DISTRIBUTED DATABASES (R22CSE3146)

III B.Tech I Semester (Information Technology)

UNIT – V

Distributed Object Database Management Systems:
Fundamental object concepts and models, object distributed design, architectural issues, object management, distributed object storage, object query processing.
Object Oriented Data Model:
Inheritance, object identity, persistent programming languages, persistence of objects, comparison OODBMS and ORDBMS.

Distributed Object Database Management System

Fundamental Object Concepts and Object Models

- An object DBMS is a system that uses an "object" as the fundamental modeling, in which information is represented in the form of objects.

Object

- All object DBMSs are built around the fundamental concept of an *object*.
- An object represents a real entity in the system that is being modeled.
- It is represented as a tuple (OID, state, interface
- In which OID is the objectidentifier, the corresponding state is some representation of the current state of the object, and the interface defines the behavior of the object.
- Object identifier is an invariant property of an object which permanently distinguishes it logically and physically from all other objects, regardless of its state.
- The *state* of an object is commonly defined as either an atomic value or a constructed value (e.g., tuple or set).

A *value* is defined as follows:

1. An element of D is a value, called an *atomic value*.
2. $[a_1 : v_1, \ldots, a_n : v_n]$, in which a_i is an element of A and v_i is either a value oran element of I, is called a *tuple value*. [ ] is known as the tuple constructor.
3. $\{v_1, \ldots, v_n\}$, in which v_i is either a value or an element of I, is called a *setvalue*. { } is known as the set constructor.

- These models consider object identifiers as values (similar to pointers in programming languages).
- Set and tuple are data constructors that we consider essential for database applications. Other constructors, such as list or array, could also be added to increase the modeling power.

Example *:* Consider the following objects:

(i_1, 231)
(i_2, S70)
(i_3, {i_6,i_{11}})
(i_4, {1,3,5})
(i_5, [LF: i_7, RF: i_8, LR: i_9, RR: i_10])

- Objects i_1 and i_2 are atomic objects and i_3 and i_4 are constructed objects. i_3 is the OID of an object whose state consists of a set.
- The same is true of i_4. The difference between the two is that the state of i_4 consists of a set of values, while that of i_3 consists of a set of OIDs. Thus, object i_3 references other objects.
- By considering object identifiers (e.g., i_6) as values in the object model, arbitrarily complex objects may be constructed.
- Object i_5 has a tuple valued state consisting of four attributes (or instance variables), the values of each being another object.

Example : Consider the following objects:

(i_1, Volvo)
(i_2, [name: John, mycar: i_1])
(i_3, [name: Mary, mycar: i_1])

- Here, John and Mary share the object denoted by i_1 (they both own Volvo cars). Changing the value of object i_1 from "Volvo" to "Chevrolet" is automatically seen by bothobjects i_2 and i_3.

Types and Classes

- The term "class" refers to the specific object model construct and the term "type" to refer to a domain of objects (e.g., integer, string).
- A class is a template for a group of objects, thus defining a common type for these objects that conform to the template.
- We don't make a distinction between primitive system objects (i.e., values), structural (tuple or set) objects, and user-defined objects.
- A class describes the type of data by providing a domain of data with the same structure, as well as methods applicable to elements of that domain.
- The abstraction capability of classes, commonly referred to as *encapsulation*, hides the

implementation details of the methods, which can be written in a general-purpose programming language.

- Some (possibly proper) subset of its class structure and methods make up the publicly visible interface of objects that belong to that class.

Example : In this example, demonstrates the power of object models.

- We will model a car that consists of various parts (engine, bumpers, tires) and will store other information such as make, model, serial number, etc.
- The type definition of Car can be as follows using this abstract syntax:

```
type Car
  attributes
        engine : Engine
        bumpers : {Bumper}
        tires : [lf: Tire, rf: Tire, lr: Tire, rr: Tire]
        make : Manufacturer
        model : String
        year : Date
        serial_no : String
        capacity : Integer
  methods
        age: Real
        replaceTire(place, tire)
```

- The class definition specifies that Carhas eight attributes and two method. Fourof the attributes (model, year, serial no, capacity) are value-based, while the others(engine, bumpers, tires and make) are object-based (i.e., have other objects as theirvalues).
- Attribute bumpers is set valued (i.e., uses the set constructor), and attribute tires is tuple-valued where the left front (lf), right front (rf), left rear (lr) and right rear (rr) tires are individually identified.
- Incidentally, we follow a notation where the attributes are lower case and types are capitalized. Thus, engine is an attribute and Engineis a type in the system.

Composition (Aggregation)

- Composition is one of the most powerful features of object models.
- It allows sharingof objects, commonly referred to as *referential sharing*, since objects "refer" to each other by their OIDs as values of object-based attributes.

Example : Assume that c_1 is one instance of Cartype. If the following is true:

(i_2, [name: John, mycar: c_1])
(i_3, [name: Mary, mycar: c_1])

- Then this indicates that John and Mary own the same car.
- The composite object relationship between types can be represented by a *composition (aggregation) graph* (or *composition (aggregation) hierarchy* in the case of complex objects).
- There is an edge from instance variable I of type T_1 to type T_2 if the domain of I is T_2.

Subclassing and Inheritance

- Object systems provide extensibility by allowing user-defined classes to be defined and managed by the system.
- This is accomplished in two ways: by the definition of classes using type constructors or by the definition of classes based on existing classes through the process of *subclassing*.
- Subclassing is based on the *specialization* relationship among classes (or types that they define).
- A class A is a *specialization* of another class B if its interface is a superset of B's interface. Thus, a specialized class is more defined (or more specified) than the class from which it is specialized.
- A class may be a specialization of a number of classes; it is explicitly specified as a *subclass* of a subset of them.
- Some object models require that a class is specified as a subclass of only one class, in which case the model supports *single subclassing*; others allow *multiple subclassing*, where a class may be specified as a subclass of more than one class.
- Subclassing and specialization indicate an **is-a** relationship between classes (types).
- In the above example, A **is-a** B, resulting in *substitutability*: an instance of a subclass (A) can be substituted in place of an instance of any of its *superclasses* (B) in any expression.

Example : Consider the Cartype we defined earlier.

- A car can be modeledas a special type of Vehicle. Thus, it is possible to define Car as a subtype of Vehiclewhose other subtypes may be Motorcycle, Truck, and Bus.
- In thiscase, Vehiclewould define the common properties of all of these:

type Vehicle as Objectattributes

engine : Engine
make : Manufacturer
model : String
year : Date
serial_no : String
methods age: Real

- Vehicleis defined as a subclass of Objectthat we assume is the root of theclass lattice with common methods such as Put or Store.
- Vehicle is defined with five attributes and one method that takes the date of manufacture and today's date (both of which are of system-defined type Date) and returns a real value.
- Obviously, Vehicle is a generalization of Car that we defined in Example

- Car can nowbe defined as follows:

type Car as Vehicle

attributes

bumpers : {Bumper}
tires : [LF: Tire, RF: Tire, LR: Tire, RR: Tire]
capacity : Integer

- Even though Car is defined with only two attributes, its interface is the same as the definition given in Example.
- This is because Car **is-a** Vehicle, and therefore inherits the attributes and methods of Vehicle.

Features of Object DBMS :

1. Support for complex objects
2. Support for object identity
3. Data persistence must be provided.
4. OODMS must support concurrent users.
5. OODMS must capable of recovery from hardware and software failure.
6. Support for dynamic binding.

Drawbacks of Object Database:

1. Object DB's are not as popular as RDBMS. It is difficult to find object DB developers.
2. Not many programming languages support object databases.
3. Object DB do not have a standard query language.
4. Object DB are difficult to learn for Non-programmers.

Object Distribution Design

- The two important aspects of distribution design are **fragmentation and allocation.**
- Distribution design in the object world brings new complexities due to the encapsulation of methods together with object state.
- An object is defined by its state and its methods. We can fragment the state, the method definitions, and the method implementation.
- Furthermore, the objects in a class extent can also be fragmented and placed at different sites. Each of these raise interesting problems and issues.

Fragmentation

- There are three fundamental types of fragmentation: horizontal, vertical, and hybrid.
- In addition to these two fundamental cases, Horizontal Partitioning , Vertical Partitioning , and Path Partitioning have been defined.

Horizontal Class Partitioning

- Partitioning of a class arising from the fragmentation of its subclasses. This occurs when a more specialized class is fragmented, so the results of this fragmentation should be reflected in the more general case.
- The fragmentation of a complex attribute may affect the fragmentation of its containing class.
- Fragmentation of a class based on a method invocation sequence from one class to another may need to be reflected in the design.
- Class C for partitioning, we create classes $C_1,...,C_n$, each of whichtakes the instances of C that satisfy the particular partitioning predicate.
- If these predicates are mutually exclusive, then classes $C_1,...,C_n$ are disjoint.

Example : Consider the definition of the Engine class

Class Engine as Object

attributes

no_cylinder : Integer

capacity : Real

horsepower: Integer

In this simple definition of Engine, all the attributes are simple.

Consider the partitioning predicates

p_1: horsepower ≤ 150
p_2: horsepower > 150

- In this case, Engine can be partitioned into two classes, Engine1 and Engine2, which inherit all of their properties from the Engine class, which is redefined as an abstract class (i.e,. a class that cannot have any objects in its shallow extent).
- The objects of Engineclass are distributed to the Engine1and Engine2classes based on the value of their horsepower attribute value.

Vertical Class Partitioning

- Vertical fragmentation is considerably more complicated. Given a class C, fragmenting it vertically into $C_1,...,C_m$ produces a number of classes, each of which contains some of the attributes and some of the methods.
- Thus, each of the fragments is less defined than the original class. Issues that must be addressed include the subtyping between the original class' superclasses and subclasses and the fragment classes, the relationship of the fragment classes among themselves, and the location of the

methods.

- If all the methods are simple, then methods can be partitioned easily.However, when this is not the case, the location of these methods becomes a problem.
- Adaptations of the affinity-based relational vertical fragmentation approaches have been developed for object databases.
- However, the break-up of encapsulation during vertical fragmentation has created significant doubts as to the suitability of vertical fragmentation in object DBMSs.

Path Partitioning

- The composition graph presents a representation for composite objects. For many applications, it is necessary to access the complete composite object.
- Path partitioning is a concept describing the clustering of all the objects forming a composite object into a partition.
- A path partition consists of grouping the objects of all the domain classes that correspond to all the instance variables in the subtree rooted at the composite object.
- A path partition can be represented as a hierarchy of nodes forming a structural index.
- Each node of the index points to the objects of the domain class of the component object.
- The index thus contains the references to all the component.

Class Partitioning Algorithms

- The main issue in class partitioning is to improve the performance of user queries and applications by reducing the irrelevant data access.
- Thus, class partitioning is a logical database design technique that restructures the object database schema based on the application semantics.
- It should be noted that class partitioning is more complicated than relation fragmentation, and is also NP-complete.
- The algorithms for class partitioning are based on affinity-based and cost-driven approaches.

Affinity-based Approach

- Affinity among attributes is used to vertically fragment relations. Similarly, affinity among instance variables and methods, and affinity among multiple methods can be used for horizontal and vertical class partitioning.
- Horizontal and vertical class partitioning algorithms have been developed that are based on classifying instance variables and methods as being either simple or complex.
- A complex instance variable is an object-based instance variable and is part of the class composition hierarchy.
- An alternative is a method- induced partitioning scheme, which applies the method semantics and appropriately generates fragments that match the methods data requirements.

Cost-Driven Approach

- Though the affinity-based approach provides "intuitively" appealing partitioning schemes, it has been shown that these partitioning schemes do not always result in the greatest reduction of disk accesses required to process a set of applications.
- Therefore, a cost model for the number of disk accesses for processing both queries and methods on an object oriented database has been developed.
- Further, an heuristic "hill-climbing" approach that uses both the affinity approach (for initial solution) and the cost-driven approach (for further refinement) has been proposed.
- This work also develops structural join index hierarchies for complex object retrieval, and studies its effectiveness against pointer traversal and other approaches, such as join index hierarchies, multi-index and access support relations (see next section).

Allocation

- The data allocation problem for object databases involves allocation of both methods and classes.
- The method allocation problem is tightly coupled to the class allocation problem because of encapsulation.
- Therefore, allocation of classes will imply allocation of methods to their corresponding home classes. But since applications on object-oriented databases invoke methods, the allocation of methods affects the performance of applications. However, allocation of methods that need to access multiple classes at different sites is a problem that has been not yet been tackled.
- Four alternatives can be identified :

1. **Local behavior – local object.** This is the most straightforward case and is included to form the baseline case. The behavior, the object to which it isto be applied, and the arguments are all co-located. Therefore, no special mechanism is needed to handle this case.
2. **Local behavior – remote object.** This is one of the cases in which the behavior and the object to which it is applied are located at different sites. There are two ways of dealing with this case. One alternative is to move the remoteobject to the site where the behavior is located. The second is to ship the behavior implementation to the site where the object is located. This is possible if the receiver site can run the code.
3. **Remote behavior – local object.** This case is the reverse of case (2).
4. **Remote function – remote argument.** This case is the reverse of case (1).

Replication

- Replication adds a new dimension to the design problem. Individual objects, classes of objects, or collections of objects (or all) can be units of replication.

- Undoubtedly, the decision is at least partially object-model dependent.
- Whether ornot type specifications are located at each site can also be considered a replication problem.

Architectural Issues

- The preferred architectural model for object DBMSs has been client/server.
- The design issues related to these systems are somewhat more complicated due to the characteristics of object models. The major concerns are listed below.

1. Since data and procedures are encapsulated as objects, the unit of communication between the clients and the server is an issue. The unit can be a page, an object, or a group of objects.
2. Closely related to the above issue is the design decision regarding the functions provided by the clients and the server. This is especially important since objects are not simply passive data, and it is necessary to consider the sites where object methods are executed.
3. In relational client/server systems, clients simply pass queries to the server, which executes them and returns the result tables to the client. This is referred to as *function shipping*. In object client/server DBMSs, this may not be the best approach, as the navigation of composite/complex object structures by the application program may dictate that data be moved to the clients (called *data shipping systems*). Since data are shared by many clients, the management ofclient cache buffers for data consistency becomes a serious concern. Client cache buffer management is closely related to concurrency control, since data that are cached to clients may be shared by multiple clients, and this has to be controlled. Most commercial object DBMSs use locking for concurrencycontrol, so a fundamental architectural issue is the placement of locks, and whether or not the locks are cached to clients.
4. Since objects may be composite or complex, there may be possibilities for prefetching component objects when an object is requested. Relational client/server systems do not usually prefetch data from the server, but this may be a valid alternative in the case of object DBMSs.

- These considerations require revisiting some of the issues common to all DBMSs, along with several new ones. We will consider these issues in three sections: those directly related to architectural design (architectural alternatives, buffer management, and cache consistency).

Alternative Client/Server Architectures

- Two main types of client/server architectures have been proposed: object servers and page servers.
- The distinction is partly based on the granularity of data that are shipped between the clients and the servers, and partly on the functionality provided to the clients and servers.

Object Server :

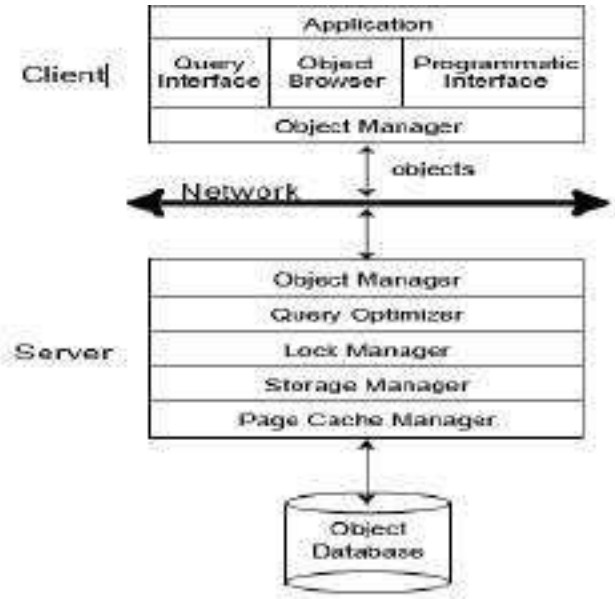

Fig. Object Server Architecture

- The first alternative is that clients request "objects" from the server, which retrieves them from the database and returns them to the requesting client. These systems are called *object servers.*
- In object servers, the server undertakes most of the DBMS services, with the client providing basically an execution environment for the applications, as well as some level of object management functionality.
- The object management layer is duplicated at both the client and the server in order to allow both to perform object functions. Object manager serves a number of functions.
- Object manager also deals with the implementation of the object identifier (logical, physical, or virtual) and the deletion of objects (either explicit deletion or garbage collection).
- The object managers at the client and the server implement an object cache (in additionto the page cache at the server). Objects are cached at the client to improve system performance by localizing accesses.
- The optimization of user queries and the synchronization of user transactions are all performed in the server, with the client receiving the resulting objects.

Page Server :

- An alternative organization is a *page server* client/server architecture, in which the unit of transfer between the servers and the clients is a physical unit of data, such as a page or segment, rather than an object.
- Page server architectures split the object processing services between the clients and the servers.
- Page servers simplify the DBMS code, since both the server and the client maintain page caches, and the representation of an object is the same all the way from the disk to the user interface.
- Thus, updates to the objects occur only in client caches and these updates are reflected on disk when the page is flushed from the client to the server.

- The server performs a limited set of functions and can therefore serve a large number of clients.
- Page servers can also exploit operating systems and even hardware functionality to deal with certain problems, such as pointer swizzling.

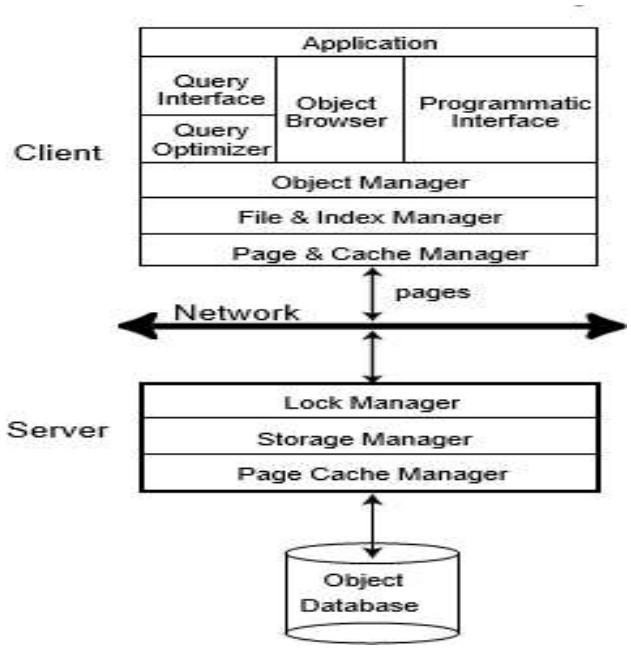

Fig. Page Server Architecture

Client Buffer Management

- The clients can manage either a page buffer, an object buffer, or a dual (i.e., page/object) buffer.
- If clients have a page buffer, then entire pages are read or written from the server every time a page fault occurs or a page is flushed. Object buffers can read/write individual objects and allow the applications object-by-object access.
- A page buffer does not encounter this problem, but if the data clustering on the disk does not match the application data access pattern, then the pages contain a great deal of unaccessed objects that use up valuable client buffer space.
- In these situations, buffer utilization of a page buffer will be lower than the buffer utilization of an object buffer.

Server Buffer Management

- The server buffer management issues in object client/server systems are not much different than their relational counterparts, since the servers usually manage a page buffer.
- We nevertheless discuss the issues here briefly in the interest of completeness.
- The pages from the page buffer are, in turn, sent to the clients to satisfy their data requests.
- A grouped object-server constructs its object groups by copying the necessary objects from the relevant server buffer pages, and sends the object groupto the clients.
- In addition to the page level buffer, the servers can also maintain a modified object buffer (MOB).

- A MOB stores objects that have been updated and returned by the clients. These updated objects have to be installed onto their corresponding data pages, which may require installation reads as described earlier.
- Finally, the modified page has to be written back to the disk.
- A MOB allows the server to amortize its disk I/O costs by batching the installation read and installation write operations.
- In a client/server system, since the clients typically absorb most of the data requests (i.e., the system has a high cache hit rate), the server buffer usually behavesmore as a staging buffer than a cache.

Cache Consistency

- Cache consistency is a problem in any data shipping system that moves data to the clients. However, the problems arise in unique ways in object DBMSs.
- The DBMS cache consistency algorithms can be classified as avoidance-based or detection-based.
- ***Avoidance-based algorithms*** prevent the access to stale cache data[5] by ensuring that clients cannot update an object if it is being read by other clients. So they ensure that stale data never exists in client caches.
- ***Detection-based algorithms*** allow access of stale cache data, because clients canupdate objects that are being read by other clients.
- However, the detection-based algorithms perform a validation step at commit time to satisfy data consistency requirements.
- We discuss each of the alternatives in the design space and comment on their performance characteristics.
- **Avoidance-based synchronous:** Callback-Read Locking (CBL) is the most common synchronous avoidance-based cache consistency algorithm [Franklinand Carey, 1994]. In this algorithm, the clients retain read locks across transactions, but they relinquish write locks at the end of the transaction. The clients send lock requests to the server and they block until the server responds. If the client requests a write lock on a page that is cached at other clients, the server issues callback messages requesting that the remote clients relinquish their read locks on the page. Callback-Read ensures a low abort rate and generally outperforms deferred avoidance-based, synchronous detection-based, andasynchronous detection-based algorithms.
- **Avoidance-based asynchronous:** Asynchronous avoidance-based cache conistency algorithms (AACC) do not have the message blocking overhead present in synchronous algorithms. Clients send lock escalation messages to the server and continue application processing. Normally, optimistic approaches such as this face high abort rates, which is reduced in avoidance-based algorithms by immediate server actions to invalidate stale cache objects at remote clients as soon as the system becomes aware of the update. Thus, asynchronous algorithms experience lower deadlock abort rates than deferred avoidance-based algorithms, which are discussed next.
- **Avoidance-based deferred:** Optimistic Two-Phase Locking (O2PL) family ofcache consistency

are deferred avoidance-based algorithms [Franklin and Carey, 1994]. In these algorithms, the clients batch their lock escalation requests andsend them to the server at commit time. The server blocks the updating client if other clients are reading the updated objects. As the data contention level increases, O2PL algorithms are susceptible to higher deadlock abort rates thanCBL algorithms.

- **Detection-based synchronous:** Caching Two-Phase Locking (C2PL) is a synchronous detection-based cache consistency algorithm [Carey et al., 1991]. In this algorithm, clients contact the server whenever they access a page in their cache to ensure that the page is not stale or being written to by other clients. C2PL's performance is generally worse than CBL and O2PL algorithms, sinceit does not cache read locks across transactions.
- **Detection-based asynchronous:** No-Wait Locking (NWL) with Notificationis an asynchronous detection-based algorithm. In this algorithm, the clients send lock escalation requests to the server, but optimistically assume that their requests will be successful. After a client transaction commits, the server propagates the updated pages to all the other clients that have also cached the affected pages. It has been shown that CBL outperformsthe NWL algorithm.
- **Detection-based deferred:** Adaptive Optimistic Concurrency Control (AOCC) is a deferred detection-based algorithm. It has been shown that AOCC can outperform callback locking algorithms even while encountering a higher abort rate if the client transaction state (data and logs) completely fits into the clientcache, and all application processing is strictly performed at the clients (purelydata-shipping architecture). Since AOCC uses deferred messages, its messaging overhead is less than CBL. Furthermore, in a purely data-shipping client/server environment, the impact of an aborting client on the performance of other clients is quite minimal. These factors contribute to AOCC's superior performance.

Object Management

- Object management includes tasks such as object identifier management, pointer swizzling, object migration, deletion of objects, method execution, and some storage management tasks at the server.

Object Identifier Management

- Object identifiers (OIDs) are system-generated and used to uniquely identify every object (transient or persistent, system-created or user-created) in the system.
- The implementation of persistent object identifier has two common solutions, based on either physical or logical identifiers, with their respective advantages and shortcomings.
- The physical identifier (POID) approach equates the OID with the physical address of the corresponding object. The address can be a disk page addressand an offset from the base address in the page.
- The advantage is that the object can be obtained directly from the OID.
- The drawback is that all parent objects and indexes must be updated whenever an object is moved to a different page.
- The logical identifier (LOID) approach consists of allocating a system-wide unique OID (i.e., a surrogate) per object.

- LOIDs can be generated either by using a system-wide unique counter (called pure LOID) or by concatenating a server identifier witha counter at each server (called pseudo-LOID).

Pointer Swizzling

- In object systems, one can navigate from one object to another using *path expressions* that involve attributes with object-based values. For example, if object c is of type Car, then c.engine.manufacturer.name is a path expression. These are basically pointers.
- Usually on disk, object identifiers are used to represent these pointers.
- However, in memory, it is desirable to use in-memory pointers for navigating from one object to another.
- The process of converting a disk version of the pointer to an in-memory version of a pointer is known as "pointer-swizzling".
- Hardware-based and software-based schemes are two types of pointer-swizzling mechanisms.
- In hardware-based schemes, the operating system's page-fault mechanism is used; when a page is brought into memory, all the pointers in it are swizzled.

Object Migration

- One aspect of distributed systems is that objects move, from time to time, between sites.
- This raises a number of issues. First is the unit of migration.
- It is possible to move the object's state without moving its methods.
- Even if individual objects are units of migration, their relocation may move them away from their type specifications and one has to decide whether types are duplicated at every site where instances reside or the types are accessed remotely when behaviors or methods are applied to objects.
- Three alternatives can be considered for the migration of classes (types):

    1. The source code is moved and recompiled at the destination,
    2. The compiled version of a class is migrated just like any other object, or
    3. The source code of the class definition is moved, but not its compiled opera-tions, for which a lazy migration strategy is used.

- Another issue is that the movements of the objects must be tracked so that they can be found in their new locations.
- A common way of tracking objects is to leave *surrogates*, or *proxy objects*.
- These are place-holder objects left at the previous site of the object, pointing to its new location. Accesses to the proxy objects are directed transparently by the system to the objects themselves at the new sites.
- The migration of objects can be accomplished based on their current state.
- Objects can be in one of four states:

1. **Ready :** Ready objects are not currently invoked, or have not received a mes-sage, but are ready to be invoked to receive a message.
2. **Active :** Active objects are currently involved in an activity in response to aninvocation or a message.
3. **Waiting :** Waiting objects have invoked (or have sent a message to) anotherobject and are waiting for a response.
4. **Suspended :** Suspended objects are temporarily unavailable for invocation.

- Objects in active or waiting state are not allowed to migrate, since the activity they are currently involved in would be broken.
- The migration involves two steps:
  1. shipping the object from the source to the destination, and
  2. creating a proxy at the source, replacing the original object.
- Two related issues must also be addressed here.
  1. One relates to the maintenance of the system directory.
  2. The second issue is that, in a highly dynamic environment where objects move frequently, the surrogate or proxy chains may become quite long.

Distributed Object Storage

- Among the many issues related to object storage, two are particularly relevant in a distributed system:
  1. Object Clustering
  2. Distributed Garbage Collection.
- For clustering data on disk such that the I/O cost of retrieving them is reduced.
- Garbage collection is a problem that arises in object databases due to reference-based sharing. Indeed, in many object DBMSs, the only way to delete an object is to delete all references to it.

Object Clustering

- An object model is essentially conceptual, and should provide high physical data independence to increase programmer productivity. The mapping of this conceptual model to a physical storage is a classical database problem.
- Object clustering refers to the grouping of objects in physical containers (i.e., disk extents) according to common properties, such as the same value of an attribute or sub-objects of the same object.
- Thus, fast access to clustered objects can be obtained.
- Object clustering is difficult for two reasons.

- First, it is not orthogonal to object identifier implementation (i.e, LOID vs. POID). LOIDs incur more overhead (an indirection table), but enable vertical partitioning of classes. POIDs yield more efficient direct object access, but require each object to contain all inherited attributes.
- Second, the clustering of complex objects along the composition relationship is more involved because of object sharing (objects with multiple parents).
- There are three basic storage models for object clustering:

1. The ***decomposition storage model*** (DSM) partitions each object class into binary relations (OID, attribute) and therefore relies on logical OID. The advantage of DSM is simplicity.
2. The ***normalized storage model*** (NSM) stores each class as a separate relation. It can be used with logical or physical OID. However, only logical OID allows the vertical partitioning of objects along the inheritance relationship.
3. The ***direct storage model*** (DSM) enables multi-class clustering of complex objects based on the composition relationship. This model generalizes the techniques of hierarchical and network databases, and works best with physical OID.

Distributed Garbage Collection

- The generality of distributed object-based systems calls for automatic distributed garbage collection.
- The basic garbage collection algorithms can be categorized as ***Reference counting*** **or Tracing-based.**

Reference counting

- In a reference counting system, each object has an associated count of the references to it.
- Each time a program creates an additional reference that points to an object, the object's count is incremented.
- When an existing reference to an object is destroyed, the corresponding count is decremented.The memory occupied by an object can be reclaimed when the object's count drops to zero and become unreachable (at which time, the object is garbage).
- In reference counting, a problem can arise where two objects only refer to each other but not referred to by anyone else; in this case, the two objects are basically unreachable (except from each other) but their reference count has not dropped to zero.

Tracing-based

- *Tracing-based* collectors are divided into *mark and sweep* and *copy-based* algorithms.
- *Mark and sweep* collectors are two-phase algorithms.

- The first phase, calledthe “mark” phase, starts from the root and marks every reachable object (for example, by setting a bit associated to each object). This mark is also called a “color”, and the collector is said to color the objects it reaches.
- The mark bit can be embedded in the objects themselves or in *color maps* that record, for every memory page, the colorsof the objects stored in that page.
- Once all live objects are marked, the memory is examined and unmarked objects are reclaimed. This is the “sweep” phase.

Object Query Processing

- There has been significant amount of work on object query processing and optimization, these have primarily focused on centralized systems.
- Almost all object query processors and optimizers that have been proposed to date use techniques developed for relational systems.
- It is possible to claim that distributed object query processing and optimization techniques require the extension of centralized object.
- Although most object query processing proposals are based on their relational counterparts, there are a number of issues that make query processing and optimization more difficult in object DBMSs
- Relational query languages operate on very simple type systems consisting ofa single type.
- Relational query optimization depends on knowledge of the physical storageof data (access paths) that is readily available to the query optimizer.
- Objects can (and usually do) have complex structures whereby the state of anobject references another object.

Object Oriented Data Model

- In Object Oriented Data Model, data and their relationships are contained in a single structure which is referred as object in this data model.
- In this, real world problems are represented as objects with different attributes.
- All objects have multiple relationships between them.
- It is combination of Object Oriented programming and Relational Database Model
- It is clear from the following figure :

Object Oriented Data Model = Combination of Object Oriented Programming + Relational database model

Components of Object Oriented Data Model :

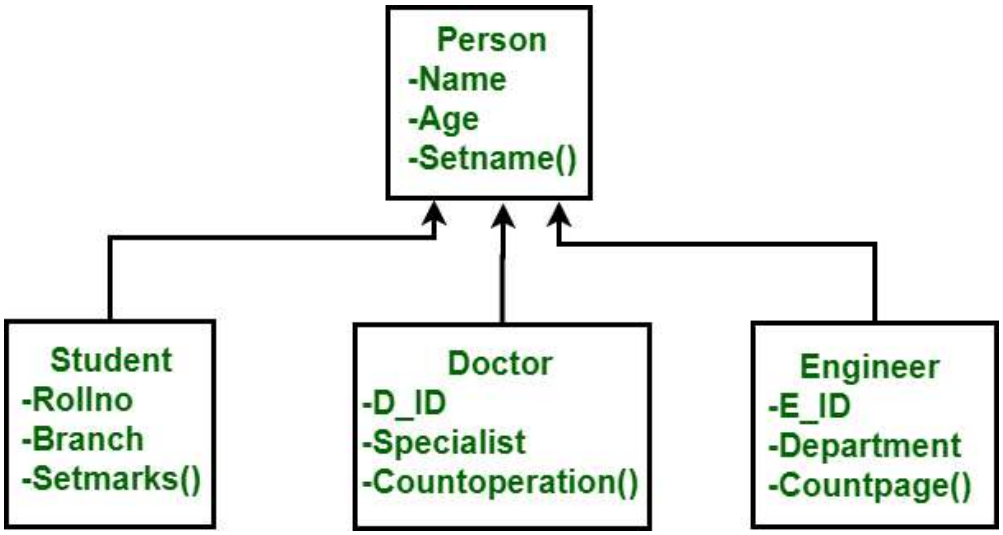

Fig : Basic Object Oriented Data Model / Hierarchy of classes

Objects

- An object is an abstraction of a real world entity or we can say it is an instance of class.
- Objects encapsulates data and code into a single unit which provide data abstraction by hiding the implementation details from the user.
- For example: Instances of student, doctor, engineer in above figure.

Attribute

- An attribute describes the properties of object.
- For example: Object is STUDENT and its attribute are Roll no, Branch, Setmarks() in the Student class.

Methods

- Method represents the behavior of an object. Basically, it represents the real-world action.
- For example: Finding a STUDENT marks in above figure as Setmarks().

Class

- A class is a collection of similar objects with shared structure i.e. attributes and behavior i.e. methods.
- An object is an instance of class.

- For example: Person, Student, Doctor, Engineer in above figure.

```
class student
{
   char Name[20];
   int roll_no;
   --
   --
   public:
   void search();
   void update();
}
```

In this example, students refers to class and S1, S2 are the objects of class which can be created in main function.

Inheritance

- By using inheritance, new class can inherit the attributes and methods of the old class i.e. base class.
- For example: as classes Student, Doctor and Engineer are inherited from the base class Person.

Characteristics of an Object Oriented Data Model

- It keeps up a direct relation between real world and database objects as if objects do not loose their integrity and identity.
- OODBs provide system generated object identifier for each object so that an object can easily be identified and operated upon.
- OODBs are extensible, which identifies new data types and the operations to be performed on them.
- Provides encapsulation, feature which, the data representation and the methods implementation are hidden from external entities.
- Also provides inheritance properties in which an object inherits the properties of other objects.

Advantages of Object Oriented Data Model :

- Codes can be reused due to inheritance.
- Easily understandable.
- Cost of maintenance can reduced due to reusability of attributes and functions because of inheritance.

Disadvantages of Object Oriented Data Model:

- It is not properly developed so not accepted by users easily.

Inheritance

- Inheritance creates a hierarchical relationship between related classes while making parts of code reusable.
- Defining new types inherits all the existing class fields and methods plus further extends them.
- The existing class is the parent class, while the child class extends the parent.
- Inheritance can be of various type
    - Substitution inheritance
    - Inclusion inheritance
    - Constraint inheritance
    - Specialization inheritance

Substitution inheritance

- ✓ Is based on behavior and not on values
- ✓ t' can be substituted anywhere for t.

Inclusion inheritance

- ✓ Denotes classification
- ✓ It is based on structure and not operations

Constraint inheritance

- ✓ Is a sub-case of inclusion
- ✓ Satisfies certain constraint (retired person sub type of person with constraint)

Specialization inheritance

- ✓ With more specific information (student object is a person with some extra attributes as course no)

Object Identity

- The basic idea behind object identity is that objects should be identified by system-generated object identifiers (OIDs) rather than by the values of their properties.
- This is in sharp contrast to the relational model, where for instance tuples are identified by the value of their primary key.
- Object identity provides a unique identifier for each object in an object database.
- This identifier distinguishes one object from another, even if they share the same attributes. Object identity ensures data integrity and consistency.
- You can reference objects using their unique identifiers.

- This capability simplifies data retrieval and manipulation. Object identity plays a crucial role in managing complex data structures.
- This has two implications:
  - ✓ One is object sharing and
  - ✓ The other one is object updates.

Object sharing:

- It is an identity-based model, two objects can share a component.
- Thus, the pictorial representation of a complex object is a graph, while it is limited to be a tree in a system without object identity.
- Consider the following example: a Person has a name, an age and a set of children. Assume Peter and Susan both have a 15-year-old child named John. In real life, two situations may arise: Susan and Peter are parent of the same child or there are two children involved. In a system without identity, Peter is represented by:

(peter, 40, {(john, 15, {})})

and Susan is represented by:

(susan, 41, {(john, 15, {})}).

- Thus, there is no way of expressing whether Peter and Susan are the parents of the same child. In an identity-based model, these two structures can share the common part (john, 15, {}) or not, thus capturing either situations.

Object updates:

- The update item (or object) operation updates an existing item or adds a new item to the table if it does not already exist.
- Assume that Peter and Susan are indeed parents of a child named John. In this case, all updates to Susan's son will be applied to the object John and, consequently, also to Peter's son.
- In a value-based system, both sub-objects must be updated separately. Object identity is also a powerful data manipulation primitive that can be the basis of set, tuple and recursive complex object manipulation.
- Supporting object identity implies offering operations such as object assignment, object copy (both deep and shallow copy) and tests for object identity and object equality (both deep and shallow equality).

Persistent Programming Language

- Programming languages that natively and seamlessly allow objects to continue existing after the program has been closed down are called **persistent programming languages**.
- A persistent programming language is a programming language extended with constructs to

handle persistent data.

- In a persistent programming language:
  - The query language is fully integrated with the host language and both share the same type system.
  - Any format changes required between the host language and the database are carried out transparently.
- Using Embedded SQL, a programmer is responsible for writing explicit code to fetch data into memory or store data back to the database.
- In a persistent programming language, a programmer can manipulate persistent data without having to write such code explicitly.
- The drawbacks of persistent programming languages include:
  - ➢ While they are powerful, it is easy to make programming errors that damage the database.
  - ➢ It is harder to do automatic high-level optimization.
  - ➢ They do not support declarative querying well.

Persistence of Object

- **Persistence** denotes a process or an object that continues to exist even after its parent process or object ceases, or the system that runs it is turned off.
- Types of persistent : There are two types of persistence:
  1. Object Persistence
  2. Process persistence.
- **Object persistence** refers to an object that is not deleted until a need emerges to remove it from the memory. Some database models provide mechanisms for storing persistent data in the form of objects.
- **Process persistence**, processes are not killed or shut down by other processes and exist until the user kills them. For example, all of the core processes of a computer system are persistent in enabling the proper functioning of the system. Persistent processes are stored in non-volatile memory. They do not need special databases like persistent objects.

Comparison of OODBMS and ORDBMS

OODBMS	ORDBMS
It stands for Object Oriented Database Management System.	It stands for Object Relational Database Management System.

In the Object Oriented Database, the data is stored in the form of objects.	In Relational Database, data is stored in the form of tables, which contains rows and columns.
OODBMS	**ORDBMS**
In OODBMS, relationships are represented by references via the object identifier (OID).	In ORDBMS, connections between two relationships are represented by foreign key attributes.
It handles larger and complex data than RDBMS.	It handles comparatively simpler data.
In OODBMS, the data management language is typically incorporated into a programming languages such as C++, C#.	In relational database systems there are data manipulation language such as SQL.
Stores data entries are described as objects.	Stores data in entries is described as tables.
Object-oriented databases, like Object Oriented Programming, represent data in the form of objects and classes.	An object-relational database is one that is based on both the relational and object-oriented database models.
OODBMSs support ODL/OQL.	ORDBMS adds object-oriented functionalities to SQL.
Every object-oriented system has a different set of constraints that it can accommodate.	Keys, entity integrity, and referential integrity are constraints of an object-oriented database.
The efficiency of query processing is low.	Processing of queries is quite effective.