# BCA204T : DATA BASE MANAGEMENT SYSTEMS

## Unit - II

Data Modelling Using the Entity-Relationship Model: High level conceptual Data Models for Database Design with and example., Entity types, Entity sets, attributes, and Keys, ER Model Concepts, Notation for ER Diagrams, Proper naming of Schema Constructs, Relationship types of degree higher than two. Record Storage and Primary File Organization: Secondary Storage Devices. Buffering of Blocks. Placing file Records on Disk. Operations on Files, File of unordered Records (Heap files), Files of Ordered Records (Sorted files), Hashing Techniques, and Other Primary file Organization.

# Unit-II

# Data Modeling Using E-R Model

### E-R Model

A logical representation of the data for an organization or for a business area is called E-R Model. It is also called has Entity-Relationship Model.

**Entity-Relationship Diagram**

A graphical representation of entity-relationship model. Also called E-R diagram or just ERD.

### ER Model: Basic Concepts

Entity relationship model defines the conceptual view of database. It works around real world entity and association among them. At view level, ER model is considered well for designing databases.

### Entity

An ***entity*** is an object that exists and which is distinguishable from other objects. An entity can be a person, a place, an object, an event, or a concept about which an organization wishes to maintain data.

For example, in a school database, student, teachers, class and course offered can be considered as entities. All entities have some attributes or properties that give them their identity.

An entity set is a collection of similar types of entities. Entity set may contain entities with attribute sharing similar values. For example, Students set may contain all the student of a school; likewise Teachers set may contain all the teachers of school from all faculties. Entities sets need not to be disjoint.

**Attributes**

An attribute is a property that describes an entity. All attributes have values. For example, a student entity may have name, class, age as attributes. There exists a domain or range of values that can be assigned to attributes. For example, a student's name cannot be a numeric value. It has to be alphabetic. A student's age cannot be negative, etc.

**Types of attributes:**

- **Simple attribute:**

  Simple attributes are atomic values, which cannot be divided further. For example, student's phone-number is an atomic value of 10 digits.

- **Composite attribute:**

  Composite attributes are made of more than one simple attribute. For example, a student's name may have Firstname and Lastname.

- **Derived attribute:**

  Derived attributes are attributes, which do not exist physical in the database, but there values are derived from other attributes presented in the database.

  For another example, Age can be derived from DOB.

- **Stored attribute:**

  An attribute whose value cannot be derived from the values of other attributes is called a stored attribute. For example, DOB

- **Single-valued attribute:**

  Single valued attributes contain on single value. For example: SocialSecurityNumber.

- **Multi-value attribute:**

  Multi-value attribute may contain more than one values.

  For example, a person can have more than one phone numbers, EmailId etc.

**Entity-set and Keys**

Key is an attribute or collection of attributes that uniquely identifies an entity among entity set.

For example, RegNo of a student makes her/him identifiable among students.

- **Super Key:** Set of attributes (one or more) that collectively identifies an entity in an entity set.
- **Candidate Key:** Minimal super key is called candidate key that is, supers keys for which no proper subset are a superkey. An entity set may have more than one candidate key.
- **Primary Key:** This is one of the candidate key chosen by the database designer to uniquely identify the entity set.

**Relationship**

The association among entities is called relationship. For example, employee entity has relation works_at with department.

Another example is for student who enrolls in some course. Here, Works_at and Enrolls are called relationship.

**Keys**

**Superkey**: an attribute or set of attributes that uniquely identifies an entity.

**Composite key**: a key requiring more than one attribute.

**Candidate key**: a superkey such that no proper subset of its attributes is also a superkey (minimal superkey – has no unnecessary attributes)

**Primary key**: the candidate key chosen to be used for identifying entities and accessing records. Unless otherwise noted "key" means "primary key"

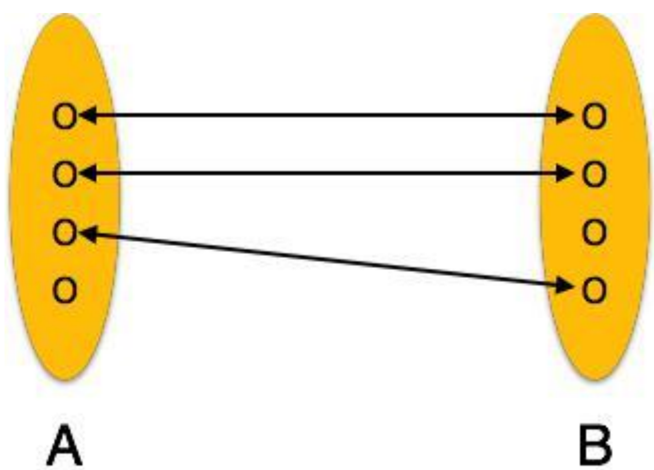
**Alternate key**: a candidate key not used for primary key

**Secondary key**: attribute or set of attributes commonly used for accessing records, but not necessarily unique

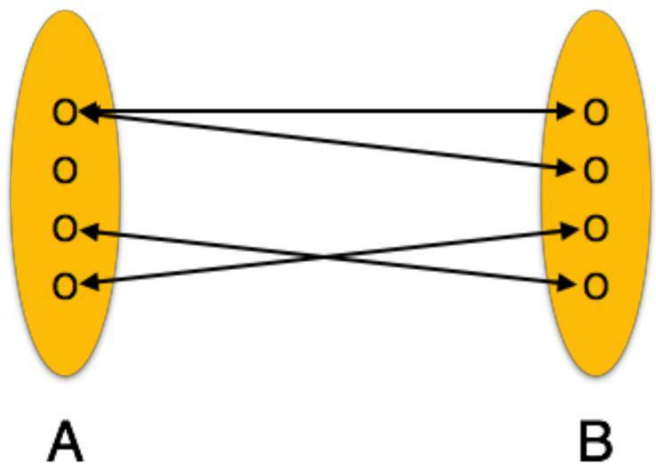
**Foreign key:** term used in relational databases (but not in the E-R model) for an attribute that is the primary key of another table and is used to establish a relationship with that table where it appears as an attribute also.
So a foreign key value occurs in the table and again in the other table.

**Mapping Cardinalities:**
**Cardinality** defines the number of entities in one entity set which can be associated to the number of entities of other set via relationship set.

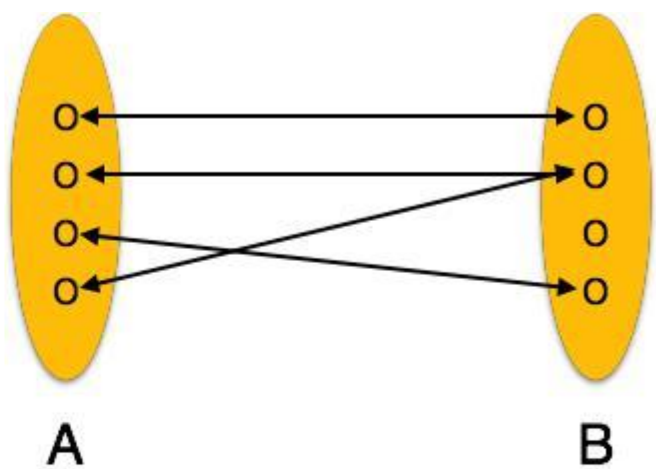
- **One-to-one:** one entity from entity set A can be associated with at most one entity of entity set B and vice versa.



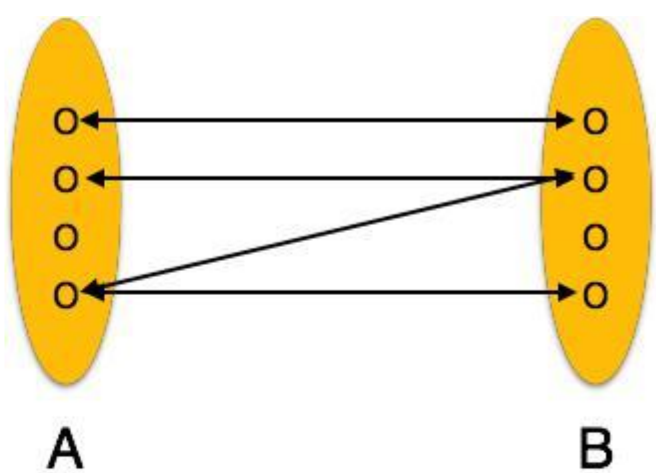

- **One-to-many:** One entity from entity set A can be associated with more than one entities of entity set B but from entity set B one entity can be associated with at most one entity.



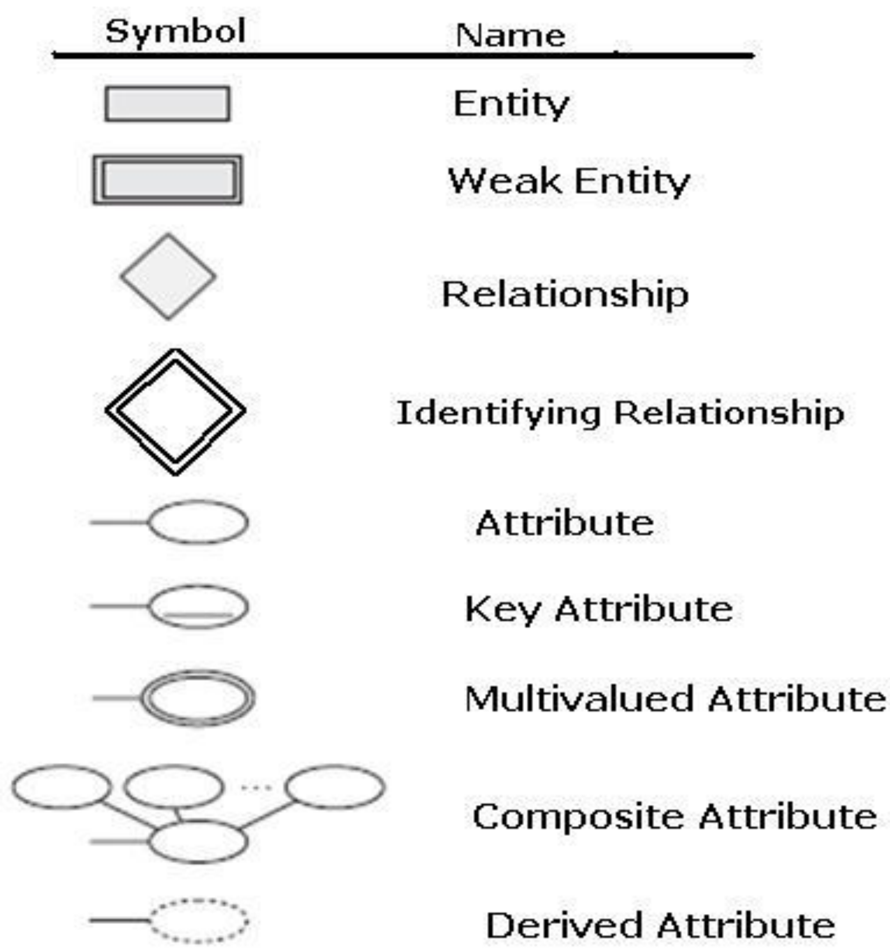

- **Many-to-one:** More than one entities from entity set A can be associated with at most one entity of entity set B but one entity from entity set B can be associated with more than one entity from entity set A.




- **Many-to-many:** one entity from A can be associated with more than one entity from B and vice versa.

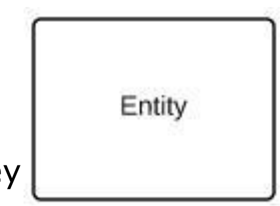


**Notations/Symbols used in E-R diagrams**




**Entity:** an entity can be any object, place person or class.

In E-R diagram entity is represented using rectangles.
For example Student is an entity.

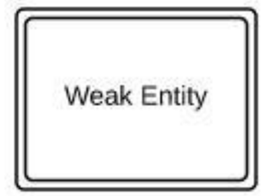



**Strong Entities** are independently from other entity types. They always possess one or more attributes that uniquely(primary key) distinguish each occurrence of the entity. For example Student is an entity.

**Weak Entities** depend on another entity.

Weak entity doesn't have key attribute of their own.

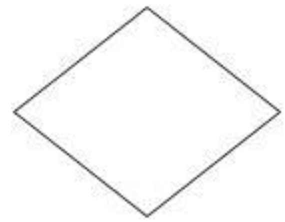
Double rectangle represents weak entity.




**Relationship**

A relationship describes relations between entities.

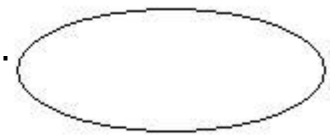
Relationship is represented using diamonds.



**Attributes:**

An attributes describes a property or characteristic of an entity.

An attribute is represented using eclipse.

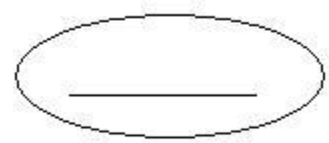


For example regno, name, course can be the attribute of student entity.

**Key Attribute**

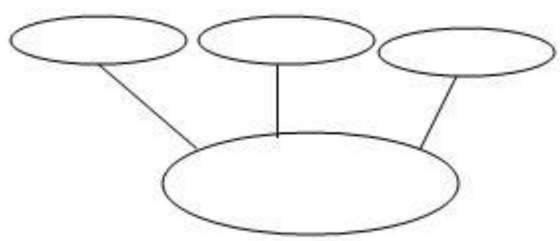
Key attribute represents the main characteristic of an entity.

It is used to represent Primary key. Ellipse with underlying lines represent key attribute.



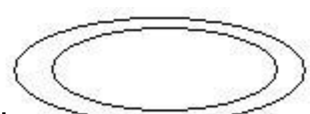
**Composite Attribute**

An attributes can be sub divided. These attributes are known as composite attribute.



**Multivalued Attribute**

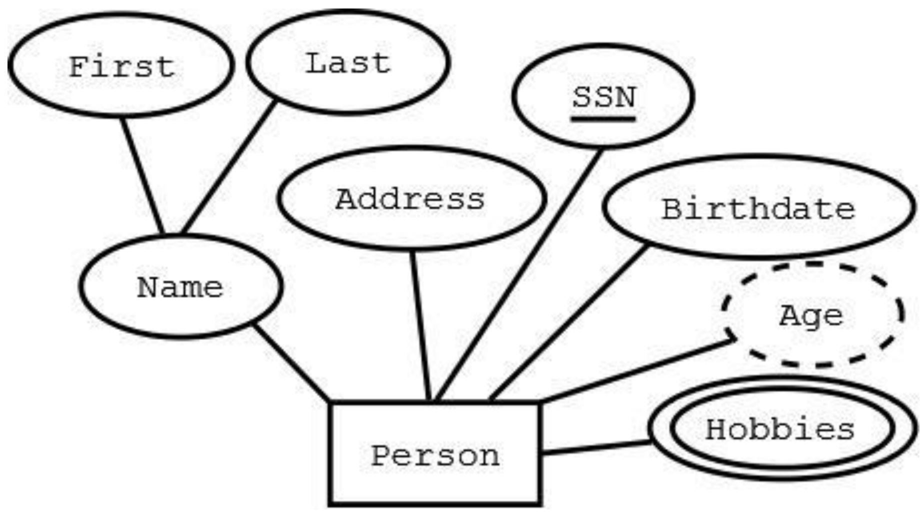
Multivalued attributes are those that are capable of taking on more than one value. It is represented by double Ellipse.



**Derived Attribute**

Derived attributes are attributes whose value can be calculated from related attribute values. They are represented by dotted ellipse.

**Example of E-R diagram**



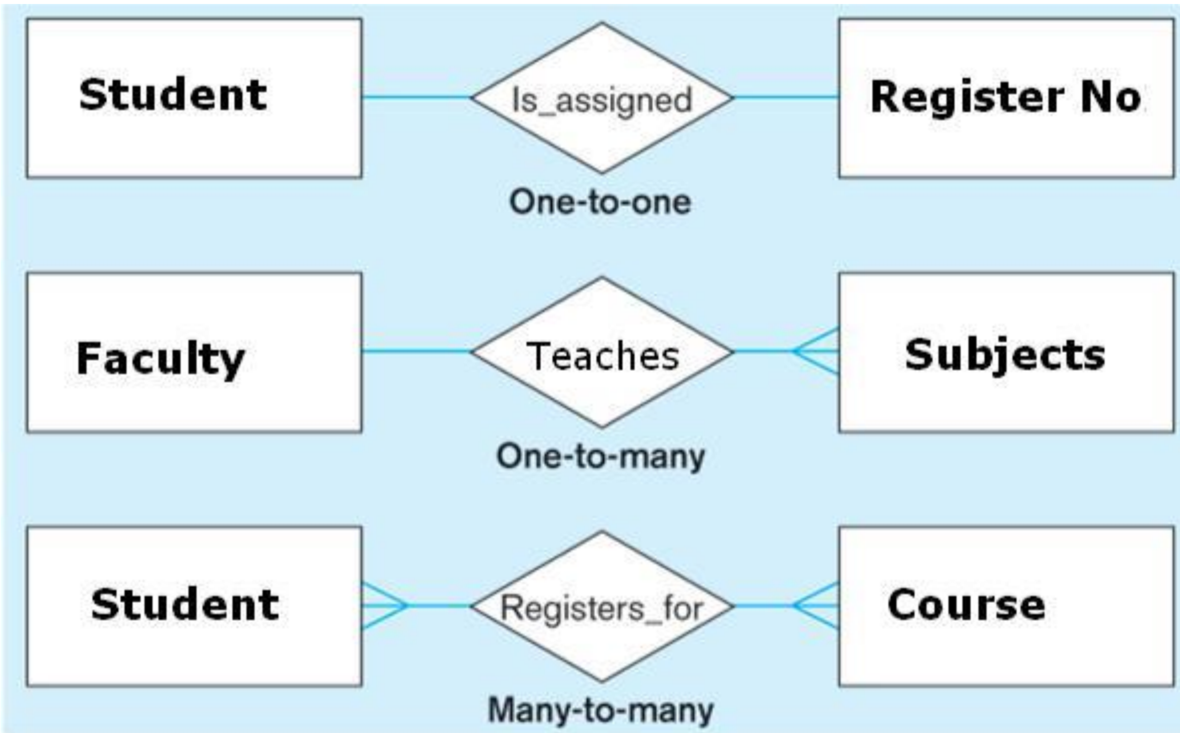

# Cardinality of Relationships

Cardinality is the number of entity instances to which another entity set can map under the relationship. This does not reflect a requirement that an entity has to participate in a relationship. Participation is another concept.

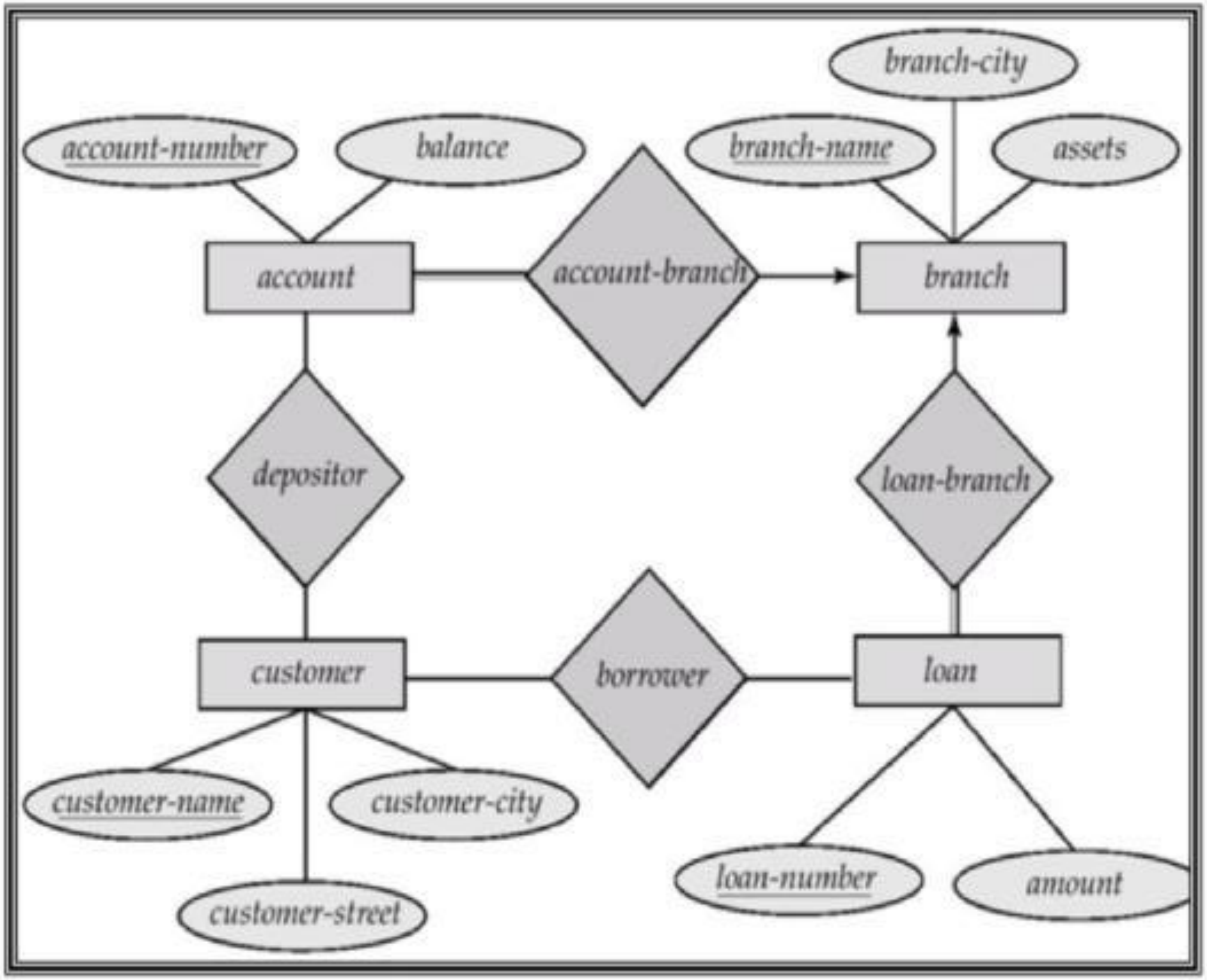
**One-to-one**: X-Y is 1:1 when each entity in X is associated with at most one entity in Y, and each entity in Y is associated with at most one entity in X.

**One-to-many**: X-Y is 1:M when each entity in X can be associated with many entities in Y, but each entity in Y is associated with at most one entity in X.

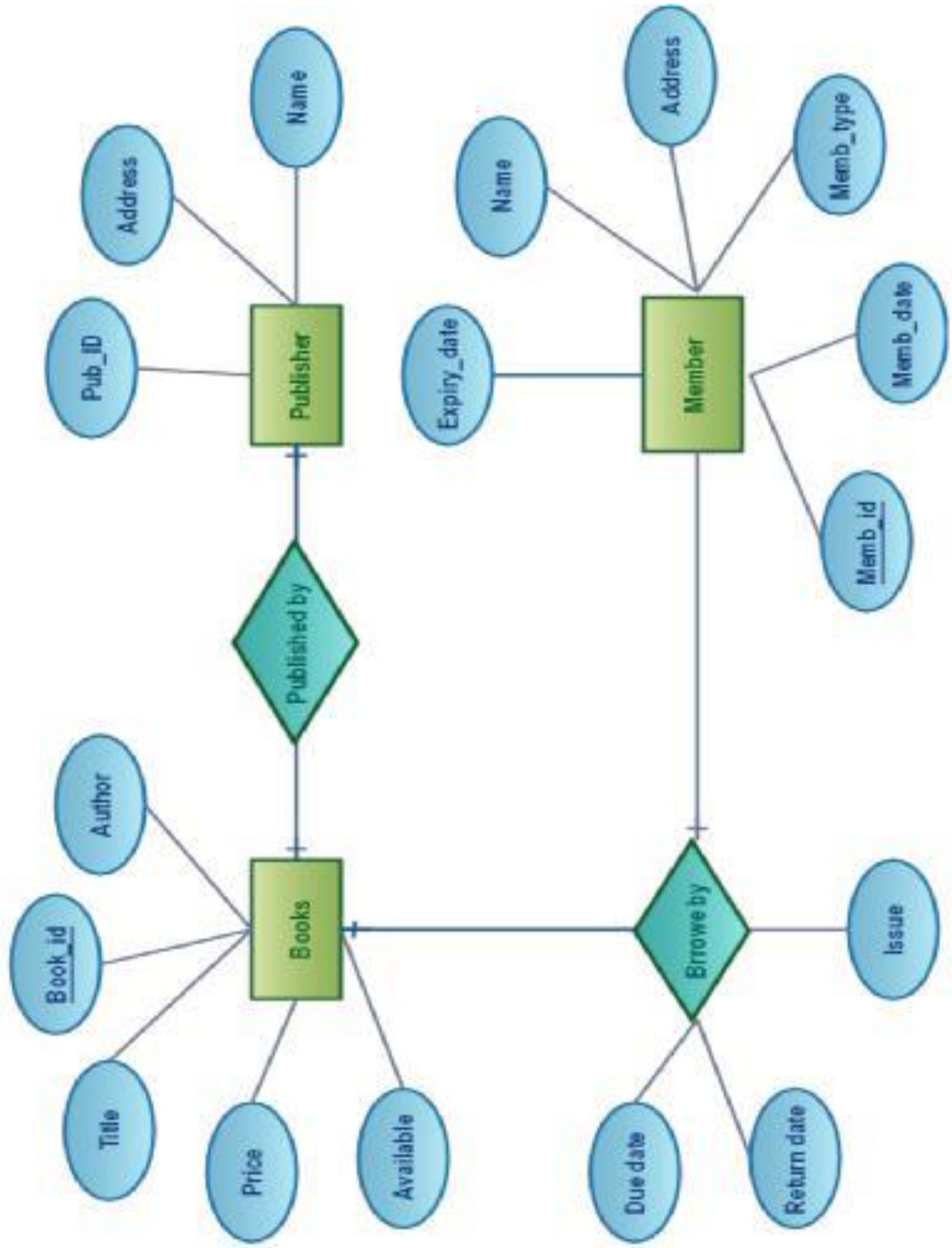
**Many-to-many**: X:Y is M:M if each entity in X can be associated with many entities in Y, and each entity in Y is associated with many entities in X ("many" =>one or more and sometimes zero)


Student
Is_assigned
Register No
One-to-one
Faculty
Teaches
Subjects
One-to-many
Student
Registers_for
Course
Many-to-many

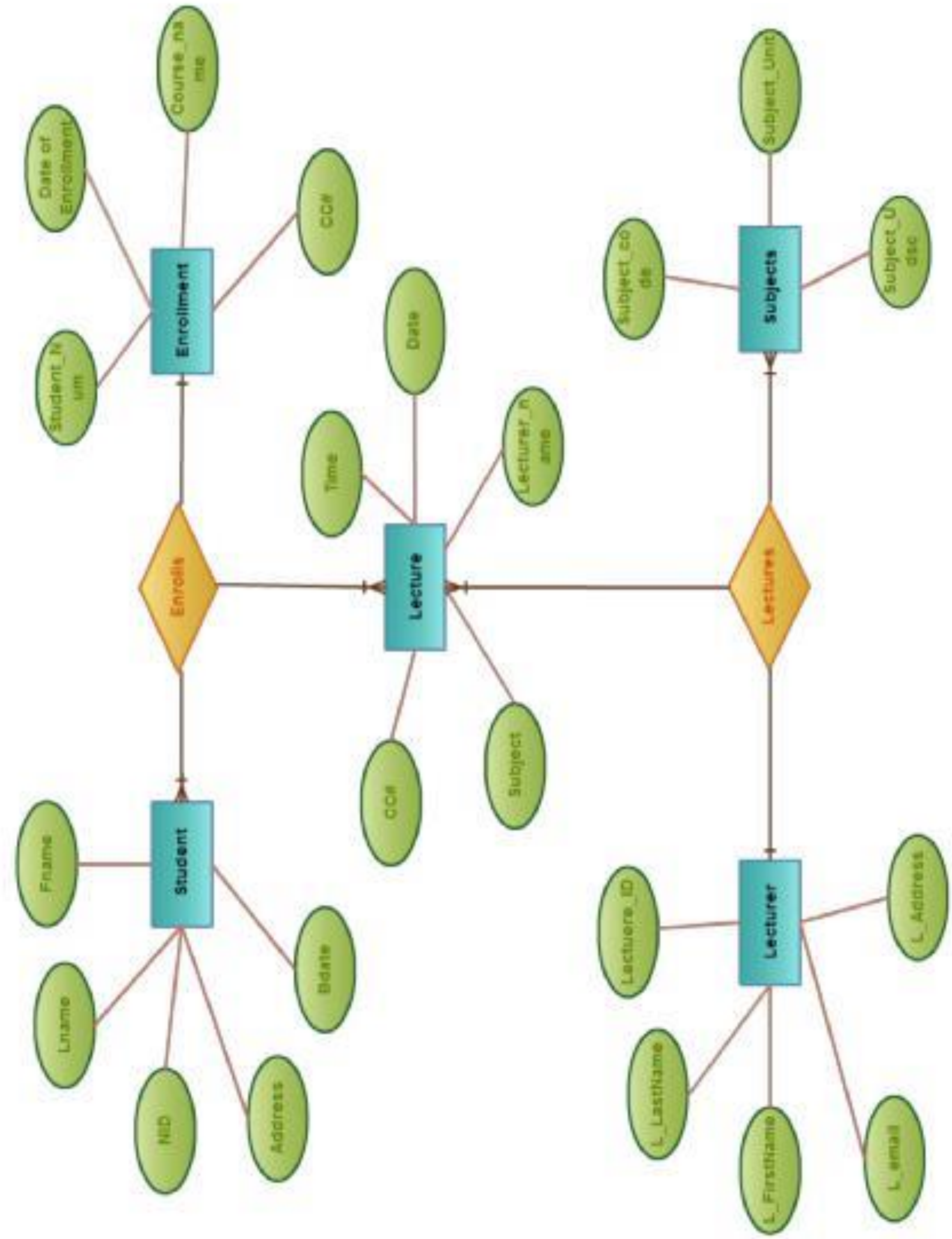
# E-R Diagram for the Banking Enterprise



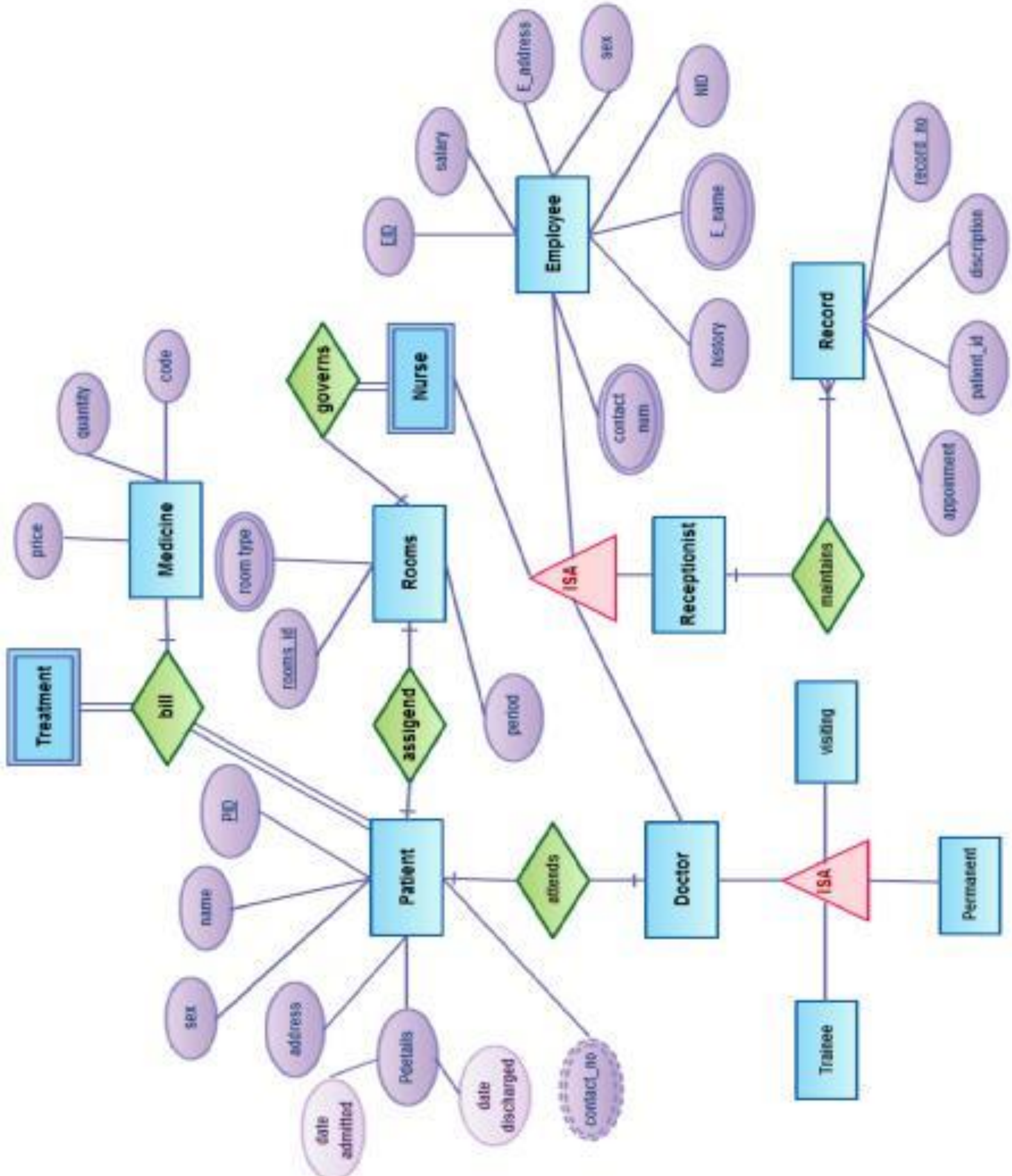
## E-R Diagram for Library Management System



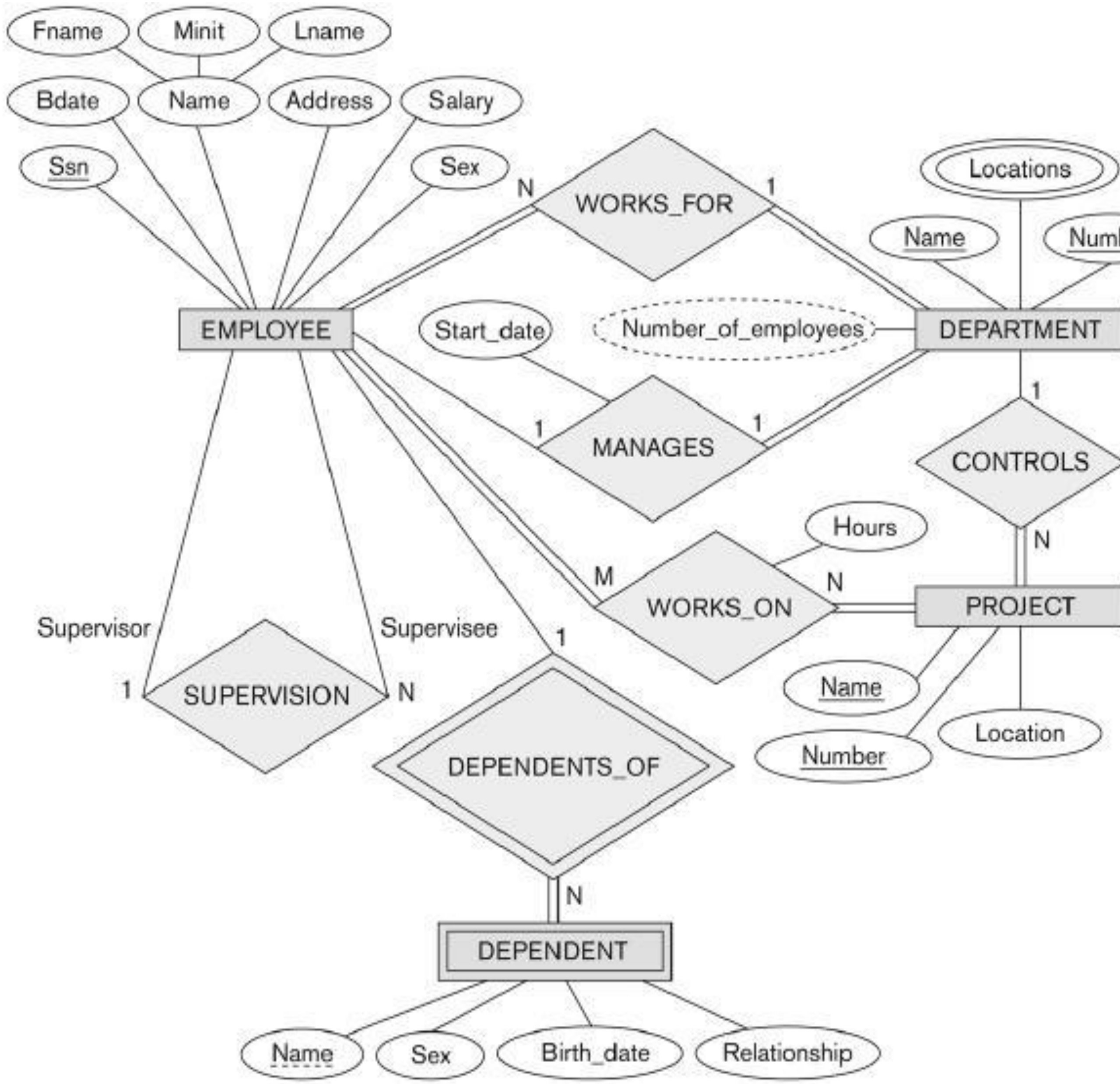
## ER DIAGRAM FOR STUDENT ENROLLMENT SYSTEM



**E-R Diagram for Hospital Management System**



**ER Diagram for Company Database**




An ER schema diagram for the COMPANY database.

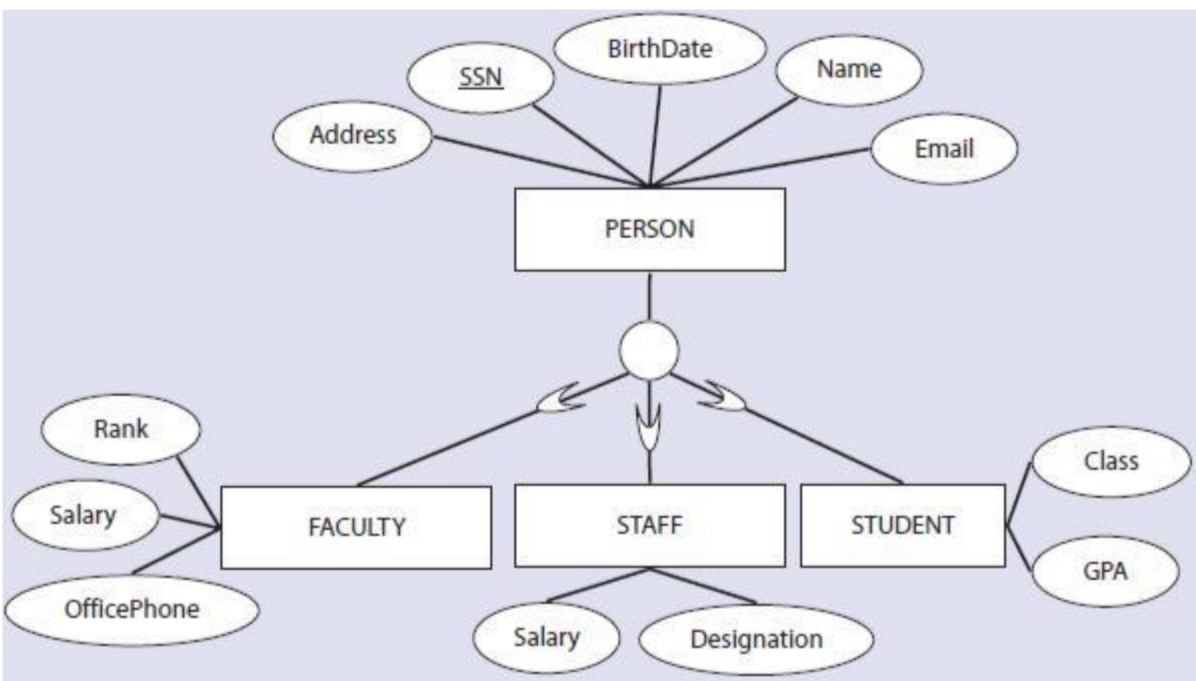
**Relationship types of degree higher than two**.

## superclass and subclass

A ***superclass*** is an entity type that has one or more distinct subgroups with unique attributes.

For example, the entity type PERSON in below fig is a superclass that includes faculty, staff, and students as its subgroups. The superclass features only those attributes that are common for all its subgroups. For example, attributes of PERSON such as *SSN*, *Name*, *Address*, and *Email* are shared by all its subgroups regardless of an individual's position as student, faculty, or staff within the university.

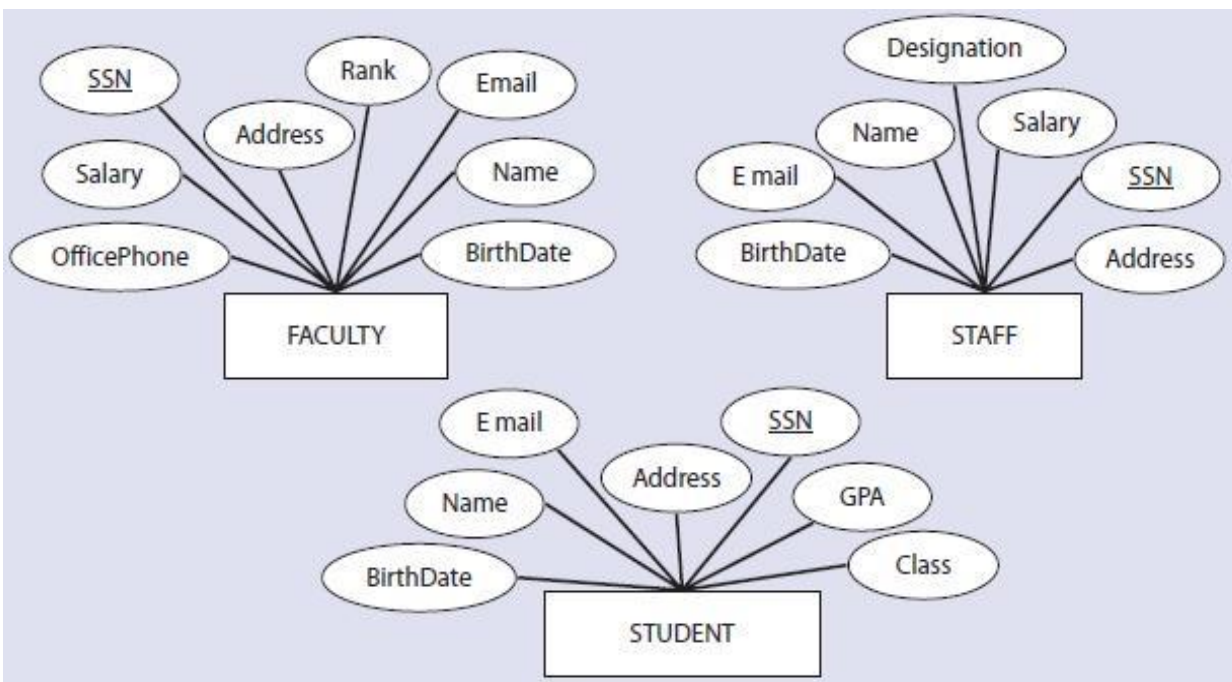
The subgroups with unique attributes are defined as ***subclasses***. The PERSON superclass thus has three subclasses: STUDENT, STAFF, and FACULTY. A subclass entity type STUDENT has attributes of its superclass along with its own attributes such as *Major*, *GPA*, and *Class* that uniquely identify the subclass. In the below fig depicts a superclass and subclasses.




## Generalization and Specialization Process

Going up in this structure is called generalization, where entities are clubbed together to represent a more generalized view. For example, a particular student named, Nisarga can be generalized along with all the students, the entity shall be student, and further a student is person. The reverse is called specialization where a person is student, and that student is Nisarga.

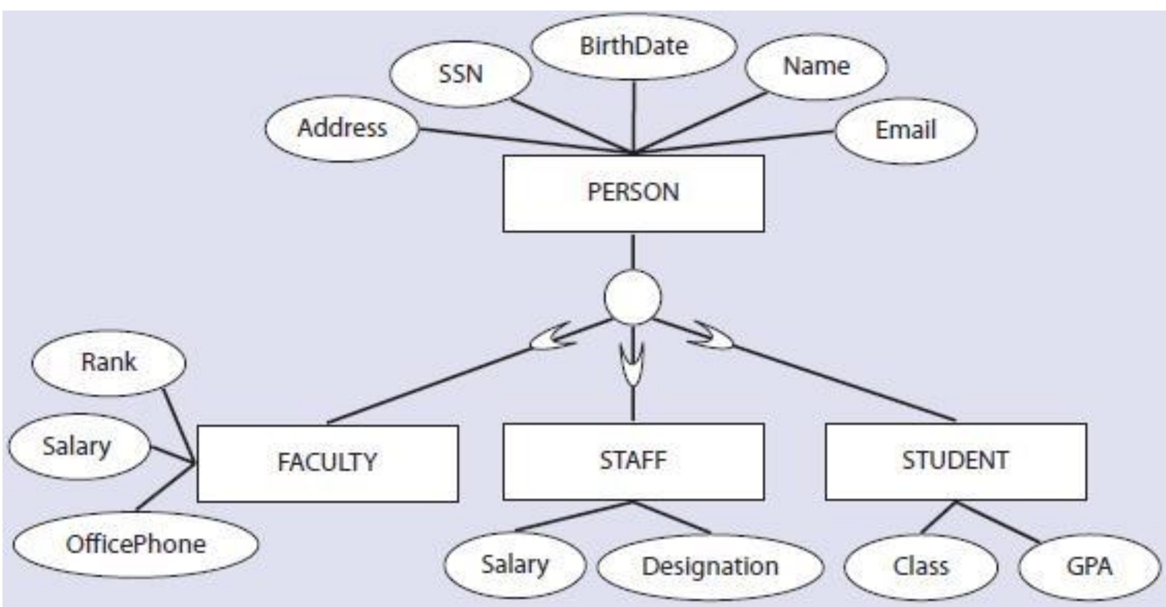
***Generalization*** is the process of defining general entity types from a set of specialized entity types by identifying their common characteristics. In other words, this process minimizes the differences between entities by *identifying a general entity type* that features the common attributes of specialized entities. Generalization is a bottom-up approach as it starts with the specialized entity types (subclasses) and forms a generalized entity type (superclass).




For example, suppose if we have given us the specialized entity types FACULTY, STAFF, and STUDENT, and we want to represent these entity types separately in the E-R model as depicted in above. However, if we examine them closely, we can observe that a number of attributes are common to all entity types, while others are specific to a particular entity.

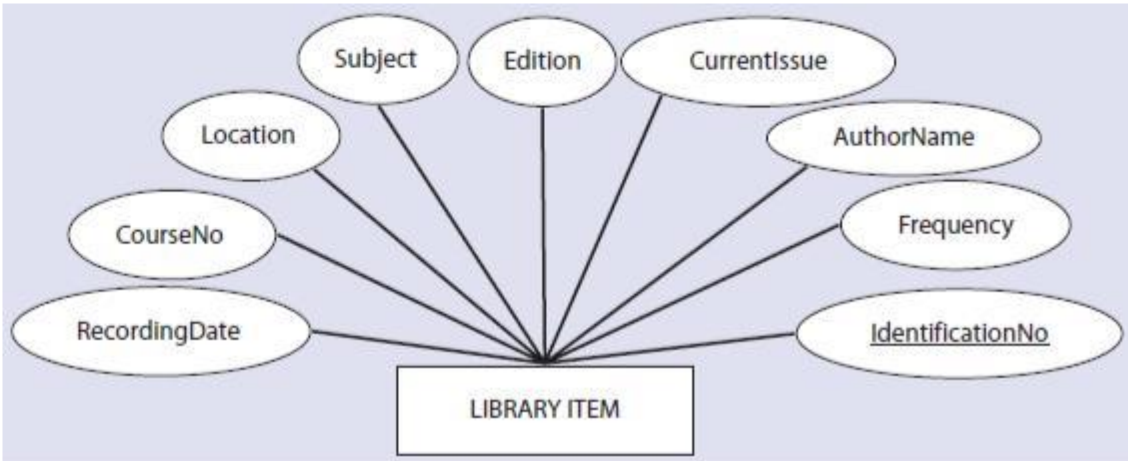
For example, FACULTY, STAFF, and STUDENT all share the attributes *Name*, *SSN*, *BirthDate*, *Address*, and *Email*. On the other hand, attributes such as *GPA*, *Class*, and *MajorDept* are specific to the STUDENTS; *OfficePhone* is specific to FACULTY, and *Designation* is specific to STAFF. Common attributes suggest that each of these three entity types is a form of a more general entity type. This general entity type is simply a PERSON superclass entity with common attributes of three subclasses (in below fig).

Thus, in the generalization process, we group specialized entity types to form one general entity type and identify common attributes of specialized entities as attributes of a general entity type. The general entity type is a superclass of specialized entity types or subclasses. Generalization is a bottom-up approach.



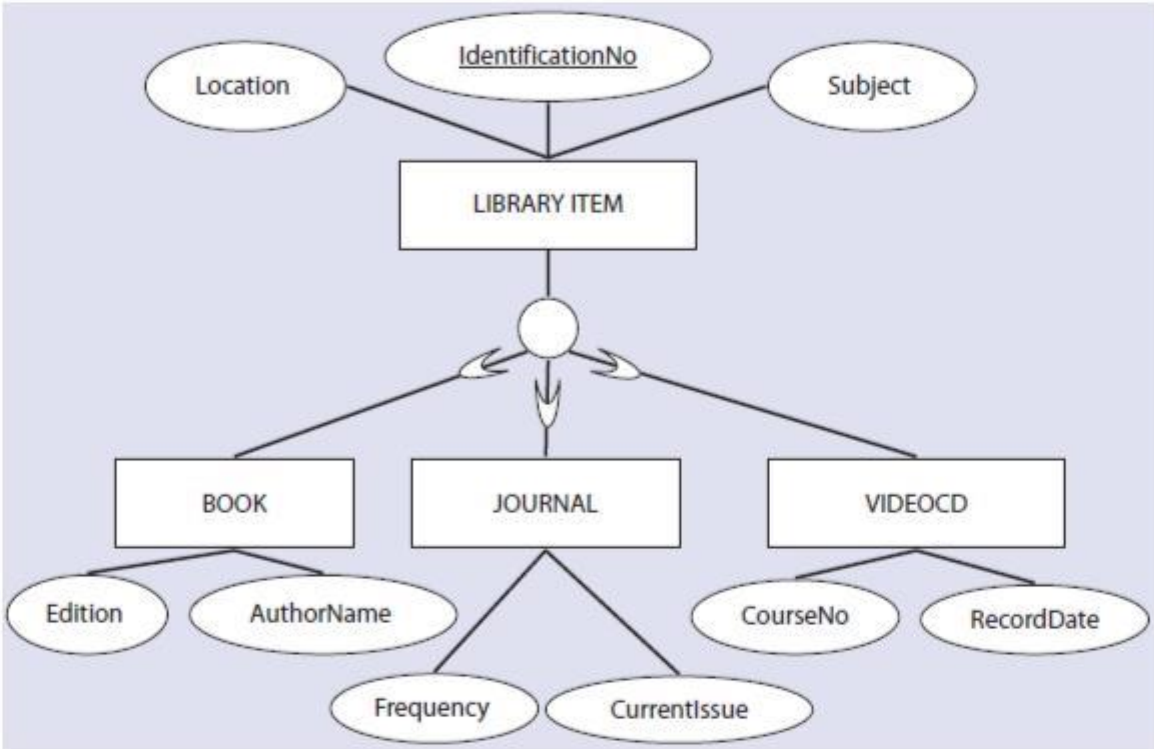

**Specialization**

Specialization is the process of defining one or more subclasses of a superclass by identifying its distinguishing characteristics. It starts with the general entity (superclass) and forms specialized entity types (subclasses) based on specialized attributes or relationships specific to a subclass.




For example, in the above Figure. LIBRARY ITEM is an entity type with several attributes such as *IdentificationNo*, *RecordingDate*, *Frequency*, and *Edition*. After careful review of these items, it should become clear that some items such as books do not have values for attributes such as *Frequency*, *RecordingDate*, and *CourseNo*, while Video CDs do not have an *Author* or an *Edition*.

In addition, all items have common attributes such as *IdentificationNo*, *Location*, and *Subject*. Someone creating a library database, then, could use the specialization process to identify superclass and subclass relationships. In this case, the original entity LIBRARY ITEM forms a superclass entity type made up of attributes shared by all items, while specialized items with distinguishing attributes, such as BOOK, JOURNALS, and VIDEOCD, form subclasses as shown in below fig. Specialization is thus a top-down approach.




**Definition**

**Generalization** is the process of defining a general entity type from a set of specialized entity types by identifying their common characteristics.

**Specialization** is a process of defining one or more subclasses of a superclass by identifying their distinguishing characteristics.

# Record Storage and Primary File Organization

The collection of data that makes up a computerized database must be stored physically on some computer **storage medium**. The DBMS software can then retrieve, update, and process this data as needed. Computer storage media form a *storage hierarchy* that includes two main categories:

**Primary storage.** This category includes storage media that can be operated on directly by the computer's *central processing unit* (CPU), such as the computer's main memory and smaller but faster cache memories. Primary storage usually provides fast access to data but is of limited storage capacity. Although main memory capacities have been growing rapidly in recent years, they are still more expensive and have less storage capacity than secondary and tertiary storage devices.

**Secondary and tertiary storage.** This category includes magnetic disks, optical disks (CD-ROMs, DVDs, and other similar storage media), and tapes. Hard-disk drives are classified as secondary storage, whereas removable media such as optical disks and tapes are considered tertiary storage. These devices usually have a larger capacity, cost less, and provide slower access to data than do primary storage devices. Data in secondary or tertiary storage cannot be processed directly by the CPU; first it must be copied into primary storage and then processed by the CPU.

**Memory Hierarchies and Storage Devices**

In a modern computer system, data resides and is transported throughout a hierarchy of storage media. The highest-speed memory is the most expensive and is therefore available with the least capacity. The lowest-speed memory is offline tape storage, which is essentially available in indefinite storage capacity.

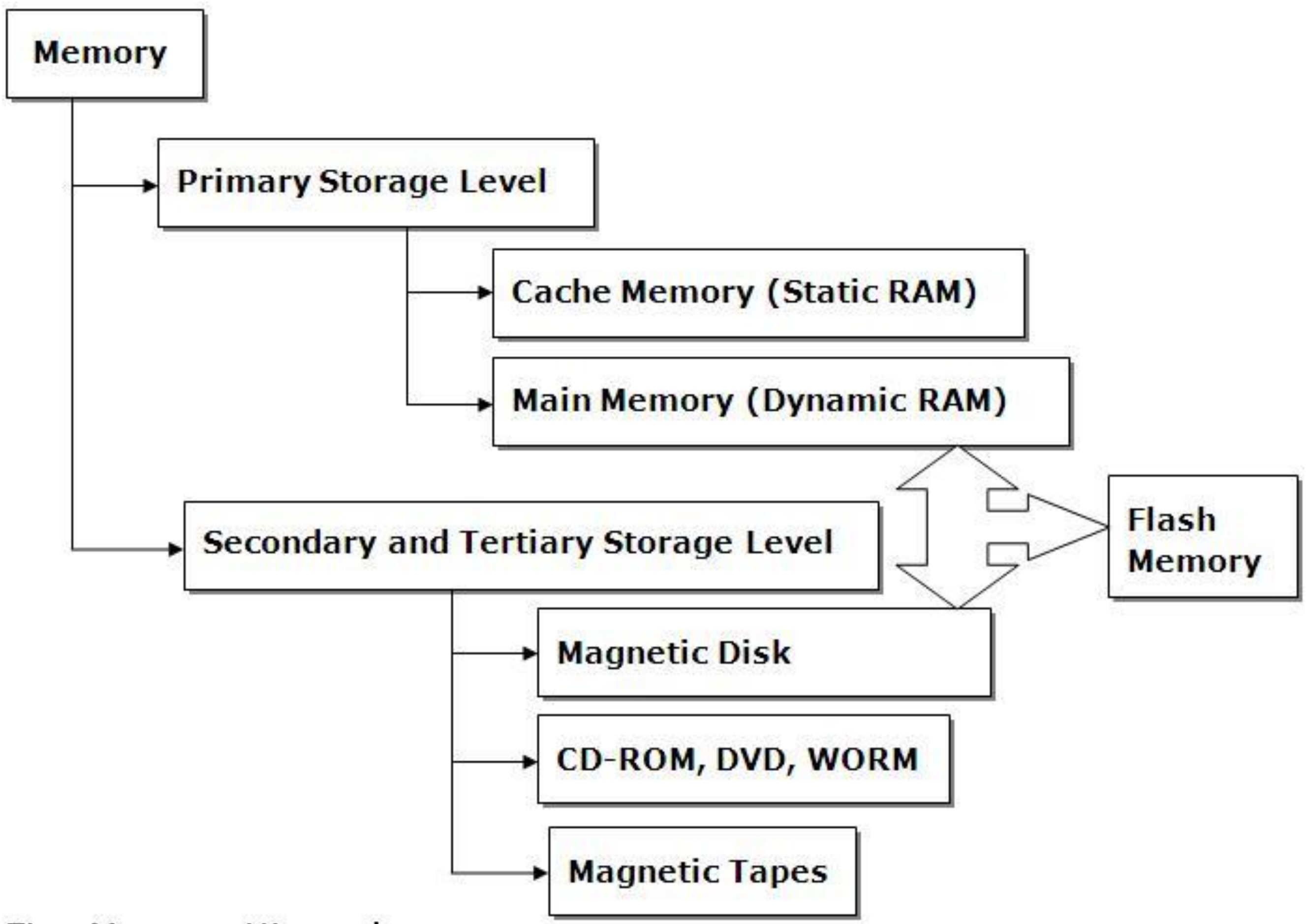



Fig: Memory Hierarchy

At the *primary storage level,* the memory hierarchy includes at the most expensive end, **cache memory**, which is a static RAM (Random Access Memory). Cache memory is typically used by the CPU to speed up execution of program instructions using techniques such as prefetching and pipelining.

The next level of primary storage is DRAM (Dynamic RAM), which provides the main work area for the CPU for keeping program instructions and data. It is popularly called **main memory**. The advantage of DRAM is its low cost, which continues to decrease; the drawback is its volatility1 and lower speed compared with static RAM.

At the *secondary and tertiary storage level,* the hierarchy includes magnetic disks, as well as **mass storage** in the form of CD-ROM (Compact Disk–Read-Only Memory) and DVD (Digital Video Disk or Digital Versatile Disk) devices, and finally tapes at the least expensive end of the hierarchy. The **storage capacity** is measured in kilobytes (Kbyte or 1000 bytes), megabytes (MB or 1 million bytes), gigabytes (GB or 1 billion bytes), and even terabytes (1000 GB). The word petabyte (1000 terabytes or 10**15 bytes) is now becoming relevant in the context of very large repositories of data in physics, astronomy, earth sciences, and other scientific applications.

Programs reside and execute in DRAM. Generally, large permanent databases reside on secondary storage, (magnetic disks), and portions of the database are read into and written from buffers in main memory as needed. Nowadays, personal computers and workstations have large main memories of hundreds of megabytes of RAM and DRAM, so it is becoming possible to load a large part of the database into main memory.

Between DRAM and magnetic disk storage, another form of memory, **flash memory**, is becoming common, particularly because it is nonvolatile. Flash memories are high-density, high-performance memories using EEPROM (Electrically Erasable Programmable Read-Only Memory) technology.

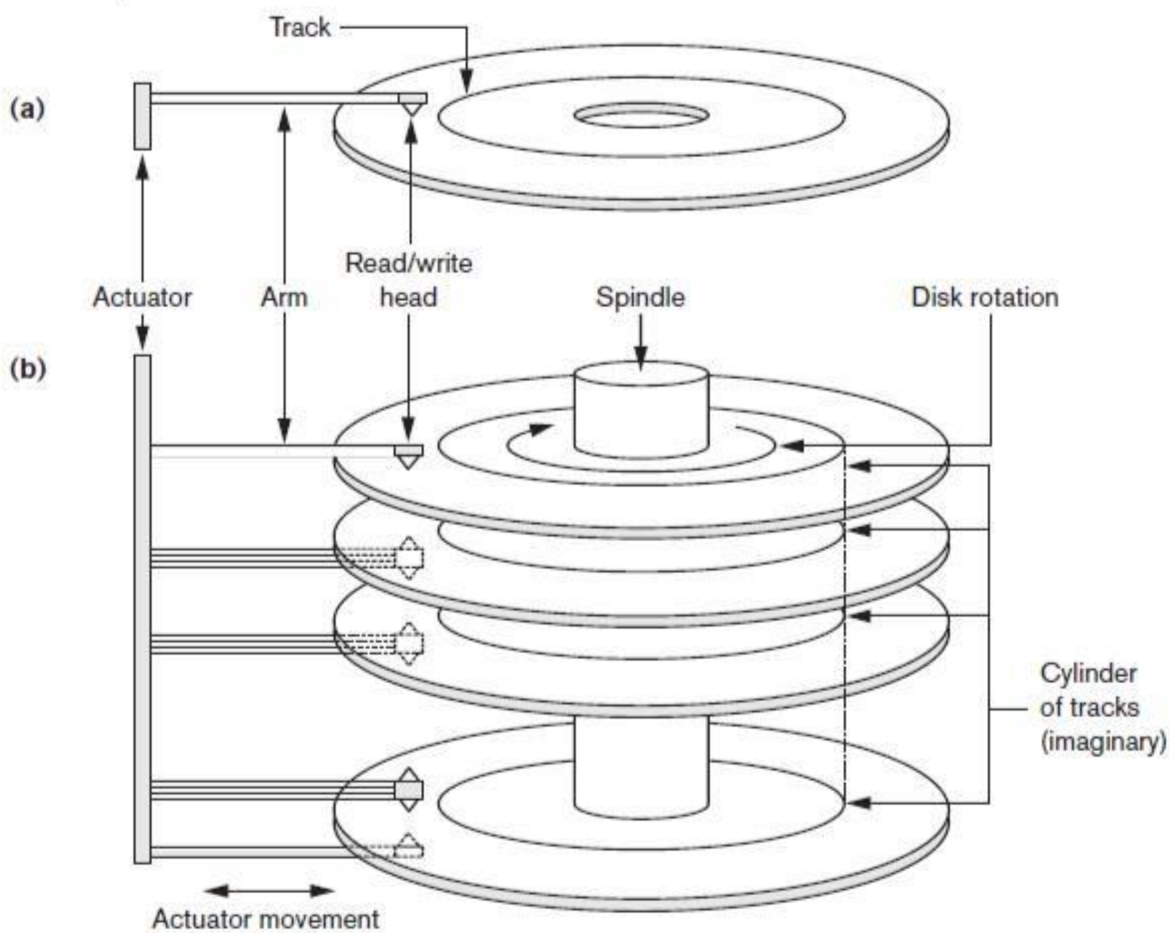
The advantage of flash memory is the fast access speed; the disadvantage is that an entire block must be erased and written over simultaneously. Flash memory cards are appearing as the data storage medium in appliances with capacities ranging from a few megabytes to a few gigabytes.

These are appearing in cameras, MP3 players, cell phones, PDAs, and so on. USB (Universal Serial Bus) flash drives have become the most portable medium for carrying data between personal computers; they have a flash memory storage device integrated with a USB interface.
**CD-ROM** (Compact Disk – Read Only Memory) disks store data optically and are read by a laser. CD-ROMs contain prerecorded data that cannot be overwritten.
**WORM** (Write-Once-Read-Many) disks are a form of optical storage used for archiving data; they allow data to be written once and read any number of times without the possibility of erasing. They hold about half a gigabyte of data per disk and last much longer than magnetic disks.
The **DVD** is another standard for optical disks allowing 4.5 to 15 GB of storage per disk. Most personal computer disk drives now read CDROM and DVD disks. Typically, drives are CD-R (Compact Disk Recordable) that can create CD-ROMs and audio CDs (Compact Disks), as well as record on DVDs.
Finally, **magnetic tapes** are used for archiving and backup storage of data. **Tape jukeboxes**—which contain a bank of tapes that are catalogued and can be automatically loaded onto tape drives—are becoming popular as **tertiary storage** to hold terabytes of data. For example, NASA's EOS (Earth Observation Satellite) system stores archived databases. Many large organizations are already finding it normal to have terabyte-sized databases.
The term **very large database** can no longer be precisely defined because disk storage capacities are on the rise and costs are declining. Very soon the term may be reserved for databases containing tens of terabytes.

**Secondary Storage Devices**

Magnetic disks are used for storing large amounts of data. The most basic unit of data on the disk is a single **bit** of information. Bits are grouped into **bytes** (or **characters**). Byte sizes are typically 4 to 8 bits, depending on the computer and the device. We assume that one character is stored in a single byte, and we use the terms *byte* and *character* interchangeably. The **capacity** of a disk is the number of bytes it can store, Hard disks for personal computers typically hold from several hundred MB up to tens of GB; and large disk packs used with servers and mainframes have capacities of hundreds of GB. Disk capacities continue to grow as technology improves.

**Figure**
(a) A single-sided disk with read/write hardware.
(b) A disk pack with read/write hardware.



Whatever their capacity, all disks are made of magnetic material shaped as a thin circular disk, as shown in Figure(a), and protected by a plastic or acrylic cover.
A disk is **single-sided** if it stores information on one of its surfaces only and **doublesided** if both surfaces are used. To increase storage capacity, disks are assembled into a **disk pack**, as shown in Figure (b), which may include many disks and therefore many surfaces. Information is stored on a disk surface in concentric circles of *small width,* each having a distinct diameter. Each circle is called a **track**. In disk packs, tracks with the same diameter on the various surfaces are called a **cylinder** because of the shape they would form if connected in space. The concept of a cylinder is important because data stored on one cylinder can be retrieved much faster than if it were distributed among different cylinders.
The number of tracks on a disk ranges from a few hundred to a few thousand, and the capacity of each track typically ranges from tens of Kbytes to 150 Kbytes. Because a track usually contains a large amount of information, it is divided into smaller blocks or sectors. The division of a track into **sectors** is hard-coded on the disk surface and cannot be changed.
The division of a track into equal-sized **disk blocks** (or **pages**) is set by the operating system during disk **formatting** (or **initialization**). Block size is fixed during initialization and cannot be changed dynamically. Typical disk block sizes range from 512 to 8192 bytes. A disk with hard-coded sectors often has the sectors subdivided into blocks during initialization. Blocks are separated by fixed-size **interblock gaps**, which include specially coded control information written during disk initialization. This information is used to determine which block on the track follows each interblock gap.

A disk is a *random access* addressable device. Transfer of data between main memory and disk takes place in units of disk blocks. The **hardware address** of a block—a combination of a cylinder number, track number (surface number within the cylinder on which the track is located), and block number (within the track) is supplied to the disk I/O (input/output) hardware.

The address of a **buffer**—a contiguous reserved area in main storage that holds one disk block—is also provided. For a **read** command, the disk block is copied into the buffer; whereas for a **write** command, the contents of the buffer are copied into the disk block. Sometimes several contiguous blocks, called a **cluster**, may be transferred as a unit. In this case, the buffer size is adjusted to match the number of bytes in the cluster.

The actual hardware mechanism that reads or writes a block is the disk **read/write head**, which is part of a system called a **disk drive**.A disk or disk pack is mounted in the disk drive, which includes a motor that rotates the disks. A read/write head includes an electronic component attached to a **mechanical arm**. Disk packs with multiple surfaces are controlled by several read/write heads—one for each surface, as shown in Figure (b). All arms are connected to an **actuator** attached to another electrical motor, which moves the read/write heads in unison and positions them precisely over the cylinder of tracks specified in a block address.
Disk drives for hard disks rotate the disk pack continuously at a constant speed (typically ranging between 5,400 and 15,000 rpm(revolutions per minute)). Once the read/write head is positioned on the right track and the block specified in the block address moves under the read/write head, the electronic component of the read/write head is activated to transfer the data. Some disk units have fixed read/write heads, with as many heads as there are tracks. These are called **fixed-head** disks, whereas disk units with an actuator are called **movable-head disks**.

For fixed-head disks, a track or cylinder is selected by electronically switching to the appropriate read/write head rather than by actual mechanical movement; consequently, it is much faster. However, the cost of the additional read/write heads is quite high, so fixed-head disks are not commonly used.

A **disk controller**, typically embedded in the disk drive, controls the disk drive and interfaces it to the computer system. The controller accepts high-level I/O commands and takes appropriate action to position the arm and causes the read/write action to take place. To transfer a disk block, given its address, the disk controller must first mechanically position the read/write head on the correct track. The time required to do this is called the **seek time**. Typical seek times are 5 to 10 msec on desktops and 3 to 8 msecs on servers. There is another delay—called the **rotational delay** or **latency**—while the beginning of the desired block rotates into position under the read/write head. It depends on the rpm of the disk. Some additional time is needed to transfer the data; this is called the **block transfer time**. Hence, the total time needed to locate and transfer an arbitrary block, given its address, is the sum of the seek time, rotational delay, and block transfer time.

**Buffering of Blocks**

When several blocks need to be transferred from disk to main memory and all the block addresses are known, several buffers can be reserved in main memory to speed up the transfer. While one buffer is being read or written, the CPU can process data in the other buffer because an independent disk I/O processor (controller) exists that, once started, can proceed to transfer a data block between memory and disk independent of and in parallel to CPU processing.

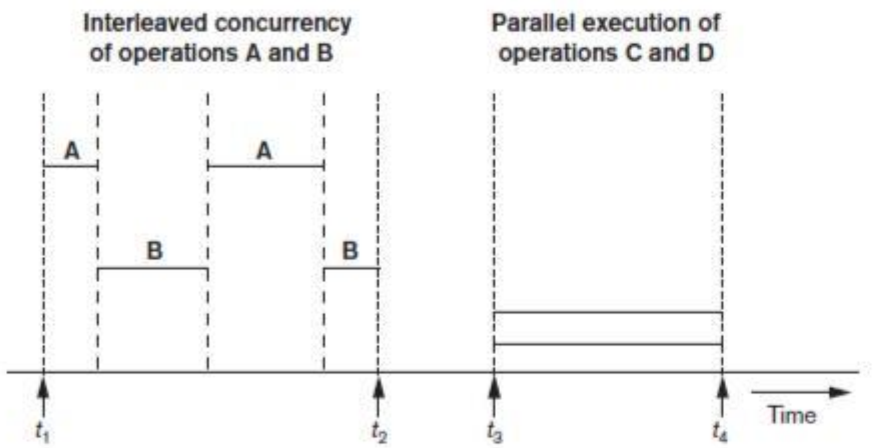



**Figure (a)**
Interleaved concurrency versus parallel execution.

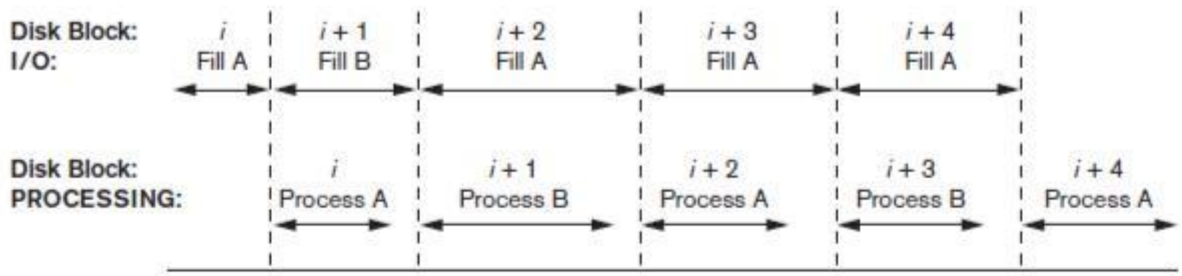



**Figure (b)**
Use of two buffers, A and B, for reading from disk.

Figure (a) illustrates how two processes can proceed in parallel. Processes A and B are running **concurrently** in an **interleaved** fashion, whereas processes C and D are running **concurrently** in a **parallel** fashion. When a single CPU controls multiple processes, parallel execution is not possible. However, the processes can still run concurrently in an interleaved way. Buffering is most useful when processes can run concurrently in a parallel fashion, either because a separate disk I/O processor is available or because multiple CPU processors exist.

Figure (b) illustrates how reading and processing can proceed in parallel when the time required to process a disk block in memory is less than the time required to read the next block and fill a buffer. The CPU can start processing a block once its transfer to main memory is completed; at the same time, the disk I/O processor can be reading and transferring the next block into a different buffer. This technique is called **double buffering** and can also be used to read a continuous stream of blocks from disk to memory. Double buffering permits continuous reading or writing of data on consecutive disk blocks, which eliminates the seek time and rotational delay for all but the first block transfer. Moreover, data is kept ready for processing, thus reducing the waiting time in the programs.

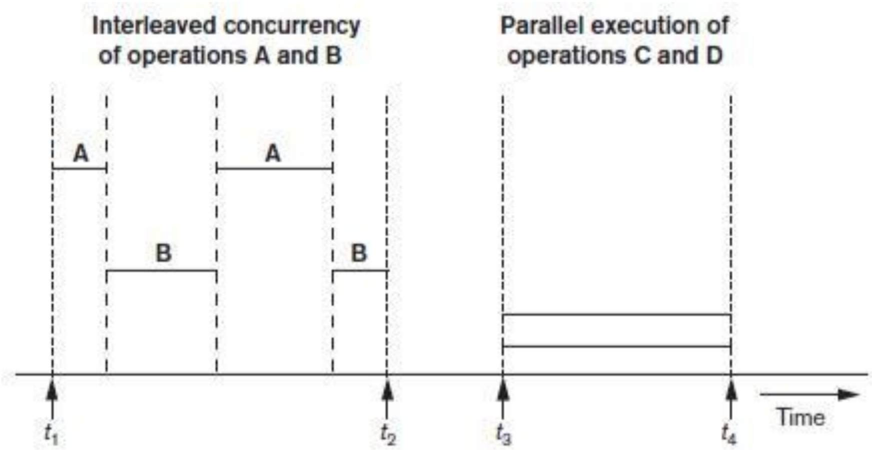



**Figure**
Interleaved concurrency versus parallel execution.

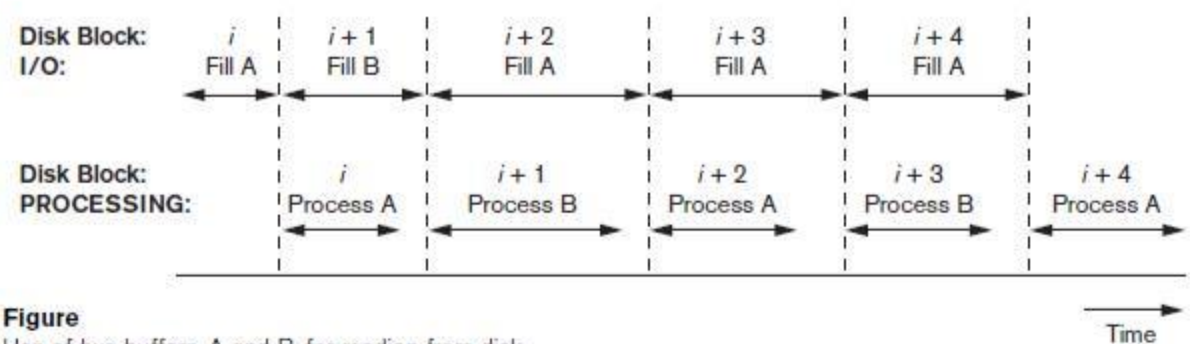



**Figure**
Use of two buffers, A and B, for reading from disk.

## Placing file Records on Disk

Data is usually stored in the form of **records**. Each record consists of a collection of related data **values** or **items**, where each value is formed of one or more bytes and corresponds to a particular **field** of the record. Records usually describe entities and their attributes.

For example, an EMPLOYEE record represents an employee entity, and each field value in the record specifies some attribute of that employee, such as EmployeeID, Name, DOB, Department, Salary.

A collection of field names and their corresponding data types constitutes a **record type** or **record format** definition.

A **data type**, associated with each field, specifies the types of values a field can take.

The data type of a field is usually include numeric (integer, long integer, or floating point), string of characters (fixed-length or varying), Boolean (having 0 and 1 or TRUE and FALSE values only), and sometimes specially coded **date** and **time** data types.

**Files, Fixed-Length Records, and Variable-Length Records**

A **file** is a *sequence* of records. In many cases, all records in a file are of the same record type. If every record in the file has exactly the same size (in bytes), the file is said to be made up of **fixed-length records**. If different records in the file have different sizes, the file is said to be made up of **variable-length records**.

**Allocating File Blocks on Disk**

There are several standard techniques for allocating the blocks of a file on disk.

In **contiguous allocation**, the file blocks are allocated to consecutive disk blocks. This makes reading the whole file very fast using double buffering, but it makes expanding the file difficult.

In **linked allocation**, each file block contains a pointer to the next file block. This makes it easy to expand the file but makes it slow to read the whole file. A combination of the two allocates **clusters** of consecutive disk blocks, and the clusters are linked. Clusters are sometimes called **file segments** or **extents**.

In **indexed allocation**, where one or more **index blocks** contain pointers to the actual file blocks. It is also common to use combinations of these techniques.

**File Headers**

A **file header** or **file descriptor** contains information about a file that is needed by the system programs that access the file records. The header includes information to determine the disk addresses of the file blocks as well as to record format descriptions, which may include field lengths and the order of fields within a record for fixed-length unspanned records and field type codes, separator characters, and record type codes for variable-length records.

**Operations on Files**

Operations on files are usually grouped into **retrieval operations** and **update operations**.

The **retrieval operations** do not change any data in the file, but only locate certain records so that their field values can be examined and processed. The latter change the file by insertion or deletion of records or by modification of field values. In either case, we may have to **select** one or more records for retrieval, deletion, or modification based on a **selection condition** (or **filtering condition**), which specifies criteria that the desired record or records must satisfy.

Actual operations for locating and accessing file records vary from system to system. DBMS software programs, access records by using these commands.

**Open.** Prepares the file for reading or writing. Allocates appropriate buffers (typically at least two) to hold file blocks from disk, and retrieves the file header. Sets the file pointer to the beginning of the file.

**Reset.** Sets the file pointer of an open file to the beginning of the file.

**Find (or Locate).** Searches for the first record that satisfies a search condition. Transfers the block containing that record into a main memory buffer (if it is not already there). The file pointer points to the record in the buffer and it becomes the *current record*

**Read (or Get).** Copies the current record from the buffer to a program variable in the user program. This command may also advance the current record pointer to the next record in the file, which may necessitate reading the next file block from disk.

**FindNext.** Searches for the next record in the file that satisfies the search condition. Transfers the block containing that record into a main memory buffer.

**Delete.** Deletes the current record and (eventually) updates the file on disk to reflect the deletion.

**Modify.** Modifies some field values for the current record and (eventually) updates the file on disk to reflect the modification.

**Insert.** Inserts a new record in the file by locating the block where the record is to be inserted, transferring that block into a main memory buffer (if it is not already there), writing the record into the buffer, and (eventually) writing the buffer to disk to reflect the insertion.

**Close.** Completes the file access by releasing the buffers and performing any other needed cleanup operations.

The preceding (except for Open and Close) are called **record-at-a-time** operations

because each operation applies to a single record. It is possible to streamline the operations Find, FindNext, and Read into a single operation, Scan, whose description is as follows:

**Scan.** If the file has just been opened or reset, *Scan* returns the first record; otherwise it returns the next record. If a condition is specified with the operation, the returned record is the first or next record satisfying the condition.

**FindAll.** Locates *all* the records in the file that satisfy a search condition.

**Find (or Locate) *n*.** Searches for the first record that satisfies a search condition and then continues to locate the next $n - 1$ records satisfying the same condition. Transfers the blocks containing the $n$ records to the main memory buffer (if not already there).

**FindOrdered.** Retrieves all the records in the file in some specified order. **Reorganize.** Starts the reorganization process. As we shall see, some file organizations require periodic reorganization. An example is to reorder the file records by sorting them on a specified field.

**Files of Unordered Records (Heap Files)**

In this simplest and most basic type of organization, records are placed in the file in the order in which they are inserted, so new records are inserted at the end of the file. Such an organization is called a **heap** or **pile file**.

**Inserting a new record** is *very efficient.* The last disk block of the file is copied into a buffer, the new record is added, and the block is then **rewritten** back to disk. The address of the last file block is kept in the file header.

**Searching for a record** using any search condition involves a **linear search** through the file block by block—an expensive procedure. If only one record satisfies the search condition, then, on the average, a program will read into memory and search half the file blocks before it finds the record. For a file of $b$ blocks, this requires searching ($b/2$) blocks, on average. If no records or several records satisfy the search condition, the program must read and search all $b$ blocks in the file.

To **delete a record**, a program must first find its block, copy the block into a buffer, delete the record from the buffer, and finally **rewrite the block** back to the disk. This leaves unused space in the disk block. Deleting a large number of records in this way results in wasted storage space. Another technique used for record deletion is to have an extra byte or bit, called a **deletion marker**, stored with each record. A record is deleted by setting the deletion marker to a certain value.

**Files of Ordered Records (Sorted Files)**

We can physically order the records of a file on disk based on the values of one of their fields—called the **ordering field**. This leads to an **ordered** or **sequential** file. If the ordering field is also a **key field** of the file—a field guaranteed to have a unique value in each record—then the field is called the **ordering key** for the file.

A **binary search** for disk files can be done on the blocks rather than on the records. Suppose that the file has *b* blocks numbered 1, 2, ..., *b*; the records are ordered by ascending value of their ordering key field; and we are searching for a record whose ordering key field value is *K*.Assuming that disk addresses of the file blocks are available in the file header, the binary search can be described by Algorithm shown below.

Binary Search on an Ordering Key of a Disk File

```
L ← 1; U ← b; (* b is the number of file blocks *)
while (U ≥ L ) do
    begin i ← (L + U) div 2;
    read block i of the file into the buffer;
    if K < (ordering key field value of the first record in block i )
        then U ← i – 1
    else if K > (ordering key field value of the last record in block i )
        then L ← i + 1
    else if the record with ordering key field value = K is in the buffer
        then goto found
    else goto notfound;
    end;
goto notfound;
```

A binary search usually accesses log2(*b*) blocks, whether the record is found or not—an improvement over linear searches, where, on the average, (*b*/2) blocks are accessed when the record is found and *b* blocks are accessed when the record is not found.

**Inserting and deleting records** are expensive operations for an ordered file because the records must remain physically ordered. To insert a record, we must find its correct position in the file, based on its ordering field value, and then make space in the file to insert the record in that position. For a large file this can be very time consuming because, on the average, half the records of the file must be moved to make space for the new record. This means that half the file blocks must be read and rewritten after records are moved among them. For record deletion, the problem is less severe if deletion markers and periodic reorganization are used.

**Modifying** a field value of a record depends on two factors: the search condition to locate the record and the field to be modified. If the search condition involves the ordering key field, we can locate the record using a binary search; otherwise we must do a linear search. A nonordering field can be modified by changing the record and rewriting it in the same physical location on disk—assuming fixed-length records. Modifying the ordering field means that the record can change its position in the file. This requires deletion of the old record followed by insertion of the modified record.

## Hashing Techniques

Hashing is a method which provides very fast access to records under certain search conditions. This organization is usually called a **hash file**. The search condition must be an equality condition on a single field, called the **hash field**. In most cases, the hash field is also a key field of the file, in which case it is called the **hash key**. The idea behind hashing is to provide a function *h*, called a **hash function** or **randomizing function**, which is applied to the hash field value of a record and yields the *address* of the disk block in which the record is stored. A search for the record within the block can be carried out in a main memory buffer. For most records, we need only a single-block access to retrieve that record.

Hashing is also used as an internal search structure within a program whenever a group of records is accessed exclusively by using the value of one field. Hashing can be used for internal files. it can be modified to store external files on disk. It can also be extended to dynamically growing files.

**Internal Hashing**

Hashing is typically implemented as a **hash table** through the use of an array of records. Let the array index range is from 0 to M−1, as shown in Figure(a); then we have *M* **slots** whose addresses correspond to the array indexes. We choose a hash function that transforms the hash field value into an integer between 0 and M-1. One common hash function is the $h(K) = K$ **mod *M*** function, which returns the remainder of an integer hash field value *K* after division by *M*; this value is then used for the record address.

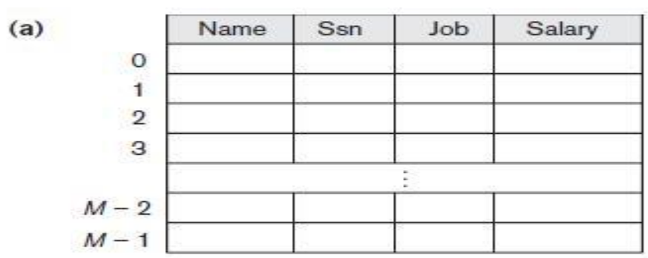
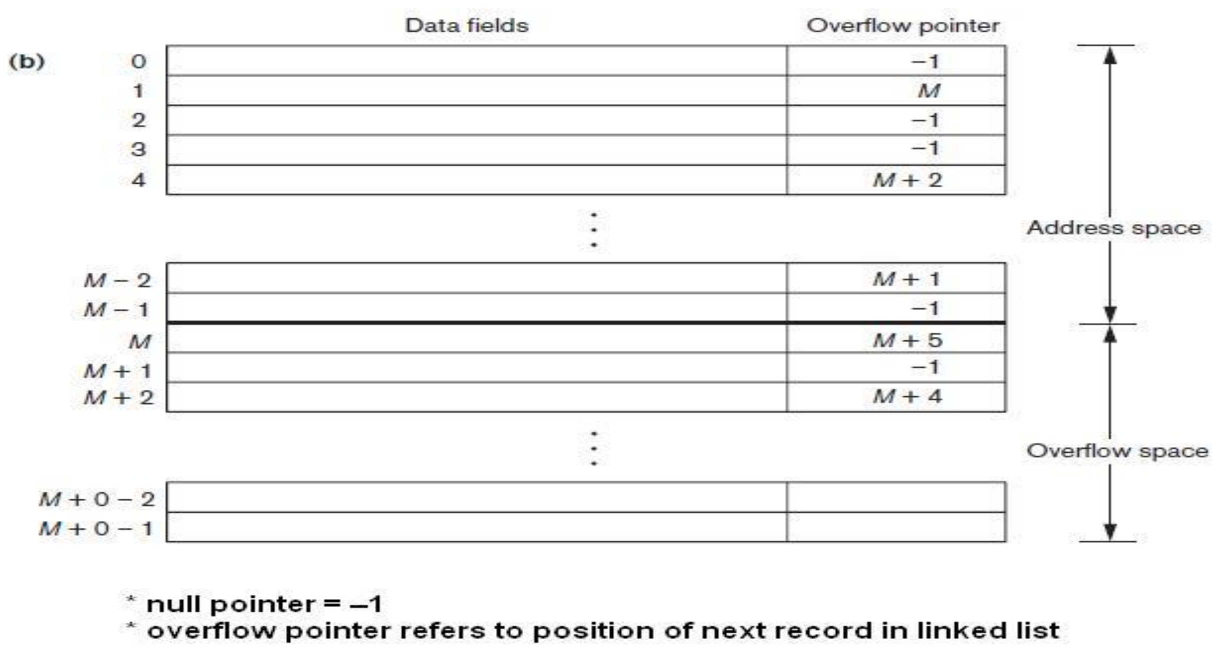



**Figure**
Internal hashing data structures. (a) Array of *M* positions for use in internal hashing. (b) Collision resolution by chaining records.



Noninteger hash field values can be transformed into integers before the mod function is applied. For character strings, the numeric (ASCII) codes associated with characters can be used in the transformation—for example, by multiplying those code values. For a hash field whose data type is a string of 20 characters, Algorithm (a) can be used to calculate the hash address.

**Algorithm** Two simple hashing algorithms: (a) Applying the mod hash function to a character string K. (b) Collision resolution by open addressing.

**(a)** *temp* ← 1;

for *i* ← 1 to 20 do *temp* ← *temp* * code(*K*[*i* ] ) mod *M* ;

*hash_address* ← *temp* mod *M*;

**(b)** *i* ← *hash_address*(*K*); *a* ← *i*;

if location *i* is occupied

then **begin** *i* ← (*i* + 1) mod *M*;

while (*i* ≠ *a*) and location *i* is occupied

do *i* ← (*i* + 1) mod *M*;

if (*i* = *a*) then all positions are full

else *new_hash_address* ← *i*;

**end**;

Other hashing functions can be used. One technique, called **folding**, involves applying an arithmetic function such as *addition* or a logical function such as *exclusive or* to different portions of the hash field value to calculate the hash address.

A **collision** occurs when the hash field value of a record that is being inserted hashes to an address that already contains a different record. In this situation, we must insert the new record in some other position. The process of finding another position is called **collision resolution**.

There are numerous methods for collision resolution, including the following:

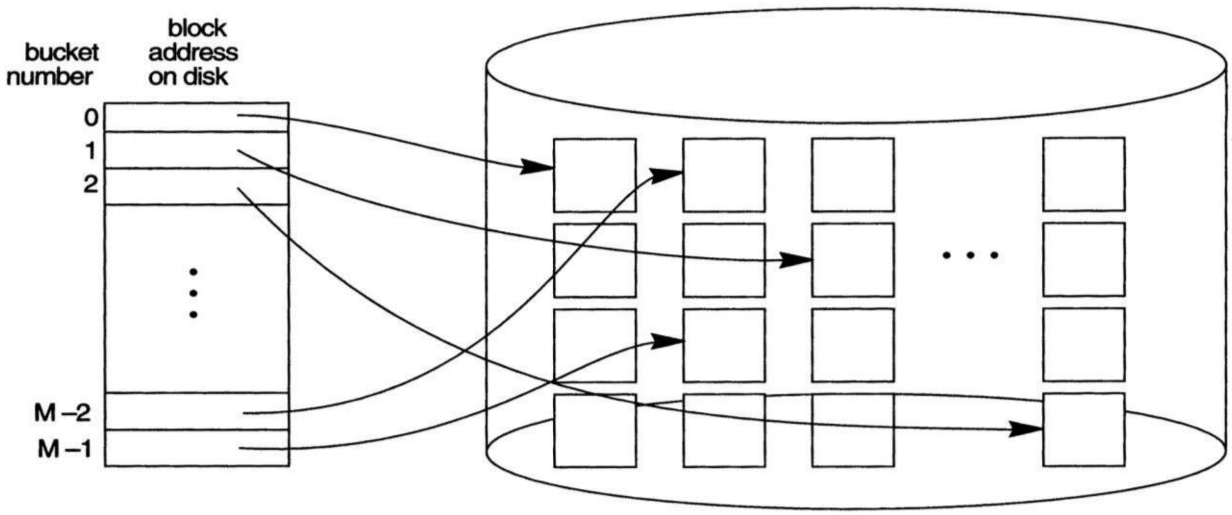
**Open addressing.** Ponce a position specified by the hash address is found to be occupied, the program checks the subsequent positions in order until an unused (empty) position is found. Algorithm (b) may be used for this purpose.

**Chaining.** For this method, various overflow locations are kept, usually by extending the array with a number of overflow positions. Additionally, a pointer field is added to each record location. A collision is resolved by placing the new record in an unused overflow location and setting the pointer of the occupied hash address location to the address of that overflow location. A linked list of overflow records for each hash address is thus maintained, as shown in Figure (b).

**Multiple hashing.** The program applies a second hash function if the first results in a collision. If another collision results, the program uses open addressing or applies a third hash function and then uses open addressing if necessary.

**External Hashing for Disk Files**

Hashing for disk files is called **external hashing**. the target address space is in external hashing is made of **buckets**, A bucket is either one disk block or a cluster of contiguous blocks. The hashing function maps the indexing field's value into a relative bucket number. A table maintained in the file header converts the bucket number into the corresponding disk block address as shown in below Figure.




The hashing scheme is called **static hashing** if a fixed number of buckets is allocated.

A major drawback of static hashing is that the number of buckets must be chosen large enough that can handle large files. That is, it is difficult to expand or shrink the file dynamically.

**Dynamic Hashing**

Two hashing techniques are

Extendible hashing and Liner hashing.

**Extendible hashing**

Basic Idea:

- Maintain a directory of bucket addresses instead of just hashing to buckets directly. (indirection)
- The directory can grow, but its size is always a power of 2.
- At any time, the directory consists of d levels, and a directory of depth d has 2d bucket pointers.
  - However, not every directory entry (bucket pointer) has to point to a unique bucket. More than one directory entry can point to the same one.
  - Each bucket has a local depth d’ that indicates how many of the d bits of

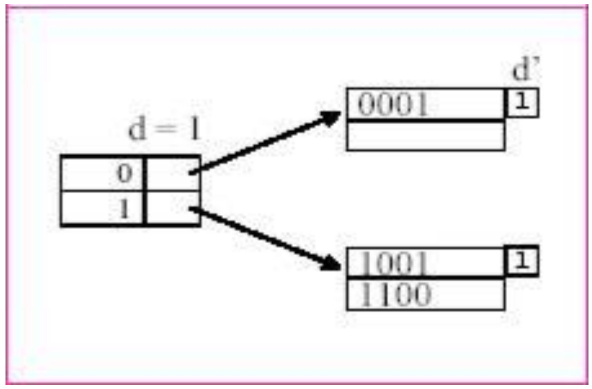
the hash value are actually used to indicate membership in the bucket.

- ➢ The depth d of the directory is based on the # of bits we use from each hash value.
  - The hash function produces an output integer which can be treated as a sequence of k bits. We use the first d bits in the hash value produced to look up an entry in the directory. The directory entry points us to the block that contains records whose keys hash to that value.

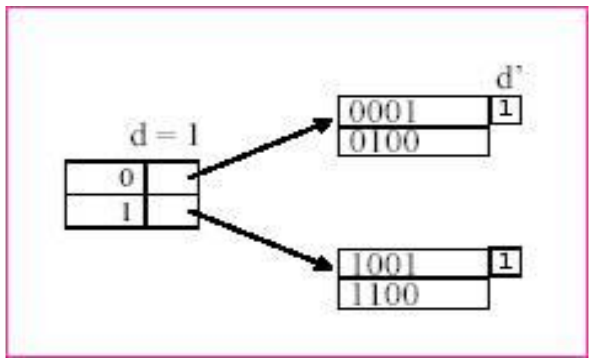
Example:

-Assume each hashed key is a sequence of four binary digits.

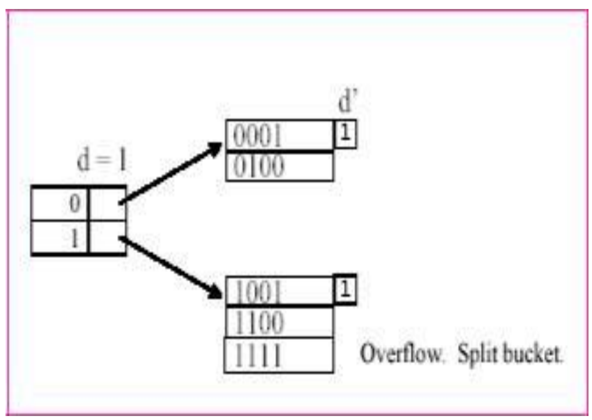
-Store values 0001, 1001, 1100.



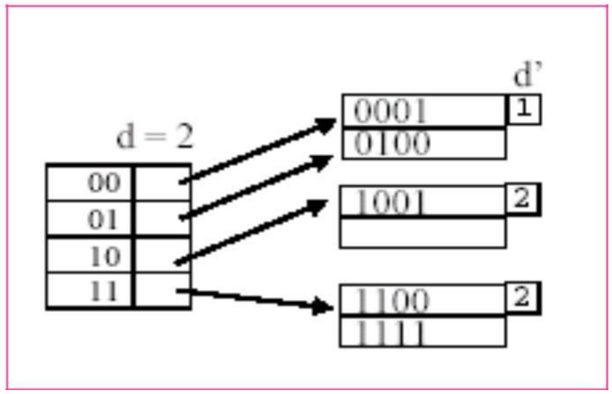

Insert 0100



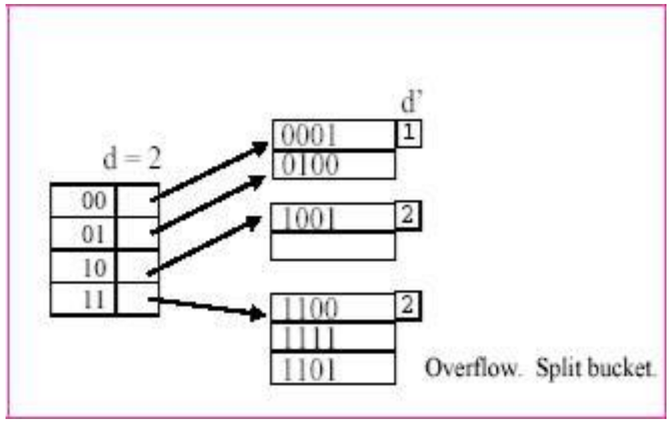

Insert 1111



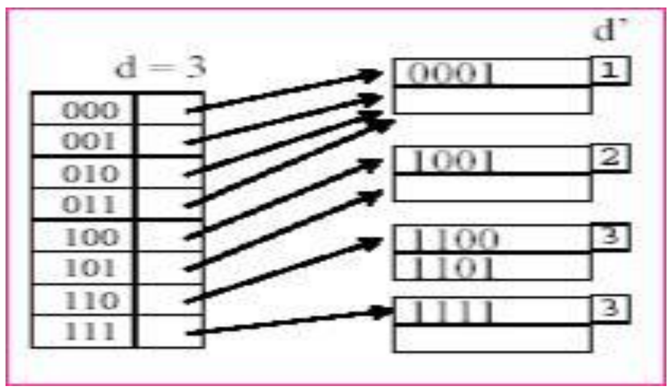
Directory grows one level.



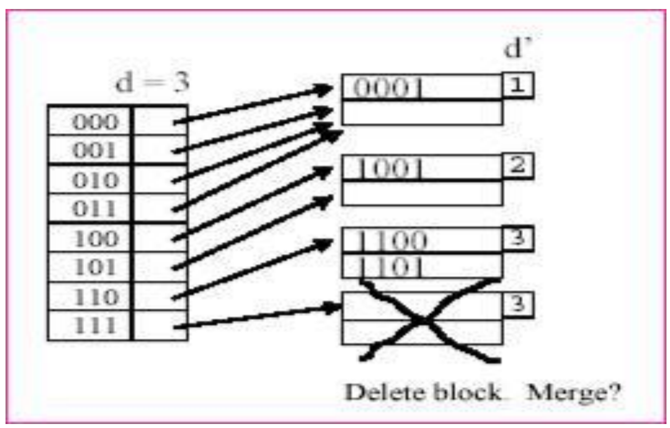

Insert 1101



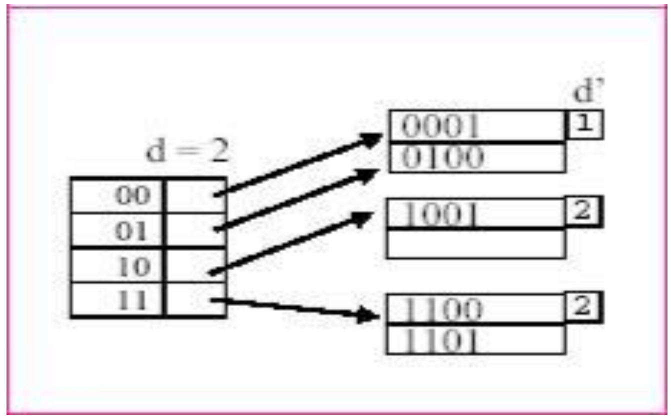

Directory grows one level.




Delete 1111



Merge Blocks and Shrink Directory




**Linear hashing**

- Linear hashing allows a hash file to expand and shrink dynamically without the need of a directory.
  - Thus, directory issues like extendible hashing are not present.
- A linear hash table starts with 2d buckets where d is the # of bits used from the hash value to determine bucket membership.
  - The size of the table will grow gradually, but not double in size.

1) Every time there is a bucket overflow.
2) When the load factor of the hash table reaches a given
point. We will examine the second growth method.

Since overflows may not always trigger growth, note that each bucket may use overflow blocks.

## Parallelizing Disk Access Using RAID Technology

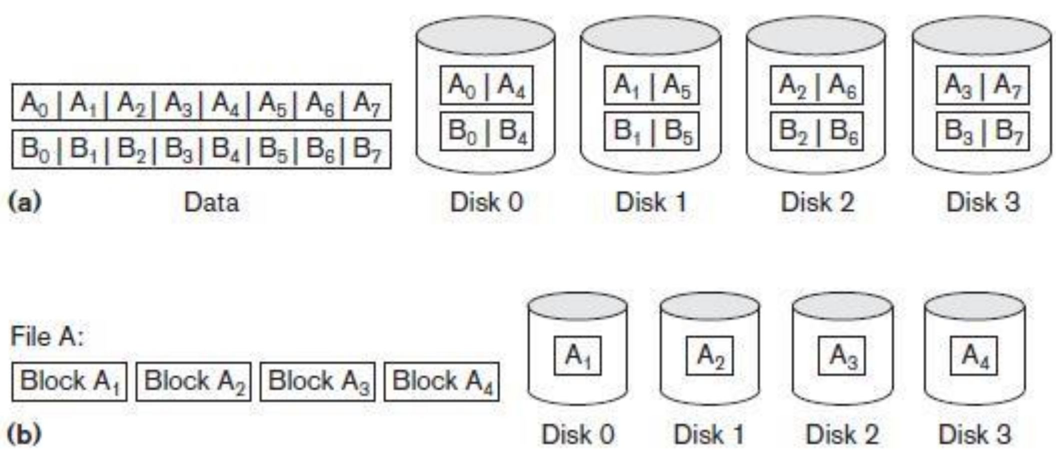
**RAID** (originally **redundant array of inexpensive disks**; now commonly **redundant array of independent disks**) is a data storage that combines multiple disk drive components into a logical unit for the purposes of data redundancy or performance improvement.

Disk striping is the process of dividing a body of data into blocks and spreading the data blocks across several partitions on several hard disks.

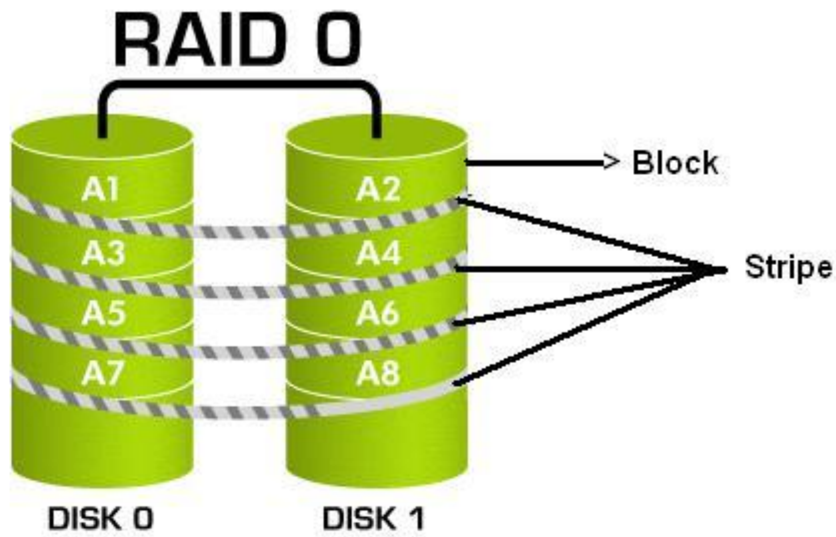
Striping can be done at the byte level, or in blocks. **Byte-level striping** means that the file is broken into "byte-sized pieces". The first byte of the file is sent to the first drive, then the second to the second drive, and so on. Sometimes byte-level striping is done as a sector of 512 bytes.

**Block-level striping** means that each file is split into blocks of a certain size and those are distributed to the various drives. The size of the blocks used is also called the stripe size (or block size, or several other names), and can be selected from a variety of choices when the array is set up.

**Figure**
Striping of data across multiple disks. (a) Bit-level striping across four disks. (b) Block-level striping across four disks.



**RAID 0**




RAID 0 performs what is called —Block Striping‖ across multiple drives. The data is fragmented, or broken up, into blocks and striped among the drives. This level increases the data transfer rate and data storage since the controller can access the hard disk drives simultaneously. However, this level has no redundancy. If single drive fails, the entire array becomes inaccessible. The more drivers in the array the higher the data transfer but higher risk of a failure. RAID level 0 requires 2 drives to implement.

**Advantages**

- I/O performance is greatly improved by spreading the I/O load across many channels and drives
- Best performance is achieved when data is striped across multiple controllers with only one drive per controller
- No parity calculation overhead is involved
- Very simple design
- Easy to implement

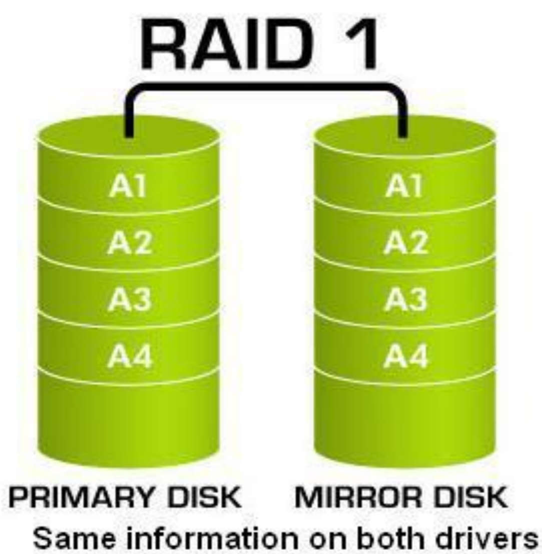
### Disadvantages

- Not a "True" RAID because it is NOT fault-tolerant
- The failure of just one drive will result in all data in an array being
- lost Should never be used in mission critical environments

### Recommended Applications

- Video Production and Editing
- Image Editing
- Pre-Press Applications
- Any application requiring high bandwidth

### RAID 1




**RAID 1** is used to create a complete mirrored set of disks. Usually employing two disks, RAID 1 writes data as a normal hard drive would but makes an exact copy of the primary drive with the second drive in the array. This mirrored copy is constantly updated as data on the primary drive changes. By keeping this mirrored backup, the array decreases the chance of failure from 5% over three years to 0.25%. Should a drive fail, the failed drive should be replaced as soon as possible

to keep the mirror updated. RAID Level 1 requires a minimum of 2 drives to implement.

**Advantages**

- Twice the Read transaction rate of single disks, same Write transaction rate as single disks
- 100% redundancy of data means no rebuild is necessary in case of a disk failure, just a copy to the replacement disk
- Transfer rate per block is equal to that of a single disk
- Under certain circumstances, RAID 1 can sustain multiple simultaneous drive failures
- Simplest RAID storage subsystem design

**Disadvantages**

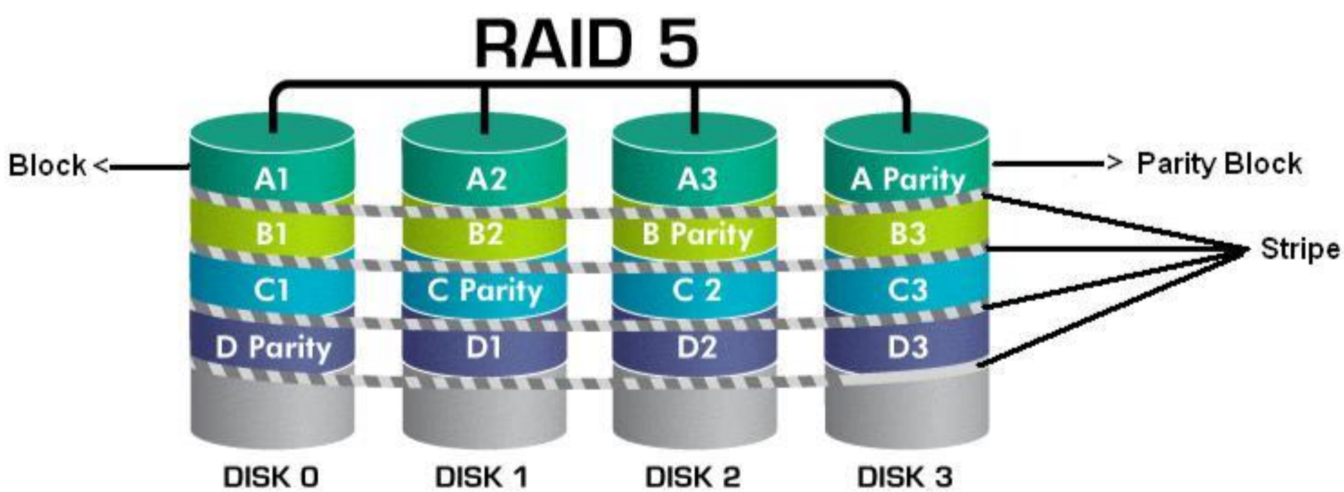
- Highest disk overhead of all RAID types (100%) - inefficient Typically
- the RAID function is done by system software, loading the CPU/Server and possibly degrading throughput at high activity levels. Hardware implementation is strongly recommended
- May not support hot swap of failed disk when implemented in "software"

**Recommended Applications**

- Accounting
- Payroll
- Financial
- Any application requiring very high availability

**RAID 5**

**RAID 5** is the most stable of the more advanced RAID levels and offers redundancy, speed, and the ability to rebuild a failed drive. RAID 5 uses the same block level striping as other RAID levels but adds another level of data protection by creating a ―parity block‖. These blocks are stored alongside the other blocks in the array in a staggered pattern and are used to check that the data has been written correctly in the drive.




when an error or failure occur, the parity block will be used to locate the information stored on the other member disks and parity blocks in the array to rebuild the data correctly. This can be done on-the-fly without interruptions to applications or other programs running on the computer. The computer will notify the user of the failed drive, but will continue to operate normally. This state of operation is known as Interim Data Recovery Mode. Performance may suffer while the disk is being rebuilt, but operation should continue. The failed drive should be replaced as soon as possible. RAID Level 5 requires a minimum of 3 drives to implement.

**Advantages**

- Highest Read data transaction rate
- Medium Write data transaction rate
- Low ratio of Error Correction Code (Parity) disks to data disks means high efficiency
- Good aggregate transfer rate

**Disadvantages**

- Disk failure has a medium impact on
- throughput Most complex controller design
- Difficult to rebuild in the event of a disk failure (as compared to RAID 1)
- Individual block data transfer rate same as single disk

**Recommended Applications**

- File and Application servers
- Database servers
- Web, E-mail, and News servers
- Intranet servers
- Most versatile RAID level