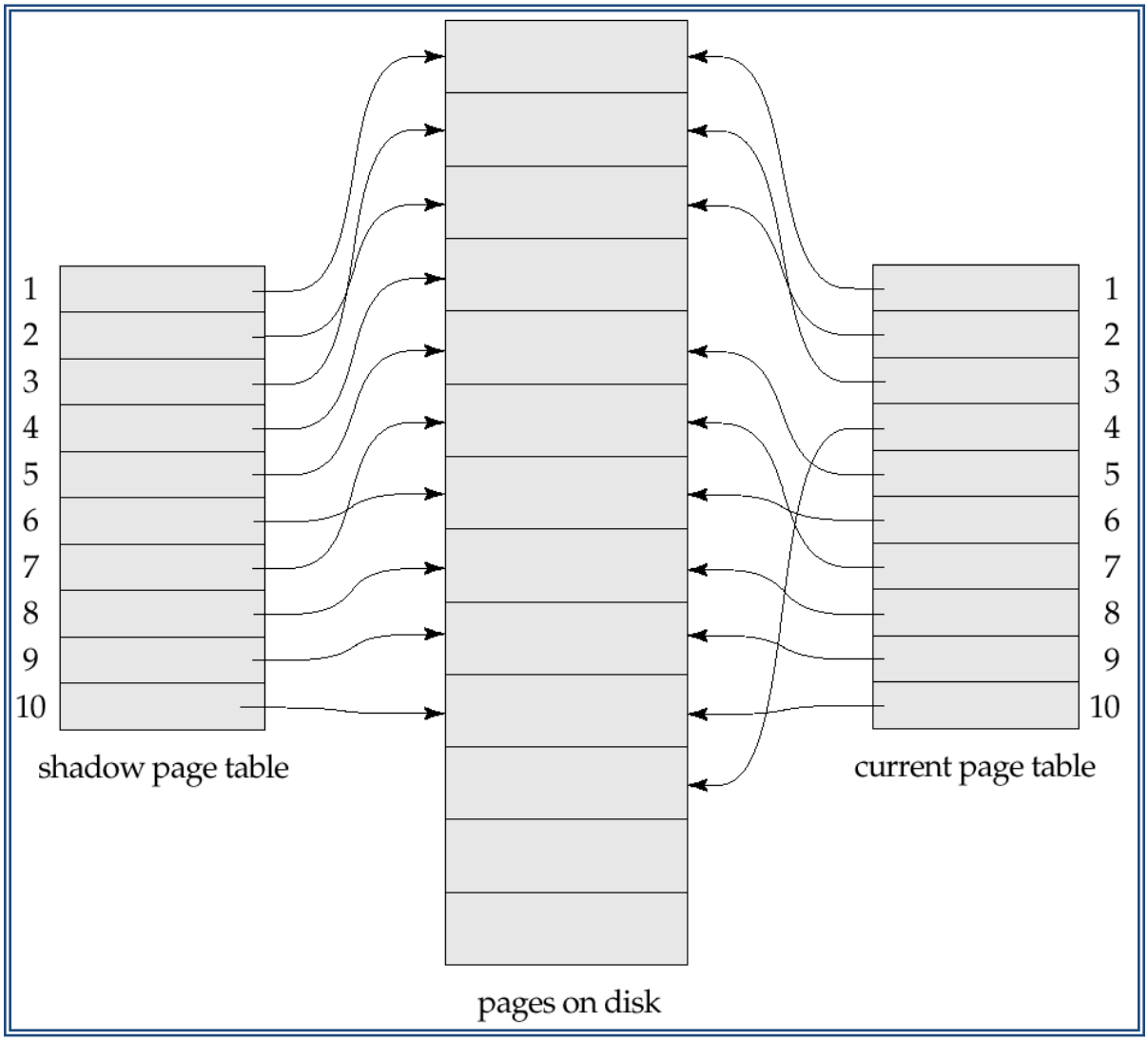
- ✓ Problems in recovery procedure as discussed earlier :
  - Searching the entire log is time-consuming
  - We might unnecessarily redo transactions which have already output their updates to the database.
- ✓ Streamline recovery procedure by periodically performing **check pointing**
  - Output all log records currently residing in main memory onto stable storage.
  - Output all modified buffer blocks to the disk.
  - Write a log record < **checkpoint**> onto stable storage.
- ✓ During recovery we need to consider only the most recent transaction T_i that started before the checkpoint and transactions that started after T_i.
  - Scan backwards from end of log to find the most recent <**checkpoint**> record
  - Continue scanning backwards till a record <T_i **start**> is found.
  - Need only consider the part of log following above **star**t record. Earlier part of log can be ignored during recovery, and can be erased whenever desired.
- ✓ After identified Transaction Ti , the redo and undo operations to be applied to the Ti and all Tj that started execution after Transaction Ti
  - For all transactions (starting from T_i or later) with no <T_i **commit**>, execute **undo (**T_i). (Done only in case of immediate modification.)
- Scanning forward in the log, for all transactions starting from T_i or later with a <T_i **commit**>, execute **redo (**T_i).



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

8.5 Shadow Paging

- ✓ **Shadow paging** is an alternative to log-based recovery; this scheme is useful if transactions execute serially
- ✓ Idea: maintain two page tables during the lifetime of a transaction –the **current page table**, and the **shadow page table**
- ✓ Store the shadow page table in nonvolatile storage, such that state of the database prior to transaction execution may be recovered.
  - Shadow page table is never modified during execution
- ✓ To start with, both the page tables are identical. Only current page table is used for data item accesses during execution of the transaction.
- ✓ Whenever any page is about to be written for the first time
  - A copy of this page is made onto an unused page.
  - The current page table is then made to point to the copy
  - The update is performed on the copy



- ✓ To commit a transaction :
  - Flush all modified pages in main memory to disk
  - Output current page table to disk
  - Make the current page table the new shadow page table, as follows:
    - ♦ Keep a pointer to the shadow page table at a fixed (known) location on disk.
    - ♦ to make the current page table the new shadow page table, simply update the pointer to point to current page table on disk
- ✓ Once pointer to shadow page table has been written, transaction is committed.
- ✓ No recovery is needed after a crash — new transactions can start right away, using the shadow page table.
- ✓ Pages not pointed to from current/shadow page table should be freed (garbage collected).
- ✓ Advantages of shadow-paging over log-based schemes
  - no overhead of writing log records
  - recovery is trivial
- ✓ Disadvantages :
  - Commit Overhead
    - ♦ Requires multiple blocks to be output- the actual data blocks, the current page table, and the disk address of the current page table. (Log based schemes need output only the log records)
    - ♦ Can be reduced by using a page table structured like a B^+-tree
      - No need to copy entire tree, only need to copy paths in the tree that lead to updated leaf nodes
    - ♦ Commit overhead is high even with above extension .Need to flush every updated page, and page table
  - Data Fragmentation:
    - ♦ Shadow Paging cause database pages to change location when they are updated
    - ♦ Data gets fragmented (related pages get separated on disk)
  - Garbage Collection:
    - ♦ After every transaction completion, the database pages containing old versions of modified data need to be garbage collected
    - ♦ Hard to extend algorithm to allow transactions to run concurrently
    - ♦ Easier to extend log based schemes

8.6 Advanced Recovery Algorithm

- ✓ Support for high-concurrency locking techniques, such as those used for B^+-tree concurrency control, which release locks early
  - Supports "logical undo"
- ✓ Recovery based on "repeating history", whereby recovery executes exactly the same actions as normal processing
  - including redo of log records of incomplete transactions, followed by subsequent undo
  - Key benefits
    - ♦ supports logical undo
    - ♦ easier to understand/show correctness

Logical Undo Logging

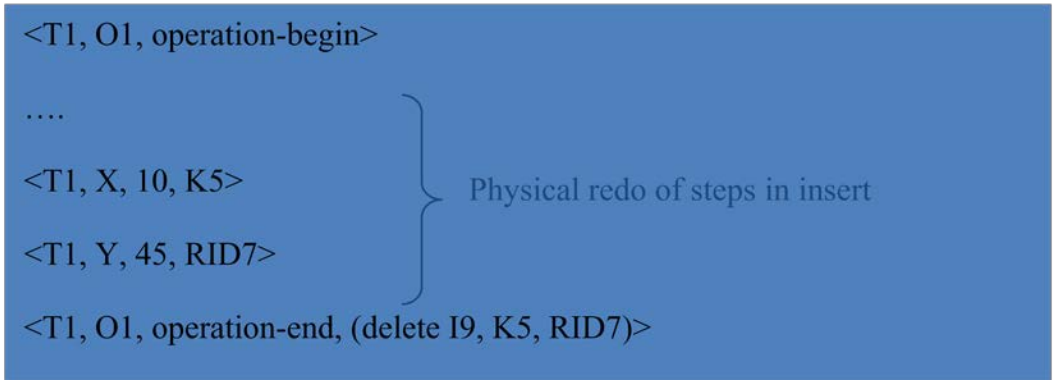
- ✓ Operations like B^+-tree insertions and deletions release locks early.
  - They cannot be undone by restoring old values (**physical undo**), since once a lock is released, other transactions may have updated the B^+-tree.
  - Instead, insertions (resp. deletions) are undone by executing a deletion (resp. insertion) operation (known as **logical undo**).
- ✓ For such operations, undo log records should contain the undo operation to be executed
  - Such logging is called **logical undo logging**, in contrast to **physical undo logging**
    - ♦ Operations are called **logical operations**
  - Other examples:
    - ♦ delete of tuple, to undo insert of tuple
      - allows early lock release on space allocation information
    - ♦ subtract amount deposited, to undo deposit
      - allows early lock release on bank balance

Physical Redo

- ✓ Redo information is logged **physically** (that is, new value for each write) even for operations with logical undo
  - Logical redo is very complicated since database state on disk may not be "operation consistent" when recovery starts
  - Physical redo logging does not conflict with early lock release

Operation Logging

- ✓ Operation logging is done as follows:
  1. When operation starts, log <T_i, O_j, **operation-begin**>. Here O_j is a unique identifier of the operation instance.
  2. While operation is executing, normal log records with physical redo and physical undo information are logged.
  3. When operation completes, <T_i, O_j, **operation-end**, U> is logged, where U contains information needed to perform a logical undo information.
- ✓ Example: insert of (key, record-id) pair (K5, RID7) into index I9



- ✓ If crash/rollback occurs before operation completes:
  - The **operation-end** log record is not found, and
  - The physical undo information is used to undo operation.
- ✓ If crash/rollback occurs after the operation completes:
  - The **operation-end** log record is found, and in this case
  - Logical undo is performed using U; the physical undo information for the operation is ignored.
- ✓ Redo of operation (after crash) still uses physical redo information.

Txn Rollback

- ✓ Rollback of transaction T_i is done as follows:
- ✓ Scan the log backwards

1. If a log record <T_i, X, V_1, V_2> is found, perform the undo and log a special **redo-only log record** <T_i, X, V_1>.
2. If a <T_i, O_j, **operation-end**, U> record is found
   - ♦ Rollback the operation logically using the undo information U.
     - Updates performed during roll back are logged just like during normal operation execution.
     - At the end of the operation rollback, instead of logging an **operation-end** record, generates a record <T_i, O_j, **operation-abort**>.

- ♦ Skip all preceding log records for T_i until the record <T_i, O_j **operation-begin**> is found

3. If a redo-only record is found ignore it
4. If a <T_i, O_j, **operation-abort**> record is found:
   - Skip all preceding log records for T_i until the record <T_i, O_j, **operation-begi**n> is found.
5. Stop the scan when the record <T_i, start> is found
6. Add a <T_i, **abort**> record to the log

✓ Some points to note:

- ♦ Cases 3 and 4 above can occur only if the database crashes while a transaction is being rolled back.
- ♦ Skipping of log records as in case 4 is important to prevent multiple rollback of the same operation.

Crash Recovery

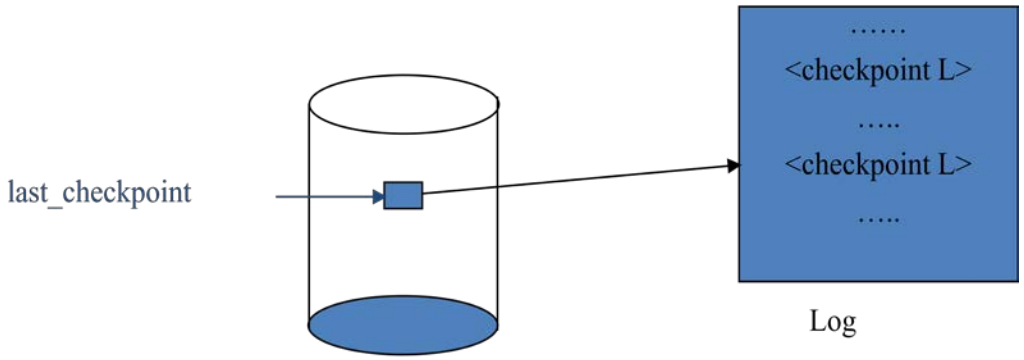
✓ The following actions are taken when recovering from system crash

1. (**Redo phase**): Scan log forward from last < **checkpoint** L> record till end of log
   - **Repeat history** by physically redoing all updates of all transactions,
   - Create an undo-list during the scan as follows
     - undo-list is set to L initially
     - Whenever <T_i **start**> is found T_i is added to undo-list
     - Whenever <T_i **commit**> or <T_i **abort**> is found, T_i is deleted from undo-list
     - This brings database to state as of crash, with committed as well as uncommitted transactions having been redone.Now undo-list contains transactions that are **incomplete**, that is, have neither committed nor been fully rolled back.
2. (**Undo phase**): Scan log backwards, performing undo on log records of transactions found in undo-list.
   - Log records of transactions being rolled back are processed as described earlier, as they are found
     - Single shared scan for all transactions being undone
   - When <T_i **start**> is found for a transaction T_i in undo-list, write a <T_i **abort**> log record.
   - Stop scan when <T_i **start**> records have been found for all T_i in undo-list

✓ This undoes the effects of incomplete transactions (those with neither **commit** nor **abort** log records). Recovery is now complete.

Check pointing

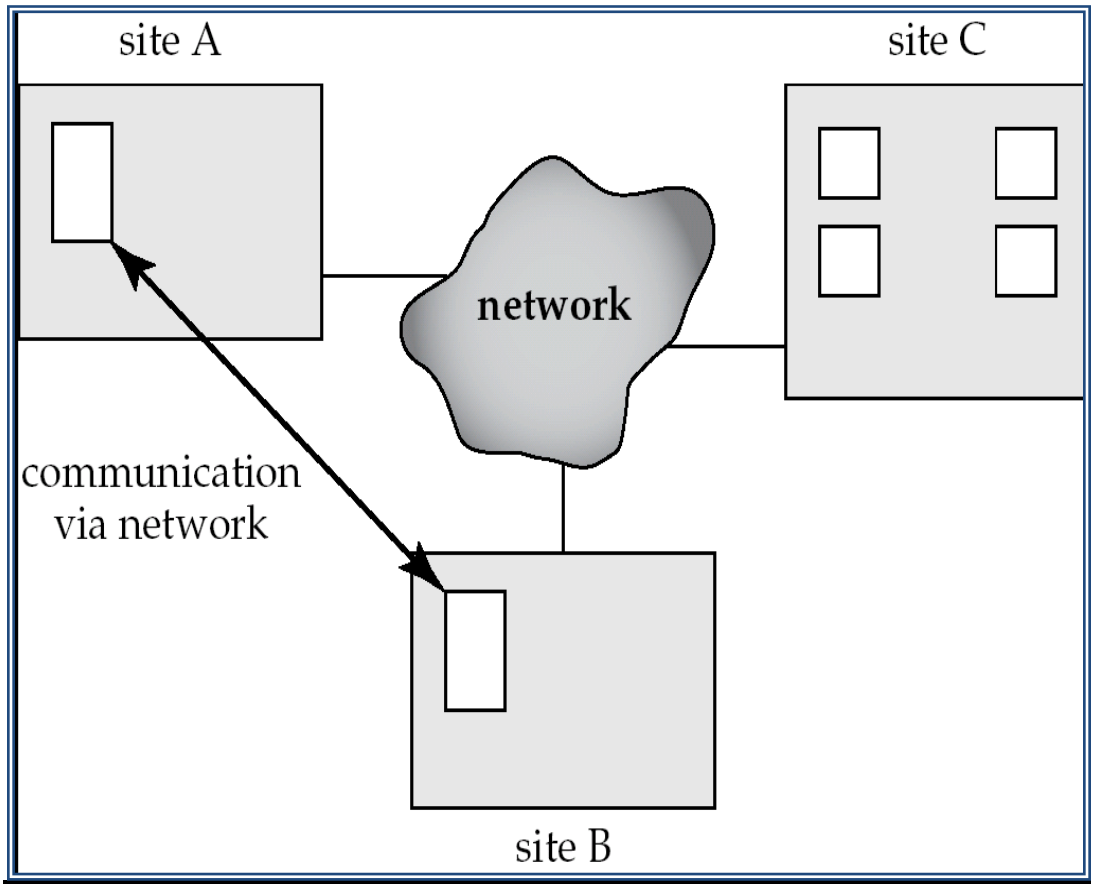
- ✓ **Check pointing** is done as follows:
  1. Output all log records in memory to stable storage
  2. Output to disk all modified buffer blocks
  3. Output to log on stable storage a < **checkpoint** L> record.
- ✓ Transactions are not allowed to perform any actions while check pointing is in progress.
- ✓ Fuzzy check pointing allows transactions to progress while the most time consuming parts of check pointing are in progress. **Fuzzy check pointing** is done as follows:
  1. Temporarily stop all updates by transactions
  2. Write a <**checkpoint** L> log record and force log to stable storage
  3. Note list M of modified buffer blocks
  4. Now permit transactions to proceed with their actions
  5. Output to disk all modified buffer blocks in list M
     - blocks should not be updated while being output
     - Follow WAL: all log records pertaining to a block must be output before the block is output
  6. Store a pointer to the **checkpoint** record in a fixed position **last_checkpoint** on disk



Chapter 9 Advanced Database Systems

9.1 Distributed database System

- ✓ A distributed database system consists of loosely coupled sites that share no physical component
- ✓ Database systems that run on each site are independent of each other
- ✓ Transactions may access data at one or more sites
- ✓ Data spread over multiple machines (also referred to as sites or nodes).
- ✓ Network interconnects the machines
- ✓ Data shared by users on multiple machines
- ✓ The main difference between the centralized and distributed system is that the data reside in on single location, where as in the later, the data reside in several locations.
- ✓ Mainly two type
  - Heterogeneous Database
  - Homogeneous Database



Homogeneous database

- ✓ All sites have identical software
- ✓ Are aware of each other and agree to cooperate in processing user requests.
- ✓ Each site surrenders part of its autonomy in terms of right to change schemas or software
- ✓ Appears to user as a single system

Heterogeneous distributed database

- ✓ Different sites may use different schemas and software
- ✓ Difference in schema is a major problem for query processing
- ✓ Difference in software is a major problem for transaction processing
- ✓ Sites may not be aware of each other and may provide only limited facilities for cooperation in transaction processing

Distributed Data Storage

- ✓ Assume relational data model stored in the database
- ✓ Two approaches to storing relation in distributed database
    1. Replication
        - System maintains multiple copies of data, stored in different sites, for faster retrieval and fault tolerance.
    2. Fragmentation
        - Relation is partitioned into several fragments stored in distinct sites
- ✓ Replication and fragmentation can be combined
        - Relation is partitioned into several fragments: system maintains several identical replicas of each such fragment.

Data Replication

- ✓ A relation or fragment of a relation is **replicated** if it is stored redundantly in two or more sites.
- ✓ **Full replication** of a relation is the case where the relation is stored at all sites.
- ✓ Fully redundant databases are those in which every site contains a copy of the entire database.
- ✓ Advantages of Replication
    - **Availability**: failure of site containing relation r does not result in unavailability of r is replicas exist.
    - **Parallelism**: queries on r may be processed by several nodes in parallel.
    - **Reduced data transfer**: relation r is available locally at each site containing a replica of r.

- ✓ Disadvantages of Replication
  - Increased cost of updates: each replica of relation *r* must be updated.
  - Increased complexity of concurrency control: concurrent updates to distinct replicas may lead to inconsistent data unless special concurrency control mechanisms are implemented.
    - ♦ One solution: choose one copy as **primary copy** and apply concurrency control operations on primary copy

Data Fragmentation

- ✓ Division of relation r into fragments r_1, r_2, …, r_n which contain sufficient information to reconstruct relation r.
- ✓ Two different schemes for fragmentation
  - **Horizontal fragmentation**:
    - ♦ each tuple of r is assigned to one or more fragments
  - **Vertical fragmentation**:
    - ♦ the schema for relation r is split into several smaller schemas
- ✓ All schemas must contain a common candidate key (or super key) to ensure lossless join property.
- ✓ A special attribute, the tuple-id attribute may be added to each schema to serve as a candidate key.
- ✓ Example : Relation account with following schema
  - Account = (branch_name, account_number, balance)
  - Horizontal Fragmentation of Account Relation

$$account_1 = \sigma_{branch\_name="Hillside"}(account)$$

$$account_2 = \sigma_{branch\_name="Valleyview"}(account)$$

  - Vertical fragmentation of Employee_Info Relation

$$deposit_1 = \Pi_{branch\_name,\ customer\_name,\ tuple\_id}(employee\_info)$$

$$deposit_2 = \Pi_{account\_number,\ balance,\ tuple\_id}(employee\_info)$$

- ✓ **Advantages of Fragmentation**
  - Horizontal:
    - ♦ allows parallel processing on fragments of a relation
    - ♦ allows a relation to be split so that tuples are located where they are most frequently accessed

- Vertical:
  - ♦ allows tuples to be split so that each part of the tuple is stored where it is most frequently accessed
  - ♦ tuple-id attribute allows efficient joining of vertical fragments
  - ♦ allows parallel processing on a relation
- Vertical and horizontal fragmentation can be mixed.
  - ♦ Fragments may be successively fragmented to an arbitrary depth.

Trade-offs in Distributed Systems

- ✓ Sharing data – users at one site able to access the data residing at some other sites.
- ✓ Autonomy – each site is able to retain a degree of control over data stored locally.
- ✓ Higher system availability through redundancy — data can be replicated at remote sites, and system can function even if a site fails.
- ✓ Disadvantage: added complexity required to ensure proper coordination among sites.
  - Software development cost.
  - Greater potential for bugs.
  - Increased processing overhead.

9.2 Parallel database systems

- ✓ Parallel database systems consist of multiple processors and multiple disks connected by a fast interconnection network.
- ✓ A coarse-grain parallel machine consists of a small number of powerful processors
- ✓ A massively parallel or fine grain parallel machine utilizes thousands of smaller processors.
- ✓ Two main performance measures:
  - throughput --- the number of tasks that can be completed in a given time interval
  - response time --- the amount of time it takes to complete a single task from the time it is submitted

- ✓ Speedup: a fixed-sized problem executing on a small system is given to a system which is N-times larger. Measured by:

$$\text{Speedup} = \frac{\text{small system elapsed time}}{\text{Large system elapsed time}}$$

- ✓ Speedup is linear if equation equals N.
- ✓ Scale up: increase the size of both the problem and the system
  - N-times larger system used to perform N-times larger job. Measured by:

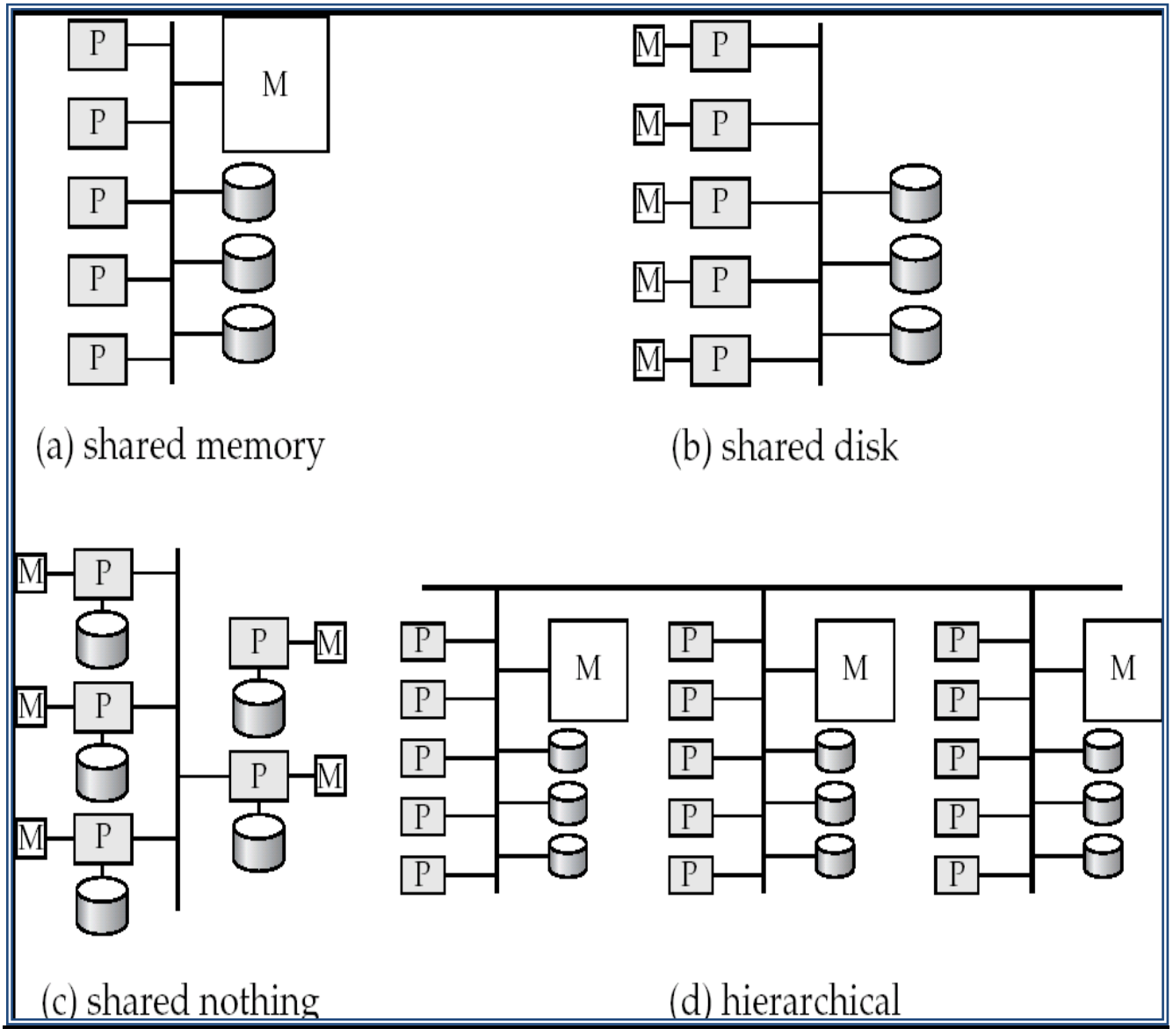
$$\text{Scale up} = \frac{\text{small system small problem elapsed time}}{\text{Big system big problem elapsed time}}$$

- ✓ Scale up is linear if equation equals 1.
- ✓ **Parallel Database Architectures**

- Shared memory -- processors share a common memory
- Shared disk -- processors share a common disk
- Shared nothing -- processors share neither a common memory nor common disk
- Hierarchical -- hybrid of the above architectures

✓ **Shared Memory**

- Processors and disks have access to a common memory, typically via a bus or through an interconnection network.
- Extremely efficient communication between processors — data in shared memory can be accessed by any processor without having to move it using software.
- Downside – architecture is not scalable beyond 32 or 64 processors since the bus or the interconnection network becomes a bottleneck
- Widely used for lower degrees of parallelism (4 to 8).



- ✓ **Shared Disk**
  - All processors can directly access all disks via an interconnection network, but the processors have private memories.
  - The memory bus is not a bottleneck
  - Architecture provides a degree of fault-tolerance — if a processor fails, the other processors can take over its tasks since the database is resident on disks that are accessible from all processors.
  - Examples: IBM Sysplex and DEC clusters (now part of Compaq) running Rdb (now Oracle Rdb) were early commercial users
  - Downside: bottleneck now occurs at interconnection to the disk subsystem.
  - Shared-disk systems can scale to a somewhat larger number of processors, but communication between processors is slower.

- ✓ **Shared Nothing**
  - Node consists of a processor, memory, and one or more disks. Processors at one node communicate with another processor at another node using an interconnection network. A node functions as the server for the data on the disk or disks the node owns.
  - Examples: Teradata, Tandem, Oracle-n CUBE
  - Data accessed from local disks (and local memory accesses) do not pass through interconnection network, thereby minimizing the interference of resource sharing.
  - Shared-nothing multiprocessors can be scaled up to thousands of processors without interference.
  - Main drawback: cost of communication and non-local disk access; sending data involves software interaction at both ends.

- ✓ **Hierarchical**
  - Combines characteristics of shared-memory, shared-disk, and shared-nothing architectures.
  - Top level is a shared-nothing architecture – nodes connected by an interconnection network, and do not share disks or memory with each other.
  - Each node of the system could be a shared-memory system with a few processors.
  - Alternatively, each node could be a shared-disk system, and each of the systems sharing a set of disks could be a shared-memory system.
  - Reduce the complexity of programming such systems by **distributed virtual-memory** architectures
    - Also called non-uniform memory architecture (NUMA)

9.3 Concept of Data Warehouse Database

- ✓ A single, complete and consistent store of data obtained from a variety of different sources made available to end users in what they can understand and use in a business context is called Data warehouse.
- ✓ A data warehouse is subject-oriented, integrated, time-variant, nonvolatile collection of data in support of management's decision making process.
- ✓ **Subject-oriented**
  - Data is arranged and optimized to provide answer to questions from diverse functional areas
  - Data is organized and summarized by topic
    - ♦ Sales / Marketing / Finance / Distribution / Etc.
  - It focuses on modeling and analysis of data for decision makers.
  - Excludes data not useful in decision support process.
- ✓ **Integrated**
  - Data Warehouse is constructed by integrating multiple heterogeneous sources.
  - Data preprocessing are applied to ensure consistency.
  - The data warehouse is a centralized, consolidated database that integrated data derived from the entire organization
    - ♦ Multiple Sources
    - ♦ Diverse Sources
    - ♦ Diverse Formats
- ✓ **Time-variant**
  - The Data Warehouse represents the flow of data through time
  - Can contain projected data from statistical models
  - Data is periodically uploaded then time-dependent data is recomputed
  - Provides information from historical perspective e.g. past 5-10 years
  - Every key structure contains either implicitly or explicitly an element of time
- ✓ **Nonvolatile**
  - Once data is entered it is NEVER removed
  - Represents the company's entire history
    - ♦ Near term history is continually added to it
    - ♦ Always growing
    - ♦ Must support terabyte databases and multiprocessors
  - Read-Only database for data analysis and query processing
  - Data warehouse requires two operations in data accessing
    - ♦ Initial loading of data
    - ♦ Access of data

Rules of a Data Warehouse

- ✓ Data Warehouse and Operational Environments are Separated
- ✓ Data is integrated
- ✓ Contains historical data over a long period of time
- ✓ Data is a snapshot data captured at a given point in time
- ✓ Data is subject-oriented
- ✓ Mainly read-only with periodic batch updates
- ✓ Development Life Cycle has a data driven approach versus the traditional process-driven approach
- ✓ Data contains several levels of detail
    - Current, Old, Lightly Summarized, Highly Summarized
- ✓ Environment is characterized by Read-only transactions to very large data sets
- ✓ System that traces data sources, transformations, and storage
- ✓ Metadata is a critical component
    - Source, transformation, integration, storage, relationships, history, etc
- ✓ Contains a chargeback mechanism for resource usage that enforces optimal use of data by end users

Phases of Data Mining

- ✓ Data Preparation
    - Identify the main data sets to be used by the data mining operation (usually the data warehouse)
- ✓ Data Analysis and Classification
    - Study the data to identify common data characteristics or patterns
        - ♦ Data groupings, classifications, clusters, sequences
        - ♦ Data dependencies, links, or relationships
        - ♦ Data patterns, trends, deviation
- ✓ Knowledge Acquisition
    - Uses the Results of the Data Analysis and Classification phase
    - Data mining tool selects the appropriate modeling or knowledge-acquisition algorithms
        - ♦ Neural Networks
        - ♦ Decision Trees
        - ♦ Rules Induction
        - ♦ Genetic algorithms
        - ♦ Memory-Based Reasoning

- ✓ Prognosis
    - Predict Future Behavior
    - Forecast Business Outcomes
    - 65% of customers who did not use a particular credit card in the last 6 months are 88% likely to cancel the account.

Need for Data Warehousing

- ✓ Industry has huge amount of operational data. Knowledge worker wants to turn this data into useful information. This information is used by them to support strategic decision making.
- ✓ It is a platform for consolidated historical data for analysis.
- ✓ It stores data of good quality so that knowledge worker can make correct decisions.
- ✓ From business perspective
    - It is latest marketing weapon
    - Helps to keep customers by learning more about their needs.
    - Valuable tool in today's competitive fast evolving world.

Application Area Of Data Warehouse

- ✓ **OLAP** (Online Analytical Processing) is a term used to describe the analysis of complex data from the data warehouse.
- ✓ **DSS** (Decision Support Systems) also known as EIS (Executive Information Systems) supports organization's leading decision makers for making complex and important decisions.
- ✓ **Data Mining** is used for knowledge discovery, the process of searching data for unanticipated new knowledge.

Characteristics of a Data Warehouse

- ✓ Subject oriented – organized based on use
- ✓ Integrated – inconsistencies removed
- ✓ Nonvolatile – stored in read-only format
- ✓ Time variant – data are normally time series
- ✓ Summarized – in decision-usable format
- ✓ Large volume – data sets are quite large
- ✓ Non normalized – often redundant
- ✓ Metadata – data about data are stored
- ✓ Data sources – comes from nonintegrated sources

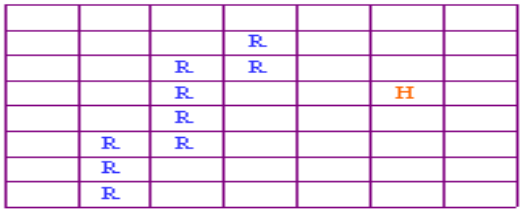
Data Warehouse Architecture

- ✓ Data Warehouse server
  - Almost always a relational DBMS ,rarely flat files
- ✓ Online Analysis Processing (OLAP) servers
  - To support and operate on multi-dimensional data structures
- ✓ Clients
  - Query and reporting tools
  - Analysis tools
  - Data mining tools

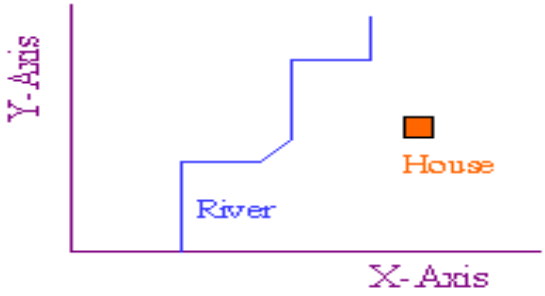
9.4 Spatial Database System

- ✓ **A spatial database system:**
  - Is a database system
    - ♦ A DBMS with additional capabilities for handling spatial data
    - ♦ Offers spatial data types (SDTs) in its data model and query language
    - ♦ Structure in space: e.g., POINT, LINE, REGION
  - Relationships among them: (l intersects r)
    - ♦ Supports SDT in its implementation providing at least
    - ♦ spatial indexing (retrieving objects in particular area without scanning the whole space)
    - ♦ efficient algorithms for spatial joins (not simply filtering the cartesian product)
- ✓ **Application area**
  - Geographical Information Systems
    - ♦ E.g. data: road network and places of interest.
    - ♦ E.g. usage: driving directions, emergency calls, standalone applications.
  - Environmental Systems
    - ♦ E.g. data: land cover, climate, rainfall, and forest fire.
    - ♦ E.g. usage: find total rainfall precipitation.
  - Corporate Decision-Support Systems
    - ♦ E.g. data: store locations and customer locations.
    - ♦ E.g. usage: determine the optimal location for a new store.
  - Battlefield Soldier Monitoring Systems
    - ♦ E.g. data: locations of soldiers (w/wo medical equipments).
    - ♦ E.g. usage: monitor soldiers that may need help from each one with medical equipment.

- ✓ **Spatial Representation**
  - **Raster model**



  - **Vector model**



- ✓ **Spatial data types**
  - Point : 2 real numbers
  - Line : sequence of points
  - Region : area included inside n-points

Modeling

- ✓ Assume 2-D and GIS application, two basic things need to be represented:
  - Objects in space: cities, forests, or rivers
    - ♦ single objects
  - Coverage/Field: say something about every point in space (e.g., partitions, thematic maps)
    - ♦ spatially related collections of objects
- ✓ Spatial primitives for objects
  - Point: object represented only by its location in space, e.g. center of a state
  - Line (actually a curve or ploy line): representation of moving through or connections in space, e.g. road, river
  - Region: representation of an extent in 2d-space, e.g. lake, city
- ✓ Coverages
  - Partition: set of **region** objects that are required to be disjoint (adjacency or region objects with common boundaries), e.g. thematic maps

- Networks: embedded graph in plane consisting of set of points (vertices) and lines (edges) objects, e.g. highways, power supply lines, rivers

✓ Spatial relationships
- Topological relationships:
  - E.g. adjacent, inside, disjoint.
  - Are invariant under topological transformations like translation, scaling, rotation
- Direction relationships:
  - E.g. above, below, or north_of, sothwest_of
- Metric relationships:
  - E.g. distance
- A valid topological relationships between two simple regions (no holes, connected): disjoint, in, touch, equal, cover, overlap

9.5 Object Oriented Database

✓ **Overview of Object-Oriented Concepts**
- OO databases try to maintain a direct correspondence between real-world and database objects so that objects do not lose their integrity and identity and can easily be identified and operated upon
- Object has two components:
  - state (value) and behavior (operations)
- OO databases, objects may have an object structure of arbitrary complexity in order to contain all of the necessary information that describes the object.
- In contrast, in traditional database systems, information about a complex object is often scattered over many relations or records, leading to loss of direct correspondence between a real-world object and its database representation.
- The internal structure of an object in OOPLs includes the specification of instance variables, which hold the values that define the internal state of the object.
- An instance variable is similar to the concept of an attribute, except that instance variables may be encapsulated within the object and thus are not necessarily visible to external users
- Some OO models insist that all operations a user can apply to an object must be predefined. This forces a complete encapsulation of objects.
- To encourage encapsulation, an operation is defined in two parts:
  - Signature or interface of the operation specifies the operation name and arguments (or parameters).
  - Method or body specifies the implementation of the operation.

- Operator polymorphism:
  - This refers to an operation's ability to be applied to different types of objects; in such a situation, an operation name may refer to several distinct implementations, depending on the type of objects it is applied to.
  - This feature is also called operator overloading
- Object-oriented databases give designer to specify
  - The structure of complex objects
  - The operations that can be applied to objects
- Maintain a direct correspondence between real-world and database objects
  - A real-world object may have different names for key attributes in different relations in traditional database systems. Example EMP_ID, SSN in different relations
- unique identity for each independent object stored in the database
  - Created by a unique, system-generated object identifier, or OID
  - The OID value of a particular object should not change

✓ **Classes and Objects**

- Class: An entity that has a well-defined role in the application domain,
- Object:
  - a particular instance of a class
  - An object has structure or state (variables) and methods (behavior/operations)
  - An object is described by four characteristics
    - Identifier: a system-wide unique id for an object
    - Name: an object may also have a unique name in DB (optional)
    - Lifetime: determines if the object is persistent or transient
    - Structure: Construction of objects using type constructors

✓ **Object Oriented Database Model**

- Represents an entity as a class
  - A class captures both attributes and behavior
- Instances of the class are called objects
- Within an object the class attributes take on specific values which distinguish one object from another
- Does not restrict to native data types such as real, integer, can use other objects
- Object Oriented Example
  - Class
    - Cat
  - Attributes
    - Color, weight, breed

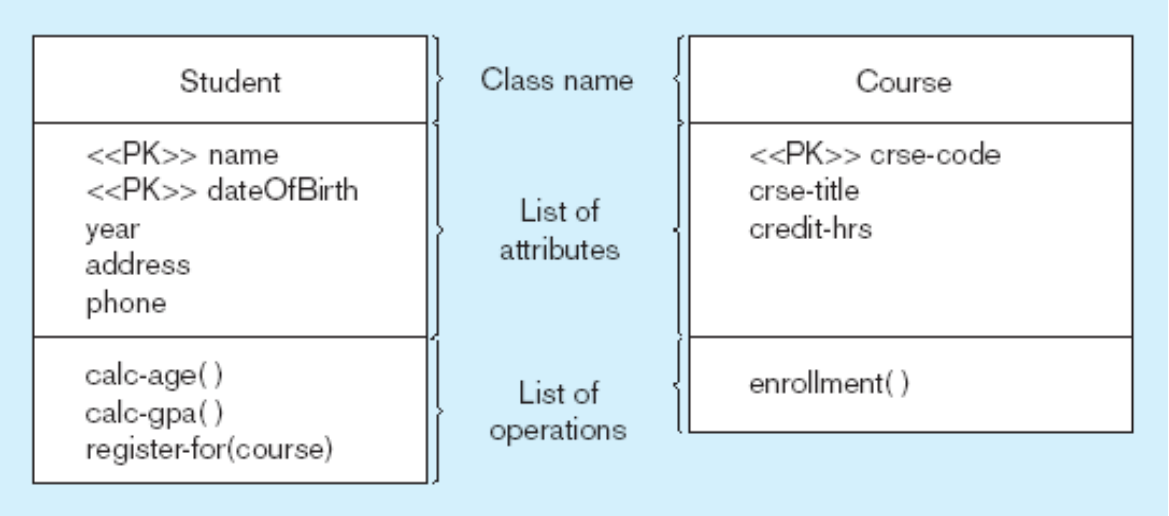
- ♦ Behavior
  - Scratches, sleeps, purrs
- ▪ An instance of the cat class is an object with specific attributes

✓ **Advantages of OODBS**

- ▪ Easier Design-Reflect Applications
- ▪ Modularity and Reusability
- ▪ Incremental refinement and abstraction
- ▪ Multiple Inheritance
- ▪ Support for multiple version And Alternatives
- ▪ Designer can specify the structure of objects and their behavior (methods)
- ▪ Better interaction with object-oriented languages such as Java and C++
- ▪ Definition of complex and user-defined types
- ▪ Encapsulation of operations and user-defined methods

✓ **Class diagram**

- ▪ Class diagram shows the static structure of an object-oriented model: object classes, internal structure, relationships.
- ▪ Class diagram showing two classes



DATABASE MANAGEMENT SYSTEMS

EX 765 06

Lecture : 3
Tutorial : 1
Practical : 3

Year : IV
Part : II

Course Objectives:

The course objective is to provide fundamental concept, theory and practices in design and implementation of Database Management System.

1. **Introduction** [3 hours]
   - 1.1. Concepts and Applications
   - 1.2. Objective and Evolution
   - 1.3. Data Abstraction and Data Independence
   - 1.4. Schema and Instances
   - 1.5. Concepts of DDL, DML and DCL

2. **Data Models** [7 hours]
   - 2.1. Logical, Physical and Conceptual
   - 2.2. E-R Model
   - 2.3. Entities and Entities sets
   - 2.4. Relationship and Relationship sets
   - 2.5. Strong and Weak Entity Sets
   - 2.6. Attributes and Keys
   - 2.7. E-R Diagram
   - 2.8. Alternate Data Model (hierarchical, network, graph)

3. **Relational Languages and Relational Model** [7 hours]
   - 3.1. Introduction to SQL
   - 3.2. Features of SQL
   - 3.3. Queries and Sub-Queries
   - 3.4. Set Operations
   - 3.5. Relations (Joined, Derived)
   - 3.6. Queries under DDL and DML Commands
   - 3.7. Embedded SQL
   - 3.8. Views
   - 3.9. Relational Algebra
   - 3.10. Database Modification
   - 3.11. QBE and domain relational calculus

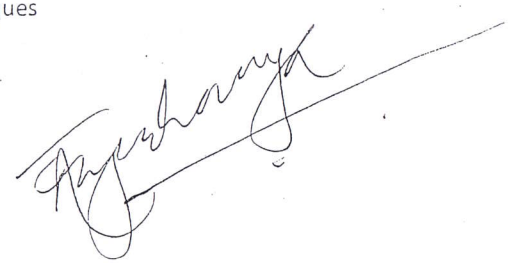
4. **Database Constraints and Normalization** [6 hours]
   - 4.1. Integrity Constraints and Domain Constraints
   - 4.2. Assertions and Triggering
   - 4.3. Functional Dependencies
   - 4.4. Multi-valued and Joined Dependencies
   - 4.5. Different Normal Forms (1st, 2nd, 3rd, BCNF, DKNF)

5. **Query Processing and Optimization** [4 hours]
   - 5.1. Query Cost Estimation
   - 5.2. Query Operations
   - 5.3. Evaluation of Expressions
   - 5.4. Query Optimization
   - 5.5. Query Decomposition
   - 5.6. Performance Tuning

6. **File Structure and Hashing** [4 hours]
   - 6.1. Records Organizations
   - 6.2. Disks and Storage
   - 6.3. Remote Backup System
   - 6.4. Hashing Concepts, Static and Dynamic Hashing
   - 6.5. Order Indices
   - 6.6. B+ tree index

7. **Transactions processing and Concurrency Control** [6 hours]
   - 7.1. ACID properties
   - 7.2. Concurrent Executions
   - 7.3. Serializability Concept
   - 7.4. Lock based Protocols
   - 7.5. Deadlock handling and Prevention

8. **Crash Recovery** [4 hours]
   - 8.1. Failure Classification
   - 8.2. Recovery and Atomicity
   - 8.3. Log-based Recovery
   - 8.4. Shadow paging
   - 8.5. Advanced Recovery Techniques



9. Advanced database Concepts [4 hours]
   9.1. Concept of Objet-Oriented and Distributed Database Model
   9.2. Properties of Parallel and Distributed Databases
   9.3. Concept of Data warehouse Database
   9.4. Concept of Spatial Database

Practical:

1: Introduction and operations of MS-Access or MySQL or any suitable DBMS

2: Database Server Installation and Configuration (MS-SQLServer, Oracle)

3: DB Client Installation and Connection to DB Server. Introduction and practice with SELECT Command with the existing DB.

4, 5: Further Practice with DML Commands

6, 7: Practice with DDL Commands. (Create Database and Tables).

8: Practice of Procedure/Trigger and DB Administration & other DBs (MySQL, PG-SQL, DB2.)

9, 10, 11: Group Project Development.

12: Project Presentation and Viva

References

1. H. F. Korth and A. Silberschatz, " *Database system concepts*", McGraw Hill, 2010.
2. A. K. Majumdar and P. Bhattacharaya, "*Database Management Systems*", Tata McGraw Hill, India, 2004.

Evaluation Scheme:

The question will cover all the chapters of the syllabus. The evaluation sche
will be as indicated in the table below:

Chapters	Hour	Marks Distribution*
1	3	4
2	7	12
3	7	12
4	6	12
5	4	8
6	4	8
7	6	12
8	4	6
9	4	6
Total	45	80

*There can be minor deviations in the numbers