UNIT I

CONCEPTUAL DATA MODELING

Database environment –Database system development lifecycle –Requirements collection – Database design --Entity-Relationship model –Enhanced-ER model –UML class diagrams.

Introduction of Data Base

What is Data?

Data is a collection of a distinct small unit of information. It can be used in a variety of forms like text, numbers, media, bytes, etc. it can be stored in pieces of paper or electronic memory, etc.

Word 'Data' is originated from the word 'datum' that means 'single piece of information.' It is plural of the word datum.

In computing, Data is information that can be translated into a form for efficient movement and processing. Data is interchangeable.

What is Database?

A database is an organized collection of data, so that it can be easily accessed and managed.

You can organize data into tables, rows, columns, and index it to make it easier to find relevant information.

Database handlers create a database in such a way that only one set of software program provides access of data to all the users.

The main purpose of the database is to operate a large amount of information by storing, retrieving, and managing data.

There are many dynamic websites on the World Wide Web nowadays which are handled through databases. For example, a model that checks the availability of rooms in a hotel. It is an example of a dynamic website that uses a database.

There are many databases available like MySQL, Sybase, Oracle, MongoDB, Informix, PostgreSQL, SQL Server, etc.

Modern databases are managed by the database management system (DBMS).

1.1 Database Environment

Advanced databases play a vital role in functioning of modern organization. Every day we all are dealing with some database applications like shopping at a super market, withdrawing cash using ATM, ordering a book online etc. Database technology not only used to improve our daily

operations but also the quality of decisions that affects our daily life. Database environment encapsulates one or more databases. This encapsulation allows us to perform the group operations between multiple databases in a single transaction. Database environment is surrounded by different types of relations, tables, file models, time sharing models to store, access and update the data in the organizational database.

Database Characteristics

Database contain a flood of data about many aspects of our life. Such as consumer preferences, telecommunications usage, credit history etc. Databases contain these sorts of facts as well as non-conventional facts such as photographs, fingerprints etc.

Some important properties of database are discussed below.

(a) Persistent: - It means data reside on stable storage such as a magnetic disk. It depends upon the intended usage.

(b) Shared: – It means the database have multiple uses and uses.

(c) Interrelated: – It means that data stored as separate units can be connected to provide a whole picture.

(d) Entity: – It is a cluster of data usually about a single subject that can be accessed together.

File Processing System:

File processing system is a type of system which is used to store and organize the information in the database. It is the process of creating, storing and accessing the content of files. It can be used to save a new file in the processing system. Through this file processing system, we can create new files and save the existing files.

Advantages File Processing System

(a) It is simple to use.

(b) It is less expensive.

(c) It fits the needs of many small businesses and home users.

(d) Popular file management systems are packaged along with the operating systems.

(e) Good for database solutions.

Disadvantages of File Processing System: There are certain disadvantages in file processing system.

(a) It typically does not support multiuser access.

(b) It is limited to smaller databases.

(c) It has limited functionality.

(d) It has redundancy.

(e) It has integrity issues.

Main Services of Relational Database Management System: The main services of Relational Database Management System are listed below.

(a) Data storage, data retrieval and data update.

(b) User accessible catalog.

(c) It supports the transaction.

(d) It controls the concurrency in the database.

(e) It is recoverable.

(f) Only authorized users can view access the data.

(g) It supports the data communication.

(h) It supports the data independence.

Need of Database: There are certain advantages of using the database.

(a) It has the chance of maximum reduction of redundancy.

(b) In database, data can be used for more than one application.

(c) It supports standardization.

(d) It has the better security option.

(e) The integrity of data is independent of several applications.

(f) Here the data can be used directly.

Time Sharing Model

Time sharing model is the main frame and terminal of database. Its environment consists of the mainframe and minicomputer. The components of time sharing model are operating systems. Here, DBMS and applications are running on a single computer. It has the interaction through terminals. The user interface in time sharing model is generated by mainframe or minicomputer. It has one or more cooperative processors for processing system.

File Server Model

File server model consists of server and personal computers. It has the environment consists of file server and work stations. The components of a file server model are DBMS and application programs which are separated from the database. It has the interaction through workstations. The screen layout is generated by workstations. In file server model, all processing is carried out by one or more intelligent workstations.

Data Integrity

Normally data integrity means consistency, validity and accuracy of data in a database. Normally, there are four types of data integrity. These are listed below.

(a) Table level integrity

(b) Field level integrity

(c) Relationship level integrity

(d) Business rules

Physical data independence.

Physical data independence is the ability to modify re internal schema without having the change to the conceptual or external schema. In physical data independence, the conceptual schema covers the users from changes in the physical storage of data. Physical data independence indicates that the physical storage structures or devices used for storing the data could be changed without necessitating a change in the conceptual view of any of the external views.

Components of a run time database manager.

The run time database manager is normally called as the database control system. It has the following components.

(a) Authorization control checks the authorization of users.

(b) Command processor processes the queries passed by authorization control module.

(c) Integrity checker checks the integrity constraints so that only valid data can be entered into a database.

(d) Query optimizer determines an optimal strategy for the query execution.

(e) Transaction manager ensures that the transaction properties are maintained by the systems.

(f) The scheduler provides an environment in which multiple users can work on.

(g) Data manager is responsible for the actual handling of data in the database.

1.2 Database Development Life Cycle

The Database Life Cycle (DBLC) contains six phases, as shown in the following Figure: database initial study, database design, implementation and loading, testing and evaluation, operation, and maintenance and evolution.

The Database Life Cycle (DBLC)

1.Database Initial Study:

In the Database initial study, the designer must examine the current system's operation within the company and determine how and why the current system fails. The overall purpose of the database initial study is to:

• Analyze the company situation.

• Define problems and constraints.

• Define objectives.

• Define scope and boundaries.

a. Analyze the Company Situation: The company situation describes the general conditions in which a company operates, its organizational structure, and its mission. To analyze the company situation, the database designer must discover what the company's operational components are, how they function, and how they interact.

b. Define Problems and Constraints: The designer has both formal and informal sources of information. The process of defining problems might initially appear to be unstructured. Company end users are often unable to describe precisely the larger scope of company operations or to identify the real problems encountered during company operations.

c. Define Objectives: A proposed database system must be designed to help solve at least the major problems identified during the problem discovery process. In any case, the database designer must begin to address the following questions:

• What is the proposed system's initial objective?

• Will the system interface with other existing or future systems in the company?

• Will the system share the data with other systems or users?

d. Define Scope and Boundaries: The designer must recognize the existence of two sets of limits: scope and boundaries. The system's scope defines the extent of the design according to operational requirements. Will the database design encompass the entire organization, one or more departments within the organization, or one or more functions of a single department? Knowing the scope helps in defining the required data structures, the type and number of entities, the physical size of the database, and so on.

The proposed system is also subject to limits known as boundaries, which are external to the system. Boundaries are also imposed by existing hardware and software.

2. Database Design:

The second phase focuses on the design of the database model that will support company operations and objectives. This is arguably the most critical DBLC phase: making sure that the final product meets user and system requirements. As you examine the procedures required to complete the design phase in the DBLC, remember these points:

• The process of database design is loosely related to the analysis and design of a larger system. The data component is only one element of a larger information system.

• The systems analysts or systems programmers are in charge of designing the other system components. Their activities create the procedures that will help transform the data within the database into useful information.

3. Implementation and Loading:

The output of the database design phase is a series of instructions detailing the creation of tables, attributes, domains, views, indexes, security constraints, and storage and performance guidelines. In this phase, you actually implement all these design specifications.

a. Install the DBMS: This step is required only when a new dedicated instance of the DBMS is necessary for the system. The DBMS may be installed on a new server or it may be installed on existing servers. One current trend is called virtualization. Virtualization is a technique that creates logical representations of computing resources that are independent of the underlying physical computing resources.

b. Create the Database(s): In most modern relational DBMSs a new database implementation requires the creation of special storage-related constructs to house the end-user tables. The constructs usually include the storage group (or file groups), the table spaces, and the tables.

c. Load or Convert the Data: After the database has been created, the data must be loaded into the database tables. Typically, the data will have to be migrated from the prior version of the system. Often, data to be included in the system must be aggregated from multiple sources. Data may have to be imported from other relational databases, non-relational databases, flat files, legacy systems, or even manual paper-and-pencil systems

4. Testing and Evaluation:

In the design phase, decisions were made to ensure integrity, security, performance, and recoverability of the database. During implementation and loading, these plans were put into place. In testing and evaluation, the DBA tests and fine-tunes the database to ensure that it performs as expected. This phase occurs in conjunction with applications programming.

a. Test the Database: During this step, the DBA tests the database to ensure that it maintains the integrity and security of the data. Data integrity is enforced by the DBMS through the proper use of primary and foreign key rules. In database testing you must check Physical security allows, Password security, Access rights, Data encryption etc.

b. Fine-Tune the Database: Although database performance can be difficult to evaluate because there are no standards for database performance measures, it is typically one of the most important factors in database implementation. Different systems will place different performance requirements on the database. Many factors can impact the database's performance on various tasks. Environmental factors, such as the hardware and software environment in which the database exists, can have a significant impact on database performance.

c. Evaluate the Database and Its Application Programs: As the database and application programs are created and tested, the system must also be evaluated from a more holistic approach. Testing and evaluation of the individual components should culminate in a variety of broader system tests to ensure that all of the components interact properly to meet the needs of the users. To ensure that the data contained in the database are protected against loss, backup and recovery plans are tested.

5. Operation

Once the database has passed the evaluation stage, it is considered to be operational. At that point, the database, its management, its users, and its application programs constitute a complete information system. The beginning of the operational phase invariably starts the process of system evolution.

6. Maintenance and Evolution

The database administrator must be prepared to perform routine maintenance activities within the database. Some of the required periodic maintenance activities include:

• Preventive maintenance (backup).

• Corrective maintenance (recovery).

• Adaptive maintenance (enhancing performance, adding entities and attributes, and so on).

• Assignment of access permissions and their maintenance for new and old users.

1.3 Requirements Analysis.

Requirements Analysis is the stage in the design cycle when you find out everything you can about the data the client needs to store in the database and the conditions under which that data needs to be accessed.

Keep in mind, too, that a single pass through this stage rarely yields all the information the database designer needs. Be prepared to return to the tasks associated with Requirements Analysis several times during the course of designing a database.

Collecting Data

Collecting data is relatively easy, but turning raw information into something useful requires that you know how to extract precisely what you need. In this module, intermediate to experienced programmers interested in data analysis will learn techniques for working with data in a business environment.

You will learn how to look at data to discover what it contains, how to capture those ideas in conceptual models, and then feed your understanding back into the organization through business plans, metrics dashboards, and other applications. Along the way, you will experiment with concepts through hands-on exercises at various points in the moule.

1. Use graphics to describe data with one, two, or dozens of variables
2. Develop conceptual models using back-of-the-envelope calculations, as well as scaling and probability arguments
3. Mine data with computationally intensive methods such as simulation and clustering
4. Make your conclusions understandable through reports, dashboards, and other metrics programs
5. Understand financial calculations, including the time value of money
6. Use dimensionality reduction techniques or predictive analytics to conquer challenging data analysis situations
7. Become familiar with different open source programming environments for data analysis

Purpose of Requirements Analysis

The overall purpose of Requirements Analysis is to gather every bit of information needed to design a database that meets the informational needs of an organization. This is accomplished by performing a series of related tasks:

1. Examine the existing database(s)
2. Conduct user interviews
3. Create a data flow diagram (if needed)
4. Determine user views
5. Document all findings

DBLC Requirements Analysis

As mentioned earlier in this course, Requirements Analysis is the most important and most labor-intensive stage in the DBLC. It is critical for the designer to approach Requirements Analysis armed with a plan for each task in the process.

Experience is the great teacher when it comes to assessing informational needs, but there is no substitute for preparation, especially for new designers. Most database designers begin Requirements Analysis by examining the existing database(s) to establish a framework for the remaining tasks.

Analyzing how an organization stores data about its business objects, and scrutinizing its perception of how it uses stored data (for example, gaining familiarity with its business rules) provides that framework.

Interview both Producers and Users of Data

The database requirements are determined by interviewing both the producers and users of data and using the information to produce a formal requirements specification. That specification includes the data required for processing, the natural data relationships, and the software platform for the database implementation.

For example, products, customers, salespersons, and orders can be formulated in the mind of the end user during the interview process.

The information needed to build a data model is gathered during the requirements analysis. Although not formally considered part of the data modeling stage by some methodologies, in reality 1) the requirements analysis and 2) the ER diagramming part of the data model are done at the same time.

Requirements Analysis

The goals of the requirements analysis are:

1. To determine the data requirements of the database in terms of primitive objects
2. To classify and describe the information about these objects
3. To identify and classify the relationships among the objects
4. To determine the types of transactions that will be executed on the database and the interactions between the data and the transactions
5. To identify rules governing the integrity of the data

The modeler works with the end users of an organization to determine the data requirements of the database. Information needed for the requirements analysis can be gathered in several ways:

Review of existing documents:

Such documents include existing forms and reports, written guidelines, job descriptions, and personal narratives. Paper documentation is a good way to become familiar with the organization or activity you need to model.

Interviews with end Users

These can be a combination of individual or group meetings. Try to keep group sessions to under five or six people. If possible, try to have everyone with the same function in one meeting. Use a blackboard or overhead transparencies to record information gathered from the interviews.

Review of existing automated systems: If the organization already has an automated system, review the system design specifications and documentation. The requirements analysis is usually done at the same time as the data modeling. As information is collected, data objects are identified and classified as either entities, attributes, or relationship. They are then assigned names and defined using terms familiar to the end-users. The objects are then modeled and analyzed using an ER diagram. The diagram can be reviewed by the modeler and the end-users to determine its completeness and accuracy. If the model is not correct, it is modified, which sometimes requires additional information to be collected. The review and edit cycle continue until the model is certified as correct.

1.4 Database Design

What is Database Design?

Database design can be generally defined as a collection of tasks or processes that enhance the designing, development, implementation, and maintenance of enterprise data management system.

Designing a proper database reduces the maintenance cost thereby improving data consistency and the cost-effective measures are greatly influenced in terms of disk storage space. Therefore, there has to be a brilliant concept of designing a database. The designer should follow the constraints and decide how the elements correlate and what kind of data must be stored.

The main objectives behind database designing are to produce physical and logical design models of the proposed database system. To elaborate this, the logical model is primarily concentrated on the requirements of data and the considerations must be made in terms of monolithic considerations and hence the stored physical data must be stored independent of the physical conditions.

On the other hand, the physical database design model includes a translation of the logical design model of the database by keep control of physical media using hardware resources and software systems such as Database Management System (DBMS).

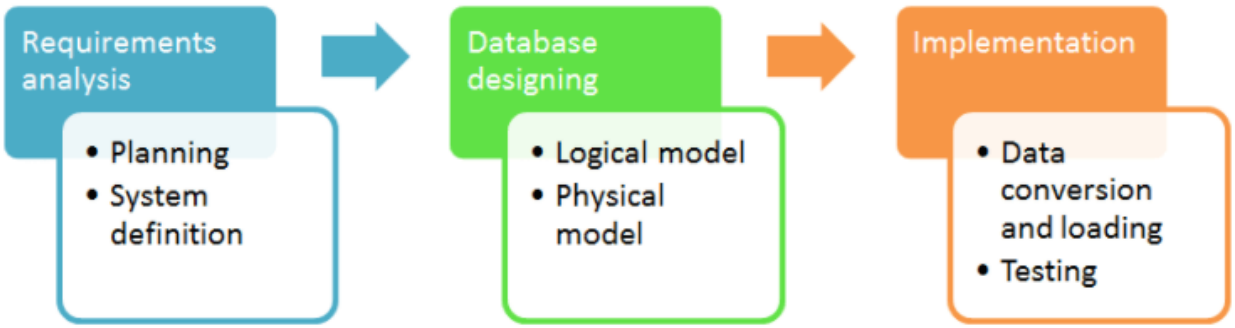
Why is Database Design important?

The important consideration that can be taken into account while emphasizing the importance of database design can be explained in terms of the following points given below.

1. Database designs provide the blueprints of how the data is going to be stored in a system. A proper design of a database highly affects the overall performance of any application.

2. The designing principles defined for a database give a clear idea of the behavior of any application and how the requests are processed.
3. Another instance to emphasize the database design is that a proper database design meets all the requirements of users.
4. Lastly, the processing time of an application is greatly reduced if the constraints of designing a highly efficient database are properly implemented.

Life Cycle



the life cycle of a database is not an important discussion that has to be taken forward in this article because we are focused on the database design. But, before jumping directly on the designing models constituting database design it is important to understand the overall workflow and life-cycle of the database.

Requirement Analysis

First of all, the planning has to be done on what are the basic requirements of the project under which the design of the database has to be taken forward. Thus, they can be defined as:-

Planning - This stage is concerned with planning the entire DDLC (Database Development Life Cycle). The strategic considerations are taken into account before proceeding.

System definition - This stage covers the boundaries and scopes of the proper database after planning.

Database Designing

The next step involves designing the database considering the user-based requirements and splitting them out into various models so that load or heavy dependencies on a single aspect are not imposed. Therefore, there has been some model-centric approach and that's where logical and physical models play a crucial role

Physical Model - The physical model is concerned with the practices and implementations of the logical model.

Logical Model - This stage is primarily concerned with developing a model based on the proposed requirements. The entire model is designed on paper without any implementation or adopting DBMS considerations.

Implementation

The last step covers the implementation methods and checking out the behavior that matches our requirements. It is ensured with continuous integration testing of the database with different data sets and conversion of data into machine understandable language. The manipulation of data is primarily focused on these steps where queries are made to run and check if the application is designed satisfactorily or not.

Data conversion and loading - This section is used to import and convert data from the old to the new system.

Testing - This stage is concerned with error identification in the newly implemented system. Testing is a crucial step because it checks the database directly and compares the requirement specifications.

Database Design Process

The process of designing a database carries various conceptual approaches that are needed to be kept in mind. An ideal and well-structured database design must be able to:

1. Save disk space by eliminating redundant data.
2. Maintains data integrity and accuracy.
3. Provides data access in useful ways.
4. Comparing Logical and Physical data models.

Logical

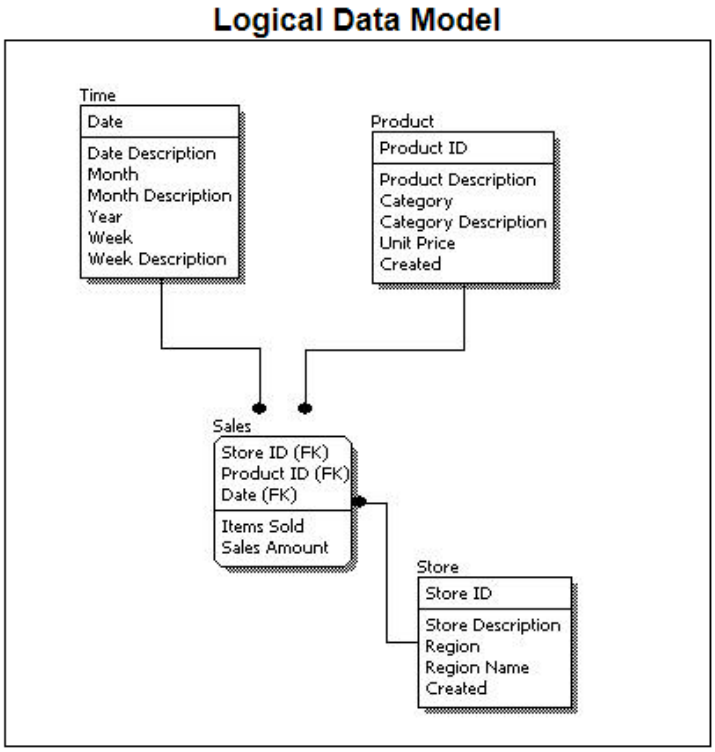
A logical data model generally describes the data in as many details as possible, without having to be concerned about the physical implementations in the database. Features of logical data model might include:

1. All the entities and relationships amongst them.
2. Each entity has well-specified attributes.
3. The primary key for each entity is specified.
4. Foreign keys which are used to identify a relationship between different entities are specified.
5. Normalization occurs at this level.

A logical model can be designed using the following approach:

1. Specify all the entities with primary keys.
2. Specify concurrent relationships between different entities.
3. Figure out each entity attributes
4. Resolve many-to-many relationships.
5. Carry out the process of normalization.

Also, one important factor after following the above approach is to critically examine the design based on requirement gathering. If the above steps are strictly followed, there are chances of creating a highly efficient database design that follows the native approach.



If we compare the logical data model as shown in the figure above with some sample data in the diagram, we can come up with facts that in a conceptual data model there are no presence of a primary key whereas a logical data model has primary keys for all of its attributes. Also, logical data model the cover relationship between different entities and carries room for foreign keys to establish relationships among them.

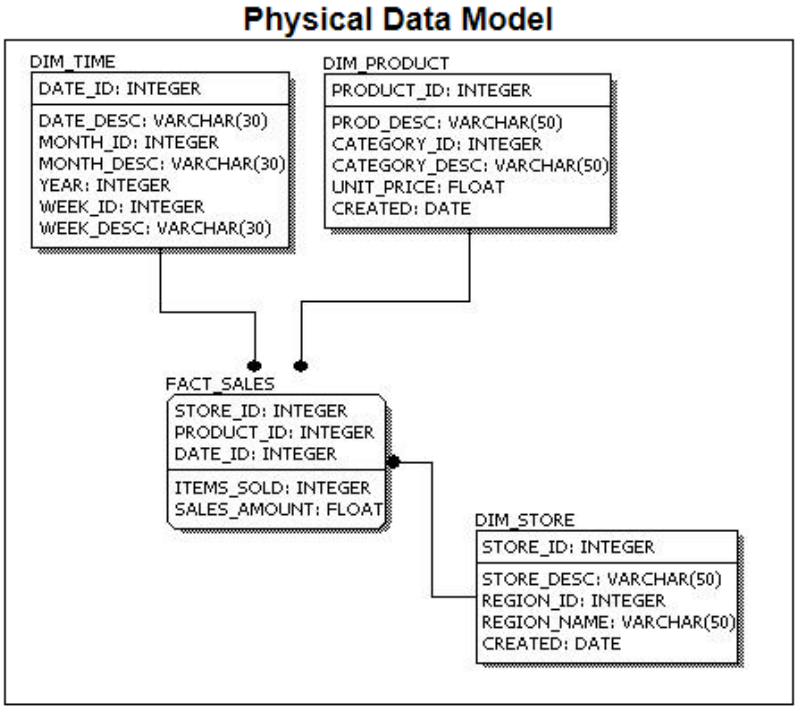
Physical

A Physical data mode generally represents how the approach or concept of designing the database. The main purpose of the physical data model is to show all the structures of the table including the column name, column data type, constraints, keys (primary and foreign), and the relationship among tables. The following are the features of a physical data model:

1. Specifies all the columns and tables.
2. Specifies foreign keys that usually define the relationship between tables.
3. Based on user requirements, de-normalization might occur.
4. Since the physical consideration is taken into account so there will straightforward reasons for difference than a logical model.
5. Physical models might be different for different RDBMS. For example, the data type column may be different in MySQL and SQL Server.

While designing a physical data model, the following points should be taken into consideration:

1. Convert the entities into tables.
2. Convert the defined relationships into foreign keys.
3. Convert the data attributes into columns.
4. Modify the data model constraints based on physical requirements.



Comparing this physical data model with the logical with the previous logical model, we might conclude the differences that in a physical database entity names are considered table names and attributes are considered column names. Also, the data type of each column is defined in the physical model depending on the actual database used.

Entity - An entity in the database can be defined as abstract data that we save in our database. For example, a customer, products.

Attributes - An attribute is a detailed form of data consisting of entities like length, name, price, etc.

Relationship - A relationship can be defined as the connection between two entities or figures. For example, a person can relate to multiple persons in a family.

Foreign key - It acts as a referral to the Primary Key of another table. A foreign key contains columns with values that exist only in the primary key column they refer to.

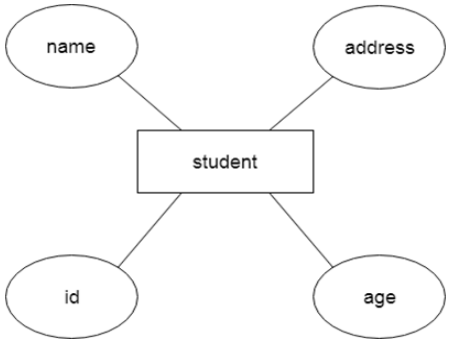
Primary key - A primary key is the pointer of records that is unique and not null and is used to uniquely identify attributes of a table.

Normalization - A flexible data model needs to follow certain rules. Applying these rules is called normalizing.

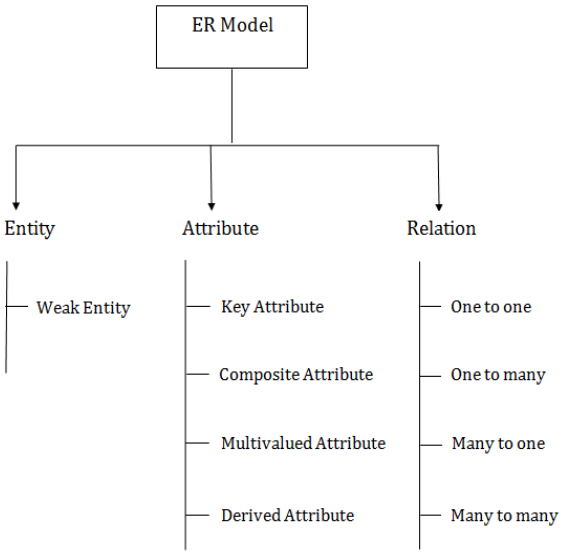
1.5 Entity-Relational model (ER model)

- ER model stands for an Entity-Relationship model. It is a high-level data model. This model is used to define the data elements and relationship for a specified system.
- It develops a conceptual design for the database. It also develops a very simple and easy to design view of data.
- In ER modeling, the database structure is portrayed as a diagram called an entity-relationship diagram.

For example, suppose we design a school database. In this database, the student will be an entity with attributes like address, name, id, age, etc. The address can be another entity with attributes like city, street name, pin code, etc. and there will be a relationship between them.



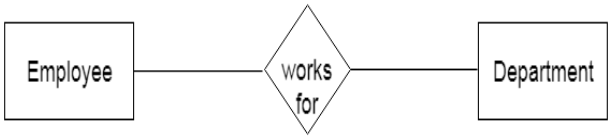
Component of ER Diagram



1. Entity:

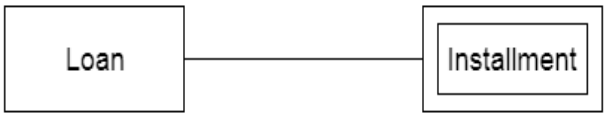
An entity may be any object, class, person or place. In the ER diagram, an entity can be represented as rectangles.

Consider an organization as an example- manager, product, employee, department etc. can be taken as an entity.



a. Weak Entity

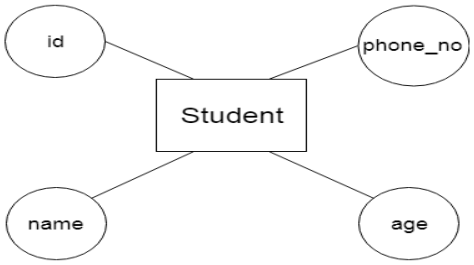
An entity that depends on another entity called a weak entity. The weak entity doesn't contain any key attribute of its own. The weak entity is represented by a double rectangle.



2. Attribute

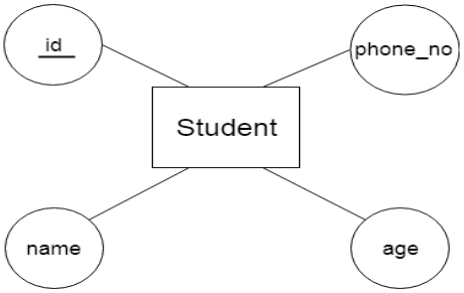
The attribute is used to describe the property of an entity. Eclipse is used to represent an attribute.

For example, id, age, contact number, name, etc. can be attributes of a student.



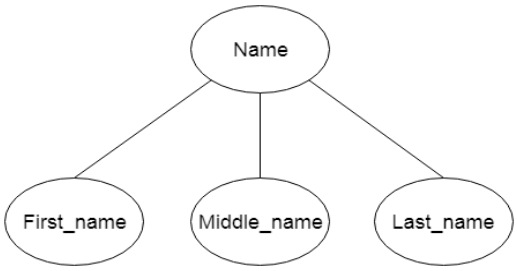
a. Key Attribute

The key attribute is used to represent the main characteristics of an entity. It represents a primary key. The key attribute is represented by an ellipse with the text underlined.



b. Composite Attribute

An attribute that composed of many other attributes is known as a composite attribute. The composite attribute is represented by an ellipse, and those ellipses are connected with an ellipse.



c. Multivalued Attribute

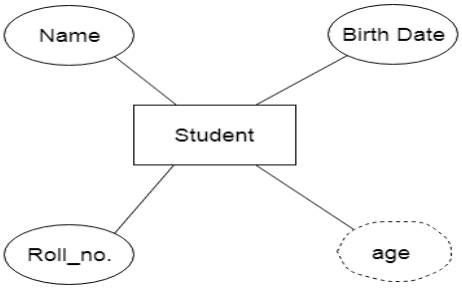
An attribute can have more than one value. These attributes are known as a multivalued attribute. The double oval is used to represent multivalued attribute.

For example, a student can have more than one phone number.

d. Derived Attribute

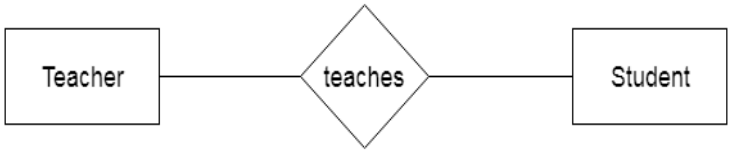
An attribute that can be derived from other attribute is known as a derived attribute. It can be represented by a dashed ellipse.

For example, a person's age changes over time and can be derived from another attribute like Date of birth



3. Relationship

A relationship is used to describe the relation between entities. Diamond or rhombus is used to represent the relationship.

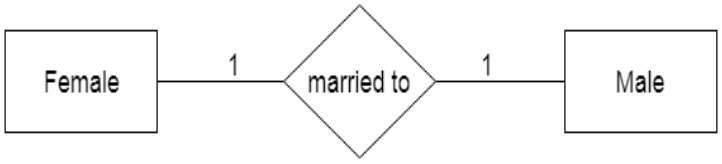


Types of relationship are as follows:

a. One-to-One Relationship

When only one instance of an entity is associated with the relationship, then it is known as one to one relationship.

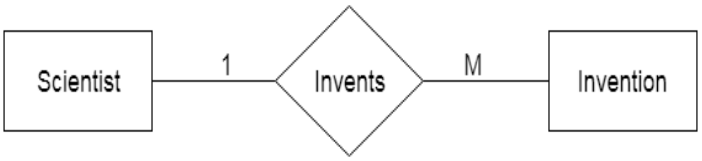
For example, a female can marry to one male, and a male can marry to one female.



b. One-to-many relationship

When only one instance of the entity on the left, and more than one instance of an entity on the right associates with the relationship then this is known as a one-to-many relationship.

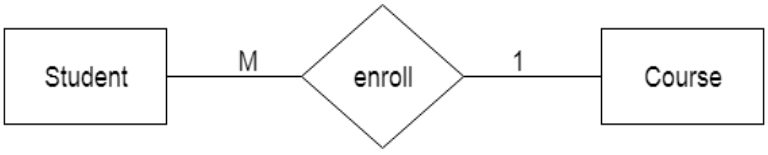
For example, Scientist can invent many inventions, but the invention is done by the only specific scientist.



c. Many-to-one relationship

When more than one instance of the entity on the left, and only one instance of an entity on the right associates with the relationship then it is known as a many-to-one relationship.

For example, Student enrolls for only one course, but a course can have many students.



d. Many-to-many relationship

When more than one instance of the entity on the left, and more than one instance of an entity on the right associates with the relationship then it is known as a many-to-many relationship.

For example, Employee can assign by many projects and project can have many employees.

1.6 Enhanced ER model

Extended ER is a high-level data model that incorporates the extensions to the original ER model. Enhanced ER models are high level models that represent the requirements and complexities of complex databases.

The extended Entity Relationship (ER) models are three types as given below −

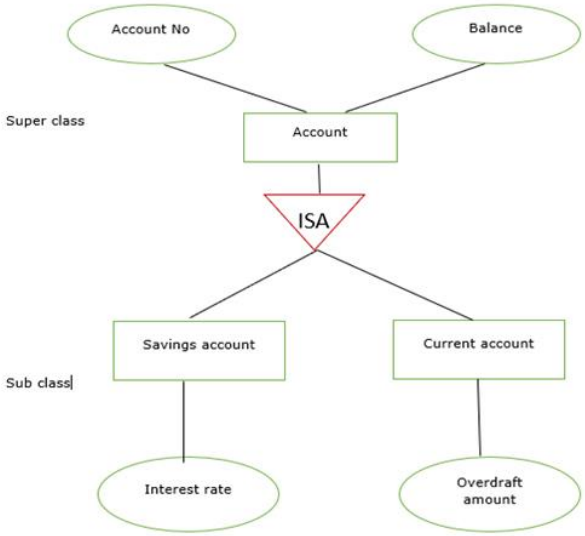
- Aggregation
- Specialization
- Generalization

Specialization

The process of designing sub groupings within an entity set is called specialization. It is a top-down process. If an entity set is given with all the attributes in which the instances of the entity set are differentiated according to the given attribute value, then that sub-classes or the sub-entity sets can be formed from the given attribute.

Example

Specialization of a person allows us to distinguish a person according to whether they are employees or customers. Specialization of account creates two entity sets: savings account and current account.



In the E-R diagram specialization is represented by triangle components labeled ISA. The ISA relationship is referred as superclass- subclass relationship as shown below –

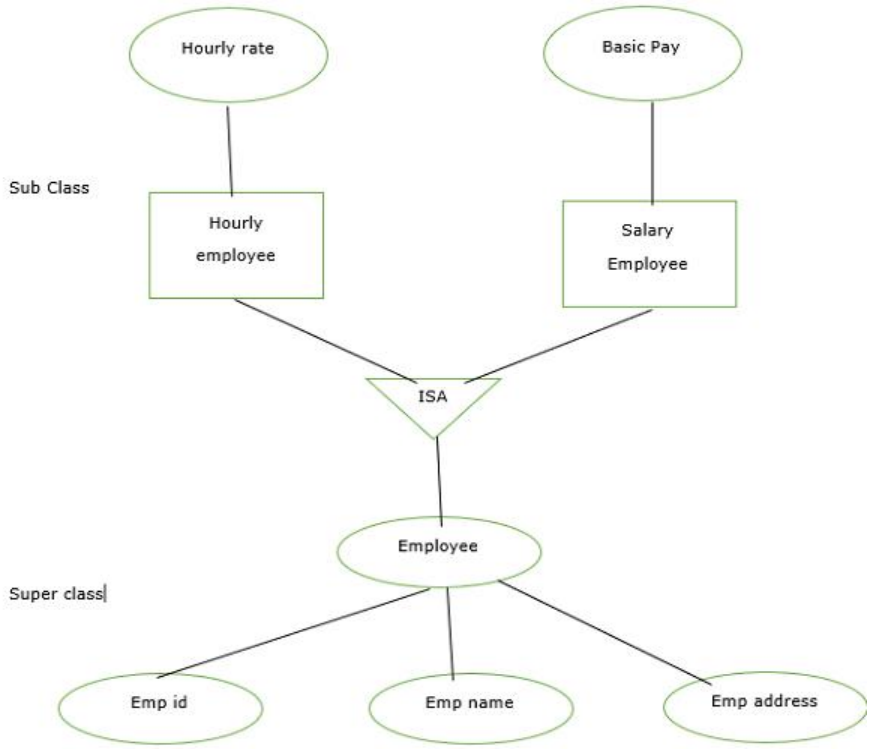
Generalization

It is the reverse process of specialization. It is a bottom-up approach.

It converts subclasses to superclasses. This process combines a number of entity sets that share the same features into higher-level entity sets.

If the sub-class information is given for the given entity set then, ISA relationship type will be used to represent the connectivity between the subclass and superclass as shown below −

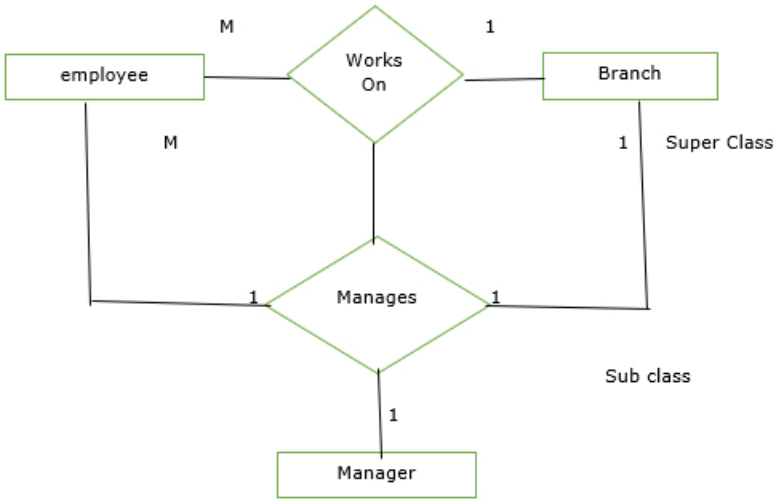
Example



Aggregation

It is an abstraction in which relationship sets are treated as higher level entity sets and can participate in relationships. Aggregation allows us to indicate that a relationship set participates in another relationship set.

Aggregation is used to simplify the details of a given database where ternary relationships will be changed into binary relationships. Ternary relation is only one type of relationship which is working between three entities.



1.7 UML Class Diagram

The UML Class diagram is a graphical notation used to construct and visualize object-oriented systems. A class diagram in the Unified Modeling Language (UML) is a type of static structure diagram that describes the structure of a system by showing the system's:

- classes,
- their attributes,
- operations (or methods),
- and the relationships among objects.

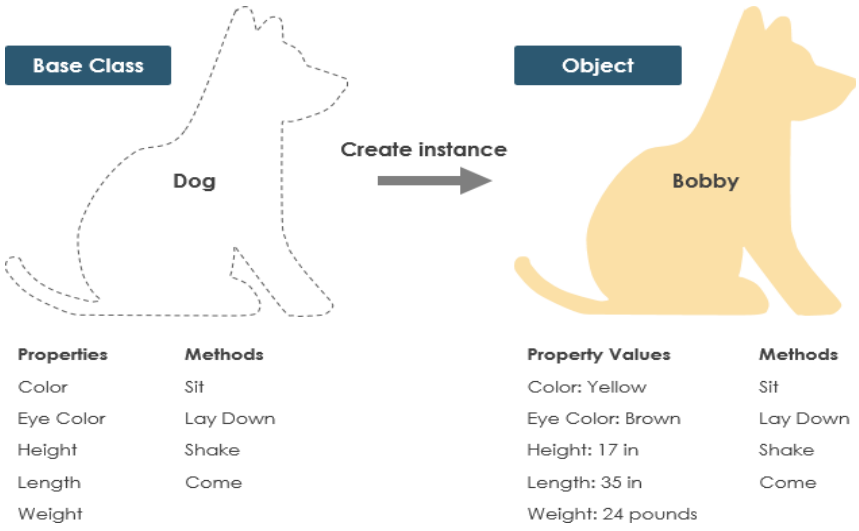
What is a Class?

A Class is a blueprint for an object. Objects and classes go hand in hand. We can't talk about one without talking about the other. And the entire point of Object-Oriented Design is not about objects, it's about classes, because we use classes to create objects. So a class describes what an object will be, but it isn't the object itself.

In fact, classes describe the type of objects, while objects are usable instances of classes. Each Object was built from the same set of blueprints and therefore contains the same components (properties and methods). The standard meaning is that an object is an instance of a class and object - Objects have states and behaviors.

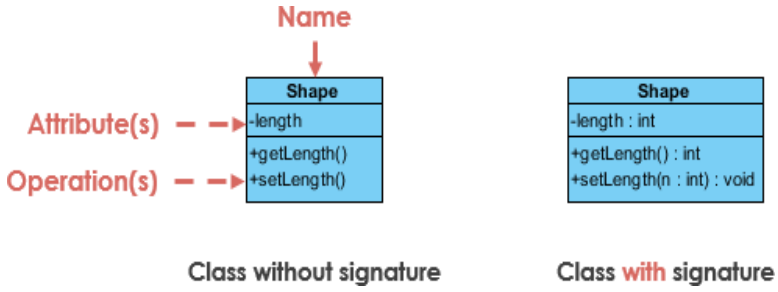
Example

A dog has states - color, name, breed as well as behaviors -wagging, barking, eating. An object is an instance of a class.



UML Class Notation

A class represent a concept which encapsulates state (attributes) and behavior (operations). Each attribute has a type. Each operation has a signature. The class name is the only mandatory information.



Class Name:

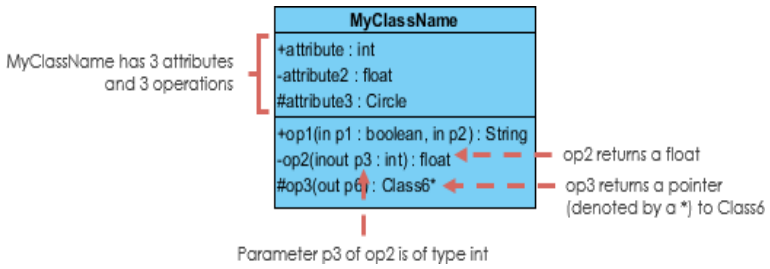
- The name of the class appears in the first partition.

Class Attributes:

- Attributes are shown in the second partition.
- The attribute type is shown after the colon.
- Attributes map onto member variables (data members) in code.

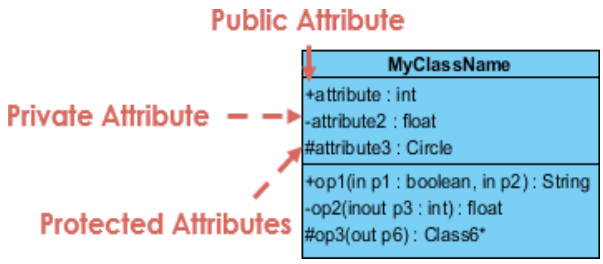
Class Operations (Methods):

- Operations are shown in the third partition. They are services the class provides.
- The return type of a method is shown after the colon at the end of the method signature.
- The return type of method parameters is shown after the colon following the parameter name. Operations map onto class methods in code



Class Visibility

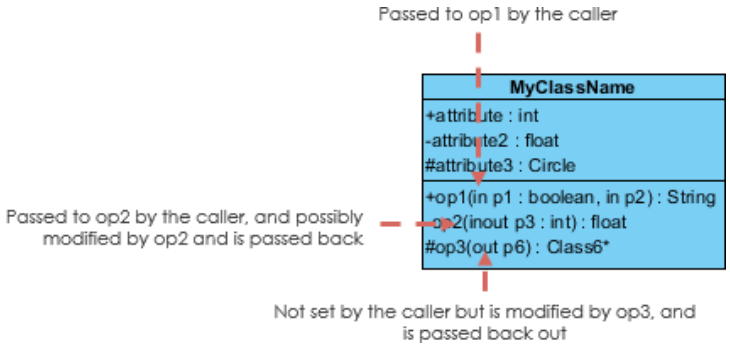
The +, - and # symbols before an attribute and operation name in a class denote the visibility of the attribute and operation.



- + denotes public attributes or operations
- - denotes private attributes or operations
- # denotes protected attributes or operations

Parameter Directionality

Each parameter in an operation (method) may be denoted as in, out or inout which specifies its direction with respect to the caller. This directionality is shown before the parameter name.



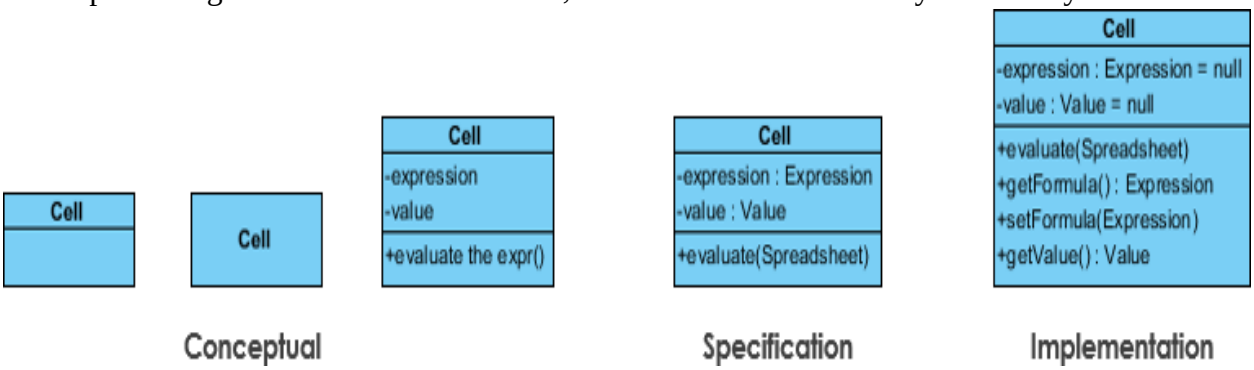
Perspectives of Class Diagram

The choice of perspective depends on how far along you are in the development process. During the formulation of a domain model, for example, you would seldom move past the conceptual perspective. Analysis models will typically feature a mix of conceptual and specification perspectives. Design model development will typically start with heavy emphasis on the specification perspective, and evolve into the implementation perspective.

A diagram can be interpreted from various perspectives:

1. **Conceptual:** represents the concepts in the domain
2. **Specification:** focus is on the interfaces of Abstract Data Type (ADTs) in the software
3. **Implementation:** describes how classes will implement their interfaces

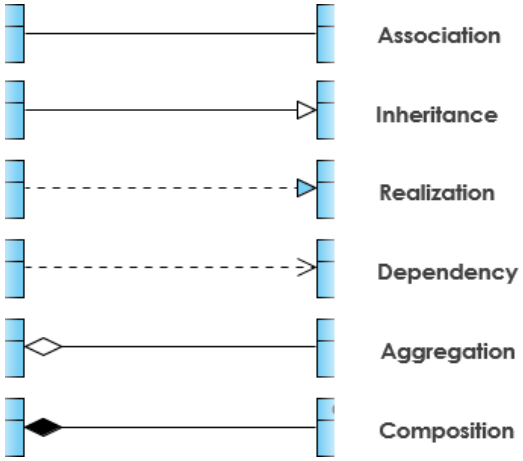
The perspective affects the amount of detail to be supplied and the kinds of relationships worth presenting. As we mentioned above, the class name is the only mandatory information.



Relationships between classes

UML is not just about pretty pictures. If used correctly, UML precisely conveys how code should be implemented from diagrams. If precisely interpreted, the implemented code will correctly reflect the intent of the designer. Can you describe what each of the relationships mean relative to your target programming language shown in the Figure below?

If you can't yet recognize them, no problem this section is meant to help you to understand UML class relationships. A class may be involved in one or more relationships with other classes. A relationship can be one of the following types:

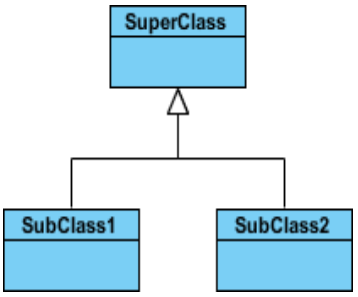


Inheritance (or Generalization): A generalization is a taxonomic relationship between a more general classifier and a more specific classifier. Each instance of the specific classifier is also an indirect instance of the general classifier. Thus, the specific classifier inherits the features of the more general classifier.

- Represents an "is-a" relationship.

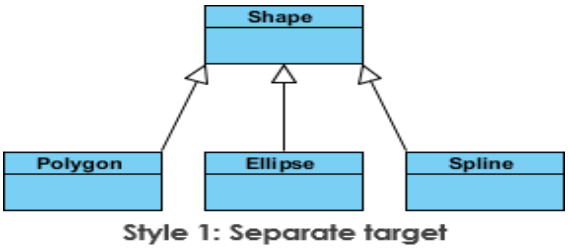
- An abstract class name is shown in italics.
- SubClass1 and SubClass2 are specializations of SuperClass.

The figure below shows an example of inheritance hierarchy. SubClass1 and SubClass2 are derived from SuperClass. The relationship is displayed as a solid line with a hollow arrowhead that points from the child element to the parent element.

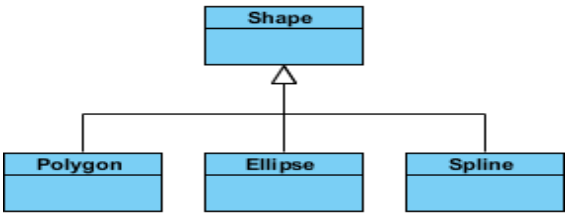


Inheritance Example - Shapes

The figure below shows an inheritance example with two styles. Although the connectors are drawn differently, they are semantically equivalent.



Style 1: Separate target



Style 2: Shared target

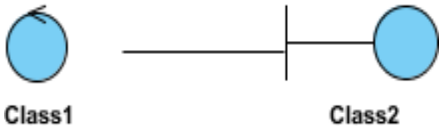
Association

Associations are relationships between classes in a UML Class Diagram. They are represented by a solid line between classes. Associations are typically named using a verb or verb phrase which reflects the real-world problem domain.

Simple Association

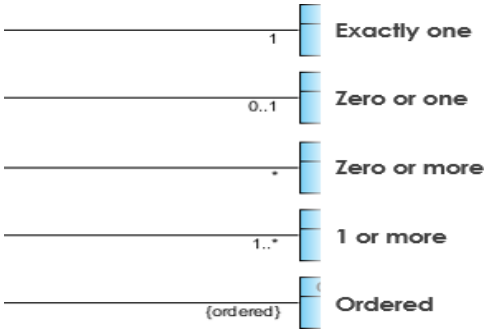
- A structural link between two peer classes.
- There is an association between Class1 and Class2

The figure below shows an example of simple association. There is an association that connects the <<control>> class Class1 and <<boundary>> class Class2. The relationship is displayed as a solid line connecting the two classes.



Cardinality: Cardinality is expressed in terms of:

- one to one
- one to many
- many to many

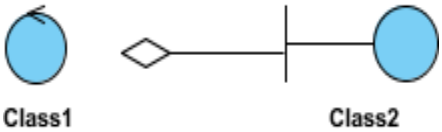


Aggregation

A special type of association.

- It represents a "part of" relationship.
- Class2 is part of Class1.
- Many instances (denoted by the *) of Class2 can be associated with Class1.
- Objects of Class1 and Class2 have separate lifetimes.

The figure below shows an example of aggregation. The relationship is displayed as a solid line with an unfilled diamond at the association end, which is connected to the class that represents the aggregate.

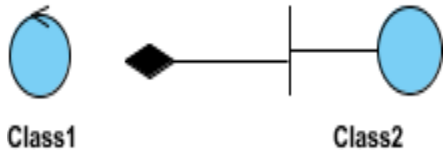


Composition

- A special type of aggregation where parts are destroyed when the whole is destroyed.
- Objects of Class2 live and die with Class1.

- Class2 cannot stand by itself.

The figure below shows an example of composition. The relationship is displayed as a solid line with a filled diamond at the association end, which is connected to the class that represents the whole or composite.

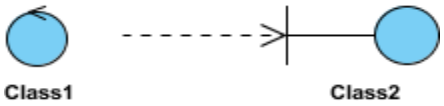


Dependency

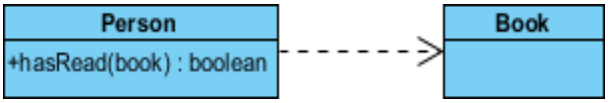
An object of one class might use an object of another class in the code of a method. If the object is not stored in any field, then this is modeled as a dependency relationship.

- A special type of association.
- Exists between two classes if changes to the definition of one may cause changes to the other (but not the other way around).
- Class1 depends on Class2

The figure below shows an example of dependency. The relationship is displayed as a dashed line with an open arrow.



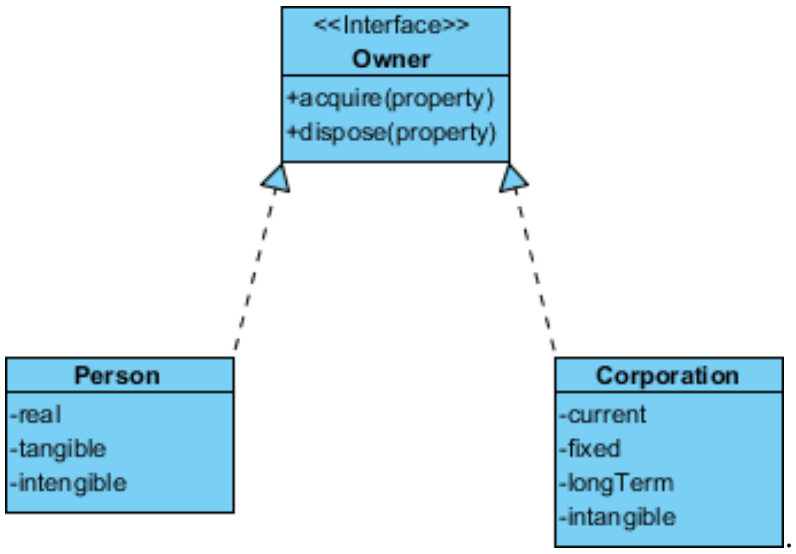
The figure below shows another example of dependency. The Person class might have a hasRead method with a Book parameter that returns true if the person has read the book (perhaps by checking some database).



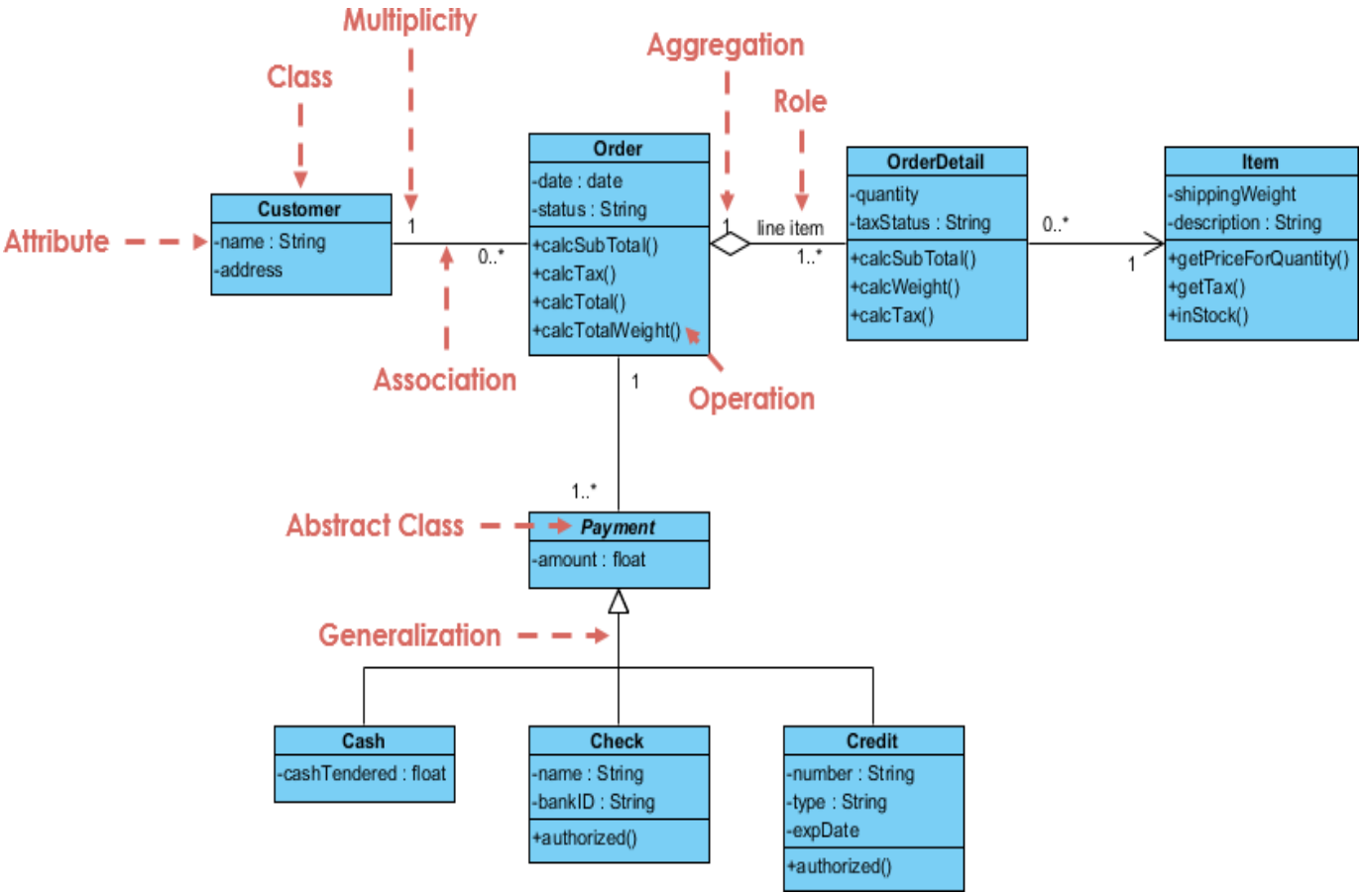
Realization

Realization is a relationship between the blueprint class and the object containing its respective implementation level details. This object is said to realize the blueprint class. In other words, you can understand this as the relationship between the interface and the implementing class.

For example, the Owner interface might specify methods for acquiring property and disposing of property. The Person and Corporation classes need to implement these methods, possibly in very different ways

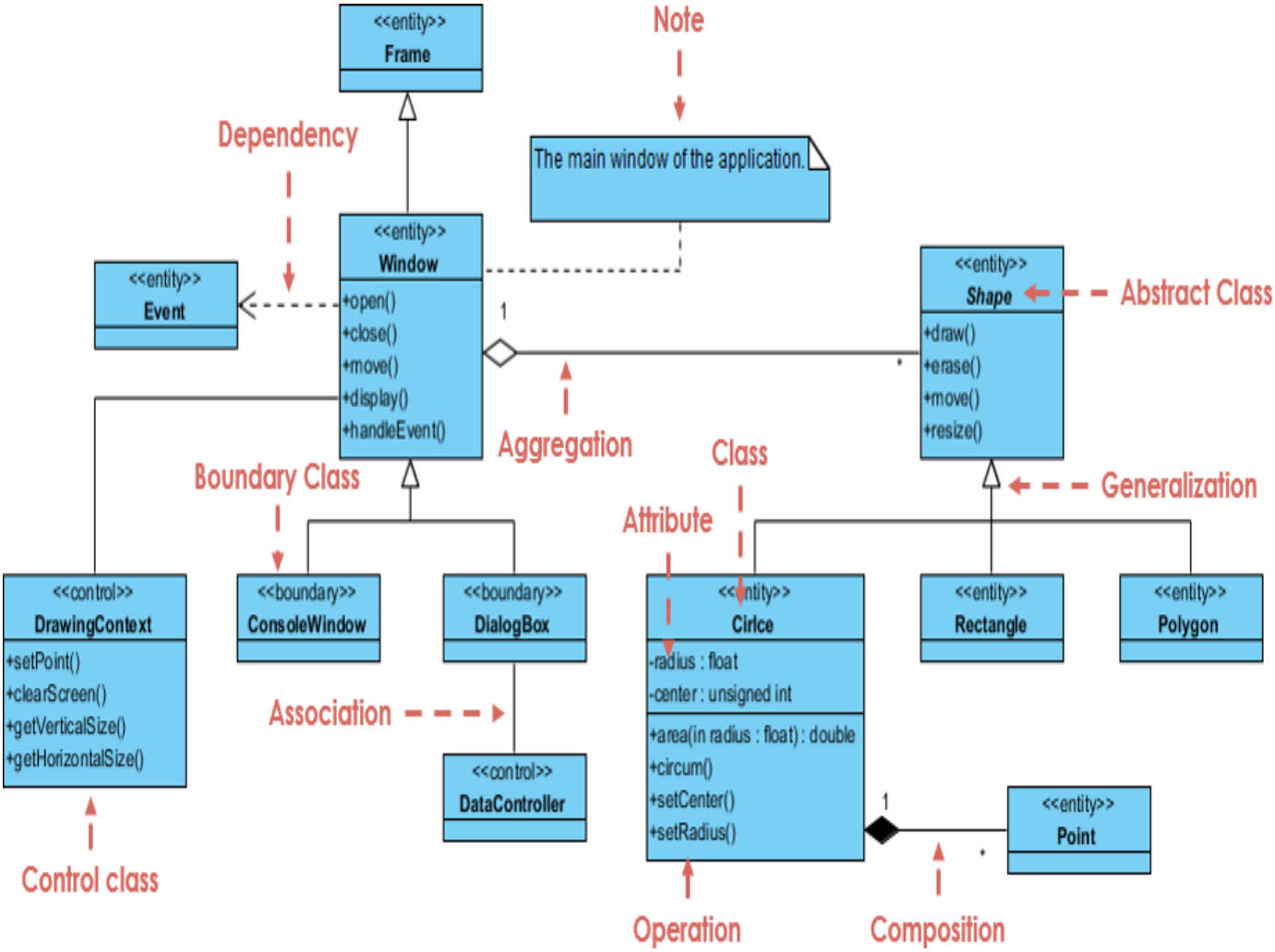


Class Diagram Example: Order System



Class Diagram Example: GUI

A class diagram may also have notes attached to classes or relationships.



UNIT II

RELATIONAL MODEL AND SQL

Relational model concepts --Integrity constraints --SQL Data manipulation –SQL Data definition –Views --SQL programming.

2.1 Relational Model concept

Relational model can represent as a table with columns and rows. Each row is known as a tuple. Each table of the column has a name or attribute.

Domain: It contains a set of atomic values that an attribute can take.

Attribute: It contains the name of a column in a particular table. Each attribute Ai must have a domain, dom(Ai)

Relational instance: In the relational database system, the relational instance is represented by a finite set of tuples. Relation instances do not have duplicate tuples.

Relational schema: A relational schema contains the name of the relation and name of all columns or attributes.

Relational key: In the relational key, each row has one or more attributes. It can identify the row in the relation uniquely.

Example: STUDENT Relation

NAME	ROLL_NO	PHONE_NO	ADDRESS	AGE
Ram	14795	7305758992	Noida	24
Shyam	12839	9026288936	Delhi	35
Laxman	33289	8583287182	Gurugram	20
Mahesh	27857	7086819134	Ghaziabad	27
Ganesh	17282	9028 9i3988	Delhi	40

- In the given table, NAME, ROLL_NO, PHONE_NO, ADDRESS, and AGE are the attributes.
- The instance of schema STUDENT has 5 tuples.
- t3 = <Laxman, 33289, 8583287182, Gurugram, 20>

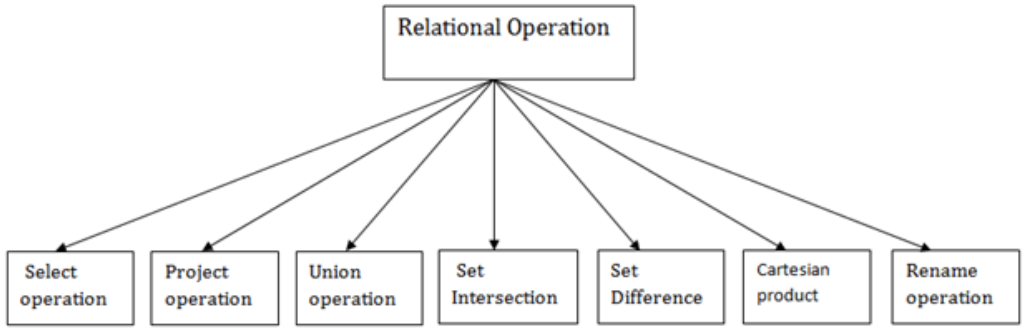
Properties of Relations

- Name of the relation is distinct from all other relations.
- Each relation cell contains exactly one atomic (single) value
- Each attribute contains a distinct name
- Attribute domain has no significance
- tuple has no duplicate value
- Order of tuple can have a different sequence

Relational Algebra

Relational algebra is a procedural query language. It gives a step by step process to obtain the result of the query. It uses operators to perform queries.

Types of Relational operation



1. Select Operation:

- The select operation selects tuples that satisfy a given predicate.
- It is denoted by sigma (σ).

Notation: σ p(r)

Where:

σ is used for selection prediction

r is used for relation

p is used as a propositional logic formula which may use connectors like: AND OR and NOT. These relational can use as relational operators like =, ≠, ≥, <, >, ≤.

BRANCH_NAME	LOAN_NO	AMOUNT
Downtown	L-17	1000
Redwood	L-23	2000
Perryride	L-15	1500
Downtown	L-14	1500
Mianus	L-13	500
Roundhill	L-11	900
Perryride	L-16	1300

Input:

σ BRANCH_NAME="perryride" (LOAN)

Output:

BRANCH_NAME	LOAN_NO	AMOUNT

Perryride	L-15	1500
Perryride	L-16	1300

2. Project Operation:

This operation shows the list of those attributes that we wish to appear in the result. Rest of the attributes are eliminated from the table.

It is denoted by ∏.

Notation: ∏ A1, A2, An (r)

Where

A1, A2, A3 is used as an attribute name of relation r.

Example: CUSTOMER RELATION

NAME	STREET	CITY
Jones	Main	Harrison
Smith	North	Rye
Hays	Main	Harrison
Curry	North	Rye
Johnson	Alma	Brooklyn
Brooks	Senator	Brooklyn

Input:

∏ NAME, CITY (CUSTOMER)

Output:

NAME	CITY
Jones	Harrison
Smith	Rye
Hays	Harrison
Curry	Rye
Johnson	Brooklyn
Brooks	Brooklyn

3. Union Operation:

- Suppose there are two tuples R and S. The union operation contains all the tuples that are either in R or S or both in R & S.
- It eliminates the duplicate tuples. It is denoted by ∪.

Notation: R ∪ S

A union operation must hold the following condition:

- R and S must have the attribute of the same number.
- Duplicate tuples are eliminated automatically.

Example:

DEPOSITOR RELATION

CUSTOMER_NAME	ACCOUNT_NO
Johnson	A-101
Smith	A-121
Mayes	A-321
Turner	A-176
Johnson	A-273
Jones	A-472
Lindsay	A-284

BORROW RELATION

CUSTOMER_NAME	LOAN_NO
Jones	L-17
Smith	L-23
Hayes	L-15
Jackson	L-14
Curry	L-93
Smith	L-11
Williams	L-17

Input:

∏ CUSTOMER_NAME (BORROW) ∪ ∏ CUSTOMER_NAME (DEPOSITOR)

Output:

CUSTOMER_NAME
Johnson
Smith
Hayes
Turner

Jones

Lindsay

Jackson

Curry

Williams

Mayes

4. Set Intersection:

- Suppose there are two tuples R and S. The set intersection operation contains all tuples that are in both R & S.
- It is denoted by intersection ∩.

Notation: R ∩ S

Example: Using the above DEPOSITOR table and BORROW table

Input:

∏ CUSTOMER_NAME (BORROW) ∩ ∏ CUSTOMER_NAME (DEPOSITOR)

Output:

CUSTOMER_NAME

Smith

Jones

5. Set Difference:

Suppose there are two tuples R and S. The set intersection operation contains all tuples that are in R but not in S.

It is denoted by intersection minus (-).

Notation: R - S

Example: Using the above DEPOSITOR table and BORROW table

Input:

∏ CUSTOMER_NAME (BORROW) - ∏ CUSTOMER_NAME (DEPOSITOR)

CUSTOMER_NAME

Jackson

Hayes

Willians

Curry

Output:

6. Cartesian product

The Cartesian product is used to combine each row in one table with each row in the other table. It is also known as a cross product.

It is denoted by X.

Notation: E X D

Example:

EMPLOYEE

EMP_ID	EMP_NAME	EMP_DEPT
1	Smith	A
2	Harry	C
3	John	B

DEPARTMENT

DEPT_NO	DEPT_NAME
A	Marketing
B	Sales
C	Legal

Input:

EMPLOYEE X DEPARTMENT

Output:

EMP_ID	EMP_NAME	EMP_DEPT	DEPT_NO	DEPT_NAME
1	Smith	A	A	Marketing
1	Smith	A	B	Sales
1	Smith	A	C	Legal

2	Harry	C	A	Marketing
2	Harry	C	B	Sales
2	Harry	C	C	Legal
3	John	B	A	Marketing
3	John	B	B	Sales
3	John	B	C	Legal

7. Rename Operation:

The rename operation is used to rename the output relation. It is denoted by rho (ρ).

Example: We can use the rename operator to rename STUDENT relation to STUDENT1.

ρ (STUDENT1, STUDENT)

8. Join Operations:

A Join operation combines related tuples from different relations, if and only if a given join condition is satisfied. It is denoted by ⋈.

Example:

EMPLOYEE

EMP_CODE	EMP_NAME
101	Stephan
102	Jack
103	Harry

SALARY

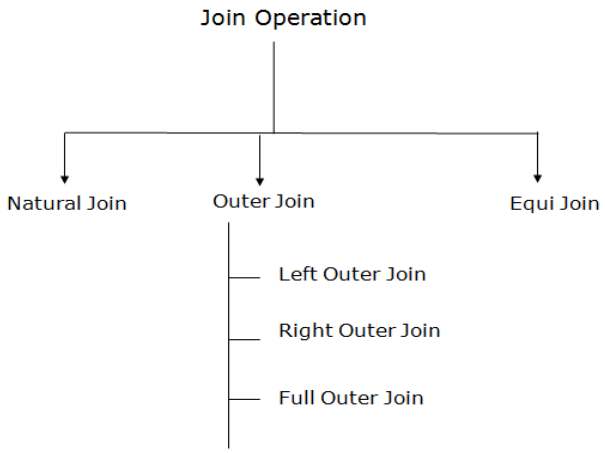
EMP_CODE	SALARY
101	50000
102	30000
103	25000

Operation: (EMPLOYEE ⋈ SALARY)

EMP_CODE	EMP_NAME	SALARY
101	Stephan	50000
102	Jack	30000

103	Harry	25000

Types of Join operations:



1. Natural Join:

A natural join is the set of tuples of all combinations in R and S that are equal on their common attribute names. It is denoted by ⋈.

Example: Let's use the above EMPLOYEE table and SALARY table:

Input: ∏EMP_NAME, SALARY (EMPLOYEE ⋈ SALARY)

Output:

EMP_NAME	SALARY
Stephan	50000
Jack	30000
Harry	25000

.

2. Outer Join:

The outer join operation is an extension of the join operation. It is used to deal with missing information

Example:

EMPLOYEE

EMP_NAME	STREET	CITY
Ram	Civil line	Mumbai
Shyam	Park street	Kolkata
Ravi	M.G. Street	Delhi
Hari	Nehru nagar	Hyderabad

FACT_WORKERS

EMP_NAME	BRANCH	SALARY
Ram	Infosys	10000
Shyam	Wipro	20000
Kuber	HCL	30000
Hari	TCS	50000

Input: (EMPLOYEE ⋈ FACT_WORKERS)

Output:

EMP_NAME	STREET	CITY	BRANCH	SALARY
Ram	Civil line	Mumbai	Infosys	10000
Shyam	Park street	Kolkata	Wipro	20000
Hari	Nehru nagar	Hyderabad	TCS	50000

An outer join is basically of three types:

1. Left outer join
2. Right outer join
3. Full outer join

a. Left outer join:

- Left outer join contains the set of tuples of all combinations in R and S that are equal on their common attribute names.
- In the left outer join, tuples in R have no matching tuples in S.
- It is denoted by ⟕.

Example: Using the above EMPLOYEE table and FACT_WORKERS table

Input:

EMPLOYEE ⋈ FACT_WORKERS

EMP_NAME	STREET	CITY	BRANCH	SALARY
Ram	Civil line	Mumbai	Infosys	10000
Shyam	Park street	Kolkata	Wipro	20000
Hari	Nehru street	Hyderabad	TCS	50000
Ravi	M.G. Street	Delhi	NULL	NULL

b. Right outer join:

- Right outer join contains the set of tuples of all combinations in R and S that are equal on their common attribute names.
- In right outer join, tuples in S have no matching tuples in R.
- It is denoted by ⋊.

Example: Using the above EMPLOYEE table and FACT_WORKERS Relation

Input:

EMPLOYEE ⋊ FACT_WORKERS

Output:

EMP_NAME	BRANCH	SALARY	STREET	CITY
Ram	Infosys	10000	Civil line	Mumbai
Shyam	Wipro	20000	Park street	Kolkata
Hari	TCS	50000	Nehru street	Hyderabad
Kuber	HCL	30000	NULL	NULL

c. Full outer join:

- Full outer join is like a left or right join except that it contains all rows from both tables.
- In full outer join, tuples in R that have no matching tuples in S and tuples in S that have no matching tuples in R in their common attribute name.
- It is denoted by ⟗.

Example:

Using the above EMPLOYEE table and FACT_WORKERS table

Input:

EMPLOYEE ⟗ FACT_WORKERS

Output:

EMP_NAME	STREET	CITY	BRANCH	SALARY
Ram	Civil line	Mumbai	Infosys	10000
Shyam	Park street	Kolkata	Wipro	20000
Hari	Nehru street	Hyderabad	TCS	50000
Ravi	M.G. Street	Delhi	NULL	NULL
Kuber	NULL	NULL	HCL	30000

3. Equi join:

It is also known as an inner join. It is the most common join. It is based on matched data as per the equality condition. The equi join uses the comparison operator (=).

Example:

CUSTOMER RELATION

CLASS_ID	NAME
1	John
2	Harry
3	Jackson

PRODUCT

PRODUCT_ID	CITY
1	Delhi
2	Mumbai
3	Noida

Input:

CUSTOMER ⋈ PRODUCT

Output:

CLASS_ID	NAME	PRODUCT_ID	CITY
1	John	1	Delhi
2	Harry	2	Mumbai
3	Harry	3	Noida

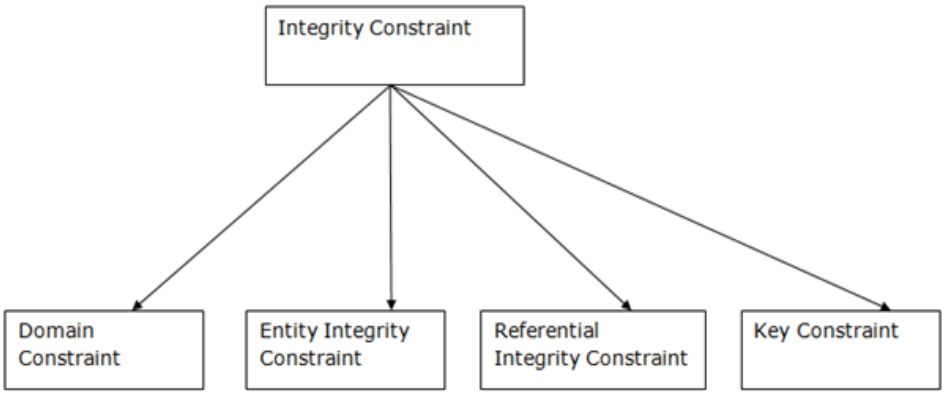
2.2 Integrity Constraints

Integrity constraints are a set of rules. It is used to maintain the quality of information.

Integrity constraints ensure that the data insertion, updating, and other processes have to be performed in such a way that data integrity is not affected.

Thus, integrity constraint is used to guard against accidental damage to the database.

Types of Integrity Constraint



1. Domain constraints

Domain constraints can be defined as the definition of a valid set of values for an attribute.

The data type of domain includes string, character, integer, time, date, currency, etc. The value of the attribute must be available in the corresponding domain.

Example:

ID	NAME	SEMENSTER	AGE
1000	Tom	1st	17
1001	Johnson	2nd	24
1002	Leonardo	5th	21
1003	Kate	3rd	19
1004	Morgan	8th	A

Not allowed. Because AGE is an integer attribute

2. Entity integrity constraints

The entity integrity constraint states that primary key value can't be null.

This is because the primary key value is used to identify individual rows in relation and if the primary key has a null value, then we can't identify those rows.

A table can contain a null value other than the primary key field.

Example:

EMPLOYEE

EMP_ID	EMP_NAME	SALARY
123	Jack	30000
142	Harry	60000
164	John	20000
	Jackson	27000

Not allowed as primary key can't contain a NULL value

3. Referential Integrity Constraints

A referential integrity constraint is specified between two tables.

In the Referential integrity constraints, if a foreign key in Table 1 refers to the Primary Key of Table 2, then every value of the Foreign Key in Table 1 must be null or be available in Table 2.

Example:

(Table 1)

EMP_NAME	NAME	AGE	D_No
1	Jack	20	11
2	Harry	40	24
3	John	27	18
4	Devil	38	13

Foreign key

Not allowed as D_No 18 is not defined as a Primary key of table 2 and In table 1, D_No is a foreign key defined

Relationships

(Table 2)

Primary Key

D_No	D_Location
11	Mumbai
24	Delhi
13	Noida

4. Key constraints

Keys are the entity set that is used to identify an entity within its entity set uniquely.

An entity set can have multiple keys, but out of which one key will be the primary key. A primary key can contain a unique and null value in the relational table.

Example:

ID	NAME	SEMENSTER	AGE
1000	Tom	1st	17
1001	Johnson	2nd	24
1002	Leonardo	5th	21
1003	Kate	3rd	19
1002	Morgan	8th	22

Not allowed. Because all row must be unique

SQL

SQL is the Structured Query Language used to store, manipulate, and retrieve data present in a database server.

MySQL is a relational database management system. It supports large databases and is customizable. This article will discuss the important commands in SQL.

SQL Commands

SQL commands are instructions that are used to communicate with the database. It is used to perform specific tasks, work, and functions with data in the database.

2.3 SQL Data Manipulate Language

Data Manipulation Language (DML) is the language that gives users the ability to access, or manipulate the condition that the appropriate data model has inherited.

Here are some SQL commands that come under DML:

- INSERT
- UPDATE
- DELETE

1. INSERT

INSERT command is used to insert new rows or records in a table.

Syntax:

INSERT INTO TABLE_NAME (column1, column2, column3...columnN)

VALUES (value1, value2, value3...valueN);

Example:

INSERT INTO Employees (Emp_Id, Emp_Name) VALUES (04, “Sam Tully”);

2. UPDATE

This command is used to update or modify the value of a column in the table.

Syntax:

UPDATE table_name

SET column1 = value1, column2 = value2...., columnN = valueN

WHERE [condition];

Example:

UPDATE Employees

SET Salary = 1000

WHERE Emp_Id = 04;

The above code will modify the salary of the employee with Emp_ID= 04;

3. DELETE

DELETE is used for removing one or more rows from the table.

Syntax:

DELETE FROM TableName

WHERE Condition;

Example:

DELETE FROM Employees

WHERE Emp_Id = 04;

This will delete the record of the Employees whose Emp_ID is 4

2.4 Data Definition Language

Data Definition Language helps you to define the database structure or schemas. DDL commands are capable of creating, deleting, and modifying data.

Here are some commands that come under DDL:

- CREATE
- ALTER
- DROP
- TRUNCATE

1. CREATE

It is used to create a new table in the database.

Syntax:

CREATE TABLE TableName (Column1 datatype, Column2 datatype, Column3 datatype,ColumnN datatype);

Example:

CREATE TABLE Employees {

Emp_Id int(3), Emp_Name varchar (20) };

This will create a table Employees with Emp_ID and Emp_Name.

2. ALTER

Alter is used to alter the structure of the database.

Syntax:

ALTER TABLE TableName

ADD ColumnName Datatype;

ALTER TABLE TableName

DROP COLUMN ColumnName;

Example:

ALTER TABLE EmployeesADD BloodGroup varchar (255);

This will add a column BloodGroup to the existing table Employees.

ALTER TABLE EmployeesDROP BloodGroup varchar (255);

This will drop the column BloodGroup from the existing table.

3. DROP

It is used to delete both the structure and record in the table.

Syntax:

DROP TABLE TableName;

Example:

DROP TABLE Employees;

This SQL command will remove the table structure along with its data from the database.

4. TRUNCATE

This truncate command is used to delete all the rows from the table and free the space.

Syntax:

TRUNCATE TABLE TableName;

Example:

TRUNCATE TABLE Employees;

2.5 SQL VIEWS

A view is nothing more than a SQL statement that is stored in the database with an associated name. A view is actually a composition of a table in the form of a predefined SQL query.

A view can contain all rows of a table or select rows from a table. A view can be created from one or many tables which depends on the written SQL query to create a view.

Views, which are a type of virtual tables allow users to do the following –

- Structure data in a way that users or classes of users find natural or intuitive.
- Restrict access to the data in such a way that a user can see and (sometimes) modify exactly what they need and no more.
- Summarize data from various tables which can be used to generate reports.

Creating Views

Database views are created using the CREATE VIEW statement. Views can be created from a single table, multiple tables or another view.

To create a view, a user must have the appropriate system privilege according to the specific implementation.

The basic CREATE VIEW syntax is as follows –

CREATE VIEW view_name AS SELECT column1, column2....FROM table _name WHERE [condition];

You can include multiple tables in your SELECT statement in a similar way as you use them in a normal SQL SELECT query.

Example

Consider the CUSTOMERS table having the following records –

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	32	Ahmedabad	2000.00
2	Khilan	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00

Following is an example to create a view from the CUSTOMERS table. This view would be used to have customer name and age from the CUSTOMERS table.

SQL > CREATE VIEW CUSTOMERS_VIEW AS

SELECT name, age FROM CUSTOMERS;

Now, you can query CUSTOMERS_VIEW in a similar way as you query an actual table. Following is an example for the same.

SQL > SELECT * FROM CUSTOMERS_VIEW;

This would produce the following result.

name	age
Ramesh	32
Khilan	25
kaushik	23
Chaitali	25
Hardik	27
Komal	22
Muffy	24

The WITH CHECK OPTION

The WITH CHECK OPTION is a CREATE VIEW statement option. The purpose of the WITH CHECK OPTION is to ensure that all UPDATE and INSERTs satisfy the condition(s) in the view definition.

If they do not satisfy the condition(s), the UPDATE or INSERT returns an error.

The following code block has an example of creating same view CUSTOMERS_VIEW with the WITH CHECK OPTION.

CREATE VIEW CUSTOMERS_VIEW AS SELECT name, age FROM CUSTOMERS WHERE age IS NOT NULL WITH CHECK OPTION;

The WITH CHECK OPTION in this case should deny the entry of any NULL values in the view's AGE column, because the view is defined by data that does not have a NULL value in the AGE column.

Updating a View

A view can be updated under certain conditions which are given below −

- The SELECT clause may not contain the keyword DISTINCT.
- The SELECT clause may not contain summary functions.
- The SELECT clause may not contain set functions.
- The SELECT clause may not contain set operators.
- The SELECT clause may not contain an ORDER BY clause.
- The FROM clause may not contain multiple tables.
- The WHERE clause may not contain subqueries.
- The query may not contain GROUP BY or HAVING.
- Calculated columns may not be updated.
- All NOT NULL columns from the base table must be included in the view in order for the INSERT query to function.

So, if a view satisfies all the above-mentioned rules then you can update that view. The following code block has an example to update the age of Ramesh.

SQL > UPDATE CUSTOMERS_VIEW SET AGE = 35 WHERE name = 'Ramesh';

This would ultimately update the base table CUSTOMERS and the same would reflect in the view itself. Now, try to query the base table and the SELECT statement would produce the following result.

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	35	Ahmedabad	2000.00

2	Khilan	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
6	Komal	22	MP	4500.00
7	Muffy	24	Indore	10000.00

Inserting Rows into a View

Rows of data can be inserted into a view. The same rules that apply to the UPDATE command also apply to the INSERT command.

Here, we cannot insert rows in the CUSTOMERS_VIEW because we have not included all the NOT NULL columns in this view, otherwise you can insert rows in a view in a similar way as you insert them in a table.

Deleting Rows into a View

Rows of data can be deleted from a view. The same rules that apply to the UPDATE and INSERT commands apply to the DELETE command.

Following is an example to delete a record having AGE = 22

SQL > DELETE FROM CUSTOMERS_VIEW WHERE age = 22;

This would ultimately delete a row from the base table CUSTOMERS and the same would reflect in the view itself. Now, try to query the base table and the SELECT statement would produce the following result.

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	35	Ahmedabad	2000.00
2	khilan	25	Delhi	1500.00
3	Kaushik	23	Kota	2000.00
4	Chaitali	25	Mumbai	6500.00
5	Hardik	27	Bhopal	8500.00
7	Muffy	24	Indore	10000.00

Dropping Views

Obviously, where you have a view, you need a way to drop the view if it is no longer needed. The syntax is very simple and is given below −

DROP VIEW view_name;

Following is an example to drop the CUSTOMERS_VIEW from the CUSTOMERS table.

DROP VIEW CUSTOMERS_VIEW;

2.6 SQL Programming

SQL is a standard language for storing, manipulating and retrieving data in databases.

Our SQL tutorial will teach you how to use SQL in: MySQL, SQL Server, MS Access, Oracle, Sybase, Informix, PostgreSQL, and other database systems.

Example: SELECT * FROM Customers;

What is SQL?

- SQL stands for Structured Query Language
- SQL lets you access and manipulate databases
- SQL became a standard of the American National Standards Institute (ANSI) in 1986, and of the International Organization for Standardization (ISO) in 1987

What Can SQL do?

- SQL can execute queries against a database
- SQL can retrieve data from a database
- SQL can insert records in a database
- SQL can update records in a database
- SQL can delete records from a database
- SQL can create new databases
- SQL can create new tables in a database
- SQL can create stored procedures in a database
- SQL can create views in a database
- SQL can set permissions on tables, procedures, and views

SQL is a Standard - BUT....

Although SQL is an ANSI/ISO standard, there are different versions of the SQL language.

However, to be compliant with the ANSI standard, they all support at least the major commands (such as SELECT, UPDATE, DELETE, INSERT, WHERE) in a similar manner.

Using SQL in Your Web Site

To build a web site that shows data from a database, you will need:

- An RDBMS database program (i.e. MS Access, SQL Server, MySQL)
- To use a server-side scripting language, like PHP or ASP
- To use SQL to get the data you want
- To use HTML / CSS to style the page

RDBMS

RDBMS stands for Relational Database Management System.

RDBMS is the basis for SQL, and for all modern database systems such as MS SQL Server, IBM DB2, Oracle, MySQL, and Microsoft Access.

The data in RDBMS is stored in database objects called tables. A table is a collection of related data entries and it consists of columns and rows.

Example

SELECT * FROM Customers;

Every table is broken up into smaller entities called fields. The fields in the Customers table consist of Customer ID, Customer Name, Contact Name, Address, City, Postal Code and Country. A field is a column in a table that is designed to maintain specific information about every record in the table.

A record, also called a row, is each individual entry that exists in a table. For example, there are 91 records in the above Customers table. A record is a horizontal entity in a table.

A column is a vertical entity in a table that contains all information associated with a specific field in a table.

SQL INSERT INTO Statement

The SQL INSERT INTO Statement

The INSERT INTO statement is used to insert new records in a table.

INSERT INTO Syntax

It is possible to write the INSERT INTO statement in two ways:

1. Specify both the column names and the values to be inserted:

INSERT INTO table_name (column1, column2, column3, ...) VALUES (value1, value2, value3, ...);

2. If you are adding values for all the columns of the table, you do not need to specify the column names in the SQL query. However, make sure the order of the values is in the same order as the columns in the table. Here, the INSERT INTO syntax would be as follows:

INSERT INTO table name VALUES (value1, value2, value3, ...);

INSERT INTO Example

The following SQL statement inserts a new record in the "Customers" table:

Example

INSERT INTO Customers (Customer Name, Contact Name, Address, City, Postal Code, Country)

VALUES ('Cardinal', 'Tom B. Erichsen', 'Skagen 21', 'Stavanger', '4006', 'Norway');

The selection from the "Customers" table will now look like this:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
89	White Clover Markets	Karl Jablonski	305 - 14th Ave. S. Suite 3B	Seattle	98128	USA
90	Wilman Kala	Matti Karttunen	Keskuskatu 45	Helsinki	21240	Finland
91	Wolski	Zbyszek	ul. Filtrowa 68	Walla	01-012	Poland
92	Cardinal	Tom B. Erichsen	Skagen 21	Stavanger	4006	Norway

Insert Data Only in Specified Columns

It is also possible to only insert data in specific columns.

The following SQL statement will insert a new record, but only insert data in the "CustomerName", "City", and "Country" columns (CustomerID will be updated automatically):

Example

INSERT INTO Customers (CustomerName, City, Country) VALUES ('Cardinal', 'Stavanger', 'Norway');

The SQL UPDATE Statement

The UPDATE statement is used to modify the existing records in a table.

UPDATE Syntax

UPDATE table_name SET column1 = value1, column2 = value2, ...WHERE condition;

Note: Be careful when updating records in a table! Notice the WHERE clause in the UPDATE statement. The WHERE clause specifies which record(s) that should be updated. If you omit the WHERE clause, all records in the table will be updated!

UPDATE Table

The following SQL statement updates the first customer (CustomerID = 1) with a new contact person and a new city.

Example

UPDATE Customers

SET ContactName = 'Alfred Schmidt', City= 'Frankfurt'

WHERE CustomerID = 1;

The selection from the "Customers" table will now look like this:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Alfred Schmidt	Obere Str. 57	Frankfurt	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico
4	Around the Horn	Thomas Hardy	120 Hanover Sq.	London	WA1 1DP	UK

5	Berglunds snabbköp	Christina Berglund	Berguvsvägen 8	Luleå	S-958 22	Sweden

The SQL DELETE Statement

The DELETE statement is used to delete existing records in a table.

DELETE Syntax

DELETE FROM table_name WHERE condition;

SQL DELETE Example

The following SQL statement deletes the customer "Alfreds Futterkiste" from the "Customers" table:

Example

DELETE FROM Customers WHERE CustomerName='Alfreds Futterkiste';

The "Customers" table will now look like this:

CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico
4	Around the Horn	Thomas Hardy	120 Hanover Sq.	London	WA1 1DP	UK

5	Berglunds snabbköp	Christina Berglund	Berguvsvägen 8	Luleå	S-95822	Sweden

Delete All Records

It is possible to delete all rows in a table without deleting the table. This means that the table structure, attributes, and indexes will be intact:

DELETE FROM table_name;

The following SQL statement deletes all rows in the "Customers" table, without deleting the table:

Example

DELETE FROM Customers;

UNIT 3

RELATIONAL DATABASE DESIGN AND NORMALIZATION

ER and EER-to-Relational mapping –Update anomalies –Functional dependencies – Inference rules –Minimal cover –Properties of relational decomposition –Normalization (up to BCNF).

3.1 ER and EER-to-Relational mapping

What are the ER diagrams?

ER diagram is a visual representation of data based on the ER model, and it describes how entities are related to each other in the database.

What are EER diagrams?

EER diagram is a visual representation of data, based on the EER model that is an extension of the original entity-relationship (ER) model.

Entity

An entity is any singular, identifiable and separate object. It refers to individuals, organizations, systems, bits of data or even distinct system components that are considered significant in and of themselves. For example, People, Property, Organizations, Agreements and, etc. In the ER diagram, the entity is represented by a rectangle

Weak Entity

A weak entity is an entity that depends on the existence of another entity. In more technical terms it can be defined as an entity that cannot be identified by its own attributes.

Attributes

Attributes are the properties that define the entity type. For example, Roll_No, Name, DOB, Age, Address, Mobile_No are the attributes that define entity type Student. In the ER diagram, the attribute is represented by an oval.

Multi-valued Attribute

If an attribute can have more than one value it is called a multi-valued attribute.

Derived Attribute

An attribute-based on another attribute.

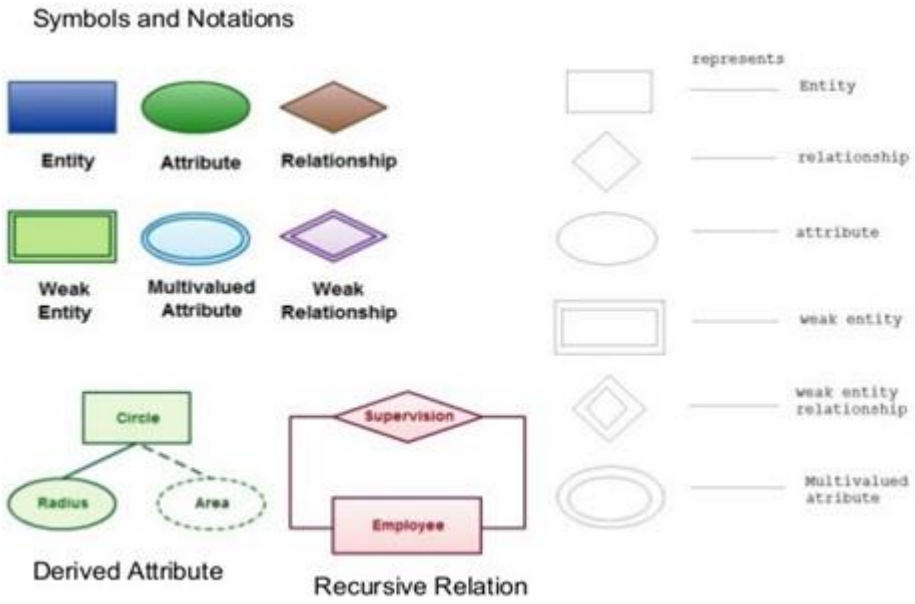
Relationships

A relationship is an association that describes the interaction between entities.

Recursive Relationship

If the same entity participates more than once in a relationship it is known as a recursive relationship.

The following are the types of entities, attributes, and relationships.



Cardinality

Cardinality refers to the maximum number of times an instance in one entity can relate to instances of another entity. There are three types of cardinalities.

one to one (1 to 1)

one to many (1 to N)

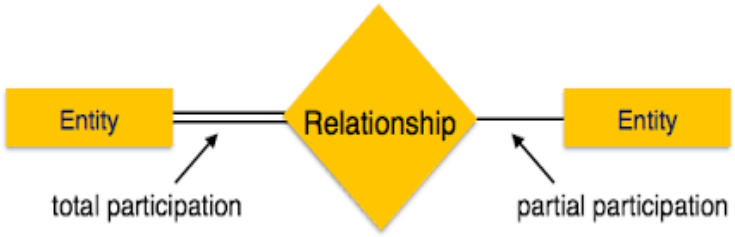
many to many (M to N)

Participation

Participation constraint specifies the existence of an entity when it is related to another entity in a relationship type. There are two types. Partial and Total participation.

Total Participation − Each entity is involved in the relationship. Total participation is represented by double lines.

Partial participation − Not all entities are involved in the relationship. Partial participation is represented by single lines.



There are three steps in database development.

Requirements Analysis- Understand the data problem through user interviews, forms, reports, observation and etc.

Component design stage- Create a data model that is a graphical representation of what will be finally will be implemented.

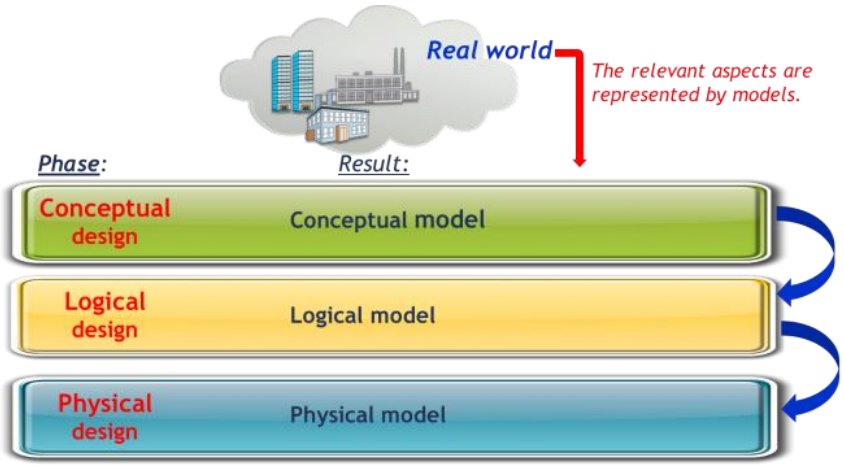
Implementation stage- Actual development of the database which leads to database actually being used in the real environment.

In the above stages after the first stage(Requirements analysis) you have to follow the database design stages.

Conceptual design

Logical design

Physical design



In the **Conceptual design**, we identify all entities, we define attributes and their properties and we define relationships between entities.

In the **Logical design**, we transform the conceptual design into relational, transform entities as tables, transform entity attributes into table column, and normalization.

In the **Physical design**, we specify internal storage structure and paths, assign one or more indexes and tune indexes.

At the conceptual design stage, we design the ER or EER diagrams.

Here are some rules for drawing ER and EER diagrams

- Write all entities in the singular form
- Write all relationships in a singular form.
- Write all attributes in the singular form.
- If you want to separate words use underscore mark.

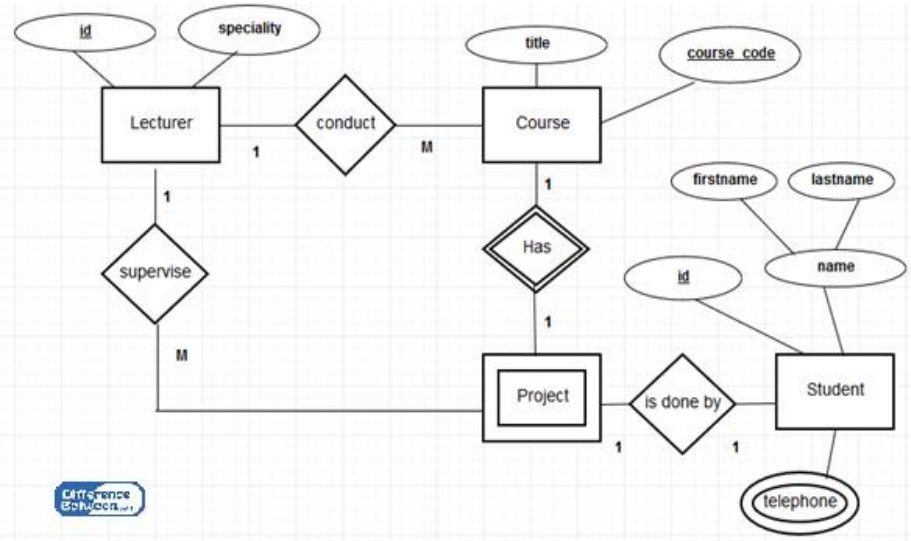
Now, let's see how to draw ER diagrams and EER diagrams.

Drawing ER and EER diagrams

Below points show how to go about creating an ER diagram.

1. Identify all the entities in the system. An entity should appear only once in a particular diagram. Create rectangles for all entities and name them properly.
2. Identify relationships between entities. Connect them using a line and add a diamond in the middle describing the relationship.
3. Add attributes for entities. Give meaningful attribute names so they can be understood easily.
4. Mark the cardinalities and participation between the entities.

example of ER diagrams.



Now let's see how to draw EER diagrams.

Here we just need to add a few things to above.

1. As in drawing ER diagrams first, we have to identify all entities.

After we found entities from the scenario you should check whether those entities have sub-entities. If so you have to mark sub-entities in your diagram.

Dividing entities into sub-entities we called as specialization. And combining sub-entities to one entity is called a **generalization.**

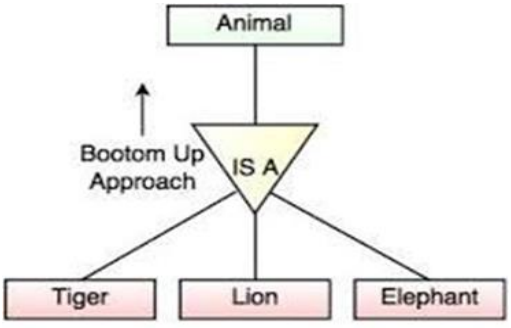


Fig. Generalization

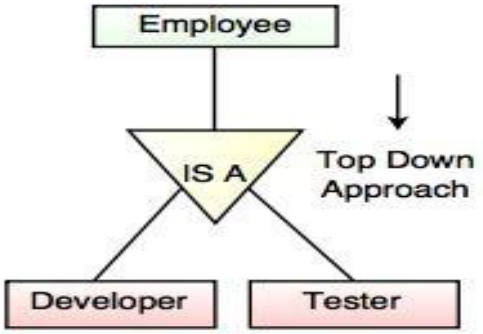


Fig. Specialization

2. Then you have to identify relationships between entities and mark them.

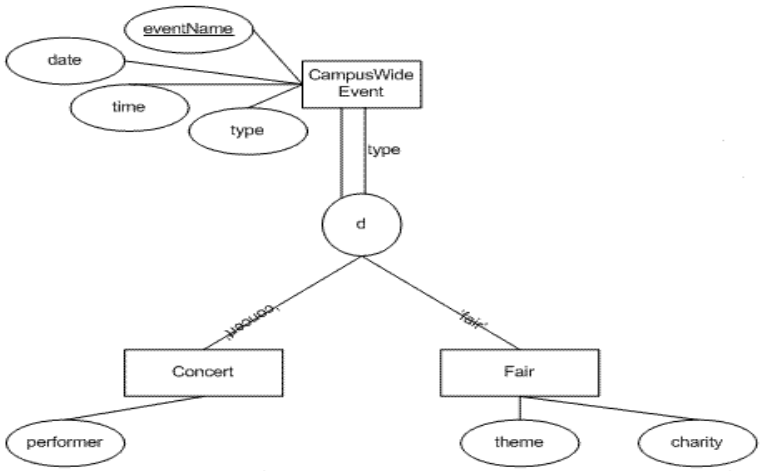
3. Add attributes for entities. Give meaningful attribute names so they can be understood easily.

4. Mark the cardinalities and participation

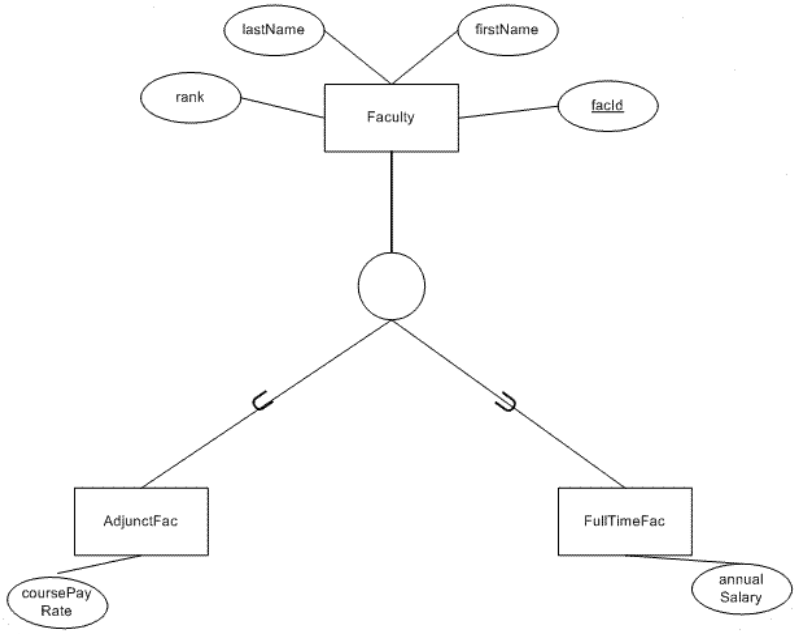
If it is an EER diagram you have to add a few to your diagram.

Here also we have to check whether sub-entities totally depend on the main entity or not. And you should mark it.

If all members in the superclass (main entity) participate in either one subclass it is known as total participation. It marks by double lines.



If at least one member in the superclass does not participate in subclass it is known as partial participation. It is denoted by one single line.



If all the members in the superclass participate for only one subclass it is known as disjoint and denoted by “d”. If all the members in the superclass participate in more than one subclass it is known as overlap and denoted by “o”.

Benefits of ER and EER diagrams.

Easy to understand and does not require a person to undergo extensive recently training to be able to work with it and accurately.

Readily translatable to relational tables which can be used to quickly build databases

Can directly be used by database developers as the blueprint for implementing databases in specific software application

It can be applied in other contexts such as describing the different relationships and operations within an organization.

Now let's move on. Our next topic is map ER and EER diagrams into relational schemas.

Mapping ER and EER diagrams into relational schemas

First I'll tell you how to map the ER diagram into a relational schema.

Mapping ER diagrams into relational schemas

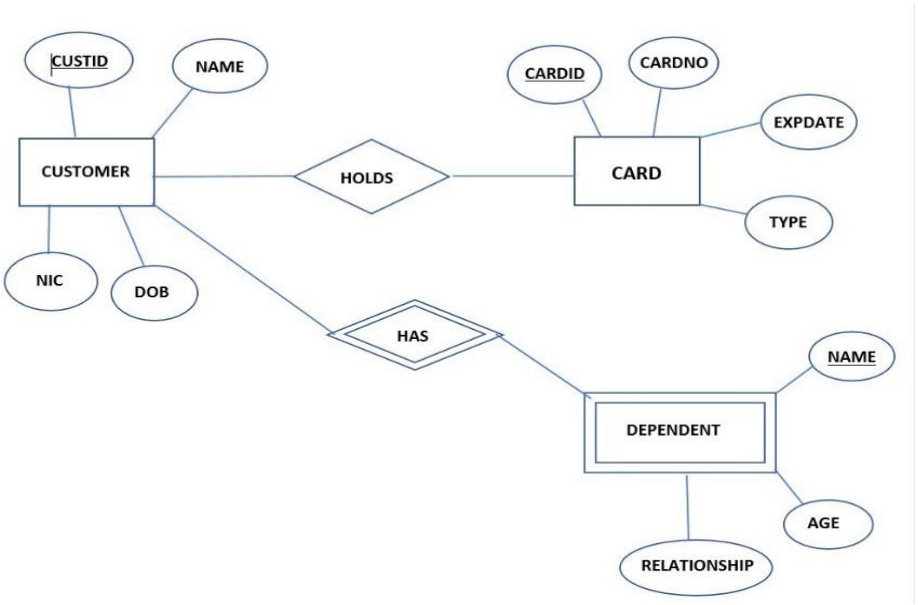
Follow the steps one by one to get it done.

1. Mapping strong entities.
2. Mapping weak entities.
3. Map binary one-to-one relations.
4. Map binary one-to-many relations
5. Map binary many-to-many relations.
6. Map multivalued attributes.
7. Map N-ary relations

Let's go deep with the examples.

1. Mapping strong entities.

2. Mapping weak entities.



Above it shows an ER diagram with its relationships.

You can see there are two strong entities with relationships and a weak entity with a weak relationship.

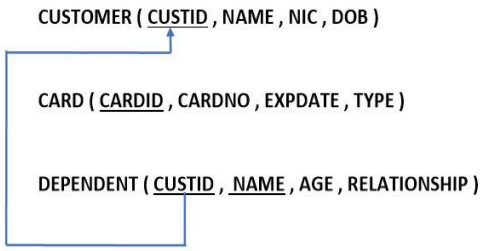
When you going to make a relational schema first you have to identify all entities with their attributes. You have to write attributes in brackets as shown below.

Definitely you have to underline the primary keys. In the above DEPENDENT is a weak entity.

To make it strong go through the weak relationship and identify the entity which connects with this.

Then you have written that entity's primary key inside the weak entity bracket.

Then you have to map the primary key to where you took from as shown below.

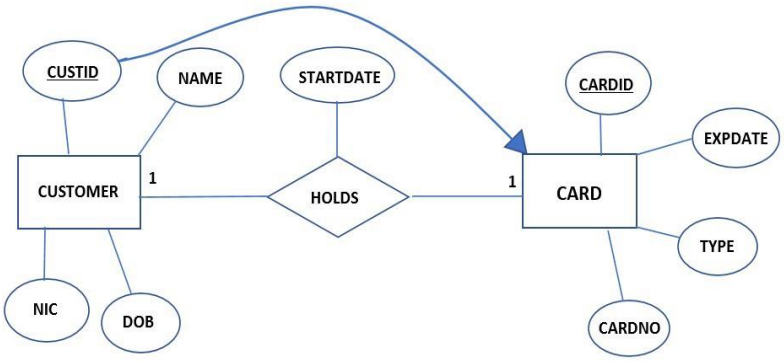


3. Map binary one to one relations.

Let's assume that the relationship between CUSTOMER and CARD is one to one.

There are three occasions where one to one relations take place according to the participation constraints.

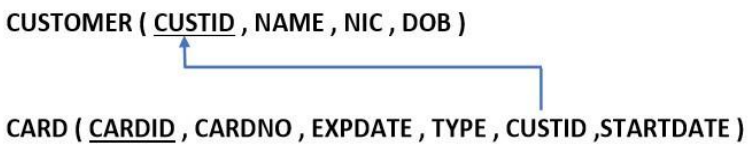
I. Both sides have partial participation.



When both sides have partial participation you can send any of the entity's primary key to others.

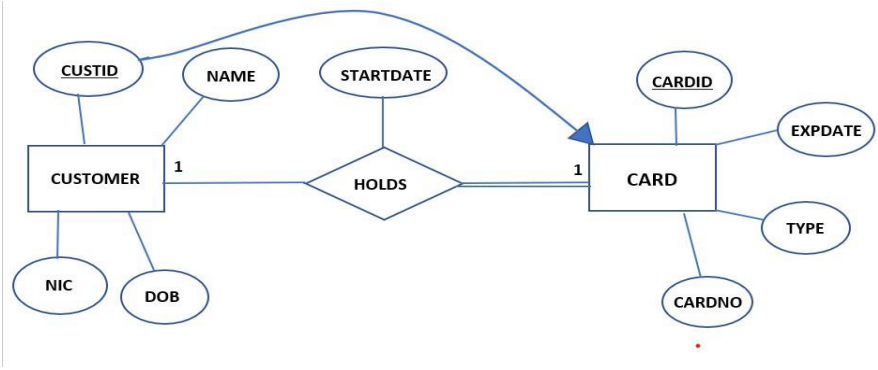
At the same time, if there are attributes in the relationship between those two entities, it is also sent to other entity as shown above.

So, now let us see how we write the relational schema.



Here you can see I have written CUSTID and STARTDATE inside the CARD table. Now you have to map CUSTID from where it comes. That's it.

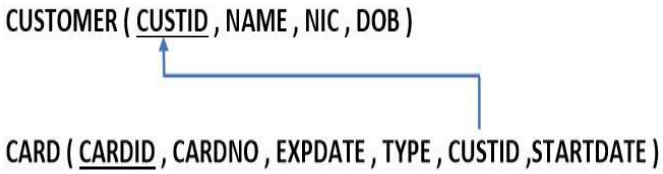
II. One side has partial participation.



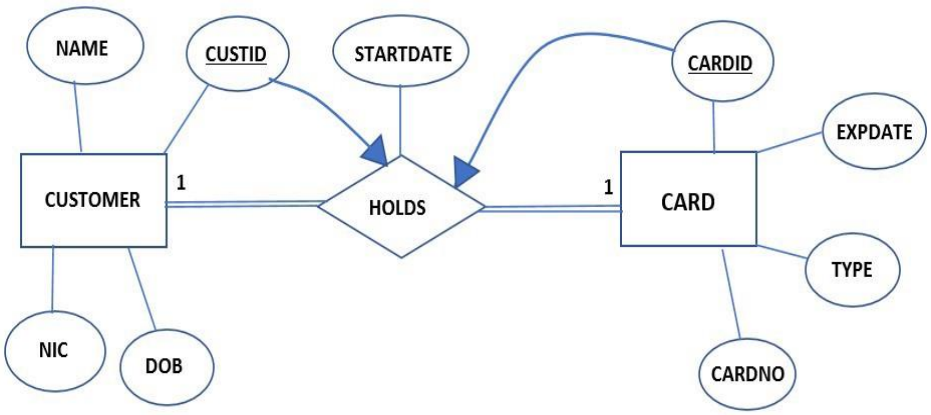
You can see between the relationship and CARD entity it has total participation.

When there is total participation definitely the primary of others comes to this. And also if there are attributes in the relationship it also comes to total participation side.

Then you have to map them as below.



III. Both sides have total participation



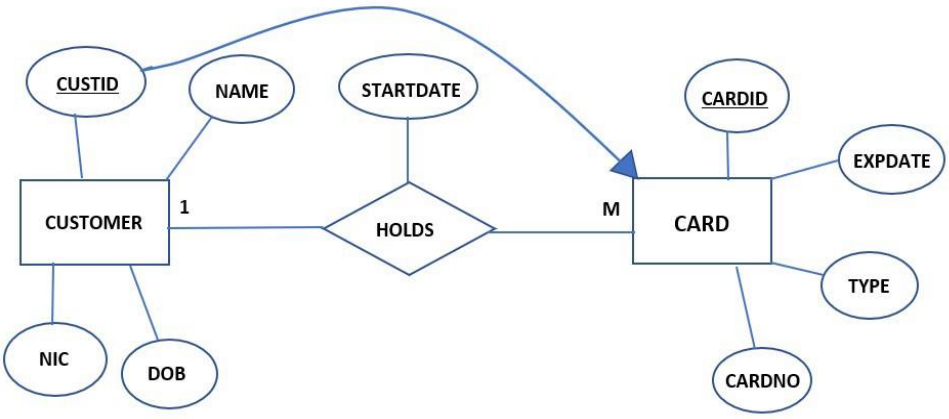
If both sides have total participation you need to make a new relationship with a suitable name and merge entities and the relationship.

Following it shows how we should write the relation.

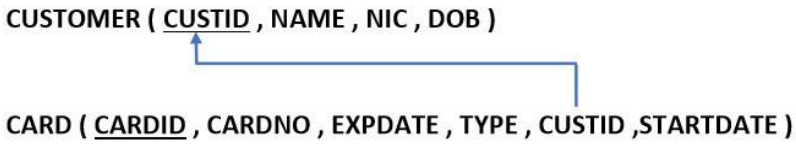
CUST_HOLD(CUSTID , NAME , NIC , DOB , CARDID , CARDNO , EXPDATE , TYPE ,STARTDATE)

Now let us see how to map one to many relations.

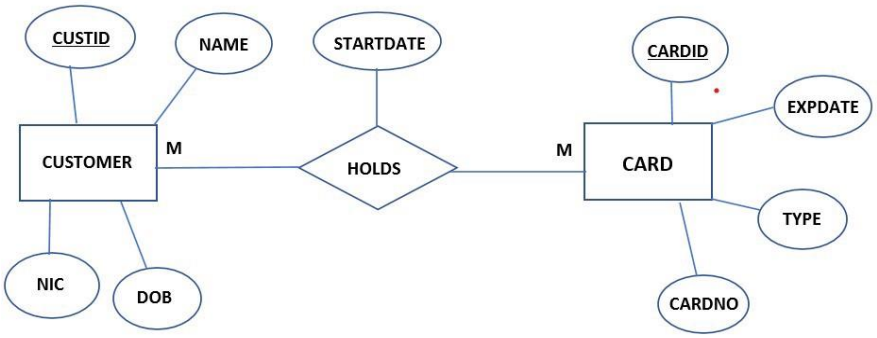
4. Map binary one-to-many relations



If it is one-to-many, always to the many side other entities' primary keys and the attributes in the relationship go to the many side. No matter about participation. And then you have to map the primary key.

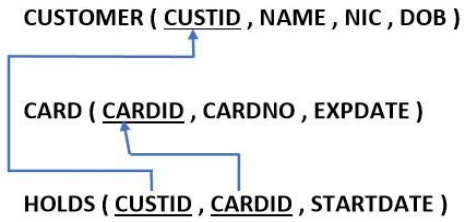


5. Map binary many to many relations.

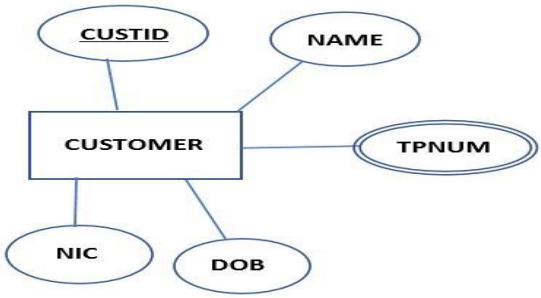


If it is many to many relations, you should always make a new relationship with the name of the relationship between the entities.

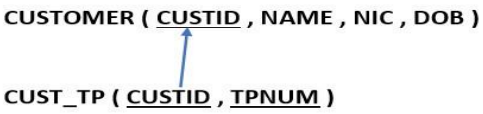
And there you should write both primary keys of the entities and attributes in the relationship and map them to the initials as shown below.



6. Map multivalued attributes.



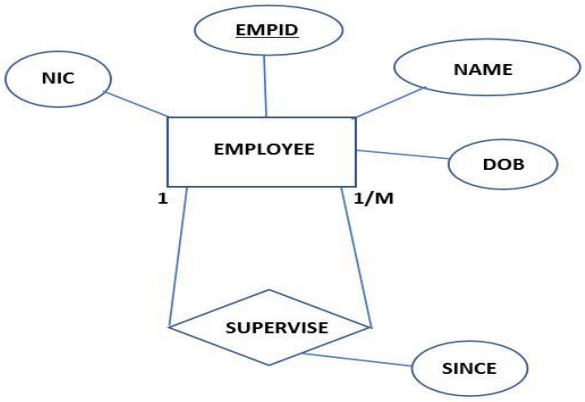
If there are multivalued attributes you have to make a new relationship with a suitable name and write the primary key of the entity which belongs to the multivalued attribute and also the name of the multivalued attribute as shown below.



7. Map N-ary relations.

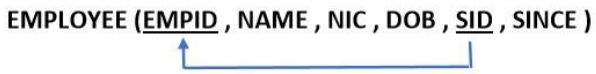
First, let us consider unary relationships. We categorized them into two.

I. one-to-one and one to many relations.

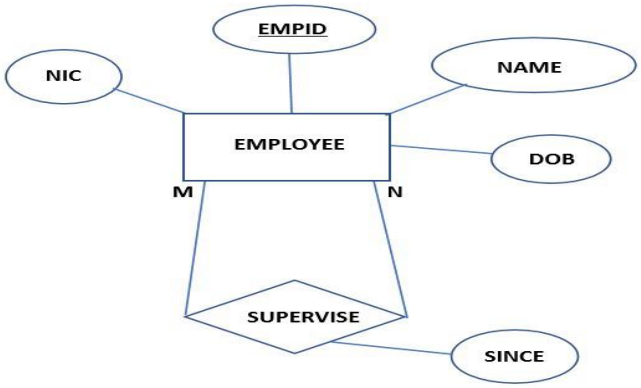


If it is unary and one to one or one to many relations, you do not need to make a new relationship you just want to add a new primary key to the current entity as shown below and map it to the

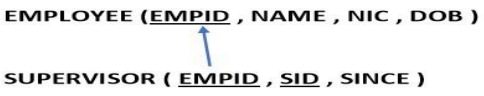
initial. For example, in the above diagram, the employee is supervised by the supervisor. Therefore, we need to make a new primary key as SID and map it to EMPID. Because of SID also an EMPID.



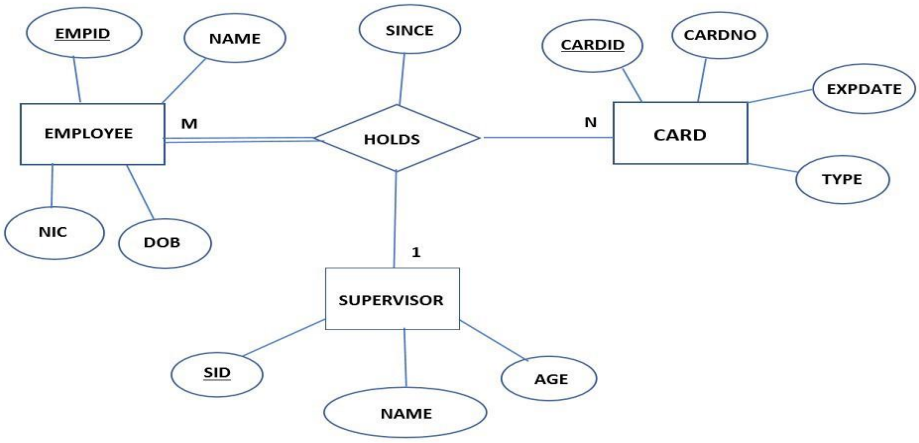
II. many-to-many relations.



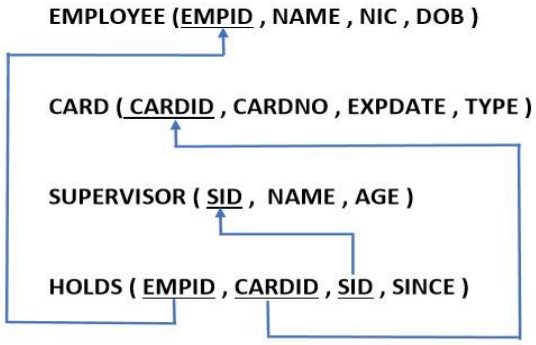
If it is unary and many to many relations, you need to make a new relationship with a suitable name. Then you have to give it a proper primary key and it should map to where it comes as shown below.



Now let us see how to map relations with more than two entities.

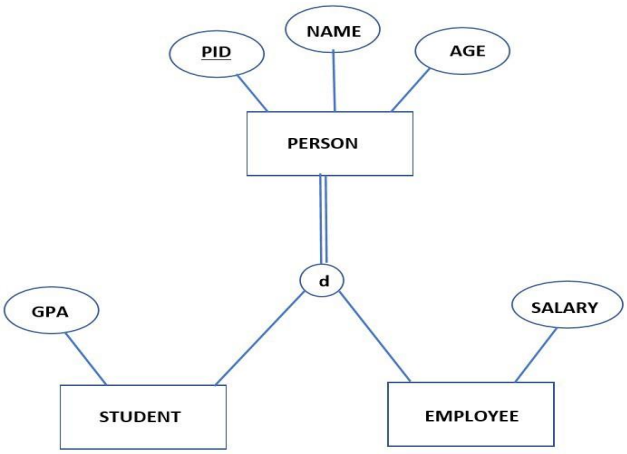


If there are more than three entities for a relationship you have to make a new relation table and put all primary keys of connected entities and all attributes in the relationship. And in the end, you have to map them as shown below.



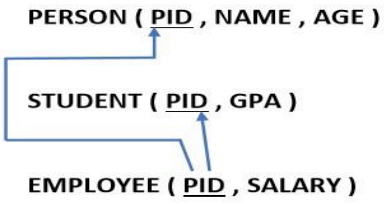
Mapping EER diagrams into relational schemas.

Let us go through the following diagram.



There are four ways to draw relational schema for an EER. You have to choose the most suitable one. In the end, I'll give you the guidelines on how to choose the best and most suitable way.

First method



Here we write separate relations to all superclass entities and subclass entities. And here we have to write the superclass entities' primary key to all subclass entities and then map them as shown above. Note that we write only the attributes belongs to each entity.

Second method

STUDENT (PID , GPA , NAME , AGE)

EMPLOYEE (PID , SALARY , NAME , AGE)

Here we do not write the superclass entity but in each subclass entity, we write all attributes that are in superclass entity.

Third method

PERSON (PID , NAME , AGE , SALARY , GPA , PERSONTYPE)

Here we write only the superclass entity and write all the attributes which belong to subclass entities. Specialty in here is that to identify that PERSON is an EMPLOYEE or STUDENT we add a column as PERSONTYPE. After the table creates we can mark as a STUDENT or EMPLOYEE.

Fourth method

PERSON (PID , NAME , AGE , SALARY , GPA , STUDENT , EMPLOYEE)

Here instead of PERSONTYPE, we write STUDENT and EMPLOYEE both.

The reason for that is sometime PERSON will belong to both categories.

3.2 Anomalies

There are different types of anomalies which can occur in referencing and referenced relation which can be discussed as:

STUDENT

STUD_NO	STUD_NAME	STUD_PHONE	STUD_STATE	STUD_COUNTRY	STUD_AGE
1	RAM	9716271721	Haryana	India	20
2	RAM	9898291281	Punjab	India	19
3	SUJIT	7898291981	Rajsthan	India	18
4	SURESH		Punjab	India	21

Table 1

STUDENT_COURSE

STUD_NO	COURSE_NO	COURSE_NAME
1	C1	DBMS
2	C2	Computer Networks
1	C2	Computer Networks

Table 2

Insertion anomaly: If a tuple is inserted in referencing relation and referencing attribute value is not present in referenced attribute, it will not allow inserting in referencing relation. For Example, if we try to insert a record in STUDENT_COURSE with STUD_NO =7, it will not allow.

Deletion and Updating Anomaly: If a tuple is deleted or updated from referenced relation and referenced attribute value is used by referencing attribute in referencing relation, it will not allow deleting the tuple from referenced relation. For Example, If we try to delete a record from STUDENT with STUD_NO =1, it will not allow. To avoid this, following can be used in query:

ON DELETE/UPDATE SET NULL: If a tuple is deleted or updated from referenced relation and referenced attribute value is used by referencing attribute in referencing relation, it will delete/update the tuple from referenced relation and set the value of referencing attribute to NULL.

ON DELETE/UPDATE CASCADE: If a tuple is deleted or updated from referenced relation and referenced attribute value is used by referencing attribute in referencing relation, it will delete/update the tuple from referenced relation and referencing relation as well.

3.3 Functional Dependency

The functional dependency is a relationship that exists between two attributes. It typically exists between the primary key and non-key attribute within a table.

$X \rightarrow Y$

The left side of FD is known as a determinant; the right side of the production is known as a dependent.

For example:

Assume we have an employee table with attributes: Emp_Id, Emp_Name, Emp_Address.

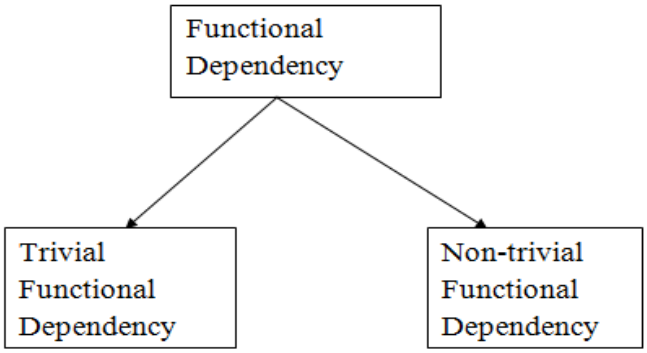
Here Emp_Id attribute can uniquely identify the Emp_Name attribute of employee table because if we know the Emp_Id, we can tell that employee name associated with it.

Functional dependency can be written as:

Emp_Id → Emp_Name

We can say that Emp_Name is functionally dependent on Emp_Id.

Types of Functional dependency



1. Trivial functional dependency

A → B has trivial functional dependency if B is a subset of A.

The following dependencies are also trivial like: A → A, B → B

Example:

Consider a table with two columns Employee_Id and Employee_Name.

{Employee_id, Employee_Name} → Employee_Id is a trivial functional dependency as

Employee_Id is a subset of {Employee_Id, Employee_Name}.

Also, Employee_Id → Employee_Id and Employee_Name → Employee_Name are trivial dependencies too.

2. Non-trivial functional dependency

A → B has a non-trivial functional dependency if B is not a subset of A.

When A Intersection B is NULL, then A → B is called as complete non-trivial.

Example:

ID → Name,

Name → DOB

3.4 Inference Rule (IR):

The Armstrong's axioms are the basic inference rule. Armstrong's axioms are used to conclude functional dependencies on a relational database.

The inference rule is a type of assertion. It can apply to a set of FD(functional dependency) to derive other FD.

Using the inference rule, we can derive additional functional dependency from the initial set.

The Functional dependency has 6 types of inference rule:

1. Reflexive Rule (IR1)

In the reflexive rule, if Y is a subset of X, then X determines Y.

If $X \supseteq Y$, then $X \rightarrow Y$

Example:

X = {a, b, c, d, e}

Y = {a, b, c}

2. Augmentation Rule (IR2)

The augmentation is also called as a partial dependency. In augmentation, if X determines Y, then XZ determines YZ for any Z.

If $X \rightarrow Y$, then $XZ \rightarrow YZ$

Example:

For R(ABCD), if $A \rightarrow B$ then $AC \rightarrow BC$

3. Transitive Rule (IR3)

In the transitive rule, if X determines Y and Y determine Z, then X must also determine Z.

If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$

4. Union Rule (IR4)

Union rule says, if X determines Y and X determines Z, then X must also determine Y and Z.

If $X \rightarrow Y$ and $X \rightarrow Z$, then $X \rightarrow YZ$

Proof:

1. $X \rightarrow Y$ (given)

2. $X \rightarrow Z$ (given)

3. $X \rightarrow XY$ (using IR2 on 1 by augmentation with X. Where XX = X)

4. $XY \rightarrow YZ$ (using IR2 on 2 by augmentation with Y)

5. $X \rightarrow YZ$ (using IR3 on 3 and 4)

5. Decomposition Rule (IR5)

Decomposition rule is also known as project rule. It is the reverse of union rule.

This Rule says, if X determines Y and Z, then X determines Y and X determines Z separately.

If X → YZ then X → Y and X → Z

Proof:

1. X → YZ (given)

2. YZ → Y (using IR1 Rule)

3. X → Y (using IR3 on 1 and 2)

6. Pseudo transitive Rule (IR6)

In Pseudo Transitive Rule, if X determines Y and YZ determines W, then XZ determines W.

If X → Y and YZ → W, then XZ → W

Proof:

1. X → Y (given)

2. WY → Z (given)

3. WX → WY (using IR2 on 1 by augmenting with W)

4. WX → Z (using IR3 on 3 and 2)

3.5 Minimal Cover

A minimal cover of a set of functional dependencies (FD) E is a minimal set of dependencies F that is equivalent to E.

The formal definition is: A set of FD F to be minimal if it satisfies the following conditions −

Every dependency in F has a single attribute for its right-hand side.

We cannot replace any dependency X->A in F with a dependency Y->A, where Y is a proper subset of X, and still have a set of dependencies that is equivalent to F.

We cannot remove any dependency from F and still have a set of dependencies that are equivalent to F.

Canonical cover is called **minimal cover** which is called the minimum set of FDs. A set of FD FC is called canonical cover of F if each FD in FC is a −

Simple FD.

Left reduced FD.

Non-redundant FD.

Simple FD − X->Y is a simple FD if Y is a single attribute.

Left reduced FD − X->Y is a left reduced FD if there are no extraneous attributes in X. {extraneous attributes: Let XA->Y then, A is a extraneous attribute if X_>Y}

Non-redundant FD − X->Y is a Non-redundant FD if it cannot be derived from F- {X->y}.

Example

Consider an example to find canonical cover of F.

The given functional dependencies are as follows −

A -> BC

B -> C

A -> B

AB -> C

Minimal cover: The minimal cover is the set of FDs which are equivalent to the given FDs. Canonical cover: In canonical cover, the LHS (Left Hand Side) must be unique.

First of all, we will find the minimal cover and then the canonical cover.

First step − Convert RHS attribute into singleton attribute.

A -> B

A -> C

B -> C

A -> B

AB -> C

Second step − Remove the extra LHS attribute

Find the closure of A.

A+ = {A, B, C}

So, AB -> C can be converted into A -> C

A -> B

A -> C

B -> C

A -> B

A -> C

Third step − Remove the redundant FDs.

A -> B

B -> C

Now, we will convert the above set of FDs into canonical cover.

The canonical cover for the above set of FDs will be as follows −

A -> BC

B -> C

3.5 Properties of Relational Decomposition

When a relation in the relational model is not appropriate normal form then the decomposition of a relation is required. In a database, breaking down the table into multiple tables termed as decomposition. The properties of a relational decomposition are listed below:

1. **Attribute Preservation:**

Using functional dependencies the algorithms decompose the universal relation schema R in a set of relation schemas D = { R1, R2, ….. Rn } relational database schema, where 'D' is called the Decomposition of R.

The attributes in R will appear in at least one relation schema Ri in the decomposition, i.e., no attribute is lost. This is called the Attribute Preservation condition of decomposition.

2. **Dependency Preservation:**

If each functional dependency X->Y specified in F appears directly in one of the relation schemas Ri in the decomposition D or could be inferred from the dependencies that appear in some Ri. This is the Dependency Preservation.

If a decomposition is not dependency preserving some dependency is lost in decomposition. To check this condition, take the JOIN of 2 or more relations in the decomposition.

For example:

R = (A, B, C)

F = {A ->B, B->C}

Key = {A}

R is not in BCNF.

Decomposition R1 = (A, B), R2 = (B, C)

R1 and R2 are in BCNF, Lossless-join decomposition, Dependency preserving.

Each Functional Dependency specified in F either appears directly in one of the relations in the decomposition.

It is not necessary that all dependencies from the relation R appear in some relation Ri.

It is sufficient that the union of the dependencies on all the relations Ri be equivalent to the dependencies on R.

3. Non Additive Join Property:

Another property of decomposition is that D should possess is the Non Additive Join Property, which ensures that no spurious tuples are generated when a NATURAL JOIN operation is applied to the relations resulting from the decomposition.

4. No redundancy:

Decomposition is used to eliminate some of the problems of bad design like anomalies, inconsistencies, and redundancy. If the relation has no proper decomposition, then it may lead to problems like loss of information.

5. Lossless Join:

Lossless join property is a feature of decomposition supported by normalization. It is the ability to ensure that any instance of the original relation can be identified from corresponding instances in the smaller relations.

For example:

R: relation, F: set of functional dependencies on R,

X, Y: decomposition of R,

A decomposition {R1, R2, …, Rn} of a relation R is called a lossless decomposition for R if the natural join of R1, R2, …, Rn produces exactly the relation R.

A decomposition is lossless if we can recover:

R (A, B, C) -> Decompose -> R1(A, B) R2(A, C) -> Recover -> R’ (A, B, C)

Thus, R’ = R

Decomposition is lossless if:

X intersection Y -> X, that is: all attributes common to both X and Y functionally determine ALL the attributes in X.

X intersection Y -> Y, that is: all attributes common to both X and Y functionally determine ALL the attributes in Y

If X Intersection Y forms a superkey of either X or Y, the decomposition of R is a lossless decomposition.

3.6 Normalization

Normalization is the process of organizing the data in the database.

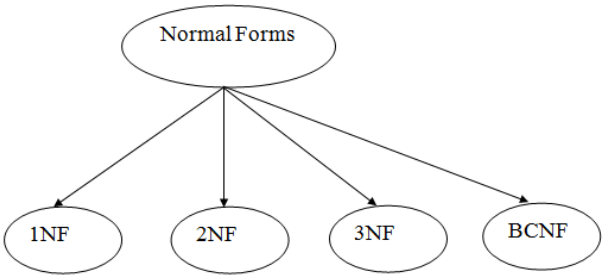
Normalization is used to minimize the redundancy from a relation or set of relations. It is also used to eliminate the undesirable characteristics like Insertion, Update and Deletion Anomalies.

Normalization divides the larger table into the smaller table and links them using relationship.

The normal form is used to reduce redundancy from the database table.

Types of Normal Forms

There are the four types of normal forms:



Normal Form- Description

1NF- A relation is in 1NF if it contains an atomic value.

2NF- A relation will be in 2NF if it is in 1NF and all non-key attributes are fully functional dependent on the primary key.

3NF- A relation will be in 3NF if it is in 2NF and no transition dependency exists.

4NF- A relation will be in 4NF if it is in Boyce Codd normal form and has no multi-valued dependency.

5NF- A relation is in 5NF if it is in 4NF and not contains any join dependency and joining should be lossless.

First Normal Form (1NF)

A relation will be 1NF if it contains an atomic value.

It states that an attribute of a table cannot hold multiple values. It must hold only single-valued attribute.

First normal form disallows the multi-valued attribute, composite attribute, and their combinations.

Example: Relation EMPLOYEE is not in 1NF because of multi-valued attribute EMP_PHONE.

EMPLOYEE table:

EMP_ID	EMP_NAME	EMP_PHONE	EMP_STATE
14	John	7272826385, 9064738238	UP
20	Harry	8574783832	Bihar
12	Sam	7390372389, 8589830302	Punjab

The decomposition of the EMPLOYEE table into 1NF has been shown below:

EMP_ID	EMP_NAME	EMP_PHONE	EMP_STATE
14	John	7272826385,	UP
14	John	9064738238	UP
20	Harry	8574783832	Bihar
12	Sam	7390372389,	Punjab
12	Sam	8589830302	Punjab

Second Normal Form (2NF)

In the 2NF, relational must be in 1NF.

In the second normal form, all non-key attributes are fully functional dependent on the primary key

Example: Let's assume, a school can store the data of teachers and the subjects they teach. In a school, a teacher can teach more than one subject.

TEACHER table

TEACHER_ID	SUBJECT	TEACHER_AGE
25	Chemistry	30
25	Biology	30
47	English	35
83	Maths	38
83	Computer	38

In the given table, non-prime attribute TEACHER_AGE is dependent on TEACHER_ID which is a proper subset of a candidate key. That's why it violates the rule for 2NF.

To convert the given table into 2NF, we decompose it into two tables:

TEACHER_DETAIL table:

TEACHER_ID	TEACHER_AGE
25	30
47	35
83	38

TEACHER_SUBJECT table:

TEACHER_ID	SUBJECT
25	Chemistry
25	Biology
47	English
83	Maths
83	Computer

Third Normal Form (3NF)

A relation will be in 3NF if it is in 2NF and not contain any transitive partial dependency.

3NF is used to reduce the data duplication. It is also used to achieve the data integrity.

If there is no transitive dependency for non-prime attributes, then the relation must be in third normal form.

A relation is in third normal form if it holds atleast one of the following conditions for every non-trivial function dependency $X \rightarrow Y$.

X is a super key.

Y is a prime attribute, i.e., each element of Y is part of some candidate key.

Example:

EMPLOYEE_DETAIL table:

EMP_ID	EMP_NAME	EMP_ZIP	EMP_STATE	EMP_CITY
222	Harry	201010	UP	Noida
333	Stephan	02228	US	Boston

444	Lan	60007	US	Chicago
555	Katharine	06389	UK	Norwich
666	John	462007	MP	Bhopal

Super key in the table above:

{EMP_ID}, {EMP_ID, EMP_NAME}, {EMP_ID, EMP_NAME, EMP_ZIP} ....so on

Candidate key: {EMP_ID}

Non-prime attributes: In the given table, all attributes except EMP_ID are non-prime.

Here, EMP_STATE & EMP_CITY dependent on EMP_ZIP and EMP_ZIP dependent on EMP_ID. The non-prime attributes (EMP_STATE, EMP_CITY) transitively dependent on super key(EMP_ID). It violates the rule of third normal form.

That's why we need to move the EMP_CITY and EMP_STATE to the new <EMPLOYEE_ZIP> table, with EMP_ZIP as a Primary key.

EMPLOYEE table:

EMP_ID	EMP_NAME	EMP_ZIP
222	Harry	201010
333	Stephan	02228
444	Lan	60007
555	Katharine	06389
666	John	462007

EMPLOYEE_ZIP table:

EMP_ZIP	EMP_STATE	EMP_CITY
201010	UP	Noida
02228	US	Boston
60007	US	Chicago
06389	UK	Norwich
462007	MP	Bhopal

Boyce Codd Normal Form (BCNF)

BCNF is the advance version of 3NF. It is stricter than 3NF.

A table is in BCNF if every functional dependency X → Y, X is the super key of the table.

For BCNF, the table should be in 3NF, and for every FD, LHS is super key.

Example: Let's assume there is a company where employees work in more than one department.

EMPLOYEE table:

EMP_ID	EMP_COUNTRY	EMP_DEPT	DEPT_TYPE	EMP_DEPT_NO
264	India	Designing	D394	283
264	India	Testing	D394	300
364	UK	Stores	D283	232
364	UK	Developing	D283	549

In the above table Functional dependencies are as follows:

EMP_ID → EMP_COUNTRY

EMP_DEPT → {DEPT_TYPE, EMP_DEPT_NO}

Candidate key: {EMP-ID, EMP-DEPT}

The table is not in BCNF because neither EMP_DEPT nor EMP_ID alone are keys.

To convert the given table into BCNF, we decompose it into three tables:

EMP_COUNTRY table:

EMP_ID	EMP_COUNTRY
264	India
264	India

EMP_DEPT table:

EMP_DEPT	DEPT_TYPE	EMP_DEPT_NO
Designing	D394	283
Testing	D394	300
Stores	D283	232
Developing	D283	549

EMP_DEPT_MAPPING table:

EMP_ID	EMP_DEPT
D394	283
D394	300
D283	232
D283	549

Functional dependencies:

EMP_ID → EMP_COUNTRY

EMP_DEPT → {DEPT_TYPE, EMP_DEPT_NO}

Candidate keys:

For the first table: EMP_ID

For the second table: EMP_DEPT

For the third table: {EMP_ID, EMP_DEPT}

Now, this is in BCNF because left side part of both the functional dependencies is a key.

UNIT IV TRANSACTION MANAGEMENT

TRANSACTION CONCEPTS AND CONCURRENCY CONTROL IN DBMS

Transaction concepts –properties –Schedules –Serializability –Concurrency Control –Two-phase locking techniques.

4.1 Transaction concepts

Concurrency Control deals with interleaved execution of more than one transaction. In the next article, we will see what is serializability and how to find whether a schedule is serializable or not.

What is Transaction?
A set of logically related operations is known as transaction. The main operations of a transaction are:

Read(A): Read operations Read(A) or R(A) reads the value of A from the database and stores it in a buffer in main memory.

Write (A): Write operation Write(A) or W(A) writes the value back to the database from buffer.

(Note: It doesn't always need to write it to database back it just writes the changes to buffer this is the reason where dirty read comes into picture)

Let us take a debit transaction from an account which consists of following operations:

R(A);

A=A-1000;

W(A);

Assume A's value before starting of transaction is 5000.

The first operation reads the value of A from database and stores it in a buffer.

Second operation will decrease its value by 1000. So, buffer will contain 4000.

Third operation will write the value from buffer to database. So, A's final value will be 4000.

But it may also be possible that transaction may fail after executing some of its operations. The failure can be because of hardware, software or power etc. For example, if debit transaction discussed above fails after executing operation 2, the value of A will remain 5000 in the database which is not acceptable by the bank. To avoid this, Database has two important operations:

Commit: After all instructions of a transaction are successfully executed, the changes made by transaction are made permanent in the database.

Rollback: If a transaction is not able to execute all operations successfully, all the changes made by transaction are undone.

Properties of a transaction

Atomicity:

As a transaction is set of logically related operations, either all of them should be executed or none. A debit transaction discussed above should either execute all three operations or none. If debit transaction fails after executing operation 1 and 2 then its new value 4000 will not be updated in the database which leads to inconsistency.

Consistency:

If operations of debit and credit transactions on same account are executed concurrently, it may leave database in an inconsistent state.

For Example, T1 (debit of Rs. 1000 from A) and T2 (credit of 500 to A) executing concurrently, the database reaches inconsistent state.

Let us assume Account balance of A is Rs. 5000. T1 reads A (5000) and stores the value in its local buffer space. Then T2 reads A (5000) and also stores the value in its local buffer space.

T1 performs A=A-1000 (5000-1000=4000) and 4000 is stored in T1 buffer space. Then T2 performs A=A+500 (5000+500=5500) and 5500 is stored in T2 buffer space. T1 writes the value from its buffer back to database.

A's value is updated to 4000 in database and then T2 writes the value from its buffer back to database. A's value is updated to 5500 which shows that the effect of debit transaction is lost and database has become inconsistent.

To maintain consistency of database, we need concurrency control protocols which will be discussed in next article. The operations of T1 and T2 with their buffers and database have been shown in Table 1.

T1	T1's buffer space	T2	T2's buffer space	Database
				A=5000
R(A);	A=5000			A=5000
	A=5000	R(A);		A=5000
A=A-1000;	A=4000		A=5000	A=5000
	A=4000	A=A+500	A=5500	
W(A);			A=5500	A=4000

		W(A);		A=5500

TABLE 1

Isolation:

Result of a transaction should not be visible to others before transaction is committed. For example, let us assume that A's balance is Rs. 5000 and T1 debits Rs. 1000 from A. A's new balance will be 4000. If T2 credits Rs. 500 to A's new balance, a will become 4500 and after this T1 fails. Then we have to rollback T2 as well because it is using value produced by T1. So a transaction results are not made visible to other transactions before it commits.

Durable:

Once database has committed a transaction, the changes made by the transaction should be permanent. e.g.; If a person has credited $500000 to his account, bank can't say that the update has been lost. To avoid this problem, multiple copies of database are stored at different locations.

What is a Schedule in transaction?

A schedule is a series of operations from one or more transactions.

A schedule can be of two types:

Serial Schedule: When one transaction completely executes before starting another transaction, the schedule is called serial schedule. A serial schedule is always consistent. e.g.; If a schedule S has debit transaction T1 and credit transaction T2, possible serial schedules are T1 followed by T2 (T1->T2) or T2 followed by T1 ((T2->T1). A serial schedule has low throughput and less resource utilization.

Concurrent Schedule: When operations of a transaction are interleaved with operations of other transactions of a schedule, the schedule is called Concurrent schedule. e.g.; Schedule of debit and credit transaction shown in Table 1 is concurrent in nature. But concurrency can lead to inconsistency in the database. The above example of a concurrent schedule is also inconsistent.

Question: Consider the following transaction involving two bank accounts x and y:

```
read(x);

x := x – 50;

write(x);

read(y);

y := y + 50;
```

write(y);

The constraint that the sum of the accounts x and y should remain constant is that of?

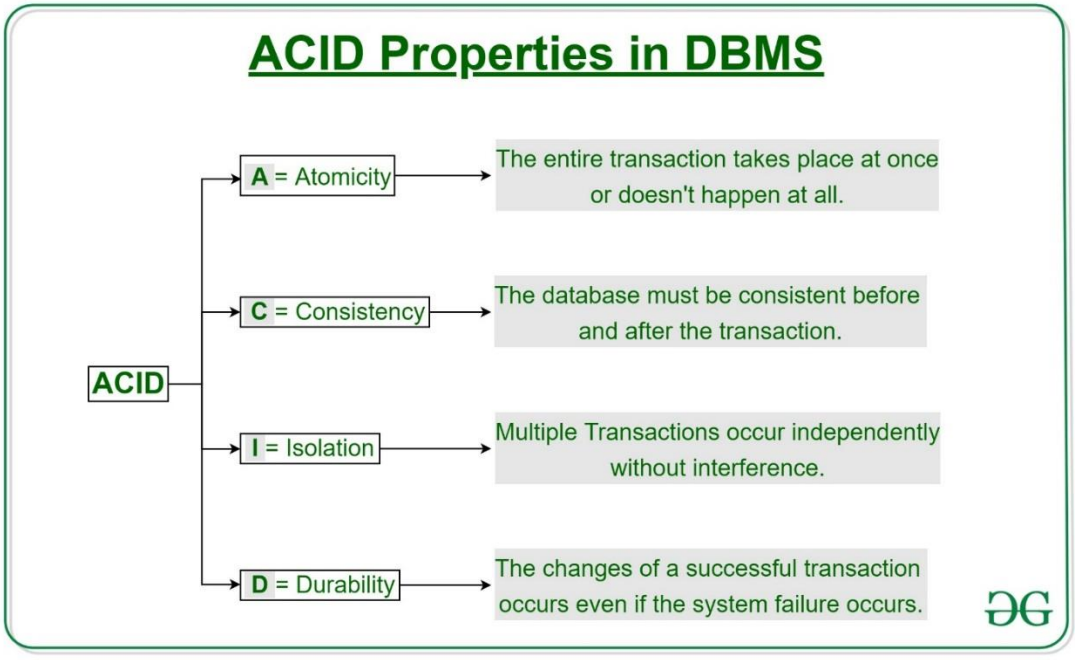
- Atomicity
- Consistency
- Isolation
- Durability

Solution: As discussed in properties of transactions, consistency properties says that sum of accounts x and y should remain constant before starting and after completion of transaction. So, the correct answer is B.

4.2 ACID Properties in DBMS

A transaction is a single logical unit of work which accesses and possibly modifies the contents of a database. Transactions access data using read and write operations.

In order to maintain consistency in a database, before and after the transaction, certain properties are followed. These are called ACID properties.



Atomicity

By this, we mean that either the entire transaction takes place at once or doesn't happen at all. There is no midway i.e. transactions do not occur partially. Each transaction is considered as one unit and either runs to completion or is not executed at all. It involves the following two operations.

—Abort: If a transaction aborts, changes made to database are not visible.

—Commit: If a transaction commits, changes made are visible.

Atomicity is also known as the ‘All or nothing rule’.

Consider the following transaction T consisting of T1 and T2: Transfer of 100 from account X to account Y.

Before: X : 500	Y: 200
Transaction T	
T1	**T2**
Read (X) X: = X − 100 Write (X)	Read (Y) Y: = Y + 100 Write (Y)
After: X : 400	Y : 300

If the transaction fails after completion of T1 but before completion of T2. (say, after write(X) but before write(Y)), then amount has been deducted from X but not added to Y. This results in an inconsistent database state. Therefore, the transaction must be executed in entirety in order to ensure correctness of database state.

Consistency

This means that integrity constraints must be maintained so that the database is consistent before and after the transaction. It refers to the correctness of a database. Referring to the example above,

The total amount before and after the transaction must be maintained.

Total before T occurs = 500 + 200 = 700.

Total after T occurs = 400 + 300 = 700.

Therefore, database is consistent. Inconsistency occurs in case T1 completes but T2 fails. As a result, T is incomplete.

Isolation

This property ensures that multiple transactions can occur concurrently without leading to the inconsistency of database state. Transactions occur independently without interference. Changes occurring in a particular transaction will not be visible to any other transaction until that particular change in that transaction is written to memory or has been committed. This property ensures that the execution of transactions concurrently will result in a state that is equivalent to a state achieved these were executed serially in some order.

Let X= 500, Y = 500.

Consider two transactions T and T".

Suppose T has been executed till Read (Y) and then T'' starts. As a result, interleaving of operations takes place due to which T'' reads correct value of X but incorrect value of Y and sum computed by

T'': (X+Y = 50, 000+500=50, 500)

is thus not consistent with the sum at end of transaction:

T: (X+Y = 50, 000 + 450 = 50, 450).

This results in database inconsistency, due to a loss of 50 units. Hence, transactions must take place in isolation and changes should be visible only after they have been made to the main memory.

T	T''
Read (X) X: = X*100 Write (X) Read (Y) Y: = Y – 50 Write(Y)	Read (X) Read (Y) Z: = X + Y Write (Z)

Durability:

This property ensures that once the transaction has completed execution, the updates and modifications to the database are stored in and written to disk and they persist even if a system failure occurs. These updates now become permanent and are stored in non-volatile memory. The effects of the transaction, thus, are never lost.

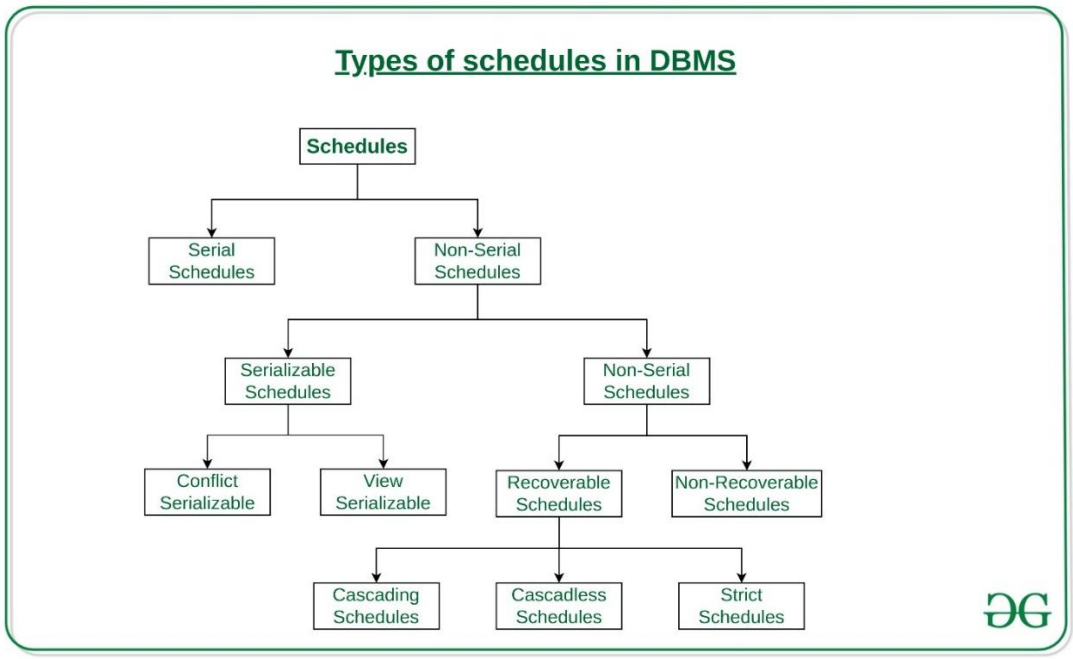
The **ACID** properties, in totality, provide a mechanism to ensure correctness and consistency of a database in a way such that each transaction is a group of operations that acts a single unit, produces consistent results, acts in isolation from other operations and updates that it makes are durably stored.

4.3 Schedules

Schedule, is a process of lining the transactions and executing them one by one.

When there are multiple transactions that are running in a concurrent manner and the order of operation is needed to be set so that the operations do not overlap each other, Scheduling is brought into play and the transactions are timed accordingly.

The basics of Transactions and Schedules is discussed in Concurrency Control (Introduction), and Transaction Isolation Levels in DBMS articles. Here we will discuss various types of schedules.



Serial Schedules:

Schedules in which the transactions are executed non-interleaved, i.e., a serial schedule is one in which no transaction starts until a running transaction has ended are called serial schedules.

Example: Consider the following schedule involving two transactions T1 and T2.

T1	T2
R(A)	
W(A)	
R(B)	
	W(B)
	R(A)
	W(B)

where R(A) denotes that a read operation is performed on some data item ‘A’.This is a serial schedule since the transactions perform serially in the order T1 —> T2

Non-Serial Schedule:

This is a type of Scheduling where the operations of multiple transactions are interleaved. This might lead to a rise in the concurrency problem. The transactions are executed in a non-serial manner, keeping the end result correct and same as the serial schedule. Unlike the serial schedule where one transaction must wait for another to complete all its operation, in the non-serial schedule, the other transaction proceeds without waiting for the previous transaction to complete. This sort of schedule does not provide any benefit of the concurrent transaction. It can be of two types namely, Serializable and Non-Serializable Schedule.

The Non-Serial Schedule can be divided further into Serializable and Non-Serializable.

4.4 Serializable:

This is used to maintain the consistency of the database. It is mainly used in the Non-Serial scheduling to verify whether the scheduling will lead to any inconsistency or not. On the other hand, a serial schedule does not need the serializability because it follows a transaction only when the previous transaction is complete. The non-serial schedule is said to be in a serializable schedule only when it is equivalent to the serial schedules, for an n number of transactions. Since concurrency is allowed in this case thus, multiple transactions can execute concurrently. A serializable schedule helps in improving both resource utilization and CPU throughput. These are of two types:

1. **Conflict Serializable:**

A schedule is called conflict serializable if it can be transformed into a serial schedule by swapping non-conflicting operations. Two operations are said to be conflicting if all conditions satisfy:

- They belong to different transactions
- They operate on the same data item
- At Least one of them is a write operation

2. **View Serializable:**

A Schedule is called view serializable if it is view equal to a serial schedule (no overlapping transactions). A conflict schedule is a view serializable but if the serializability contains blind writes, then the view serializable does not conflict serializable.

Non-Serializable:

The non-serializable schedule is divided into two types, Recoverable and Non-recoverable Schedule.

Recoverable Schedule:

Schedules in which transactions commit only after all transactions whose changes they read commit are called recoverable schedules. In other words, if some transaction Tj is reading value

updated or written by some other transaction Ti, then the commit of Tj must occur after the commit of Ti.

Example – Consider the following schedule involving two transactions T1 and T2.

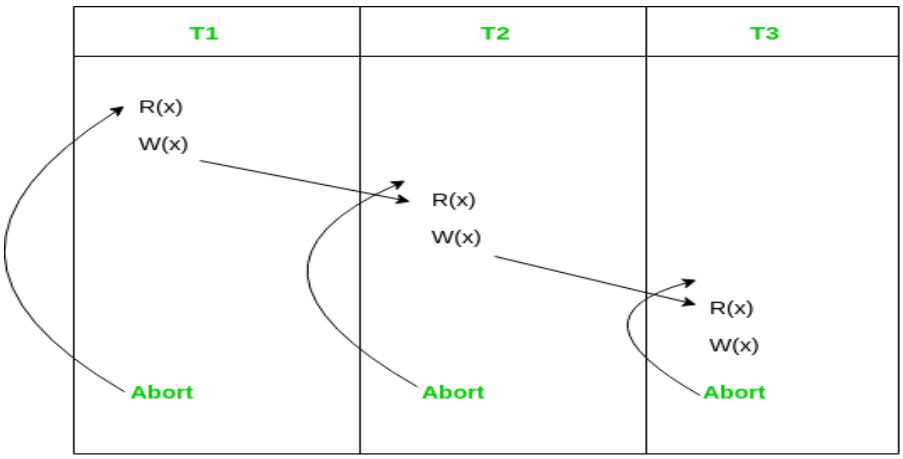
T1	T2
R(A)	
W(A)	
	W(A)
	R(A)
Commit	
	Commit

This is a recoverable schedule since T1 commits before T2, that makes the value read by T2 correct.

There can be three types of recoverable schedule:

Cascading Schedule:

Also called Avoids cascading aborts/rollbacks (ACA). When there is a failure in one transaction and this leads to the rolling back or aborting other dependent transactions, then such scheduling is referred to as Cascading rollback or cascading abort. Example:



Cascadeless Schedule:

Schedules in which transactions read values only after all transactions whose changes they are going to read commit are called cascadeless schedules. Avoids that a single transaction abort

leads to a series of transaction rollbacks. A strategy to prevent cascading aborts is to disallow a transaction from reading uncommitted changes from another transaction in the same schedule.

In other words, if some transaction Tj wants to read value updated or written by some other transaction Ti, then the commit of Tj must read it after the commit of Ti.

Example: Consider the following schedule involving two transactions T1 and T2.

T1	T2
R(A)	
W(A)	
	W(A)
Commit	
	R(A)
	Commit

This schedule is cascadeless. Since the updated value of A is read by T2 only after the updating transaction i.e. T1 commits.

Example: Consider the following schedule involving two transactions T1 and T2.

T_1	T_2
R(A)	
W(A)	
	R(A)
	W(A)
abort	
	abort

It is a recoverable schedule but it does not avoid cascading aborts. It can be seen that if T1 aborts, T2 will have to be aborted too in order to maintain the correctness of the schedule as T2 has already read the uncommitted value written by T1.

Strict Schedule:

A schedule is strict if for any two transactions Ti, Tj, if a write operation of Ti precedes a conflicting operation of Tj (either read or write), then the commit or abort event of Ti also precedes that conflicting operation of Tj.

In other words, Tj can read or write updated or written value of Ti only after Ti commits/aborts.

Example: Consider the following schedule involving two transactions T1 and T2.

T1	T2
R(A)	
	R(A)
W(A)	
commit	
	W(A)
	R(A)
	commit

This is a strict schedule since T2 reads and writes A which is written by T1 only after the commit of T1.

Non-Recoverable Schedule:

Example: Consider the following schedule involving two transactions T1 and T2.

T1	T2
R(A)	
W(A)	
	W(A)
	R(A)
	Commit
Abort	

T2 read the value of A written by T1, and committed. T1 later aborted, therefore the value read by T2 is wrong, but since T2 committed, this schedule is non-recoverable.

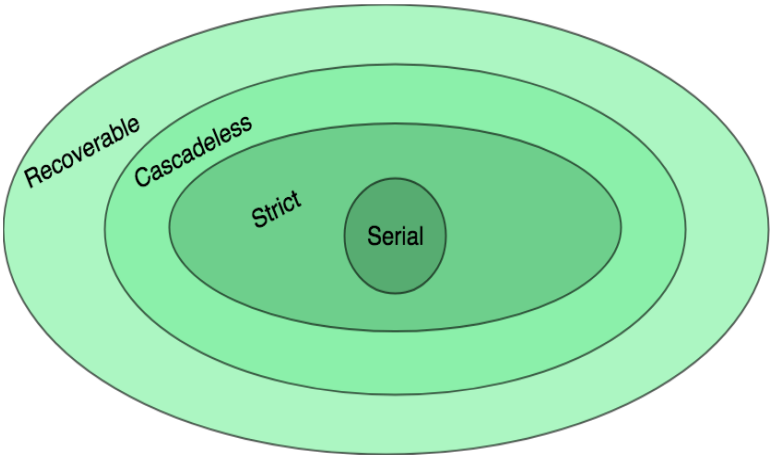
Note – It can be seen that:

Cascadeless schedules are stricter than recoverable schedules or are a subset of recoverable schedules.

Strict schedules are stricter than cascadeless schedules or are a subset of cascadeless schedules.

Serial schedules satisfy constraints of all recoverable, cascadeless and strict schedules and hence is a subset of strict schedules.

The relation between various types of schedules can be depicted as:



Example: Consider the following schedule:

S: R1(A), W2(A), Commit2, W1(A), W3(A), Commit3, Commit1

Which of the following is true?

(A) The schedule is view serializable schedule and strict recoverable schedule

(B) The schedule is non-serializable schedule and strict recoverable schedule

(C) The schedule is non-serializable schedule and is not strict recoverable schedule.

(D) The Schedule is serializable schedule and is not strict recoverable schedule

Solution: The schedule can be re-written as: -

T1	T2	T3
R(A)		
	W(A)	
	Commit	
W(A)		
		W(A)
		Commit
Commit		

First of all, it is a view serializable schedule as it has view equal serial schedule T1 —> T2 —> T3 which satisfies the initial and updated reads and final write on variable A which is required for view serializability. Now we can see there is write – write pair done by transactions T1 followed by T3 which is violating the above-mentioned condition of strict schedules as T3 is supposed to do write operation only after T1 commits which is violated in the given schedule. Hence the given schedule is serializable but not strict recoverable.

So, option (D) is correct.

4. 5 Concurrency control

Concurrency control concept comes under the Transaction in database management system (DBMS).

It is a procedure in DBMS which helps us for the management of two simultaneous processes to execute without conflicts between each other, these conflicts occur in multi user systems.

Concurrency Control is the working concept that is required for controlling and managing the concurrent execution of database operations and thus avoiding the inconsistencies in the database. Thus, for maintaining the concurrency of the database, we have the concurrency control protocols.

Concurrency can simply be said to be executing multiple transactions at a time. It is required to increase time efficiency.

If many transactions try to access the same data, then inconsistency arises. Concurrency control required to maintain consistency data.

For example, if we take ATM machines and do not use concurrency, multiple persons cannot draw money at a time in different places. This is where we need concurrency.

Advantages

The advantages of concurrency control are as follows −

- Waiting time will be decreased.
- Response time will decrease.
- Resource utilization will increase.
- System performance & Efficiency is increased.

Concurrent Execution in DBMS

- In a multi-user system, multiple users can access and use the same database at one time, which is known as the concurrent execution of the database. It means that the same database is executed simultaneously on a multi-user system by different users.

- While working on the database transactions, there occurs the requirement of using the database by multiple users for performing different operations, and in that case, concurrent execution of the database is performed.
- The thing is that the simultaneous execution that is performed should be done in an interleaved manner, and no operation should affect the other executing operations, thus maintaining the consistency of the database. Thus, on making the concurrent execution of the transaction operations, there occur several challenging problems that need to be solved.

Problems with Concurrent Execution

In a database transaction, the two main operations are READ and WRITE operations. So, there is a need to manage these two operations in the concurrent execution of the transactions as if these operations are not performed in an interleaved manner, and the data may become inconsistent. So, the following problems occur with the Concurrent Execution of the operations:

Problem 1: Lost Update Problems (W - W Conflict)

The problem occurs when two different database transactions perform the read/write operations on the same database items in an interleaved manner (i.e., concurrent execution) that makes the values of the items incorrect hence making the database inconsistent.

For example:

Consider the below diagram where two transactions TX and TY, are performed on the same account A where the balance of account A is $300.

Time	T_x	T_y
t_1	READ (A)	—
t_2	A = A - 50	
t_3	—	READ (A)
t_4	—	A = A + 100
t_5	—	—
t_6	WRITE (A)	—
t_7		WRITE (A)

LOST UPDATE PROBLEM

- At time t1, transaction TX reads the value of account A, i.e., $300 (only read).
- At time t2, transaction TX deducts $50 from account A that becomes $250 (only deducted and not updated/write).
- Alternately, at time t3, transaction TY reads the value of account A that will be $300 only because TX didn't update the value yet.

- At time t4, transaction TY adds $100 to account A that becomes $400 (only added but not updated/write).
- At time t6, transaction TX writes the value of account A that will be updated as $250 only, as TY didn't update the value yet.
- Similarly, at time t7, transaction TY writes the values of account A, so it will write as done at time t4 that will be $400. It means the value written by TX is lost, i.e., $250 is lost.

Hence data becomes incorrect, and database sets to inconsistent.

Problem 2: Dirty Read Problems (W-R Conflict)

The dirty read problem occurs when one transaction updates an item of the database, and somehow the transaction fails, and before the data gets rollback, the updated database item is accessed by another transaction.

There comes the Read-Write Conflict between both transactions.

For example:

Consider two transactions TX and TY in the below diagram performing read/write operations on account A where the available balance in account A is $300:

Time	T_x	T_y
t_1	READ (A)	—
t_2	A = A + 50	—
t_3	WRITE (A)	—
t_4	—	READ (A)
t_5	SERVER DOWN ROLLBACK	—

DIRTY READ PROBLEM

- At time t1, transaction TX reads the value of account A, i.e., $300.
- At time t2, transaction TX adds $50 to account A that becomes $350.
- At time t3, transaction TX writes the updated value in account A, i.e., $350.
- Then at time t4, transaction TY reads account A that will be read as $350.
- Then at time t5, transaction TX rollbacks due to server problem, and the value changes back to $300 (as initially).
- But the value for account A remains $350 for transaction TY as committed, which is the dirty read and therefore known as the Dirty Read Problem.

Problem 3: Unrepeatable Read Problem (W-R Conflict)

Also known as Inconsistent Retrievals Problem that occurs when in a transaction, two different values are read for the same database item.

For example: Consider two transactions, TX and TY, performing the read/write operations on account A, having an available balance = $300. The diagram is shown below:

Time	T_x	T_y
t_1	READ (A)	—
t_2	—	READ (A)
t_3	—	A = A + 100
t_4	—	WRITE (A)
t_5	READ (A)	—

UNREPEATABLE READ PROBLEM

- At time t1, transaction TX reads the value from account A, i.e., $300.
- At time t2, transaction TY reads the value from account A, i.e., $300.
- At time t3, transaction TY updates the value of account A by adding $100 to the available balance, and then it becomes $400.
- At time t4, transaction TY writes the updated value, i.e., $400.
- After that, at time t5, transaction TX reads the available value of account A, and that will be read as $400.
- It means that within the same transaction TX, it reads two different values of account A, i.e., $ 300 initially, and after updation made by transaction TY, it reads $400. It is an unrepeatable read and is therefore known as the Unrepeatable read problem.

Thus, in order to maintain consistency in the database and avoid such problems that take place in concurrent execution, management is needed, and that is where the concept of Concurrency Control comes into role.

Concurrency control techniques

The concurrency control techniques are as follows –

1. **Lock-Based Protocol**

In this type of protocol, any transaction cannot read or write data until it acquires an appropriate lock on it. There are two types of lock:

1. Shared lock:

- It is also known as a Read-only lock. In a shared lock, the data item can only read by the transaction.
- It can be shared between the transactions because when the transaction holds a lock, then it can't update the data on the data item.

2. Exclusive lock:

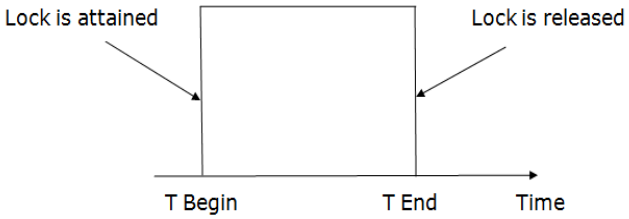
- In the exclusive lock, the data item can be both reads as well as written by the transaction.
- This lock is exclusive, and in this lock, multiple transactions do not modify the same data simultaneously.

There are four types of lock protocols available:

1. Simplistic lock protocol- It is the simplest way of locking the data while transaction. Simplistic lock-based protocols allow all the transactions to get the lock on the data before insert or delete or update on it. It will unlock the data item after completing the transaction.

2. Pre-claiming Lock Protocol

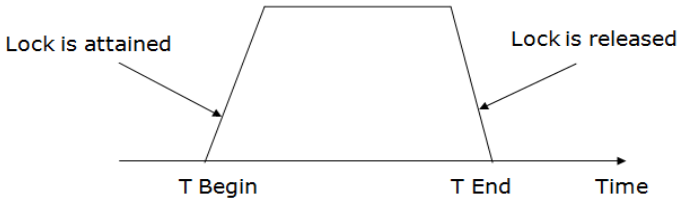
- Pre-claiming Lock Protocols evaluate the transaction to list all the data items on which they need locks.
- Before initiating an execution of the transaction, it requests DBMS for all the lock on all those data items.
- If all the locks are granted, then this protocol allows the transaction to begin. When the transaction is completed then it releases all the lock.
- If all the locks are not granted, then this protocol allows the transaction to rolls back and waits until all the locks are granted.



-

4.6 Two-phase locking (2PL)

- The two-phase locking protocol divides the execution phase of the transaction into three parts.
- In the first part, when the execution of the transaction starts, it seeks permission for the lock it requires.
- In the second part, the transaction acquires all the locks. The third phase is started as soon as the transaction releases its first lock.
- In the third phase, the transaction cannot demand any new locks. It only releases the acquired locks.



There are two phases of 2PL:

Growing phase: In the growing phase, a new lock on the data item may be acquired by the transaction, but none can be released.

Shrinking phase: In the shrinking phase, existing lock held by the transaction may be released, but no new locks can be acquired.

In the below example, if lock conversion is allowed then the following phase can happen:

1. Upgrading of lock (from S(a) to X (a)) is allowed in growing phase.
2. Downgrading of lock (from X(a) to S(a)) must be done in shrinking phase.

Example:

	T1	T2
0	LOCK-S(A)	
1		LOCK-S(A)
2	LOCK-X(B)	
3	——	——
4	UNLOCK(A)	
5		LOCK-X(C)
6	UNLOCK(B)	
7		UNLOCK(A)
8		UNLOCK(C)
9	——	——

The following way shows how unlocking and locking work with 2-PL.

Transaction T1:

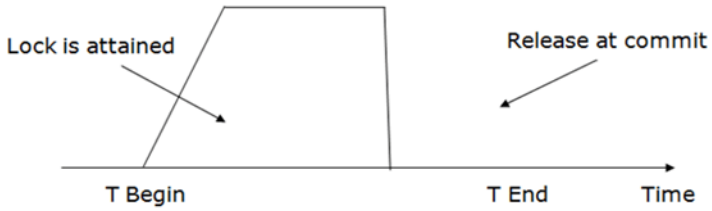
- Growing phase: from step 1-3
- Shrinking phase: from step 5-7
- Lock point: at 3

Transaction T2:

- Growing phase: from step 2-6
- Shrinking phase: from step 8-9
- Lock point: at 6

Strict Two-phase locking (Strict-2PL)

- The first phase of Strict-2PL is similar to 2PL. In the first phase, after acquiring all the locks, the transaction continues to execute normally.
- The only difference between 2PL and strict 2PL is that Strict-2PL does not release a lock after using it.
- Strict-2PL waits until the whole transaction to commit, and then it releases all the locks at a time.
- Strict-2PL protocol does not have shrinking phase of lock release.



Timestamp Ordering Protocol

- The Timestamp Ordering Protocol is used to order the transactions based on their Timestamps. The order of transaction is nothing but the ascending order of the transaction creation.
- The priority of the older transaction is higher that's why it executes first. To determine the timestamp of the transaction, this protocol uses system time or logical counter.
- The lock-based protocol is used to manage the order between conflicting pairs among transactions at the execution time. But Timestamp based protocols start working as soon as a transaction is created.

Let's assume there are two transactions T1 and T2. Suppose the transaction T1 has entered the system at 007 times and transaction T2 has entered the system at 009 times. T1 has the higher priority, so it executes first as it is entered the system first.

The timestamp ordering protocol also maintains the timestamp of last 'read' and 'write' operation on a data.

Basic Timestamp ordering protocol works as follows:

1.Check the following condition whenever a transaction Ti issues a Read (X) operation:

If *W_TS(X) >TS(Ti)* then the operation is rejected.

If *W_TS(X) <= TS(Ti)* then the operation is executed.

Timestamps of all the data items are updated.

2. Check the following condition whenever a transaction Ti issues a Write(X) operation:

If *TS(Ti) < R_TS(X)* then the operation is rejected.

If *TS(Ti) < W_TS(X)* then the operation is rejected and Ti is rolled back otherwise the operation is executed.

Where,

TS(TI) denotes the timestamp of the transaction Ti.

R_TS(X) denotes the Read time-stamp of data-item X.

W_TS(X) denotes the Write time-stamp of data-item X.

Advantages and Disadvantages of TO protocol:

TO protocol ensures serializability since the precedence graph is as follows:

Image: Precedence Graph for TS ordering

1. **Validation Based Protocol**

Validation phase is also known as optimistic concurrency control technique. In the validation-based protocol, the transaction is executed in the following three phases:

1. **Read phase:** In this phase, the transaction T is read and executed. It is used to read the value of various data items and stores them in temporary local variables. It can perform all the write operations on temporary variables without an update to the actual database.
2. **Validation phase**: In this phase, the temporary variable value will be validated against the actual data to see if it violates the serializability.
3. **Write phase:** If the validation of the transaction is validated, then the temporary results are written to the database or system otherwise the transaction is rolled back.

Here each phase has the following different timestamps:

- **Start (Ti):** It contains the time when Ti started its execution.

- **Validation (Ti):** It contains the time when Ti finishes its read phase and starts its validation phase.
- **Finish (Ti):** It contains the time when Ti finishes its write phase.

This protocol is used to determine the time stamp for the transaction for serialization using the time stamp of the validation phase, as it is the actual phase which determines if the transaction will commit or rollback.

Hence TS(T) = validation(T).

The serializability is determined during the validation process. It can't be decided in advance.

While executing the transaction, it ensures a greater degree of concurrency and also a smaller number of conflicts.

Thus, it contains transactions which have a smaller number of rollbacks.

Thomas write Rule

Thomas Write Rule provides the guarantee of serializability order for the protocol. It improves the Basic Timestamp Ordering Algorithm.

The basic Thomas write rules are as follows:

- If TS(T) < R_TS(X) then transaction T is aborted and rolled back, and operation is rejected.
- If TS(T) < W_TS(X) then don't execute the W_item(X) operation of the transaction and continue processing.
- If neither condition 1 nor condition 2 occurs, then allowed to execute the WRITE operation by transaction Ti and set W_TS(X) to TS(T).
- If we use the Thomas write rule then some serializable schedule can be permitted that does not conflict serializable as illustrate by the schedule in a given figure:

T1	T2
R(A) W(A) Commit	 W(A) Commit

Figure: A Serializable Schedule that is not Conflict Serializable

In the above figure, T1's read and precedes T1's write of the same data item. This schedule does not conflict serializable.

Thomas write rule checks that T2's write is never seen by any transaction. If we delete the write operation in transaction T2, then conflict serializable schedule can be obtained which is shown in below figure.

T1	T2
R(A) W(A) Commit	Commit

Figure: A Conflict Serializable Schedule

Multiple Granularity

Granularity: It is the size of data item allowed to lock.

Multiple Granularity:

It can be defined as hierarchically breaking up the database into blocks which can be locked.

The Multiple Granularity protocol enhances concurrency and reduces lock overhead.

It maintains the track of what to lock and how to lock.

It makes easy to decide either to lock a data item or to unlock a data item. This type of hierarchy can be graphically represented as a tree.

For example: Consider a tree which has four levels of nodes.

- The first level or higher level shows the entire database.
- The second level represents a node of type area.
- The higher-level database consists of exactly these areas.
- The area consists of children nodes which are known as files.
- No file can be present in more than one area.
- Finally, each file contains child nodes known as records.
- The file has exactly those records that are its child nodes.
- No records represent in more than one file.

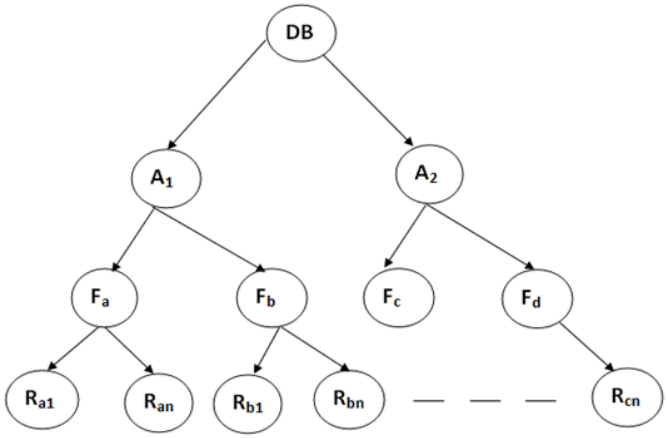


Figure: Multi Granularity tree Hierarchy

Hence, the levels of the tree starting from the top level are as follows:

1. Database
2. Area
3. File
4. Record

In this example, the highest level shows the entire database. The levels below are file, record, and fields.

There are three additional lock modes with multiple granularity:

1. **Intention-shared (IS):** It contains explicit locking at a lower level of the tree but only with shared locks.
2. **Intention-Exclusive (IX):** It contains explicit locking at a lower level with exclusive or shared locks.
3. **Shared & Intention-Exclusive (SIX):** In this lock, the node is locked in shared mode, and some node is locked in exclusive mode by the same transaction.

Compatibility Matrix with Intention Lock Modes: The below table describes the compatibility matrix for these lock modes:

	IS	IX	S	SIX	X
IS	✓	✓	✓	✓	X
IX	✓	✓	X	X	X
S	✓	X	✓	X	X
SIX	✓	X	X	X	X
X	X	X	X	X	X

It uses the intention lock modes to ensure serializability. It requires that if a transaction attempts to lock a node, then that node must follow these protocols:

- Transaction T1 should follow the lock-compatibility matrix.
- Transaction T1 firstly locks the root of the tree. It can lock it in any mode.
- If T1 currently has the parent of the node locked in either IX or IS mode, then the transaction T1 will lock a node in S or IS mode only.
- If T1 currently has the parent of the node locked in either IX or SIX modes, then the transaction T1 will lock a node in X, SIX, or IX mode only.
- If T1 has not previously unlocked any node only, then the Transaction T1 can lock a node.
- If T1 currently has none of the children of the node-locked only, then Transaction T1 will unlock a node.

Observe that in multiple-granularity, the locks are acquired in top-down order, and locks must be released in bottom-up order.

- If transaction T1 reads record Ra9 in file Fa, then transaction T1 needs to lock the database, area A1 and file Fa in IX mode. Finally, it needs to lock Ra2 in S mode.
- If transaction T2 modifies record Ra9 in file Fa, then it can do so after locking the database, area A1 and file Fa in IX mode. Finally, it needs to lock the Ra9 in X mode.
- If transaction T3 reads all the records in file Fa, then transaction T3 needs to lock the database, and area A in IS mode. At last, it needs to lock Fa in S mode.
- If transaction T4 reads the entire database, then T4 needs to lock the database in S mode.

4. Recovery with Concurrent Transaction

- Whenever more than one transaction is being executed, then the interleaved of logs occur. During recovery, it would become difficult for the recovery system to backtrack all logs and then start recovering.
- To ease this situation, 'checkpoint' concept is used by most DBMS.

As we have discussed checkpoint

in Transaction Processing Concept of this tutorial, so you can go through the concepts again to make things clearer.

UNIT V

OBJECT RELATIONAL AND NO-SQL DATABASES

Mapping EER to ODB schema –Object identifier –reference types –row types –UDTs – Subtypes and super types –user-defined routines –Collection types –Object Query

Language; No-SQL: CAP theorem –Document-based: MongoDB data model and CRUD operations; Column-based: Hbase data model and CRUD operations.

5.1 Mapping EER to ODB schema

It is relatively straightforward to design the type declarations of object classes for an ODBMS from an EER schema that contains neither categories nor n-ary relation-

ships with n > 2. However, the operations of classes are not specified in the EER dia- gram and must be added to the class declarations after the structural mapping is completed. The outline of the mapping from EER to ODL is as follows:

Step 1. Create an ODL class for each EER entity type or subclass. The type of the ODL class should include all the attributes of the EER class. Multivalued attributes are typically declared by using the set, bag, or list constructors.

If the values of the multivalued attribute for an object should be ordered, the list constructor is chosen; if duplicates are allowed, the bag constructor should be chosen; otherwise, the set constructor is chosen. Composite attributes are mapped into a tuple constructor (by using a struct declaration in ODL).

Declare an extent for each class, and specify any key attributes as keys of the extent. (This is possible only if an extent facility and key constraint declarations are available in the ODBMS.)

Step 2. Add relationship properties or reference attributes for each binary relationship into the ODL classes that participate in the relationship. These may be created in one or both directions.

If a binary relationship is represented by references in both directions, declare the references to be relationship properties that are inverses of one another, if such a facility exists. If a binary relationship is represented by a reference in only one direction, declare the reference to be an attribute in the referencing class whose type is the referenced class name.

Depending on the cardinality ratio of the binary relationship, the relationship properties or reference attributes may be single-valued or collection types. They will be single-valued for binary relationships in the 1:1 or N:1 direction; they are collection types (set-valued or list-valued) for relationships in the 1: N or M: N direction. An alternative way to map binary M: N relationships is discussed in step 7.

If relationship attributes exist, a tuple constructor (struct) can be used to create a structure of the form < reference, relationship attributes >, which may be included instead of the reference attribute. However, this does not allow the use of the inverse constraint. Additionally, if this choice is represented in both directions, the attribute values will be represented twice, creating redundancy.

This implicitly uses a tuple constructor at the top level of the type declaration, but in general, the tuple constructor is not explicitly shown in the ODL class declarations. Further analysis of the

application domain is needed to decide which constructor to use because this information is not available from the EER schema.

The ODL standard provides for the explicit definition of inverse relationships. Some ODBMS products may not provide this support; in such cases, programmers must maintain every relationship explicitly by coding the methods that update the objects appropriately. The decision whether to use set or list is not available from the EER schema and must be determined

Step 3. Include appropriate operations for each class. These are not available from the EER schema and must be added to the database design by referring to the origi- nal requirements.

A constructor method should include program code that checks any constraints that must hold when a new object is created. A destructor method should check any constraints that may be violated when an object is deleted. Other methods should include any further constraint checks that are relevant.

Step 4. An ODL class that corresponds to a subclass in the EER schema inherits (via extends) the type and methods of its superclass in the ODL schema. Its specific (noninherited) attributes, relationship references, and operations are specified, as discussed in steps 1, 2, and 3.

Step 5. Weak entity types can be mapped in the same way as regular entity types.

An alternative mapping is possible for weak entity types that do not participate in any relationships except their identifying relationship; these can be mapped as though they were composite multivalued attributes of the owner entity type, by using the set < struct < ... >> or list < struct < ... >> constructors. The attributes of the weak entity are included in the struct < ... > construct, which corresponds to a tuple constructor. Attributes are mapped as discussed in steps 1 and 2.

Step 6. Categories (union types) in an EER schema are difficult to map to ODL. It is possible to create a mapping similar to the EER-to-relational mapping by declaring a class to represent the category and defining 1:1 relationship between the category and each of its superclasses. Another option is to use a union type, if it is available**.**

Step 7. An n-ary relationship with degree n > 2 can be mapped into a separate class, with appropriate references to each participating class.

These references are based on mapping a 1: N relationship from each class that represents a participating entity type to the class that represents the n-ary relationship. An M: N binary relationship, especially if it contains relationship attributes, may also use this mapping option, if desired.

5.2 Object identifier

An object identifier (OID) is an unambiguous, long-term name for any type of object or entity.

The OID mechanism finds application in diverse scenarios, particularly in security, and is endorsed by the International Telecommunication Union (ITU), the Internet Engineering Task Force (IETF), and ISO.

In computing, an OID appears as a group of characters that allows a server or end user to retrieve an object without needing to know the physical location of the data. This approach is useful for automating and streamlining data storage in cloud computing environments. On the Internet, an OID takes the form of a Universal Unique Identifier (UUID), a 128-bit number used to uniquely identify an object or entity. In a database, an OID is a set of integers that uniquely identifies each Row (or record) in a table.

What is an OID?

An object identifier (OID) is an extensively used identification mechanism jointly developed by ITU-T and ISO/IEC for naming any type of object, concept or "thing" with a globally unambiguous name which requires a persistent name (long life-time). It is not intended to be used for transient naming. OIDs, once allocated, should not be re-used for a different object/thing.

It is based on a hierarchical name structure based on the "OID tree". This naming structure uses a sequence of names, of which the first name identifies a top-level "node" in the OID tree, and the next provides further identification of arcs leading to sub-nodes beneath the top-level, and so on to any depth.

A critical feature of this identification mechanism is that it makes OIDs available to a great many organizations and specifications for their own use (including countries, ITU-T Recommendations, ISO and IEC International Standards, specifications from national, regional or international organizations, etc.).

How are OIDs allocated and what is a registration authority?

At each node, including the root, there is a requirement for some organization or standard to be responsible for allocating arcs to sub-nodes and recording that allocation (together with the organization the subordinate node has been allocated to), not necessarily publicly. This activity is called a Registration Authority (RA).

In the OID tree, RAs are generally responsible only for allocation of sub-arcs to other RAs that then control their own sub-nodes. In general, the RA for a sub-node operates independently in allocating further sub-arcs to other organizations, but can be constrained by rules imposed by its superior, should the superior so wish.

The registration tree is indeed managed in a completely decentralized way (a node gives full power to its children).

The registration tree is defined and managed following the ITU-T X.660 & X.670 Recommendation series (or the ISO/IEC 9834 series of International Standards)

What is an OID repository?

Initially, it was left for each Registration Authority (RA)in the hierarchy to maintain its own record of allocation beneath that RA, and to keep those allocations private if it so chose. There was never any policing of this. An RA in the hierarchy was its own master and operated autonomously.

In the early 1990s Orange developed software for their internal use which was generic enough to provide a publicly available repository of OID allocations.

Information on OIDs is often buried inside the databases (perhaps sometimes paper) maintained by an immense number of RAs. The information can be hard to access and is sometimes private. Today this OID repository is regarded as the easiest way to access a large amount of the publicly available information on OIDs: Many OIDs are recorded but it does not contain all existing OIDs.

This OID repository is not an official Registration Authority, so any OID described on this web site has to be officially allocated by the RA of its parent OID. The accuracy and completeness of this OID repository rely on crowdsourcing, i.e., each user is welcome to contribute data.

5.3 reference type

In SQL, a <reference type> is a pointer; a scalar constructed SQL <data type>. It points to a row of a Base table that has the with REF value property – that is, a <reference type> points to a UDT value.

Reference <data type>s

A <reference type> is defined by a descriptor that contains three pieces of information:

1. The <data type>'s name: REF.
2. The name of the UDT that the <reference type> is based on. (The UDT is known as the referenced type.)
3. The scope of the <reference type>: a (possibly empty) list of the names of the Base tables that make up the <reference type>'s scope.

REF

The required syntax for a <reference type> specification is as follows.

<reference type>:: =

REF (<UDT name>)

[SCOPE <Table name> [reference scope check]]

<reference scope check> ::=

REFERENCES ARE [NOT] CHECKED

[ON DELETE

{CASCADE | SET NULL | SET DEFAULT | RESTRICT | NO ACTION}]

A <reference type> specification defines a pointer: its value is a value that references some site. (A site either does or does not have a REF value.) For example, this REF specification defines a <reference type> based on a UDT (the "referenced type") called my_udt:

REF(my_udt)

As already mentioned, a REF is a pointer. The value in a REF coloumn "refers" to a row in a Base table that has the with REF value property (this is a known as a typed table). The row that the REF value points to contains a value of the UDT that the REF Column is based on.

If you're putting a REF specification in an SQL-Schema statement, the <AuthorizationID> that owns the containing Schema must have the USAGE Privilege on "<UDT name>".

If you're putting a REF specification in any other SQL statement, then your current <AuthorizationID> must have the USAGE Privilege on "<UDT name>".

For each site that has a REF value and is defined to hold a value of the referenced UDT, there is exactly one REF value – at any time, it is distinct from the REF value of any other site in your SQL-environment. The <data type> of the REF value is REF (UDT).

[NON-PORTABLE] The data type and size of a REF value in an application program must be some number of octets but is non-standard because the SQL Standard requires implementors to define the octet-length of a REF value.

A REF value might have a scope: it determines the effect of a dereference operator on that value. A REF value's scope is a list of Base table names and consists of every row in every one of those Base tables.

The optional SCOPE clause of a <reference type> specification identifies REF's scope. Each Table named in the SCOPE clause must be a referenceable Base table with a structured type that is the same as the structured type of the UDT that REF is based on. Here is an examples:

```
CREATE TABLE Table_1 (
  column_1 SMALLINT,
  column_2 REF(my_udt) SCOPE Table_2);
```

If you omit the SCOPE clause, the scope defaults to the Table that owns the Column you're defining.

If your REF specification with a SCOPE clause is part of a <Field definition>, it must include this <reference scope checks>: REFERENCES ARE [NOT] CHECKED ON DELETE NO ACTION.

If a REF specification with a SCOPE clause is part of a <Column definition>, it must include a <reference scope checks> with or without the optional ON DELETE sub-clause.

The <reference scope check> clause may not be used under any other circumstances.

A <reference type> is a subtype of a <data type> if (a) both are <reference type>s and (b) the UDT referenced by the first <reference type> is a subtype of the UDT referenced by the second <reference type>.

If you want to restrict your code to Core SQL, don't use the REF <data type>.

Reference Operations

A <reference type> is compatible with, and comparable to, all other <reference type>s of the same referenced type – that is, <reference type>s are mutually comparable and mutually assignable if they are based on the same UDT.

CAST

In SQL, CAST is a scalar operator that converts a given scalar value to a given scalar <data type>. CAST, howver, can´t be used with <reference type>s. To cast REF values, you´ll have to use a user-defined cast.

It isn't, of course, possible to convert the values of every <data type> into the values of every other <data type>. You can cast a <reference type> source to a UDT target and to any SQL predefined <data type> target (except for <collection type>s and <row type>s) provided that a user-defined cast exist for this purpose and your current <AuthorizationID> has the EXECUTE Privilege on that user-defined cast. When you cast a <reference type> to any legal target, your DBMS incokes the user-defined cast. When you cast a <reference type> to any legal target, your DBMS invokes the user-defined cast routine´s argument. The cast result in the value returned by the user-defined cast.

Assignment

In SQL, when a <reference type> is assigned to a <reference type> target, the assignment is straightforward – however, assignment is possible only if your source's UDT is a subtype of the UDT of your target.

[Obscure Rule] Since only SQL accepts null values, if your source is NULL, then your target's value is not changed. Instead, your DBMS will set its indicator parameter to -1, to indicate that an assignment of the null value was attempted. If your target doesn't have an indicator parameter, the assignment will fail: your DBMS will return the SQLSTATE error 22002 "data exception-null value, no indicator parameter". Going the other way, there are two ways to assign a null value to an SQL-data target. Within SQL, you can use the <keyword> NULL in an INSERT or an UPDATE statement to indicate that the target should be set to NULL; that is, if your source is NULL, your DBMS will set your target to vNULL``. Outside of SQL, if your source has an indicator parameter that is set to -1, your DBMS will set your target to NULL (regardless of the value of the source). (An indicator parameter with a value less than -1 will cause an error: your DBMS will return the SQLSTATE error 22010 "data exception-invalid indicator parameter value".) We'll talk more about indicator parameters in our chapters on SQL binding styles.

Comparison

SQL provides only two scalar comparison operators – = and <> – to perform operations on <reference type>s. Both will be familiar; there are equivalent operators in other computer languages. Two REF values are comparable if they're both based on the same UDT. If either of the comparands are NULL, the result of the operation is UNKNOWN.

Other Operations

With SQL, you have several other operations that you can perform on <reference type>s.

Scalar functions

SQL provides two scalar functions that operate on or return a <reference type>: the <dereference operation> and the <reference resolution>.

<dereference operation>

The required syntax for a <dereference operation> is as follows.

<dereference operation>:: = reference_argument -> <Attribute name>

The <dereference operation> operates on two operands — the first must evaluate to a <reference type> that has a non-empty scope and the second must be the name of an Attribute of the <reference type>'s UDT.

The <dereference operation> allows you to access a Column of the row identified by a REF value; it returns a result whose <data type> is the <data type> of <Attribute name> and whose value is the value of the system-generated Column of the Table in the <reference type>'s scope (where the system-generated Column is equal to reference_argument). That is, given a REF value, the <dereference operation> returns the value at the site referenced by that REF value. If the REF value doesn't identify a site (perhaps because the site it once identified has been destroyed), the <dereference operation> returns NULL.

If you want to restrict your code to Core SQL, don't use the <dereference operation>.

<reference resolutions>

The required syntax for a <dereference operation> is as follows.

<dereference operation>:: = reference_argument -> <Attribute name>

DEREF operates on any expression that evaluates to a <reference type> that has a non-empty scope. It returns the value referenced by a REF value. Your current <AuthorizationID> must have the SELECT WITH HIERARCHY Privilege on reference_argument's scope Table.

If you want to restrict your code to Core SQL, don't use DEREF.

Set Functions

SQL provides three set functions that operate on a <reference type>: COUNT and GROUPING. Since none of these operate exclusively with REF arguments, we won't discuss them here; look for them in our chapter on set functions.

Predicates

In addition to the comparison operators, SQL provides eight other predicates that operate on <reference type>s: the <between predicate>, the <in predicate>, the <null predicate>, the <exists predicate>, the <unique predicate>, the <match predicate>, the <quantified predicate> and the <distinct predicate>. Each will return a boolean value: either TRUE, FALSE or UNKNOWN. Since none of them operates strictly on <reference type>s, we won't discuss them here. Look for them in our chapters on search conditions.

5.4 ROWTYPE Attribute

Row <data type>s

A <row type> is defined by a descriptor that contains three pieces of information:

The <data type>'s name: **ROW.**

The <data type>'s degree: the number of Fields that belong to the row.

A descriptor for every Field that belongs to the row. The Field descriptor contains the name of the Field, the Field's ordinal position in the <row type>, the Field's <data type> and nullability attribute (or, if the Field is based on a Domain, the name of that Domain), the Field's Character set and default Collation (for character string <data type>s) and the Field's <reference scope check> (for <reference type>s).

ROW

Example:

The required syntax for a <row type> specification is as follows.

<row type> ::= ROW (<Field definition> [{,<Field definition>}...])

<Field definition> ::= <Field name> {<data type> | <Domain name>}

[<reference scope check>]

[COLLATE <Collation name>]

A <row type> specification defines a row of data: it consists of a sequence of one or more parenthesized {<Field name>,<data type>} pairs, known as Fields. The degree of a row is the number of Fields it contains. A value of a row consists of one value for each of its Fields, while a value of a Field is a value of the Field's <data type>. Each Field in a row must have a unique name.

Example of a <row type> specification:

ROW (field_1 INT, field_2 DATE, field_3 INTERVAL (4) YEAR)

A <Field name> identifies a Field and is either a <regular identifier> or a <delimited identifier> that is unique (for all Fields and Columns) within the Table it belongs to. You can define a Field's <data type> either by putting a <data type> specification after <Field name> or by putting a <Domain name> after the <Field name>. The <data type> of a Field can be any type other than a <reference type> – in particular, it can itself be a <row type>.

Example, of a <row type> specification;

It defines a row with one Field (called **field_1**) whose defined <data type> is **DATE**:

ROW (field_1 DATE)

[Obscure Rule] If the <data type> of a Field is **CHAR**, **VARCHAR** or **CLOB,** the Character set that the Field's values must belong to is determined as follows:

- If the <Field definition> contains a <data type> specification that includes a **CHARACTER SET** clause, the Field's Character set is the Character set named. Your current <AuthorizationID> must have the **USAGE** Privilege on that Character set.
- If the <Field definition> does not include a <data type> specification, but the Field is based on a Domain whose definition includes a **CHARACTER SET** clause, the Field's Character set is the Character set named.
- If the <Field definition> does not include any **CHARACTER SET** clause at all – either through a <data type> specification or through a Domain definition – the Field's Character set is the Character set named in the **DEFAULT CHARACTER SET** clause of the **CREATE SCHEMA** statement that defines the Schema that the Field belongs to.

For example, the effect of these two SQL statements:

```
CREATE SCHEMA bob AUTHORIZATION bob
 DEFAULT CHARACTER SET INFORMATION_SCHEMA.LATIN1;
  CREATE TABLE Table_1 (
    column_1 ROW(
      field_1 CHAR(10),
      field_2 INT));
```

is to create a Table in Schema **bob**. The Table has a Column with a **ROW** <data type>, containing two Fields.

The character string Field's set of valid values are fixed length character strings, exactly 10 characters long, all of whose characters must be found in the

INFORMATION_SCHEMA.LATIN1 Character set – the Schema's default Character set. The effect of these two SQL statements:

```
CREATE SCHEMA bob AUTHORIZATION bob
 DEFAULT CHARACTER SET INFORMATION_SCHEMA.LATIN1;
 CREATE TABLE Table_1 (
   column_1 ROW(
     field_1 CHAR(10) CHARACTER SET INFORMATION_SCHEMA.SQL_CHARACTER,
     field_2 INT));
```

is to create the same Table with one difference: this time, the character string Field's values must consist only of characters found in the **INFORMATION_SCHEMA.SQL_CHARACTER** Character set – the explicit Character set specification in **CREATE TABLE** constrains the Field's set of values. The Schema's default Character set does not.

[Obscure Rule] If the <data type> of a Field is CHAR, VARCHAR, CLOB, NCHAR, NCHAR VARYING or NCLOB, and your <Field definition> does not include a COLLATE clause, the Field has a coercibility attribute of COERCIBLE – but if your <Field definition> includes a COLLATE clause, the Field has a coercibility attribute of IMPLICIT. In either case, the Field's default Collation is determined as follows:

- If the <Field definition> includes a COLLATE clause, the Field's default Collation is the Collation named. Your current <Authorization ID> must have the USAGE Privilege on that Collation.
- If the <Field definition> does not include a COLLATE clause, but does contain a <data type> specification that includes a COLLATE clause, the Field's default Collation is the Collation named. Your current <Authorization ID> must have the USAGE Privilege on that Collation.
- If the <Field definition> does not include a COLLATE clause, but the Field is based on a Domain whose definition includes a COLLATE clause, the Field's default Collation is the Collation named.
- If the <Field definition> does not include any COLLATE clause at all – either explicitly, through a <data type> specification or through a Domain definition – the Field's default Collation is the default Collation of the Field's Character set.

[Obscure Rule] If the <data type> of a Field is REF(UDT), your current <AuthorizationID> must have the USAGE Privilege on that UDT. If the <data type> of a Field includes REF with a <scope clause>, your <Field definition> must also include this <reference scope check> clause: REFERENCES ARE [NOT] CHECKED ON DELETE NO ACTION – to indicate whether references are to be checked or not. Do not add a <reference scope check> clause under any other circumstances.

- If a Field is defined with REFERENCES ARE CHECKED, and a <scope clause> is included in the <Field definition>, then there is an implied DEFERRABLE INITIALLY IMMEDIATE Constraint on the Field. This Constraint checks that the Field´s values are also found in the corresponding Field of the system-generated Column of the Table named in the <scope clause>.
- If the <data type> of a Field in a row is a UDT, then the current <AuthorizationID> must have the USAGE Privilege on that UDT.
- A <row type> is a subtype of a <data type> if (a) both are <row type>s with the same degree and (b) for every pair of corresponding <Field definition>s, the <Field name>s are the same and the <data type> of the Field in the first <row type> is a supertype of the <data type> of the Field in the second <row type>.

<row reference>

A <row reference> returns a row. The required syntax for a <row reference> is as follows.

<row reference> ::= ROW {<Table name> | <query name> | <Correlation name>}

A row of data values belonging to a Table (or a query result, which is also a Table) is also considered to be a <row type>.

In a Table, each Column of a data row corresponds to a Field of the <row type>: the Column and Field have the same ordinal positions in the Table and <row type>, respectively.

A <row reference> allows you to access a specific row of a Table or a query result. Here is an example of a <row reference> that would return a row of a Table named **TABLE_1**:

ROW(Table_1)

<Field reference>

A <Field reference> returns a Field of a row. The required syntax for a <Field reference> is as follows.

<Field reference> ::= row_argument.<Field name>

A <Field reference> allows you to access a specific Field of a row. It operates on two arguments: the first must evaluate to a <row type> and the second must be the name of a Field belonging to that row.

If the value of row_argument is NULL, then the specified Field is also NULL.

If row_argument has a non-null value, the value of the Field reference is the value of the specified Field in row_argument. Here is an example of a <Field reference> that would return the value of a Field named FIELD_1 that belongs to a row of TABLE_1:

ROW(Table_1).field_1

<row value constructor>

An <row value constructor> is used to construct a row of data. The required syntax for a <row value constructor> is as follows.

<row value constructor> ::= element_expression |

[ROW] (element_expression [{,element_expression}...]) |

(<query expression>)

element_expression ::=

element_expression |

NULL |

ARRAY[] |

ARRAY??(??) |

DEFAULT

A <row value constructor> allows you to assign values to the Fields of a row, using either a list of **element_expressions** of the result of a subquery. An **element_expression** may be any expression that evaluates to a scalar value with a <data type> that is assignable to the corresponding Field's <data type>. A subquery – (<query expression>) – is discussed in our chapter on complex queries.

The result is a row whose n-th Field value is the value of the n-th **element_expression** (or whose value is the value of the subquery) you specify. If your **element_expression** is **NULL,** the corresponding Field is assigned the null value. If your **element_expression** is **ARRAY []** or **ARRAY??(??),** the corresponding Field is assigned an empty array. If your **element_expression** is **DEFAULT**, the corresponding Field is assigned its default value. Here is an example of a <row value constructor>:

ROW ('hello',567, DATE '1994-07-15', NULL, DEFAULT, ARRAY [])

This example constructs a row with six Fields. The first Field has a character string value of 'hello', the second has a numeric value of 567, the third has a date value of '1994-07-15', the fourth has a null value, the fifth has a value that is the fifth Field's default value and the sixth has a value that is an empty array. This <row value constructor> would be valid for this <row type> specification:

ROW (field_1 CHAR (5),

field_2 SMALLINT,

field_3 DATE,

field_4 BIT (4),

field_5 domains_1,

field_6 INT ARRAY [4])

A <row value constructor> serves the same purpose for a row as a <literal> does for a predefined <data type>. It has the same format as the <row type>'s ROW () – but instead of a series of <Field definition>s inside the size delimiters, it contains comma-delimited values of the correct <data type> for each Field. For example, if your <row type> specification is:

ROW (field_1 INT, field_2 CHAR (5), field_3 BIT (4))

a valid <row value constructor> would be:

ROW (20,'hello', B'1011')

If you construct a row with a subquery, the row takes on the <data type> of the subquery's result. An empty subquery result constructs a one-Field row whose value is **NULL.** A non-empty subquery result constructs a one-Field row whose value is the subquery result.

If you want to restrict your code to Core SQL, (a) don't use the **ROW** <data type> or <row reference>s and <Field reference>s and, when using a <row value constructor>, (b) don't use **ARRAY[]** or **ARRAY??(??)** as an **element_expression,(**c) don't construct a row with more than one Field,(d) don't use the ROW <keyword> in front of your **element_expression,** and (e) don't use a subquery to construct your row.

Row Operations

A row is compatible with, and comparable to, any row with compatible Fields – that is, rows are mutually comparable and mutually assignable only if they have the same number of Fields and each corresponding pair of Fields are mutually comparable and mutually assignable. Rows may not be directly compared with, or directly assigned to, any other <data type> class, though implicit type conversions of their Fields can occur in expressions, SELECTs, INSERTs, DELETEs and UPDATEs. Explicit row type conversions are not possible.

Assignment

In SQL, when a <row type> is assigned to a <row type> target, the assignment is done one Field at a time – that is, the source's first Field value is assigned to the target's first Field, the source's second Field value is assigned to the target's second Field, and so on. Assignment of a <row type> to another <row type> is possible only if (a) both <row type>s have the same number of Fields and (b) each corresponding pair of Fields have <data type>s that are mutually assignable.

[Obscure Rule] Since only SQL accepts null values, if your source is NULL, then your target's value is not changed. Instead, your DBMS will set its indicator parameter to -1, to indicate that an assignment of the null value was attempted.

If your target doesn't have an indicator parameter, the assignment will fail: your DBMS will return the SQLSTATE error 22002 "data exception-null value, no indicator parameter". Going the other way, there are two ways to assign a null value to an SQL-data target. Within SQL, you can use the <keyword> NULL in an INSERT or an UPDATE statement to indicate that the target

should be set to NULL; that is, if your source is NULL, your DBMS will set your target to NULL.

Outside of SQL, if your source has an indicator parameter that is set to -1, your DBMS will set your target to NULL (regardless of the value of the source). (An indicator parameter with a value less than -1 will cause an error: your DBMS will return the SQLSTATE error 22010 "data exception-invalid indicator parameter value".) We'll talk more about indicator parameters in our chapters on SQL binding styles.

Comparison

SQL provides the usual scalar comparison operators – = and <> and < and <= and > and >= – to perform operations on rows. All of them will be familiar; there are equivalent operators in other computer languages. Two rows are comparable if (a) both have the same number of Fields and (b) each corresponding pair of Fields have <data type>s that are mutually comparable.

Comparison is between pairs of Fields in corresponding ordinal positions – that is, the first Field of the first row is compared to the first Field of the second row, the second Field of the first row is compared to the second Field of the second row, an so on. If either comparand is NULL the result of the operation is UNKNOWN.

The result of a <row type> comparison depends on two things: (a) the comparison operator and (b) whether any Field is NULL. The order of comparison is:

If the comparison operator is = or <>: First the Field pairs which don´t include NULLs, then the pairs which do.

If the comparison operator is anything other than = or <>: Field pairs from left to right. Comparison stops when the result is unequal or UNKNOWN, or when there are no more Fields. The result of the row comparison is the result of the last Field pair comparison.

Here are the possibilities.

If the comparison operator is =. The row comparison is (a) TRUE if the comparison is TRUE for every pair of Fields, (b) FALSE if any non-null pair is not equal, and (c) UNKNOWN if at least one Field is NULL and all non-null pairs are equal. For example:

```
ROW (1,1,1) = ROW (1,1,1)         -- returns TRUE

ROW (1,1,1) = ROW (1,2,1)         -- returns FALSE

ROW (1, NULL,1) = ROW (2,2,1)     -- returns FALSE

ROW (1, NULL,1) = ROW (1,2,1)     -- returns UNKNOWN
```

Comparison operator is <>. The row comparison is (a) TRUE if any non-null pair is not equal, (b) FALSE if the comparison is FALSE for every pair of Fields, and (c) UNKNOWN if at least one Field is NULL and all non-null pairs are equal. For example:

```
ROW (1,1,1) <> ROW (1,2,1)        -- returns TRUE
```

ROW (1, NULL,2) <> ROW (2,2,1) -- returns TRUE

ROW (2,2,1) <> ROW (2,2,1) -- returns FALSE

ROW (1, NULL,1) <> ROW (1,2,1) -- returns UNKNOWN

Comparison operator is anything other than = or <>.

The row comparison is

(a) TRUE if the comparison is TRUE for at least one pair of Field and every pair before the TRUE result is equal,

(b) FALSE uf the comparison is FALSE for at least one pair of Fields and every pair before the FALSE result is equal, and

(c) UNKNOWN if the comparison is UNKNWON for at least one pair of Fields and every pair before the UNKNOWN result is equal. Comparison stops as soon as any of these results (TRUE, FALSE, or UNKNOWN) is established. For example:

ROW (1,1,1) < ROW (1,2,1) -- returns TRUE

ROW (1, NULL,1) < ROW (2, NULL,0) -- returns TRUE

ROW (1,1,1) < ROW (1,1,1) -- returns FALSE

ROW (3, NULL,1) < ROW (2, NULL,0) -- returns FALSE

ROW (2, NULL,1) < ROW (1,2,0) -- returns UNKNOWN

ROW (NULL,1,1) < ROW (2,1,0) -- returns UNKNOWN

SQL also provides three quantifiers – ALL, SOME, ANY – which you can use along with a comparison operator to compare a row value with the collection of values returned by a <table subquery>. Place the quantifier after the comparison operator, immediately before the <table subquery>. For example:

```
SELECT row_column
FROM   Table_1
WHERE row_column < ALL (
  SELECT row_column
  FROM   Table_2);
```

ALL returns TRUE either (a) if the collection is an empty set (i.e.: if it contains zero rows) or (b) if the comparison operator returns TRUE for every value in the collection. ALL returns FALSE if the comparison operator returns FALSE for at least one value in the collection.

SOME and ANY are synonyms. They return TRUE if the comparison operator returns TRUE for at least one value in the collection. They return FALSE either (a) if the collection is an empty set

or (b) if the comparison operator returns FALSE for every value in the collection. The search condition = ANY (collection) is equivalent to “IN (collection)``

5.5 UDTs

A UDT is defined by a descriptor that contains twelve pieces of information:

1. The <UDT name>, qualified by the <Schema name> of the Schema it belongs to.
2. Whether the UDT is ordered.
3. The UDT’s ordering form: either EQUALS, FULL or NONE.
4. The UDT’s ordering category: either RELATIVE, HASH or STATE.
5. The <specific routine designator> that identifies the UDT’s ordering function.
6. If the UDT is a direct subtype of one or more other UDTs, then the names of those UDTs.
7. If the UDT is a distinct type, then the descriptor of the <data type> it’s based on; otherwise an Attribute descriptor for each of the UDT’s Attributes.
8. The UDT’s degree: the number of its Attributes.
9. Whether the UDT is instantiable or not instantiable.
10. Whether the UDT is final or not final.
11. The UDT’s Transform descriptor.
12. If the UDT’s definition includes a method signature list, a descriptor for each method signature named.

To create a UDT, use the CREATE TYPE statement (either as a stand-alone SQL statement or within a CREATE SCHEMA statement). CREATE TYPE specifies the enclosing Schema, names the UDT and identifies the UDT’s set of valid values.

To destroy a UDT, use the DROP TYPE statement. None of SQL3’s UDT syntax is Core SQL, so if you want to restrict your code to Core SQL, don’t use UDTs.

UDT Names

A <UDT name> identifies a UDT. The required syntax for a <UDT name> is:

<UDT name> ::= [<Schema name>.] unqualified name

A <UDT name> is a <regular identifier> or a <delimited identifier> that is unique (for all Domains and UDTs) within the Schema it belongs to. The <Schema name> which qualifies a <UDT name> names the Schema that the UDT belongs to and can either be explicitly stated, or a default will be supplied by your DBMS as follows:

- If a <UDT name> in a CREATE SCHEMA statement isn’t qualified, the default qualifier is the name of the Schema you’re creating.
- If the unqualified <UDT name> is found in any other SQL statement in a Module, the default qualifier is the name of the Schema identified in the SCHEMA clause or AUTHORIZATION clause of the MODULE statement that defines that Module

UDT Example

Here's an example of a UDT definition:

```
CREATE TYPE book_udt AS                    -- the UDT name will be book_udt
title CHAR (40),                           -- title is the first attribute
buying_price DECIMAL (9,2),                -- buying_price is the second attribute
selling_price DECIMAL (9,2)                -- selling_price is the third attribute
NOT FINAL                                  -- this is a mandatory Finality Clause
METHOD profit () RETURNS DECIMAL (9,2); -- profit is a method, defined later
```

This CREATE TYPE statement results in a UDT named BOOK_UDT. The components of the UDT are three attributes (named TITLE, BUYING_PRICE and SELLING_PRICE) and one method (named PROFIT).

The three name-and-data-type pairs title CHAR (40) and buying_price DECIMAL (9,2) and selling_price DECIMAL (9,2) are the UDT's Attribute definitions.

The words NOT FINAL matter only for subtyping, which we'll get to later. Briefly, though, if a UDT definition doesn't include an UNDER clause, the finality clause must specify NOT FINAL.

The clause METHOD profit () RETURNS DECIMAL (9,2) is a teaser. Like an Attribute, a "method" is a component of a UDT. However, this method – PROFIT – is actually a declaration that a function named PROFIT exists.

This function isn't defined further in the UDT definition – there is a separate SQL statement for defining functions: CREATE METHOD. All we can see at this stage is that PROFIT has a name and a (predefined) data type>, just as regular Attributes do. Some people would call PROFIT a "derived Attribute".

5.6 Super type and Sub type

Purpose of the Supertypes and Subtypes

Supertypes and subtypes occur frequently in the real world:

- food order types (eat in, to go)
- grocery bag types (paper, plastic)
- payment types (check, cash, credit)

You can typically associate 'choices' of something with supertypes and subtypes.

For example, what will be the method of payment – cash, check or credit card?

Understanding real world examples helps us understand how and when to model them.

Evaluating Entities

Often some instances of an entity have attributes and/or relationships that other instances do not have.

Imagine a business which needs to track payments from customers.

Customers can pay by cash, by check, or by credit card.

All payments have some common attributes: payment date, payment amount, and so on.

But only credit cards would have a “card number” attribute.

And for credit card and check payments, we may need to know which CUSTOMER made the payment, while this is not needed for cash payments

Should we create a single PAYMENT entity or three separate entities CASH, CHECK, and CREDIT CARD?

And what happens if in the future we introduce a fourth method of payment?

Subdivide an Entity

Sometimes it makes sense to subdivide an entity into subtypes.

This may be the case when a group of instances has special properties, such as attributes or relationships that exist only for that group.

In this case, the entity is called a “supertype” and each group is called a “subtype”.

Subtype Characteristics

A subtype:

Inherits all attributes of the supertype

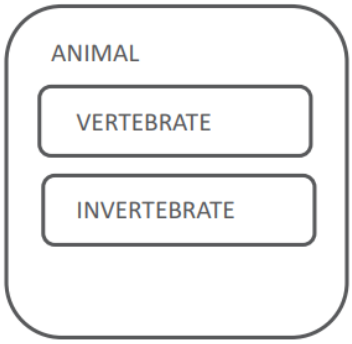
Inherits all relationships of the supertype

Usually has its own attributes or relationships

Is drawn within the supertype

Never exists alone

May have subtypes of its own

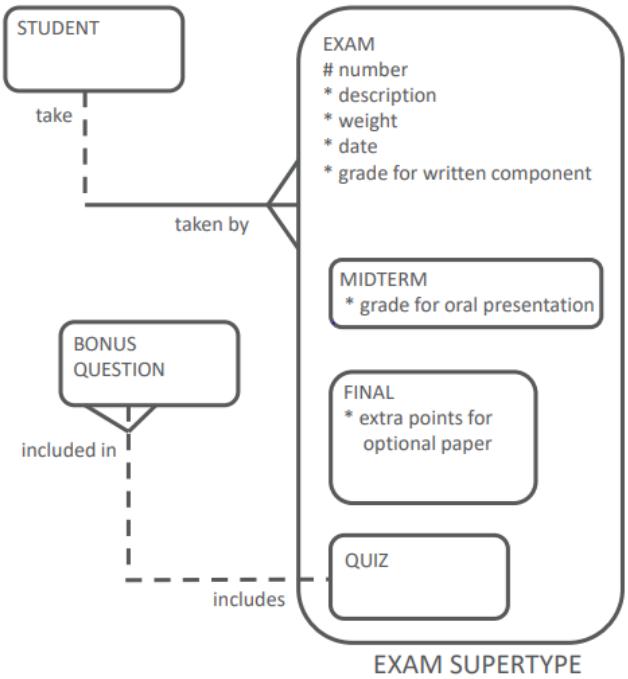


Supertype Example

EXAM is a supertype of QUIZ, MIDTERM, and FINAL.

The subtypes have several attributes in common.

These common attributes are listed at the supertype level.



The same applies to relationships.

Subtypes inherit all attributes and relationships of the supertype entity.

Read the diagram as: Every QUIZ, MIDTERM, or FINAL is an EXAM (and thus has attributes such as description, weight, date, and grade).

Conversely: Every EXAM is either a QUIZ, a MIDTERM, or a FINAL.

Always More Than One Subtype

When an ER model is complete, subtypes never stand alone. In other words, if an entity has a subtype, a second subtype must also exist. This makes sense.

A single subtype is exactly the same as the supertype.

This idea leads to the two subtype rules:

Exhaustive: Every instance of the supertype is also an instance of one of the subtypes. All subtypes are listed without omission.

Mutually Exclusive: Each instance of a supertype is an instance of only one possible subtype.

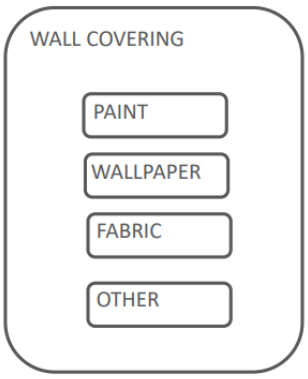
At the conceptual modeling stage, it is good practice to include an OTHER subtype to make sure that your subtypes are exhaustive — that you are handling every instance of the supertype.

Subtypes Always Exist

Any entity can be subtyped by making up a rule that subdivides the instances into groups.

But being able to subtype is not the issue—having a reason to subtype is the issue.

When a need exists within the business to show similarities and differences between instances, then subtype.



WALLCOVERING SUPERTYPE

Correctly Identifying Subtypes

When modeling supertypes and subtypes, you can use three questions to see if the subtype is correctly identified:

Is this subtype a kind of supertype?

Have I covered all possible cases? (exhaustive)

Does each instance fit into one and only one subtype? (mutually exclusive)

SNOWSPORT SUPERTYPE

Nested Subtypes

You can nest subtypes.

For ease of reading — "readability" — you would usually show subtypes with only two levels, but there is no rule that would stop you from going beyond two levels.

5.7 User-Defined routines (UDR)

User-defined routines (UDR) are functions that perform specific actions that you can define in your SIL™ programs for a later use. These can considerably improve the readability and maintainability of your code.

Syntax

```
function <name>(<type> param1, <type> param2, ...) {
   Instruction1;
   ...
   InstructionN;
   return <value>;
}
```

Example

```
function zero () {
   return 0;
}
number a = zero ();
```

Parameters

The list of parameters in the definition of a UDR can be of any length (including 0) and their respective types can be any valid SIL™ type.

Eample:

```
function zero () {
   return 0;
}
function doSomething(string s, number n1, number [] n2, boolean flag, string [] oneMore){
   ....
}
```

Constant Parameters

Parameters of user-defined routines can be made read-only in the scope of the routine by adding the keyword "const" before the parameter definition in the signature of the routine.

```
function f (const string s) {
   ...
}
```

Variable visibility

There are three categories of variables that can be used in a UDR:

Local variables

These are the variables you define in the body of the UDR. These can be used throughout the body of the UDR. On exit, the values of these variables are lost.

```
function example () {
   number a = 3;
   number b = a + 10;
   // use here variables a and b
}
```

Parameter variables

These are the values passed to the UDR in the list of parameters. Because SIL™ uses a "pass-by-value" policy, even though you modify the value of these variables in the body of the function, on exit, their original values will be restored.

```
function increment (number a) {
   a = a + 1; // the value of a is only modified locally
   return a;
}
number b = 0;
number c = increment(b); // the value of b does not change
print(b); // this prints 0
print(c); // this prints 1
```

Global variables

These are the variables that are already defined and can be used right away (issue fields, customfields and any variables defined before the routine).

You can use issue fields and custom fields anywhere in your code (including in the UDR body) without having to declare them.

```
function print Key () {
   print(key);
}
```

Return value

Return values can be used to communicate with the context that called the UDR or to halt its execution.

Examples

```
function isEven(number a){
   return (a % 2 == 0);
}
function increment (number a) {
   return a + 1;
}
number b = increment (2);
```

Notice that there is no need to declare the type of the return value; this will be evaluated at runtime.

Therefore, even though the check on the following program will be ok, at runtime the value of d will NOT be modified because of the incompatibility between date (on the right-hand-side) and number (on the left-hand-side).

```
function increment (number a) {
   return a + 1;
}
date d = increment (2);
```

You can return simply from a routine without specifying a value. However, you should always remember that by design routines return a value, even if it is undefined. The following code is therefore valid:

```
function f (number a) {
   if (a > 0) {
      print("positive");
      return;
   }
   if (a == 0) {print("ZERO");}
}
```

//[.....]

string s =f (4); //s is still undefined, no value was returned

if(isNull(s)) {

? print ("S IS NULL!"); //this will be printed

} else {

? print ("S IS NOT NULL!");

}

Of course, the above code will print the text 'S IS NULL' in the log.

5.8 Collection types

A collection is an ordered group of elements having the same data type. Each element is identified by a unique subscript that represents its position in the collection.

PL/SQL provides three collection types −

- Index-by tables or Associative array
- Nested table
- Variable-size array or Varray

Oracle documentation provides the following characteristics for each type of collections –

Collection Type	Number of Elements	Subscript Type	Dense or Sparse	Where Created	Can Be Object Type Attribute
Associative array (or index-by table)	Unbounded	String or integer	Either	Only in PL/SQL block	No
Nested table	Unbounded	Integer	Starts dense, can become sparse	Either in PL/SQL block or at schema level	Yes
Variablesize array (Varray)	Bounded	Integer	Always dense	Either in PL/SQL block or at schema level	Yes

Both types of PL/SQL tables, i.e., the index-by tables and the nested tables have the same structure and their rows are accessed using the subscript notation.

However, these two types of tables differ in one aspect; the nested tables can be stored in a database column and the index-by tables cannot.

Index-By Table

An index-by table (also called an associative array) is a set of key-value pairs. Each key is unique and is used to locate the corresponding value. The key can be either an integer or a string.

An index-by table is created using the following syntax. Here, we are creating an index-by table named table_name, the keys of which will be of the subscript_type and associated values will be of the element_type

TYPE type_name IS TABLE OF element_type [NOT NULL] INDEX BY subscript_type;

table_name type_name;

Example

Following example shows how to create a table to store integer values along with names and later it prints the same list of names.

```
DECLARE
  TYPE salary IS TABLE OF NUMBER INDEX BY VARCHAR2(20);
  salary_list salary;
  name   VARCHAR2(20);
BEGIN
  -- adding elements to the table
  salary_list('Rajnish') := 62000;
  salary_list('Minakshi') := 75000;
  salary_list('Martin') := 100000;
  salary_list('James') := 78000;
  -- printing the table
  name := salary_list.FIRST;
  WHILE name IS NOT null LOOP
    dbms_output.put_line
    ('Salary of ' || name || ' is ' || TO_CHAR(salary_list(name)));
    name := salary_list.NEXT(name);
  END LOOP;
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

Salary of James is 78000

Salary of Martin is 100000

Salary of Minakshi is 75000

Salary of Rajnish is 62000

PL/SQL procedure successfully completed.

Example

Elements of an index-by table could also be a **%ROWTYPE** of any database table or **%TYPE** of any database table field. The following example illustrates the concept. We will use the **CUSTOMERS** table stored in our database as −

```
Select * from customers;
```

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
+----+----------+-----+-----------+----------+
```

```
DECLARE
  CURSOR c_customers is
    select name from customers;
  TYPE c_list IS TABLE of customers.Name%type INDEX BY binary_integer;
  name_list c_list;
  counter integer:=0;
BEGIN
  FOR n IN c_customers LOOP
    counter:= counter +1;
    name_list(counter):= n.name;
    dbms_output.put_line('Customer('||counter||'):'||name_lis t(counter));
  END LOOP;
```

```
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
Customer (1): Ramesh
Customer (2): Khilan
Customer (3): kaushik
Customer (4): Chaitali
Customer (5): Hardik
Customer (6): Komal
PL/SQL procedure successfully completed
```

Nested Tables

A nested table is like a one-dimensional array with an arbitrary number of elements. However, a nested table differs from an array in the following aspects −

An array has a declared number of elements, but a nested table does not. The size of a nested table can increase dynamically.

An array is always dense, i.e., it always has consecutive subscripts. A nested array is dense initially, but it can become sparse when elements are deleted from it.

A nested table is created using the following **syntax** −

```
TYPE type_name IS TABLE OF element_type [NOT NULL];
table_name type_name;
```

This declaration is similar to the declaration of an index-by table, but there is no INDEX BY clause.

A nested table can be stored in a database column. It can further be used for simplifying SQL operations where you join a single-column table with a larger table. An associative array cannot be stored in the database.

Example

The following examples illustrate the use of nested table −

```
DECLARE
   TYPE names_table IS TABLE OF VARCHAR2(10);
   TYPE grades IS TABLE OF INTEGER;
```

```
   names names_table;
   marks grades;
   total integer;
BEGIN
   names := names_table('Kavita', 'Pritam', 'Ayan', 'Rishav', 'Aziz');
   marks:= grades(98, 97, 78, 87, 92);
   total := names.count;
   dbms_output.put_line('Total '|| total || ' Students');
   FOR i IN 1 .. total LOOP
      dbms_output.put_line('Student:'||names(i)||', Marks:' || marks(i));
   end loop;
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

```
Total 5 Students
Student:Kavita, Marks:98
Student:Pritam, Marks:97
Student:Ayan, Marks:78
Student:Rishav, Marks:87
Student:Aziz, Marks:92
PL/SQL procedure successfully completed.
```

Example

Elements of a **nested table** can also be a **%ROWTYPE** of any database table or %TYPE of any database table field. The following example illustrates the concept. We will use the CUSTOMERS table stored in our database as −

```
Select * from customers;
```

```
+----+----------+-----+-----------+----------+
| ID | NAME     | AGE | ADDRESS   | SALARY   |
+----+----------+-----+-----------+----------+
|  1 | Ramesh   |  32 | Ahmedabad |  2000.00 |
```

```
|  2 | Khilan   |  25 | Delhi     |  1500.00 |
|  3 | kaushik  |  23 | Kota      |  2000.00 |
|  4 | Chaitali |  25 | Mumbai    |  6500.00 |
|  5 | Hardik   |  27 | Bhopal    |  8500.00 |
|  6 | Komal    |  22 | MP        |  4500.00 |
+----+----------+-----+-----------+----------+
```

```
DECLARE
  CURSOR c_customers is
    SELECT  name FROM customers;
  TYPE c_list IS TABLE of customerS.No.ame%type;
  name_list c_list := c_list();
  counter integer :=0;
BEGIN
  FOR n IN c_customers LOOP
    counter := counter +1;
    name_list.extend;
    name_list(counter)  := n.name;
    dbms_output.put_line('Customer('||counter||'):'||name_list(counter));
  END LOOP;
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result −

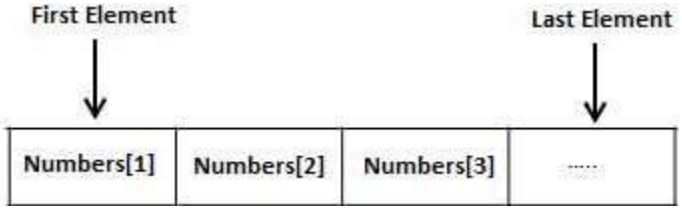
```
Customer(1): Ramesh
Customer(2): Khilan
Customer(3): kaushik
Customer(4): Chaitali
Customer(5): Hardik
Customer(6): Komal
PL/SQL procedure successfully completed.
```

Variable size array(Varray) type

The PL/SQL programming language provides a data structure called the VARRAY, which can store a fixed-size sequential collection of elements of the same type. A varray is used to store an

ordered collection of data, however it is often better to think of an array as a collection of variables of the same type.

All varrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



Varrays in PL/SQL

An array is a part of collection type data and it stands for variable-size arrays. We will study other collection types in a later chapter 'PL/SQL Collections'.

Each element in a varray has an index associated with it. It also has a maximum size that can be changed dynamically.

Creating a Varray Type

A varray type is created with the CREATE TYPE statement. You must specify the maximum size and the type of elements stored in the varray.

The basic syntax for creating a VARRAY type at the schema level is −

CREATE OR REPLACE TYPE varray_type_name IS VARRAY(n) of <element_type>

Where,

varray_type_name is a valid attribute name,

n is the number of elements (maximum) in the varray,

element_type is the data type of the elements of the array.

Maximum size of a varray can be changed using the ALTER TYPE statement.

For example,

CREATE Or REPLACE TYPE namearray AS VARRAY (3) OF VARCHAR2(10);

/

Type created.

The basic syntax for creating a VARRAY type within a PL/SQL block is −

TYPE varray_type_name IS VARRAY(n) of <element_type>

For example –

```
TYPE namearray IS VARRAY(5) OF VARCHAR2(10);
Type grades IS VARRAY(5) OF INTEGER;
```

Let us now work out on a few examples to understand the concept

Example 1

The following program illustrates the use of varrays

```
DECLARE
   type namesarray IS VARRAY(5) OF VARCHAR2(10);
   type grades IS VARRAY(5) OF INTEGER;
   names namesarray;
   marks grades;
   total integer;
BEGIN
   names := namesarray('Kavita', 'Pritam', 'Ayan', 'Rishav', 'Aziz');
   marks:= grades(98, 97, 78, 87, 92);
   total := names.count;
   dbms_output.put_line('Total '|| total || ' Students');
   FOR i in 1 .. total LOOP
      dbms_output.put_line('Student: ' || names(i) || '
      Marks: ' || marks(i));
   END LOOP;
END;
/
```

When the above code is executed at the SQL prompt, it produces the following result

```
Total 5 Students
Student: Kavita  Marks: 98
Student: Pritam  Marks: 97
Student: Ayan  Marks: 78
Student: Rishav  Marks: 87
```

Student: Aziz Marks: 92

PL/SQL procedure successfully completed.

5.9 Object Query Language; No-SQL: CAP theorem

CAP theorem

The CAP theorem is about how distributed database systems behave in the face of network instability.

When working with distributed systems over unreliable networks we need to consider the properties of consistency and availability in order to make the best decision about what to do when systems fail. The CAP theorem introduced by Eric Brewer in 2000 states that any distributed database system can have at most two of the following three desirable properties

Consistency: Consistency is about having a single, up-to-date, readable version of our data available to all clients. Our data should be consistent - no matter how many clients reading the same items from replicated and distributed partitions we should get consistent results. All writes are atomic and all subsequent requests retrieve the new value.

High availability: This property states that the distributed database will always allow database clients to make operations like select or update on items without delay. Internal communication failures between replicated data shouldn't prevent operations on it. The database will always return a value as long as a single server is running.

Partition tolerance: This is the ability of the system to keep responding to client requests even if there's a communication failure between database partitions. The system will still function even if network communication between partitions is temporarily lost.

Note that the **CAP theorem** only applies in cases when there's a connection failure between partitions in our cluster. The more reliable our network, the lower the probability we will need to think about this theorem. The CAP theorem helps us understand that once we partition our data, we must determine which options best match our business requirements: consistency or availability. Remember: at most two of the aforementioned three desirable properties can be fulfilled, so we have to select either consistency or availability.

5.10 MongoDB CRUD Operations

Data Model Design

Effective data models support your application needs. The key consideration for the structure of your documents is the decision to embed or to use references.

Embedded Data Models

With MongoDB, you may embed related data in a single structure or document. These schema are generally known as "denormalized" models, and take advantage of MongoDB's rich documents. Consider the following diagram:

Embedded data models allow applications to store related pieces of information in the same database record. As a result, applications may need to issue fewer queries and updates to complete common operations.

In general, use embedded data models when:

- you have "contains" relationships between entities. See Model One-to-One Relationships with Embedded Documents.
- you have one-to-many relationships between entities. In these relationships the "many" or child documents always appear with or are viewed in the context of the "one" or parent documents. See Model One-to-Many Relationships with Embedded Documents.

In general, embedding provides better performance for read operations, as well as the ability to request and retrieve related data in a single database operation. Embedded data models make it possible to update related data in a single atomic write operation.

To access data within embedded documents, use dot notation to "reach into" the embedded documents. See query for data in arrays and query data in embedded documents for more examples on accessing data in arrays and embedded documents.

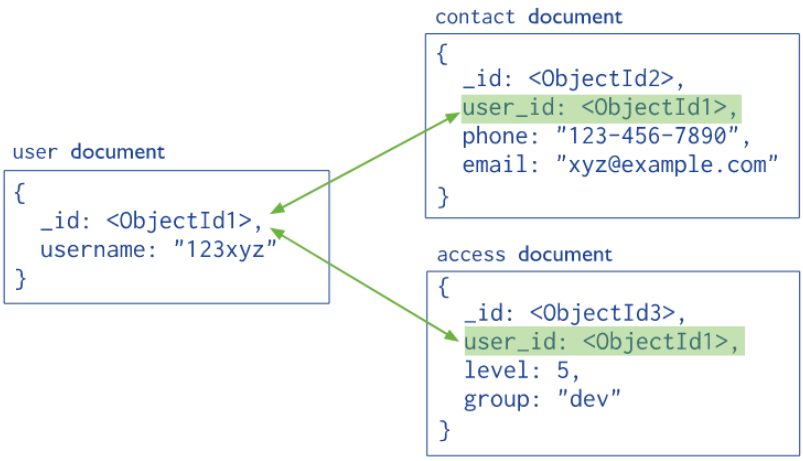
Embedded Data Model and Document Size Limit

Documents in MongoDB must be smaller than the maximum BSON document size.

For bulk binary data, consider GridFS.

Normalized Data Models

Normalized data models describe relationships using references between documents.



In general, use normalized data models:

when embedding would result in duplication of data but would not provide sufficient read performance advantages to outweigh the implications of the duplication.

to represent more complex many-to-many relationships.

to model large hierarchical data sets.

CRUD operations

CRUD operations create, read, update, and delete documents.

Create Operations: Create or insert operations add new documents to a collection. If the collection does not currently exist, insert operations will create the collection.

MongoDB provides the following methods to insert documents into a collection:

db.collection.insertOne()

db.collection.insertMany()

In MongoDB, insert operations target a single collection. All write operations in MongoDB are atomic on the level of a single document.

Read Operations: Read operations retrieve documents from a collection; i.e. query a collection for documents. MongoDB provides the following methods to read documents from a collection:

db.collection.find()

You can specify query filters or criteria that identify the documents to return.

Update Operations: Update operations modify existing documents in a collection. MongoDB provides the following methods to update documents of a collection:

db.collection.updateOne() New in version 3.2

db.collection.updateMany() New in version 3.2

db.collection.replaceOne() New in version 3.2

In MongoDB, update operations target a single collection. All write operations in MongoDB are atomic on the level of a single document.

You can specify criteria, or filters, that identify the documents to update. These filters use the same syntax as read operations.

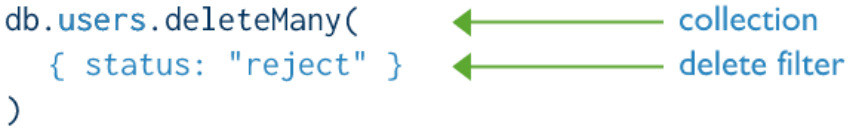
Delete Operations: Delete operations remove documents from a collection. MongoDB provides the following methods to delete documents of a collection:

db.collection.deleteOne() New in version 3.2

db.collection.deleteMany() New in version 3.2

In MongoDB, delete operations target a single collection. All write operations in MongoDB are atomic on the level of a single document.

You can specify criteria, or filters, that identify the documents to remove. These filters use the same syntax as read operations.



5.11 HBase Data Model and CRUD Operations

The HBase Data Model is designed to handle semi-structured data that may differ in field size, which is a form of data and columns. The data model's layout partitions the data into simpler components and spread them across the cluster. HBase's Data Model consists of various logical components, such as a table, line, column, family, column, column, cell, and edition.

Row Key	Customer		Sales	
Customer id	Name	City	Product	Amount
101	Ram	Delhi	Chairs	4000.00
102	Shyam	Lucknow	Lamps	2000.00
103	Gita	M.P	Desk	5000.00
104	Sita	U.K	Bed	2600.00

Table:

An HBase table is made up of several columns. The tables in HBase defines upfront during the time of the schema specification.

Row:

An HBase row consists of a row key and one or more associated value columns. Row keys are the bytes that are not interpreted. Rows are ordered lexicographically, with the first row appearing in a table in the lowest order. The layout of the row key is very critical for this purpose.

Column:

A column in HBase consists of a family of columns and a qualifier of columns, which is identified by a character: (colon).

Column Family:

Apache HBase columns are separated into the families of columns. The column families physically position a group of columns and their values to increase its performance.

Every row in a table has a similar family of columns, but there may not be anything in a given family of columns.

The same prefix is granted to all column members of a column family.

For **example,** Column courses: history and courses: math, are both members of the column family of courses.

The character of the colon (:) distinguishes the family of columns from the qualifier of the family of columns. The prefix of the column family must be made up of printable characters.

During schema definition time, column families must be declared upfront while columns are not specified during schema time.

They can be conjured on the fly when the table is up and running. Physically, all members of the column family are stored on the file system together.

Column Qualifier

The column qualifier is added to a column family. A column standard could be content (html and pdf), which provides the content of a column unit. Although column families are set up at table formation, column qualifiers are mutable and can vary significantly from row to row.

Cell:

A Cell store data and is quite a unique combination of row key, Column Family, and the Column. The data stored in a cell call its value and data types, which is every time treated as a byte [].

Timestamp:

In addition to each value, the timestamp is written and is the identifier for a given version of a number.

The timestamp reflects the time when the data is written on the Region Server. But when we put data into the cell, we can assign a different timestamp value.

CRUD Operations

1. **Create a data-Hbase**

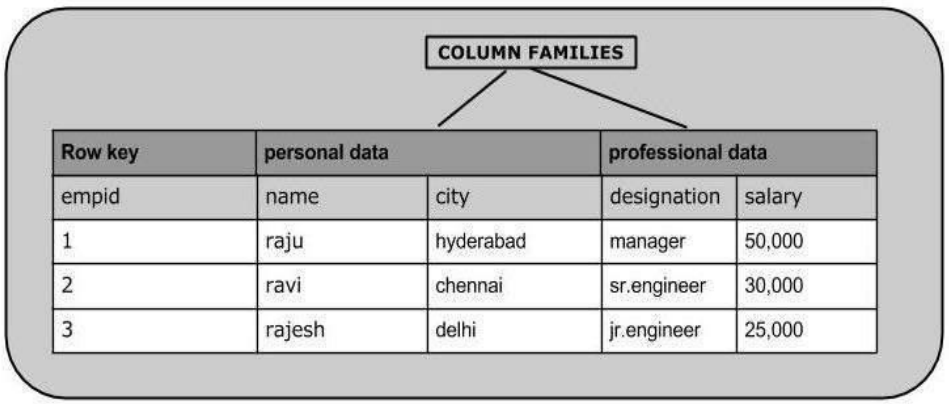
Inserting Data using HBase Shell- to create data in an HBase table. To create data in an HBase table, the following commands and methods are used:

put command,

add () method of **Put** class, and

put () method of **HTable** class.

As an example, we are going to create the following table in HBase.



Row key	personal data		professional data	
empid	name	city	designation	salary
1	raju	hyderabad	manager	50,000
2	ravi	chennai	sr.engineer	30,000
3	rajesh	delhi	jr.engineer	25,000

z

Using **put** command, you can insert rows into a table. Its syntax is as follows:

put'<table name>','row1','<colfamily:colname>','<value>'

Inserting the First Row

Let us insert the first-row values into the emp table as shown below.

hbase(main): 005:0> put 'emp','1','personal data:name','raju'

0 row(s) in 0.6600 seconds

hbase(main): 006:0> put 'emp','1','personal data:city','hyderabad'

0 row(s) in 0.0410 seconds

hbase(main): 007:0> put 'emp','1','professional

data:designation','manager'

0 row(s) in 0.0240 seconds

hbase(main): 007:0> put 'emp','1','professional data: salary','50000'

0 row(s) in 0.0240 seconds

Insert the remaining rows using the put command in the same way. If you insert the whole table, you will get the following output.

hbase(main): 022:0> scan 'emp'

ROW COLUMN+CELL

1 column=personal data:city, timestamp=1417524216501, value=hyderabad

1 column=personal data:name, timestamp=1417524185058, value=ramu

1 column=professional data:designation, timestamp=1417524232601,

value=manager

1 column=professional data:salary, timestamp=1417524244109, value=50000

2 column=personal data:city, timestamp=1417524574905, value=chennai

2 column=personal data:name, timestamp=1417524556125, value=ravi

2 column=professional data:designation, timestamp=1417524592204,

value=sr:engg

2 column=professional data:salary, timestamp=1417524604221, value=30000

3 column=personal data:city, timestamp=1417524681780, value=delhi

3 column=personal data:name, timestamp=1417524672067, value=rajesh

3 column=professional data:designation, timestamp=1417524693187,

value=jr:engg

3 column=professional data:salary, timestamp=1417524702514,

value=25000

Inserting Data Using Java API

You can insert data into Hbase using the add () method of the Put class. You can save it using the put () method of the HTable class. These classes belong to the org.apache.hadoop.hbase.client package. Below given are the steps to create data in a Table of HBase.

Step 1: Instantiate the Configuration Class

The Configuration class adds HBase configuration files to its object. You can create a configuration object using the create () method of the HbaseConfiguration class as shown below.

Configuration conf = HbaseConfiguration.create();

Step 2: Instantiate the HTable Class

You have a class called HTable, an implementation of Table in HBase. This class is used to communicate with a single HBase table. While instantiating this class, it accepts configuration object and table name as parameters. You can instantiate HTable class as shown below.

HTable hTable = new HTable(conf, tableName);

Step 3: Instantiate the PutClass

To insert data into an HBase table, the add () method and its variants are used. This method belongs to Put, therefore instantiate the put class. This class requires the row name you want to insert the data into, in string format. You can instantiate the Put class as shown below.

Put p = new Put (Bytes.toBytes("row1"));

Step 4: Insert Data

The add () method of Put class is used to insert data. It requires 3-byte arrays representing column family, column qualifier (column name), and the value to be inserted, respectively. Insert data into the HBase table using the add () method as shown below.

p.add(Bytes.toBytes("coloumn family "), Bytes.toBytes("column

name"), Bytes.toBytes("value"));

Step 5: Save the Data in Table

After inserting the required rows, save the changes by adding the put instance to the put () method of HTable class as shown below.

hTable.put(p);

Step 6: Close the HTable Instance

After creating data in the HBase Table, close the HTable instance using the close () method as shown below.

hTable.close();

Given below is the complete program to create data in HBase Table.

```
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.client.HTable;
import org.apache.hadoop.hbase.client.Put;
import org.apache.hadoop.hbase.util.Bytes;
public class InsertData{
  public static void main (String [] args) throws IOException {
    // Instantiating Configuration class
    Configuration config = HBaseConfiguration.create();
    // Instantiating HTable class
    HTable hTable = new HTable(config, "emp");
    // Instantiating Put class
    // accepts a row name.
    Put p = new Put (Bytes.toBytes("row1"));
     // adding values using add () method
```

```
    // accepts column family name, qualifier/row name, value
    p.add(Bytes.toBytes("personal"),
    Bytes.toBytes("name"), Bytes.toBytes("raju"));
    p.add(Bytes.toBytes("personal"),
    Bytes.toBytes("city"), Bytes.toBytes("hyderabad"));
    p.add (Bytes.toBytes("professional"), Bytes.toBytes("designation"),
    Bytes.toBytes("manager"));
    p.add(Bytes.toBytes("professional"),Bytes.toBytes("salary"),
    Bytes.toBytes("50000"));
    // Saving the put Instance to the HTable.
    hTable.put(p);
    System.out.println("data inserted");
    // closing HTable
    hTable.close();
  }
}
```

Compile and execute the above program as shown below.

$javac InsertData.java

$java InsertData

The following should be the output:

data inserted

2. Updating Data using HBase Shell

You can update an existing cell value using the put command. To do so, just follow the same syntax and mention your new value as shown below.

put 'table name','row','Column family:column name','new value'

The newly given value replaces the existing value, updating the row.

Example

Suppose there is a table in HBase called emp with the following data.

hbase(main): 003:0> scan 'emp'

ROW COLUMN + CELL

row1 column = personal:name, timestamp = 1418051555, value = raju

row1 column = personal:city, timestamp = 1418275907, value = Hyderabad

row1 column = professional:designation, timestamp = 14180555,value = manager

row1 column = professional:salary, timestamp = 1418035791555,value = 50000

1 row(s) in 0.0100 seconds

The following command will update the city value of the employee named 'Raju' to Delhi.

hbase(main): 002:0> put 'emp','row1','personal: city','Delhi'

0 row(s) in 0.0400 seconds

The updated table looks as follows where you can observe the city of Raju has been changed to 'Delhi'.

hbase(main): 003:0> scan 'emp'

ROW COLUMN + CELL

row1 column = personal:name, timestamp = 1418035791555, value = raju

row1 column = personal:city, timestamp = 1418274645907, value = Delhi

row1 column = professional:designation, timestamp = 141857555,value = manager

row1 column = professional:salary, timestamp = 1418039555, value = 50000

1 row(s) in 0.0100 seconds

Updating Data Using Java API

You can update the data in a particular cell using the put () method. Follow the steps given below to update an existing cell value of a table.

Step 1: Instantiate the Configuration Class

Configuration class adds HBase configuration files to its object. You can create a configuration object using the create () method of the HbaseConfiguration class as shown below.

Configuration conf = HbaseConfiguration.create();

Step 2: Instantiate the HTable Class

You have a class called HTable, an implementation of Table in HBase. This class is used to communicate with a single HBase table. While instantiating this class, it accepts the onfiguration object and the table name as parameters. You can instantiate the HTable class as shown below.

HTable hTable = new HTable(conf, tableName);

Step 3: Instantiate the Put Class

To insert data into HBase Table, the add () method and its variants are used. This method belongs to Put, therefore instantiate the put class. This class requires the row name you want to insert the data into, in string format. You can instantiate the Put class as shown below.

Put p = new Put (Bytes.toBytes("row1"));

Step 4: Update an Existing Cell

The add () method of Put class is used to insert data. It requires 3-byte arrays representing column family, column qualifier (column name), and the value to be inserted, respectively. Insert data into HBase table using the add () method as shown below.

p.add(Bytes.toBytes("coloumn family "), Bytes.toBytes("column

name"),Bytes.toBytes("value"));

p.add(Bytes.toBytes("personal"),

Bytes.toBytes("city"),Bytes.toBytes("Delih"));

Step 5: Save the Data in Table

After inserting the required rows, save the changes by adding the put instance to the put () method of the HTable class as shown below.

hTable.put(p);

Step 6: Close HTable Instance

After creating data in HBase Table, close the HTable instance using the close () method as shown below.

hTable.close();

Given below is the complete program to update data in a particular table.

import java.io.IOException;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.hbase.HBaseConfiguration;

import org.apache.hadoop.hbase.client.HTable;

import org.apache.hadoop.hbase.client.Put;

import org.apache.hadoop.hbase.util.Bytes;

public class UpdateData{

```
  public static void main (String [] args) throws IOException {
    // Instantiating Configuration class
    Configuration config = HBaseConfiguration.create();
    // Instantiating HTable class
    HTable hTable = new HTable(config, "emp");
    // Instantiating Put class
    //accepts a row name
    Put p = new Put (Bytes.toBytes("row1"));
    // Updating a cell value
    p.add(Bytes.toBytes("personal"),
    Bytes.toBytes("city"), Bytes.toBytes("Delih"));
    // Saving the put Instance to the HTable.
    hTable.put(p);
    System.out.println("data Updated");
    // closing HTable
    hTable.close();
  }
}
```

Compile and execute the above program as shown below.

```
$javac UpdateData.java
$java UpdateData
```

The following should be the output:

```
data Updated
```

3. Reading Data using HBase Shell

The get commands and the get () method of HTable class are used to read data from a table in HBase. Using get command, you can get a single row of data at a time. Its syntax is as follows:

```
get'<table name>','row1'
```

Example

The following example shows how to use the get command. Let us scan the first row of the emp table.

hbase(main): 012:0> get 'emp', '1'

COLUMN CELL

personal: city timestamp = 1417521848375, value = hyderabad

personal: name timestamp = 1417521785385, value = ramu

professional: designation timestamp = 1417521885277, value = manager

professional: salary timestamp = 1417521903862, value = 50000

4 row(s) in 0.0270 seconds

Reading a Specific Column

Given below is the syntax to read a specific column using the get method.

hbase> get 'table name', 'rowid', {COLUMN ⇒ 'column family:column name '}

Example

Given below is the example to read a specific column in HBase table.

hbase(main): 015:0> get 'emp', 'row1', {COLUMN ⇒ 'personal:name'}

COLUMN CELL

personal:name timestamp = 1418035791555, value = raju

1 row(s) in 0.0080 seconds

Reading Data Using Java API

To read data from an HBase table, use the get () method of the HTable class. This method requires an instance of the Get class. Follow the steps given below to retrieve data from the HBase table.

Step 1: Instantiate the Configuration Class

Configuration class adds HBase configuration files to its object. You can create a configuration object using the create () method of the HbaseConfiguration class as shown below.

Configuration conf = HbaseConfiguration.create();

Step 2: Instantiate the HTable Class

You have a class called HTable, an implementation of Table in HBase. This class is used to communicate with a single HBase table.

While instantiating this class, it accepts the configuration object and the table name as parameters. You can instantiate the HTable class as shown below.

HTable hTable = new HTable(conf, tableName);

Step 3: Instantiate the Get Class

You can retrieve data from the HBase table using the get () method of the HTable class. This method extracts a cell from a given row. It requires a Get class object as parameter. Create it as shown below.

Get get = new Get(toBytes("row1"));

Step 4: Read the Data

While retrieving data, you can get a single row by id, or get a set of rows by a set of row ids, or scan an entire table or a subset of rows.

You can retrieve an HBase table data using the add method variants in Get class.

To get a specific column from a specific column family, use the following method.

get.addFamily(personal)

To get all the columns from a specific column family, use the following method.

get.addColumn(personal, name)

Step 5: Get the Result

Get the result by passing your Get class instance to the get method of the HTable class. This method returns the Result class object, which holds the requested result. Given below is the usage of get () method.

Result result = table.get(g);

Step 6: Reading Values from the Result Instance

The Result class provides the getValue() method to read the values from its instance. Use it as shown below to read the values from the Result instance.

byte [] value = result. getValue(Bytes.toBytes("personal"),Bytes.toBytes("name"));

byte [] value1 = result. getValue(Bytes.toBytes("personal"),Bytes.toBytes("city"));

Given below is the complete program to read values from an HBase table.

import java.io.IOException;

import org.apache.hadoop.conf.Configuration;

import org.apache.hadoop.hbase.HBaseConfiguration;

import org.apache.hadoop.hbase.client.Get;

```
import org.apache.hadoop.hbase.client.HTable;
import org.apache.hadoop.hbase.client.Result;
import org.apache.hadoop.hbase.util.Bytes;
public class RetriveData{
  public static void main (String [] args) throws IOException, Exception {
    // Instantiating Configuration class
    Configuration config = HBaseConfiguration.create();
    // Instantiating HTable class
    HTable table = new HTable(config, "emp");
    // Instantiating Get class
    Get g = new Get (Bytes.toBytes("row1"));
    // Reading the data
    Result result = table.get(g);
    // Reading values from Result class object
    byte [] value = result. getValue(Bytes.toBytes("personal"),Bytes.toBytes("name"));
    byte [] value1 = result. getValue(Bytes.toBytes("personal"),Bytes.toBytes("city"));
    // Printing the values
    String name = Bytes.toString(value);
    String city = Bytes.toString(value1);
    System.out.println("name: " + name + " city: " + city);
  }
}
```

Compile and execute the above program as shown below.

```
$javac RetriveData.java
$java RetriveData
```

The following should be the output:

```
name: Raju city: Delhi
```

Deleting a Specific Cell in a Table

Using the delete command, you can delete a specific cell in a table. The syntax of delete command is as follows:

delete '<table name>', '<row>', '<column name >', '<time stamp>'

Example

Here is an example to delete a specific cell. Here we are deleting the salary.

hbase(main): 006:0> delete 'emp', '1', 'personal data:city',

1417521848375

0 row(s) in 0.0060 seconds

Deleting All Cells in a Table

Using the "deleteall" command, you can delete all the cells in a row. Given below is the syntax of deleteall command.

deleteall '<table name>', '<row>',

Example

Here is an example of "deleteall" command, where we are deleting all the cells of row1 of emp table.

hbase(main): 007:0> deleteall 'emp','1'

0 row(s) in 0.0240 seconds

Verify the table using the scan command. A snapshot of the table after deleting the table is given below.

hbase(main): 022:0> scan 'emp'

ROW COLUMN + CELL

2 column = personal data:city, timestamp = 1417524574905, value = chennai

2 column = personal data:name, timestamp = 1417524556125, value = ravi

2 column = professional data:designation, timestamp = 1417524204, value = sr:engg

2 column = professional data:salary, timestamp = 1417524604221, value = 30000

3 column = personal data:city, timestamp = 1417524681780, value = delhi

3 column = personal data:name, timestamp = 1417524672067, value = rajesh

3 column = professional data:designation, timestamp = 1417523187, value = jr:engg

3 column = professional data:salary, timestamp = 1417524702514, value = 25000

4. Deleting Data Using Java API

You can delete data from an HBase table using the delete () method of the HTable class. Follow the steps given below to delete data from a table.

Step 1: Instantiate the Configuration Class

Configuration class adds HBase configuration files to its object. You can create a configuration object using the create () method of the the HbaseConfiguration class as shown below.

Configuration conf = HbaseConfiguration.create();

Step 2: Instantiate the HTable Class

You have a class called HTable, an implementation of Table in HBase. This class is used to communicate with a single HBase table. While instantiating this class, it accepts the configuration object and the table name as parameters. You can instantiate the HTable class as shown below.

HTable hTable = new HTable(conf, tableName);

Step 3: Instantiate the Delete Class

Instantiate the Delete class by passing the rowid of the row that is to be deleted, in byte array format. You can also pass timestamp and Rowlock to this constructor.

Delete delete = new Delete(toBytes("row1"));

Step 4: Select the Data to be Deleted

You can delete the data using the delete methods of the Delete class. This class has various delete methods. Choose the columns or column families to be deleted using those methods. Take a look at the following examples that show the usage of Delete class methods.

delete.deleteColumn(Bytes.toBytes("personal"), Bytes.toBytes("name"));

delete.deleteFamily(Bytes.toBytes("professional"));

Step 5: Delete the Data

Delete the selected data by passing the delete instance to the delete () method of the HTable class as shown below.

table.delete(delete);

Step 6: Close the HTableInstance

After deleting the data, close the HTable Instance.

table.close();

Given below is the complete program to delete data from the HBase table.

```
import java.io.IOException;
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.HBaseConfiguration;
import org.apache.hadoop.hbase.client.Delete;
import org.apache.hadoop.hbase.client.HTable;
import org.apache.hadoop.hbase.util.Bytes;
public class DeleteData {
  public static void main (String [] args) throws IOException {
    // Instantiating Configuration class
    Configuration conf = HBaseConfiguration.create();
    // Instantiating HTable class
    HTable table = new HTable(conf, "employee");
    // Instantiating Delete class
    Delete delete = new Delete (Bytes.toBytes("row1"));
    delete.deleteColumn(Bytes.toBytes("personal"), Bytes.toBytes("name"));
    delete.deleteFamily(Bytes.toBytes("professional"));
    // deleting the data
    table.delete(delete);
    // closing the HTable object
    table.close();
    System.out.println("data deleted.....");
  }
 }
```

Compile and execute the above program as shown below.

```
$javac Deletedata.java
$java DeleteData
```

The following should be the output: ***data deleted***