AD3391 : Database Design and management: (1)

UNIT: 1 - Conceptual Data Modeling:

Database Environment - Database System development Lifecycle - Requirement collection - Database Design - Entity-Relationship model - Enhanced ER model - UML Class diagrams

Introduction:

Data:

* collection of raw facts and figures which can be processed to provide information.
* Data can be represented in the form of numbers and words which can be stored in computer language.

Data base:

* collection of interrelated data which can be stored in the form of tables.
* A database can be of any size and varying complexity
* The database relation can be easily accessed, managed and updated.

Student

Name	Roll-No	Class	Department
Ramesh	44	II	AIDS
Kumar	45	II	AIDS

DBMS:

* A database management system is a collection of interrelated data and a set of programs to access those data.

* The collection of data is usually referred to as the database

* The primary goal of a DBMS is to provide a way to store and retrieve database information that is both convenient and efficient

Characteristics of a good database:

* It should be able to store all kinds of data which exists in this real world

* It should be able to related the entities/tables in the database by means of a relationship

ie: Any two tables should be related.

* There should not be unnecessary waste of DB space [Less Redundancy required]

* DBMS has a strong query Language. Once the database is designed, it helps the user to retrieve and manipulate the data.

* Concurrency: multiple user should be able to access the same database simultaneously, without affecting the other users.

* It supports multiple views to the user, depending on his role.

* Database should also provide security at different levels.

* Database should support ACID property. [Atomicity, Consistency, Isolation & Durability]

Purpose of database systems:

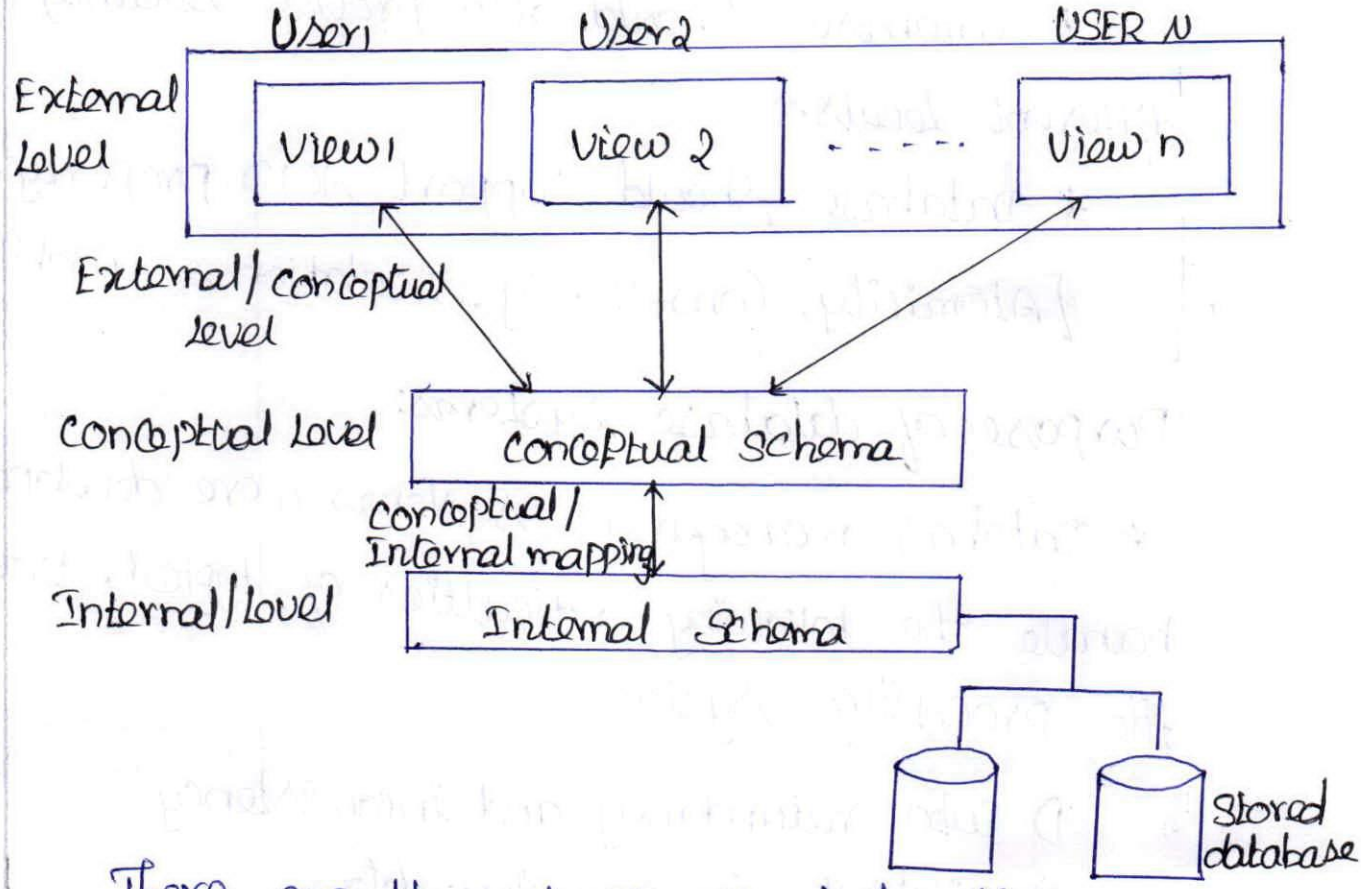
* Databas management systems were developed to handle the following difficulties of typical, traditional file-processing systems.

1) Data redundancy and inconsistency
2) Difficulty in accessing data
3) Data isolation
4) Integrity problems
5) Atomicity problems
6) Concurrent-access anomalies
7) Security problems

Views of Data / ANSI-SPARC Three level Architecture:

Database are made up of complex data structures. To ease the user interaction with database, the developer hide internal complex details from users.

* This process of hiding complex details from user is called data abstraction.



There are three levels of abstraction

physical level / Internal level:

* This is the lowest level of data abstraction, It describes how data is actually stored in database. you can get the complex data structure details at this level.

[The process of transforming requests and results between level are called mapping].

View level [External Level] :

* This is the highest level of data abstraction. This level describes the user interaction

Logical level [conceptual level]

* This is the middle level of 3level data abstraction architecture. It describes what data is stored in database.

Example:

* Let's say we are storing customer information in a customer table. At physical level, these records can be described as block of storage (bytes, gigabytes, terabytes etc) in memory. These details are often hidden from the programmers.

* At the Logical level, these records can be described as fields and attributes along with their data types, their relationship among each other can be logically implemented

* At view level, the user interact with the system with the help of GUI and enter the details at the screen. They are not aware of the how the data is stored and what data is stored. Such details are hidden from them.

Data models:

* A data model provides a way to describe the design of a database at the physical, logical and view levels. The data models can be classified into four different categories:

* Relational model
* Entity Relationship model
* object-based data model
* Semi structured data model

Relational model

* The relational model uses a collection of tables to represent both data and the relationships among those data. Each table has multiple columns and each column has a unique name. Tables are also known as relations record-based model. Record based is structured in fixed-format record of several types.

* Using certain integrity rules, two different tables can be related with each other using a common field in these tables. In relational model, the relational information can be retrived by relating a data in one table with other table.

Entity Relationship model:

* The entity-relationship (E-R) model uses a collection of entities and relationship among

these tables. The ER model is widely used in database design. It helps you to analyze data requirements systematically to produce a well-designed database.

Object-based data model:

* It object based data models, the focus is on how data is represented. The data is divided into multiple entities each of which have defining characteristics. moreover, those data entities are connected with each other through relationships.

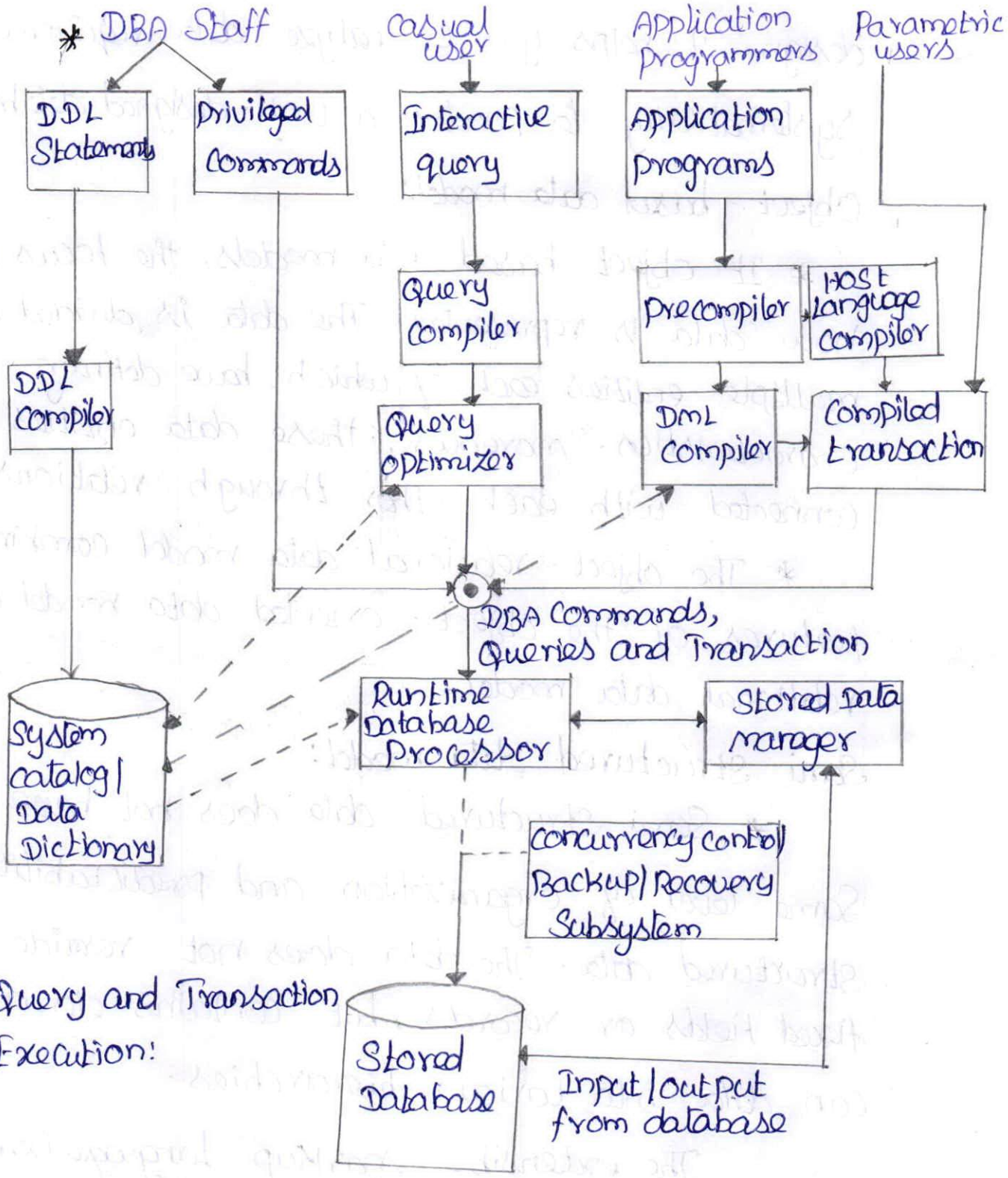
* The object-relational data model combines the features of the object-oriented data model and relational data model.

Semi structured data model:

* Semi structured data does not have the same level of organization and predictability of structured data. The data does not reside in fixed fields or records, but contains element that can data into various hierarchies.

The extensible markup language (xml) is widely used to represent Semi-structured data.

Database Architecture



The above figure illustrate the simplified form of the typical DBMS components. The figure is divided into two levels. The top half of the figures refers to various users of database environment and

* The Lower half shows the internal Structures of DBMS, which is responsible for Storage of data and processing of transactions.

* The database and the DBMS catalog are usually Stored on hard disk. Access to disk is primarily controlled by the operating System.

Stored Data manager module:

* It controls the access to DBMS information on disk at a high level.

DBA Staff:

* They work on defining the dB and tunning it by making changes to its definition using DDL and other privileged commands.

DDL Compiler:

* It process schema definitions specified in DDL and stores the meta-data in the DBMS catalog.

casual users

* They interact with some interactive query interface for the need of information.

Query compiler:

* It parses and analyses the queries generated by the casual users using query interface for correctiveness. Then the compiler compiles it to an

Internal form which is understandable by the DBMS.

Query optimizer:

* It attempts to determine the most efficient way to execute a given query by eliminating the redundancies and using of correct algorithm and indexes during execution.

Application programmers:

* They write programs is host languages such as Java, C which will be submitted to a precompiler.

Precompiler:

* It converts the source program into its appropriate SQL Function calls for DBMS.

DML Compiler:

* It converts the SQL commands from precompiler to an internal form of DBMS. Insertion, Deletion, modification commands are handled here.

Host Language Compiler [Data Sub Language]

* After extracting the SQL functions using precompiler, the rest of the programs is sent to the host language compiler. It compiles the rest into its equivalent object code. [Low level instructions]

* The DML Commands and the rest of the program are linked, forming a combined transaction to be processed by a runtime database processor.

Parametric users:

* They are the combined transaction by supplying parameters. These parameters are called runtime parameters.

(Eg:) Bank withdrawal transaction, where the account number and the amount are supplied as parameters.

Runtime Database processor:

* It executes the DBA commands, queries and combined transactions with parameters. It works with the system dictionary for updation.

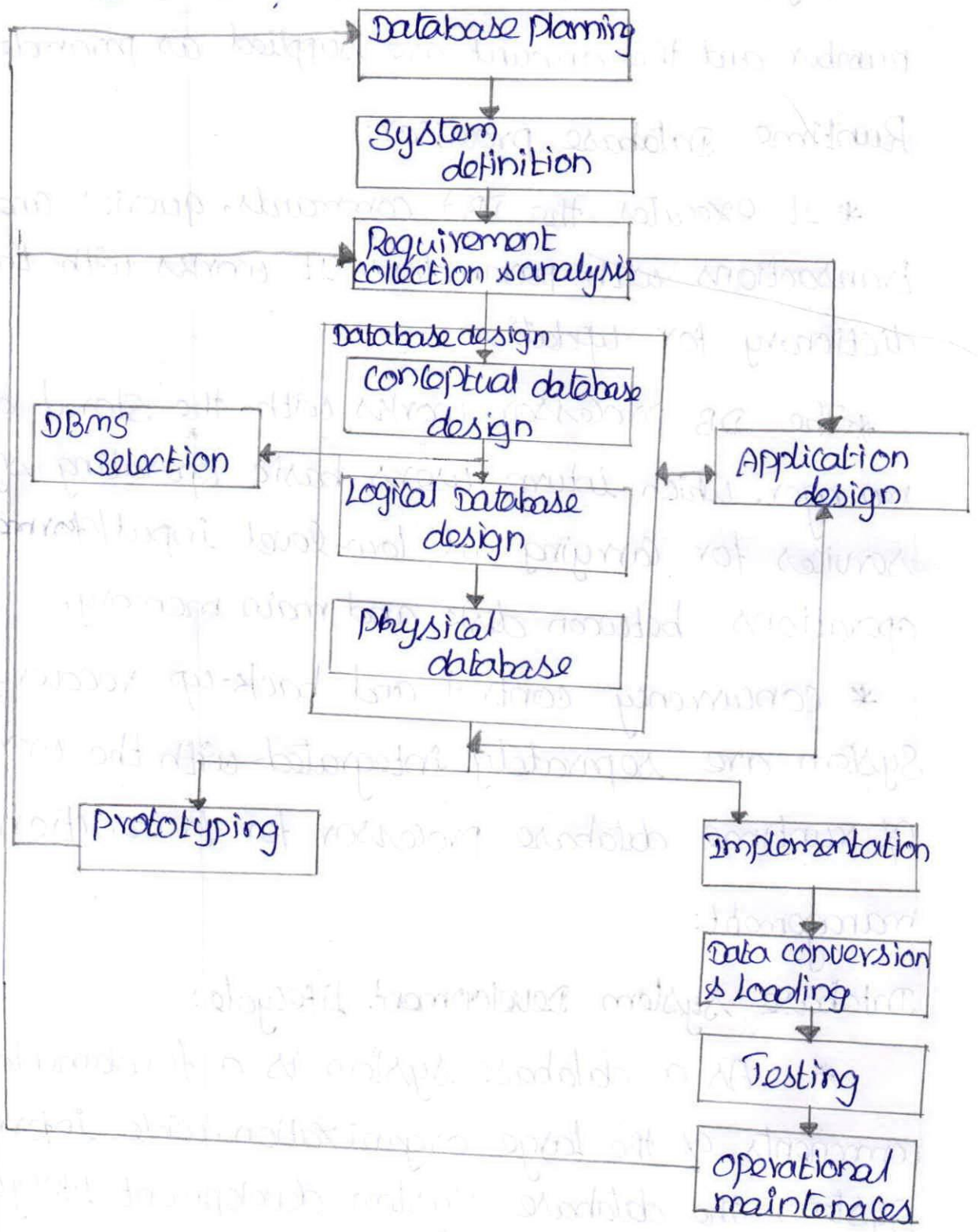
* The DB processor works with the stored data manager, which inturn uses basic operating system services for carrying out low-level input/output operations between disk and main memory.

* concurrency control and back-up recovery system are separately integrated with the working of runtime database processor for transaction management:

Database system Development Lifecycle:

* As a database system is a fundamental components of the large organization-wide information system, the database system development lifecycle is inherently associated with the life cycle of the

Information System : The resources that enables the collection, management, control and dissemination of information throughout an organization.



* It is important to recognize that the stage of the database system development lifecycle are not strictly sequential, but involve some amount of repetition of previous stages through feedback loops.

Database planning:

* Planning how the stages of the lifecycle can be realized most effectively and efficiently.

* There are three main issues involved in:

↳ Identification of enterprise plan and goal with subsequent determination of information system needs;

↳ Evaluation of current information systems to determain existing strengths and weakness.

↳ Appraisal of IT opportunities that might yield competitive advantage.

System definition:

* Specifying the scope and boundaries of the database system, including the major user views, its users, and application areas.

Requirements collection and analysis:

Collection and analysis of the requirements for the new database system.

Database design:

* conceptual, logical and physical design of the database.

DBMS Collection:

* Selecting a suitable DBMS for the database system.

Application design:

* Designing the user interface and the application programs that use and process the database.

Prototyping:

* Building a working model of the database system which allows the designs or users to visualize and evaluate how the final system will look and function

Implementation: Creating the physical database definitions and the application programs.

Data conversion and Loading: Loading data from the old system and where possible, converting any existing application to run on the new database.

Testing: Database system is tested for errors and validated against the requirements specified by the users.

Operational maintainance: Database is fully implemented The system is continuously monitored and maintained when nessary, new requirements are incorporated into the database system. Through the preceding stages of the lifecycle.

Requirements collection and Analysis:

* The process of collecting and analysising information about the part of the organization that is to be supported by the database system, and using this information to identify the requirments for the new system.

* This stage involves the collection and analysis of information about the part of the enterprise to be served by the database. There are many techniques for gathering this information called fact-finding techniques. Information is gathered for each major user view, including:

↳ a description of the data used or generated

↳ The details of how data is to be used or generated.

↳ Any additional requirements for the new database system.

* This information is then analyzed to identify the requirements to be included in the new database system. This requirements are described in documents collectively referred to as requirements specifications for the new database system.

* Requirements collection and analysis is a preliminary stage to database design. The amount of data gathered depends on the nature of the problem and the policies of the paralysis by analysis. Too little thought can result in a unnecessary waste of both time and money due to working on the wrong solution to the wrong problem.

* There are three main approaches to managing the requirements of a database system with multiple user view.

  - The centralized approach.
  - The view integration approach
  - A combination of both approach.

Centralized Approach:

* Requirements for each view are merged into a single set of requirements for the new database system. A data model representation setting all user views is created during the database design

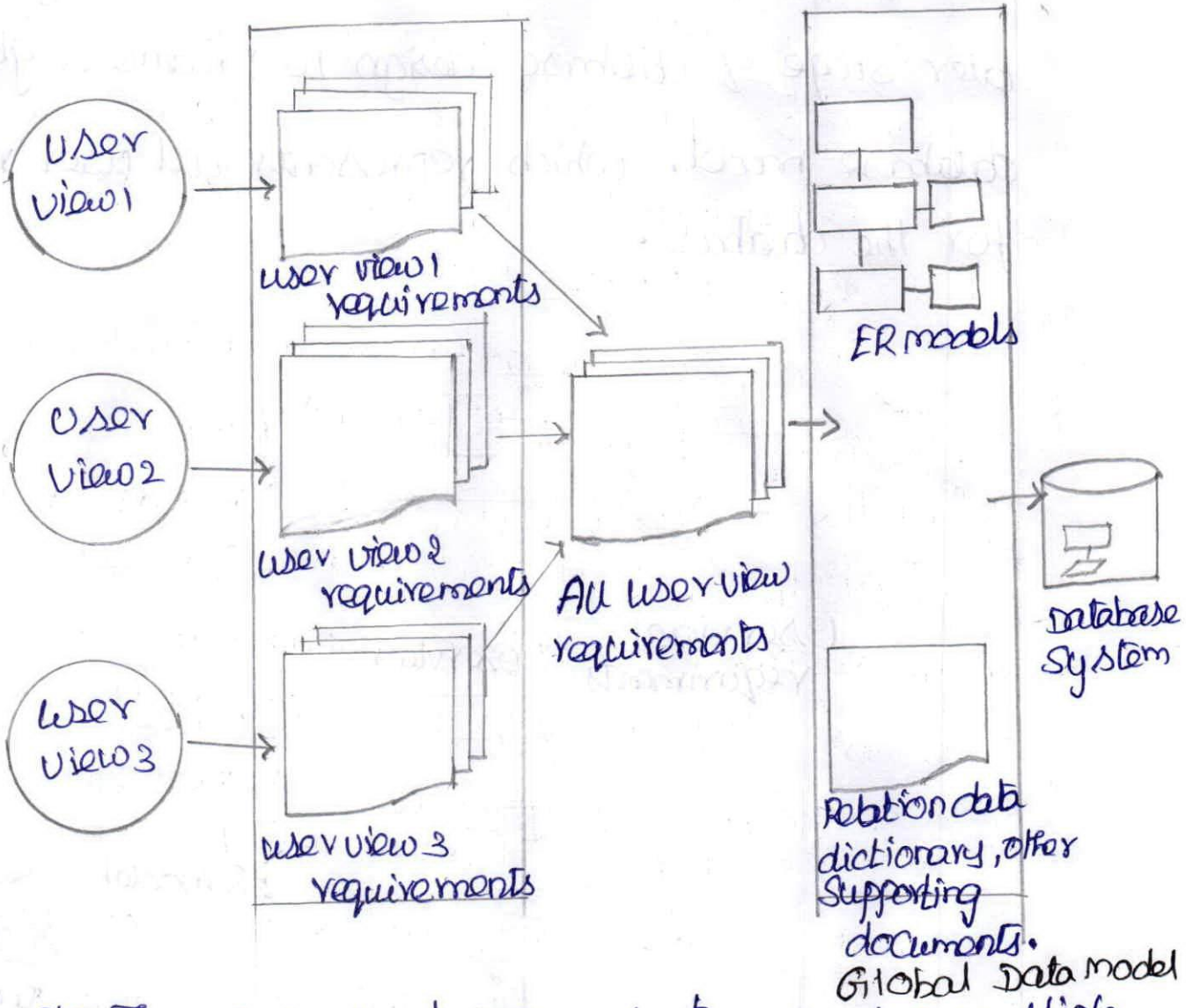


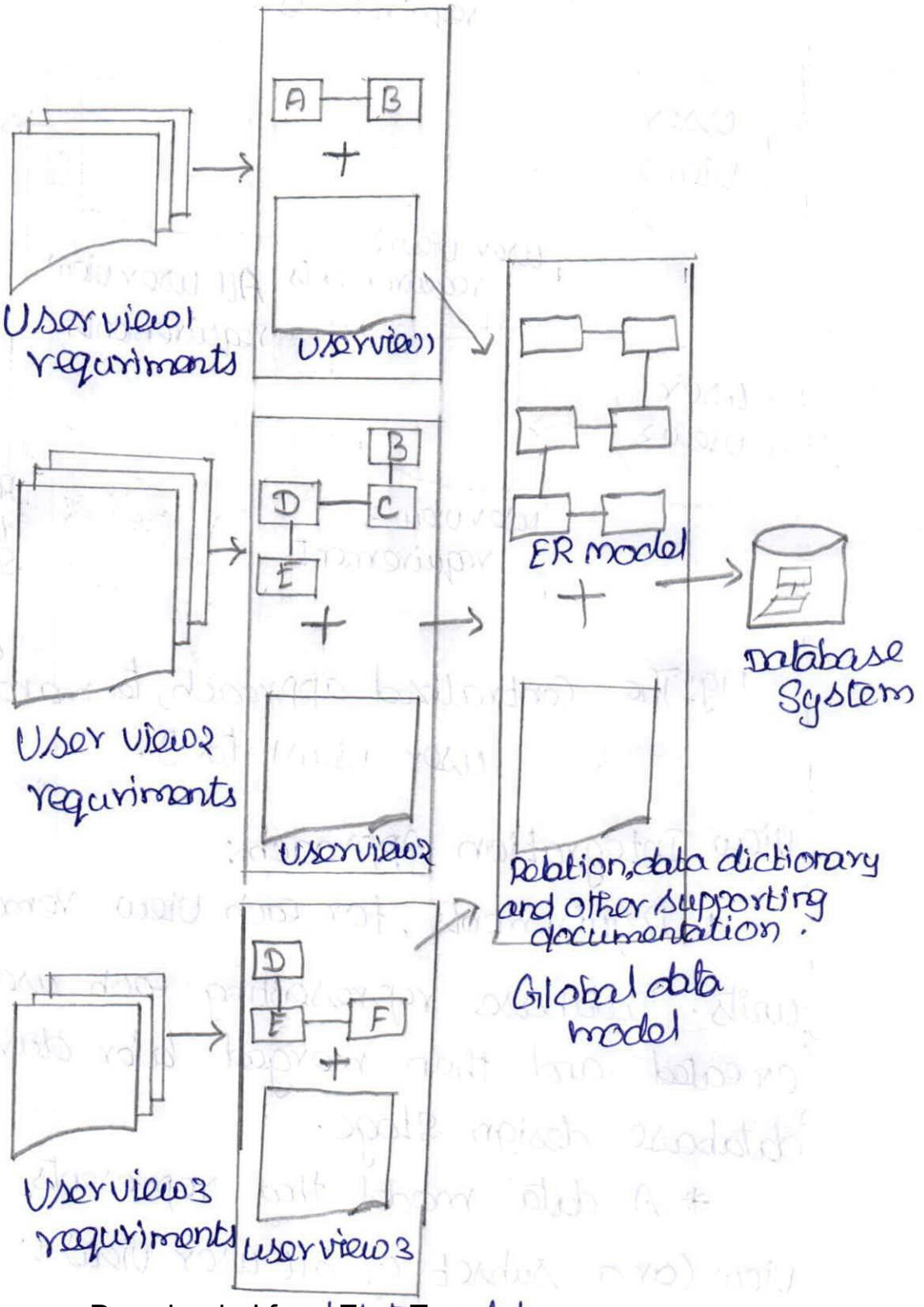
Fig: The Centralized approach to managing multiple user view1 to 3.

View Integration Approach:

* Requirements for each view remain as seperate units. Database representing each user view are created and then merged later during the database design stage.

* A data model that represents a single user view (or a subset of all user view is called a

* The local data models are then merged at a later stage of database design to produce a global database model, which represents all user requirements for the database.



* For some complex database system, it may be appropriate to use a "combination of both the centralized and view integration approaches" to manages multiple user view.

For Example: The requirements for two or more user views may be first merged using centralized approach, which is used to build a logical Local Logical data model.

* This model can be merged with other Local Logical data model using the view integration approach to produce a global logical data model.

* In this case, each Local Logical data model represents the requirements of two or more user views and the final global Logical data model represents the requirments of all user views of the database systems.

Database design:

* The process of creating a design that will support the enterprise's mission statement and mission objectives for the required database system.

Approaches to Database Design:

* The two main approaches to the design of a database are referred to as "bottom up" and

and topdown".

* The bottom up approach begins at the fundamental level of attributes, which through analysis of the associations between attributes are grouped into relations that represent types of entities and relationships between entities.

* The bottom-up approach is appropriate for the design of simple databases with a relatively small number of attributes. However, this approach becomes difficult when applied to the design of more complex database with large number of attributes, where it is difficult to establish all the functional dependencies between the attributes.

* As the conceptual and logical datamodels for complex database may contain hundreds to thousands of attributes, it is essential to establish an approach that will simplify the design process. Also, in the initial stages of establishing the data requirements for a complex database, it may be difficult to establish all the attributes tobe included in the data models.

* A more appropriate strategy for the design of complex database isto use the top-down approach.

* This approach starts with the development of data models that contains a few high-level entities and relationships and then applies successive top down refinements to identify lower-level entities, relationships and the associated attributeds.

* The top-down approach is illustrated using the concepts of the Entity-Relationship (ER) model, beginning with the identification of entites and relationship between the entities, which are of interest to the organization.

* There are other approaches to database design, such as the inside-out approach and the mixed strategy approach. The inside-out approach is related to the bottomup approach, but differs by first identifying a set of major entities and then spreading out to consider other entities, relationships and attributes associated with those first identified.

* The mixed strategy approach uses both the bottomup and top-down approach for various parts of the model before finally combining all parts together.

Data modeling:

* The two main purpose of data modeling are in the understanding of the meaning of the data and to facilitate communication about the information requirements.

* Building a data model requires answering questions about entities, relationships, and attributes. In doing so, the designers discover the semantics of the enterprise's data which exists whether or not they happen to be recorded in a formal data model.

* A data model makes it easier to understand the meaning of the data and thus we model data to ensure that we understand:

  - Ls each user's perspective of the data.
  - Ls The nature of the data itself independent of its physical representations.

* Data models can be used to convey the designer's understanding of the information requirements of the enterprise.

* provided both parties are familiar with the notation used in the model it will support communication between end users and designers.

Criteria for data models: An optimal data model should satisfy the criteria listed in the below table:-

Table: This criteria to produce an optimal data model.

Structural Validity →	consistency with the way the enterprise defines and organize information.
Simplicity →	Ease of understanding by IS professionals and non-technical users
Expressibility →	Ability to distinguish between different data, relationships between data and constraints
Non redundancy →	Exclusion of extraneous information; in particular, the representation of any one piece of information exactly once.
Sharing →	Not specific to any particular application or technology and thereby usable by many.
Extensibility →	Ability to evolve to support new requirements with minimal effect on existing users.
Integrity →	Consistency with the way the enterprise uses and manages information.
Diagrammatic representation →	ability to represent a model using an easily understood diagrammatic notation.

Entity Relationship model : [ER model]

* ER model is a high level conceptual data model diagram. Entity - Relationship model is based on the notation of real-world entities and the relationship between them.

* ER modeling helps you to analyze data requirements systematically to produce a well-designed database. So, it is considered a best practice to complete ER modeling before implementing your database.

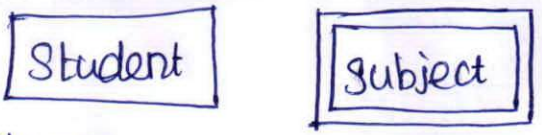
ER Diagrams:

* ER diagrams displays the relationships of entity set stored in a database. ER diagrams helps you to explain the logical structure of database. ER diagram includes special symbols, and its meanings make this model uniques

Components of ER models:

Entity, Attributes, Relationships etc from the components of E-R diagram.

Entity

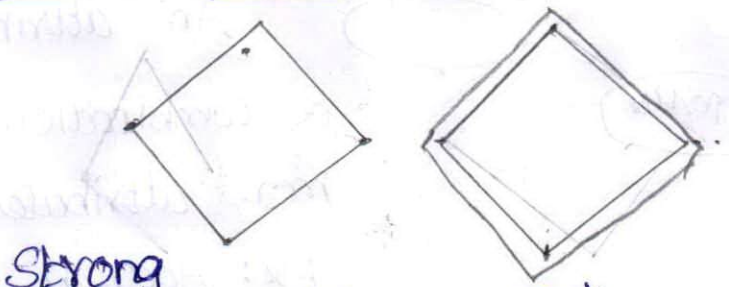


Strong Entity

Weak Entity

Entity which doesn't have any key attributes of its own is weak Entity

Relationship between Entities

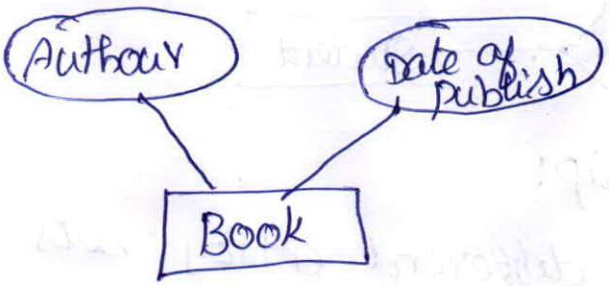


Strong relationship

weak relationship

Relationship b/w a Strong entity and a weak entity is weak relationship

Attributes for an Entity



Key Attributes (Prime Key attributes)

Derived Attribute

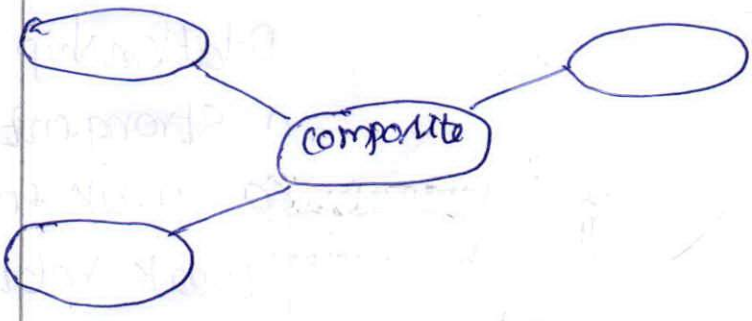
Derived attributes are those which are derived based on other attributes. Eg: Age derived from Date of birth

multi-valued attribute:

Attribute which has more than one value

Eg: mobile-no

Composite Attribute



An attribute which is a combination of two or more attributes.

Ex: Address: State, City, Zip

Relationship.

Relationship describes relation between Entities

Degree of Relationship:

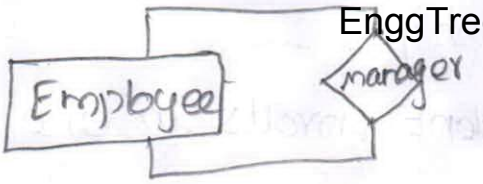
The number of different entity sets participating in a relationship is called degree of relationship.

1. unary [Recursive relationships]
2. Binary Relationship
3. Ternary Relationship.

Unary Relationship:

When there is only one entity participating in a relationship, it is unary relationship.

Eg: A person, an employee manager an



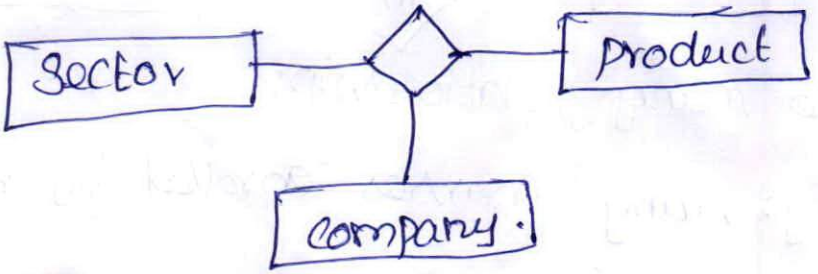
Binary relationship:

When there are two entities participating in a relationship, the relationship is called as binary relationship.

Eg: Student enrolled in course.

Ternary relation

When there are n-ary entities participating in a relationship, the relationship is called ternary relation.



Cardinality of relationship:

* The number of times an entity participates in relationship with other entities is cardinality

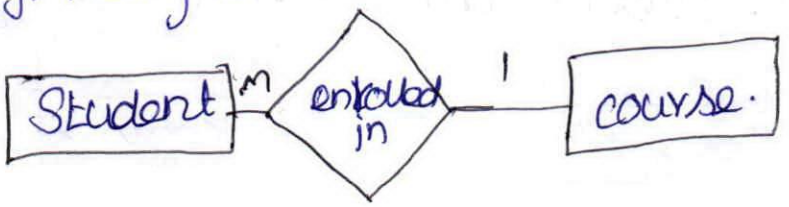
They are, ① one to one relationship
② one to many relationship
③ many to one relationship

One to One Relationship:

* Eg: one student enrolls in one course

many to one Relationship

* Eg: many Student enrolls in one courses

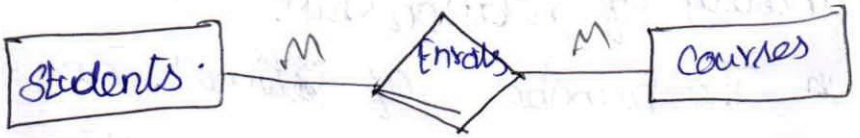


one to many Relationship

Eg: One Student enrolls in many courses

many to many Relationship:

Eg: many courses enrolled by many students



Entity Relationship (ER) Modeling -

Here we are going to design an Entity Relationship (ER) model for a college database . Say we have the following statements.

1. A college contains many departments
2. Each department can offer any number of courses
3. Many instructors can work in a department
4. An instructor can work only in one department
5. For each department there is a Head
6. An instructor can be head of only one department
7. Each instructor can take any number of courses
8. A course can be taken by only one instructor
9. A student can enroll for any number of courses
10. Each course can have any number of students

Good to go. Let's start our design.(Remember our previous topic and the notations we have used for entities, attributes, relations etc)

Step 1 : Identify the Entities

What are the entities here?

From the statements given, the entities are

1. Department
2. Course
3. Instructor
4. Student

Step 2 : Identify the relationships

1. One department offers many courses. But one particular course can be offered by only one department. hence the cardinality between department and course is One to Many (1:N)
2. One department has multiple instructors . But instructor belongs to only one department. Hence the cardinality between department and instructor is One to Many (1:N)
3. One department has only one head and one head can be the head of only one department. Hence the cardinality is one to one. (1:1)
4. One course can be enrolled by many students and one student can enroll for many courses. Hence the cardinality between course and student is Many to Many (M:N)
5. One course is taught by only one instructor. But one instructor teaches many courses. Hence the cardinality between course and instructor is Many to One (N :1)

Step 3: Identify the key attributes

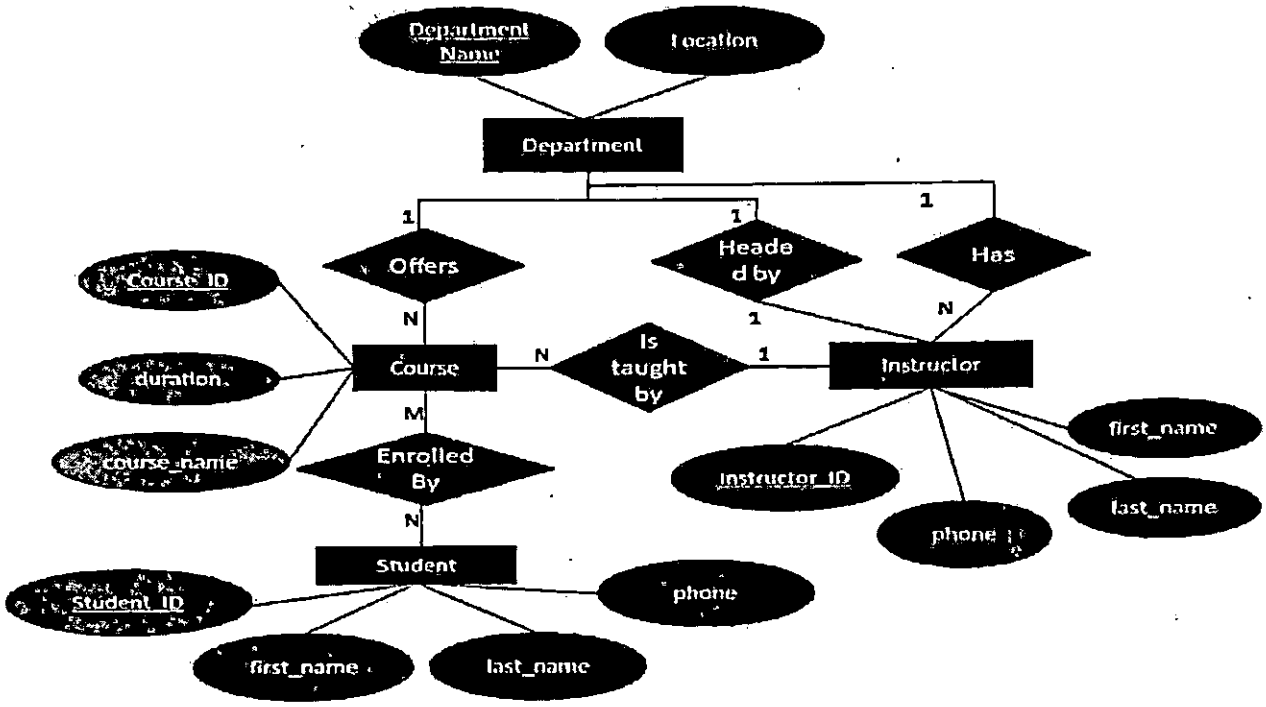
- "Departmen_Name" can identify a department uniquely. Hence Department_Name is the key attribute for the Entity "Department".
- Course_ID is the key attribute for "Course" Entity.
- Student_ID is the key attribute for "Student" Entity.
- Instructor_ID is the key attribute for "Instructor" Entity.

Step 4: Identify other relevant attributes

- For the department entity, other attributes are location
- For course entity, other attributes are course_name,duration
- For instructor entity, other attributes are first_name, last_name, phone
- For student entity, first_name, last_name, phone

Step 5: Draw complete ER diagram

By connecting all these details, we can now draw ER diagram as given below.



Enhanced Entity Relationship model [EER model]

* EER is a high level data model that incorporates the extensions to the original ER model.

It is a diagrammatic technique for displaying the following concepts

Sub class and super class
Specialization and generalization
Union or category
Aggregation

Features of EER model:

* It create a design more accurate to database schemas.
* It reflects the data properties and constraints more precisely.
* It includes all ~~modifi~~ modeling concepts of the ER model
* Diagrammatic technique helps for displaying the EER schema.
* It includes the concepts of specification and generalization.
* It is used to represent a collection of

Sub class and Super class:

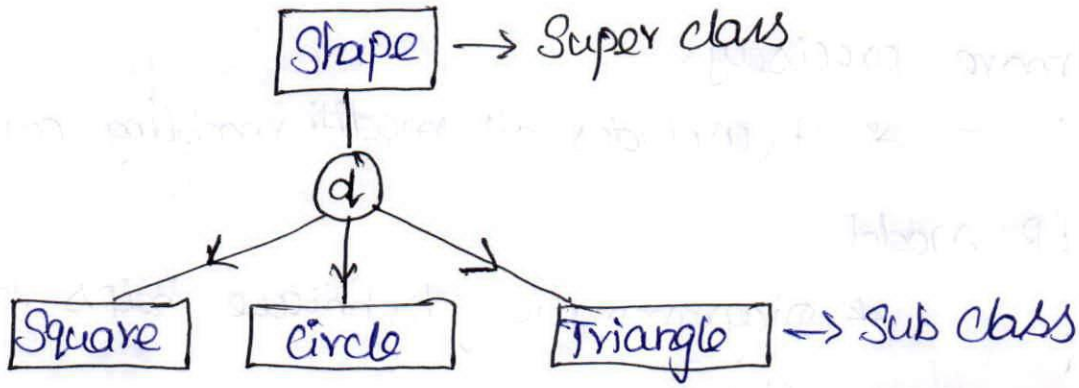
* Sub-class and Super class relationship leads the concepts of inheritance.
* The relationship between Subclass and super class is denoted with (d) Sysmbol.

Super class:

* Super class is an entity type that has a relationship with one or more sub-types.
* An entity cannot exist in database merely by being member of any super class.

Sub class:

* Sub-class is a group of entities with attributes.
* It inherits properties and attributes from its Super class.



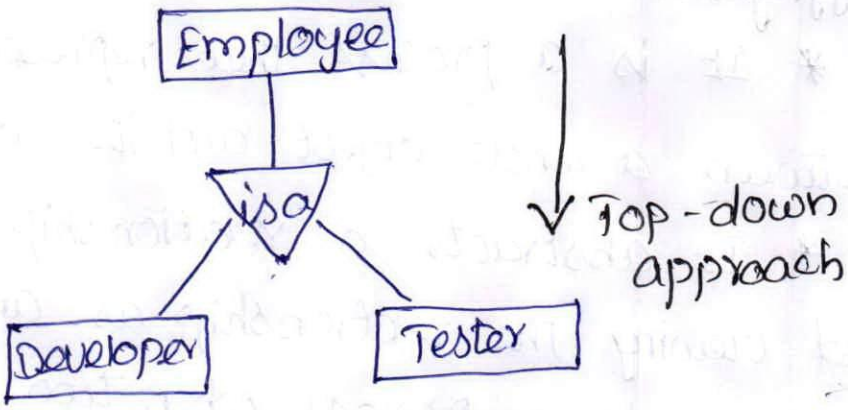
Specialization:

* It is a process which defines a group of entities which is divided into sub-groups based on their characteristics.

* It is a top down approach, in which one higher entity can be broken into two lower level entity.

* It maximizes the differences between the members of an entity by identifying the unique characteristic or attributes of each member.

* It defines one or more sub class for the super class and also forms the super class/ sub class relationship.



Specialization.

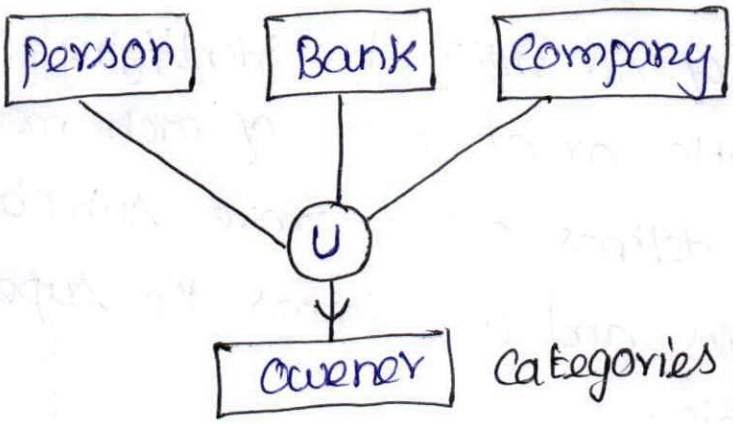
Category or union:

* Category represents a single super class or sub class relationship with more than one super class.

It can be a total or partial participations

For examples, Car booking: Car owner can be a person, a bank (holds a possession on a car) or a company. Category (sub class) → owner is a

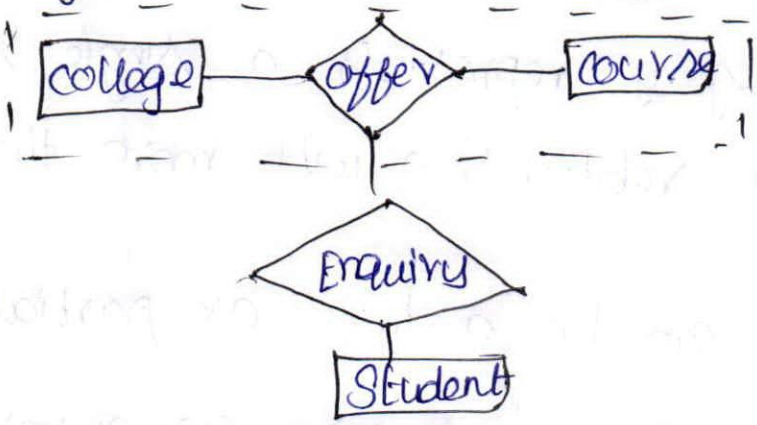
Subset of the union of the three super class → company, Bank and person. A category member must exist in at least one of its super class.



Categories (union type).

Aggregation:

* It is a process that represents a relationship between a whole object and its components parts.
* It abstracts a relationship between objects and viewing the relationship as an objects.
* It is a process when two entity is treated as a single entity.



* In the above example, the relation between college and course is acting as an entity in relation with student.

UML class diagram: [UML: Unified modeling Language]

* class diagram describes the attributes and operation of a class and also the constraints imposed on the system.

Class diagram are one of the most useful types of diagrams in UML as they clearly mapout the structure of a particular system by modelling its classes, attributes, operation and relationship between objects.

* The UML methodology is being used extensively in software design and has many types of diagram for various software design purpose.

* In UML class diagram, a class is displayed as a box that includes three section: The top section gives the class name; middle section includes the attributes; and the last section includes operations; that can be applied individua objects of the class, operation are not specified in ER diagram.

* In UML, there are two types of relationship association and aggregation. Aggregation is ~~must~~ meant to represent the relationship between a whole object and its components part and it has a

* UML also distinguishes between unidirectional and bidirectional associations. In the unidirectional case the line connecting the classes is displayed with an arrow to indicates that only one direction for accessing related objects is needed. If no arrow is displayed, the bidirectional case is assumed, which is the default.

Alternate Notations for E-R Diagrams → There are many alternative diagramming notations for displaying ER Diagrams. We have Unified Modeling Language (UML) notation class diagrams, which has been proposed as a standard for conceptual object modeling.

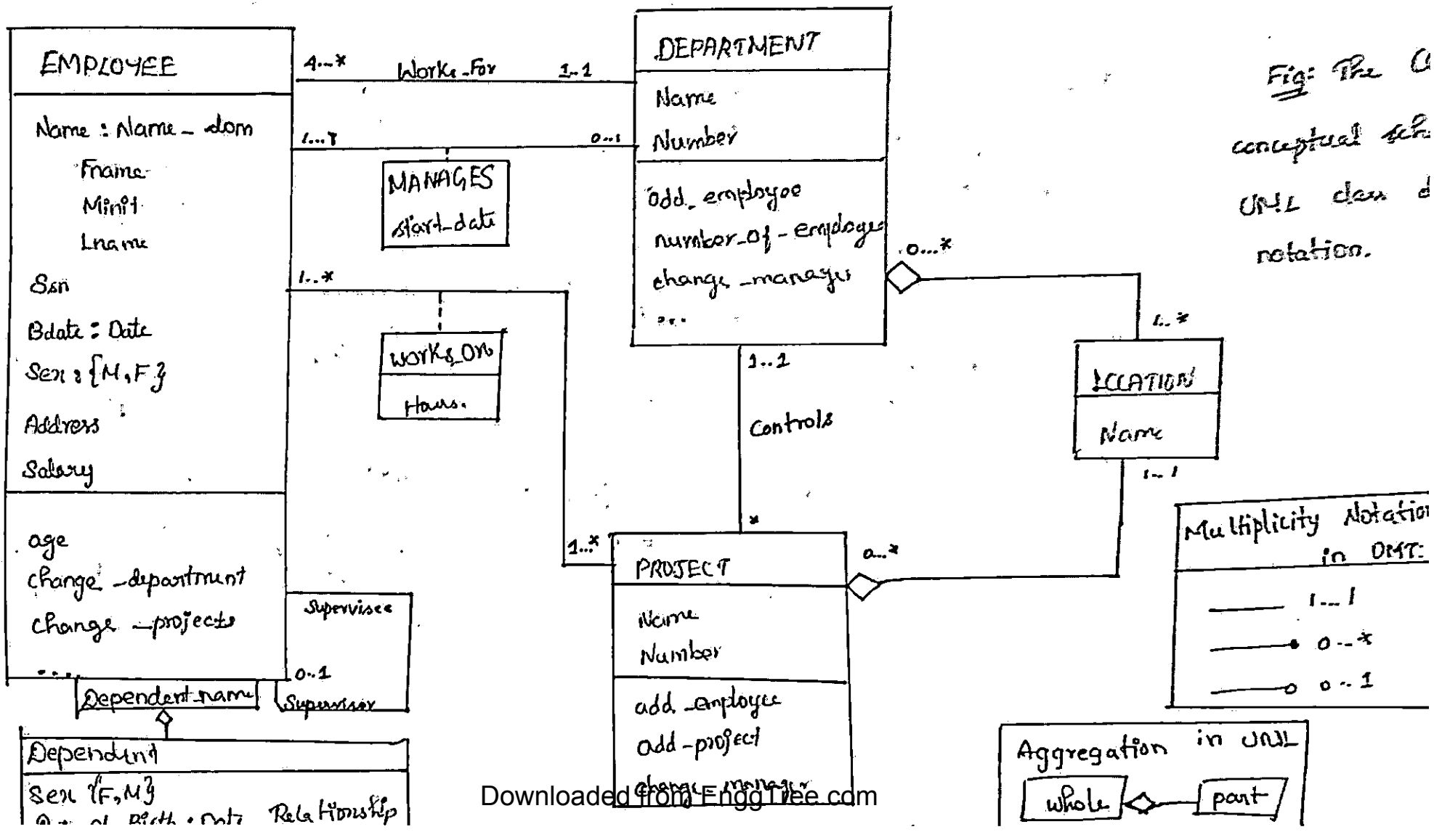
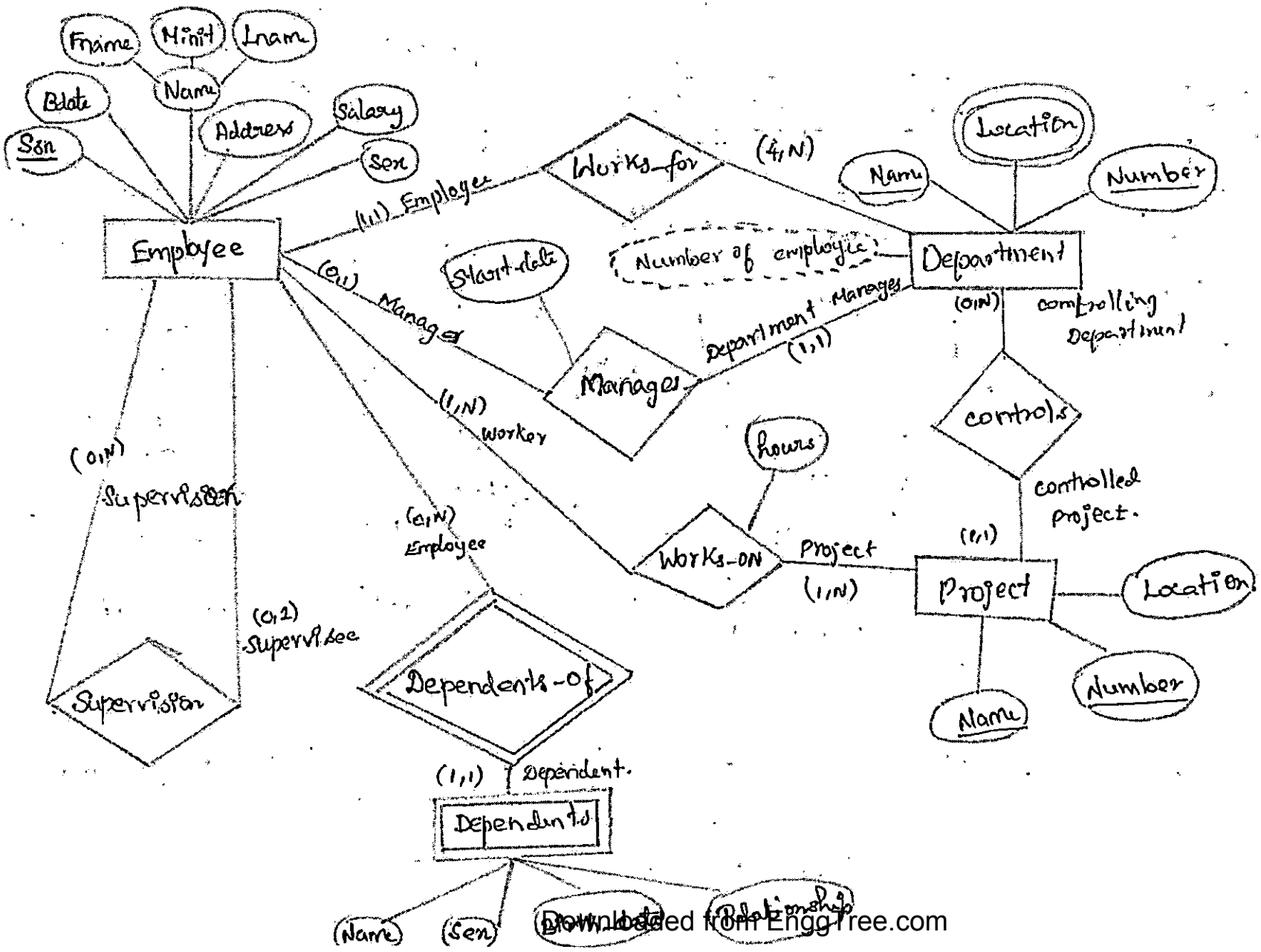


Fig: The C
conceptual sch
UML class d
notation.

ER Diagram for the company schema, with structural constaints specified u
(min, max) notation and role nam



AD8401 Database Design and Management.

UNIT- II Relational Model and SQL.

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

1) Relational Model concepts: "A relational model of data for large shared data banks" - first seminar paper presented by E.F. Codd. The relational model's objectives were specified as follows:

* To allow a high degree of data independence. Application programs must not be affected by modifications to the internal data representation, particularly by changes to file organizations, record orderings, or access paths.

* To provide substantial grounds for dealing with data representation, particularly by semantics, consistency, and redundancy problems. In particular, Codd's paper introduced the concept of normalized relations, that is relations that have no repeating groups.

* To enable the expansion of set-oriented data manipulation languages.

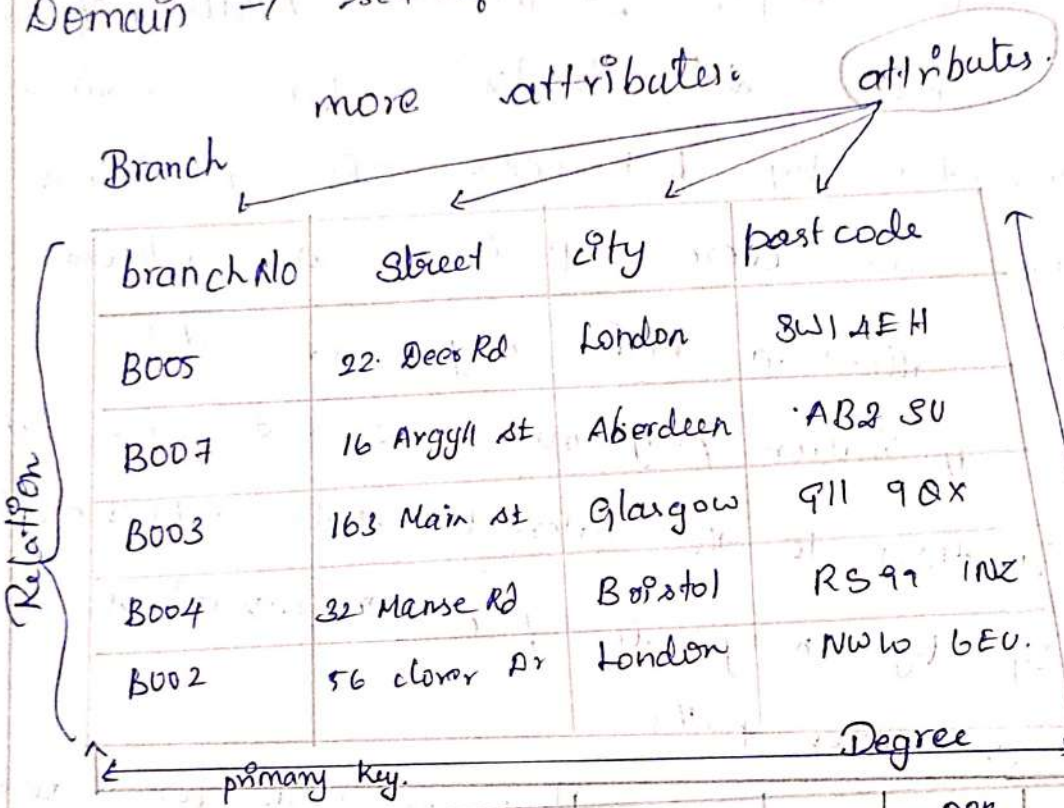
Terminology:

Relational Data Structure.

Relation → table with columns and rows.

Attribute → named column of a relation.

Domain → set of allowable values for one or more attributes.



Relation

staffNo	fName	lName	position	sex	DOB	Salary	branchNo
SL21	John	White	Manager	M	1 Oct 45	30000	B005
SG37	Ann	Beech	Assistant	F	10 Nov 60	12000	B003
SG14	David	Ford	Supervisor	M	24 Mar 58	18000	B003
SA9	Mary	Howe	Assistant	F	19 Feb 70	9000	B007
SG5	Susan	Brand	Manager	F	3 Jun 40	24000	B003
SL41	Julie	Lee	Assistant	F	13 Jun 65	9000	B005.

Fig: Domains for some attributes of the Branch and staff relation.

Tuple → A Row of a relation.

Degree → degree of a relation is the number of attributes it contains.

Cardinality → number of tuples a relation contains

Relational database → A collection of normalized relation names.

Mathematical Relations.

We have two sets D_1 and D_2

$D_1 = \{2, 4\}$ $D_2 = \{1, 3, 5\}$

The cartesian product of there two sets, written $D_1 \times D_2$ is the set of all ordered pairs such that the first element is a member of D_1 and the second element is a member of D_2. An alternative way of expressing this is to find all combinations of elements with the first from D_1 and the second from D_2.

$D_1 \times D_2 = \{(2,1), (2,3), (2,5), (4,1), (4,3), (4,5)\}$.

Any subset of this cartesian product is a relation.

Database Relations.

Relation Schema → A named relation defined by a set of attribute and domain name pairs.

Relational database schema → A set of relation schemas, each with a distinct name.

If $R_1, R_2, \ldots R_n$ are a set of relation schemas, then we can write the relational database schema, or simply relational schema, R as

$$R = \{R_1, R_2 \ldots R_3\}.$$

Properties of Relations.

* the relation has a name that is distinct from all other relation names in the relational schem
* each cell of the relation contains exactly one atomic value:
* each attribute has a distinct name.
* the ~~values~~ value of an attribute are all from the same ~~direction~~. domain
* each tuple is distinct; there are no duplicate tuples.
* the order of attributes has no significance.

* the order of tuple has no significance, theoretically.

Relation that do not contain repeating groups is said to satisfies the property, called normalized or in first normal form.

Relational Keys.

Superkey → An attribute, or set of attributes, that uniquely identifies a tuple within a relation.

Candidate Key → A superkey such that no proper subset is a superkey within the relation.

A candidate key K for a relation R has two properties:

Uniqueness: In each tuple of R, the value of K uniquely identify that tuple.

Irreducibility: No proper subset of K has the uniqueness property.

There are several candidate keys for a relation. When a key consists of more than one attribute, we call it a composite key.

Primary Key: The candidate key that is selected to identify tuple uniquely within the relation

Foreign Key: An attribute, or set of attributes, within one relation that matches the candidate key of some relation.

2) Integrity Constraints.

There are constraints that form restrictions on the set of values allowed for the attributes of relations. In addition, there are two important integrity rules, which are constraints or restrictions that apply to all instances of the database. The two principle rules for the relational model are known as entity integrity and referential integrity. Other types of integrity constraints are multiplicity.

Nulls: Represents a value for an attribute that is currently unknown or is not applicable for this tuple.

Entity Integrity: In a base relation, no attribute of a primary key can be null.

Referential Integrity: If a foreign key exists in a relation, either the foreign key value must match a candidate key value of some tuple in its home relation or the foreign key value must be wholly null.

General Constrains: Additional rules specified by the users or database administrators of a database that define or constrain some aspect of the enterprise.

3) SQL - Data Manipulation

Introduction to SQL.

Ideally, a database language should allow a user to:

* create a database and relation structure
* Peoform basic data management tasks, such as the insertion, modification, and deletion of data from the relations
* Perform both simple and complex queri

A database language must perform these tasks with minimal user effort, and its command structure and syntax must be relatively t easy to learn. Finally, the language must be portable; that is, it must conform to some recognized standard so that we can use the same command structure and syntax when we move from one DBMS to another. SQL is intended to satisfy these requirements.

SQL is an example of a transform-oriented language, or a language designed to use relations to transform inputs into required outputs. As a language, the ISO SQL standard has two major components:

* a Data Definition Language (DDL) for defining the database structure and controlling access to the data.
* a Data Manipulation Language (DML) for retriving and updating data.

SQL is a relatively easy language to learn:

* It is a non-procedural language, you specify what information you require, rather than how to get it. In other words, SQL does not require you to specify the access methods to the data.

* Like most modern languages, SQL is essentially free-format, which means that parts of statements do not have to be typed at particular locations on the screen.

* The command structure consists of standard English words such as CREATE, TABLE, INSERT, SELECT.

For example:

- CREATE TABLE staff (staffNO VARCHAR(5), lName VARCHAR(15), salary DECIMAL(7,2));
- INSERT INTO staff VALUES ('SG16', 'Brown', 8300)
- SELECT staffNO, lName, salary
  FROM staff
  WHERE salary > 10000;

* SQL can be used by a range of users including database administrators (DBA) management personnel, application developers, and many other types of end-user.

Importance of SQL:

SQL is the only standard database language to gain wide acceptance. The only other language, the Network Database Language (NDL), based on the CODASYL network model, has few followers. Nearly every major current vendor provides database products based on SQL or with an SQL interface, and most are represented on at least one of the standard-making bodies. There is a huge investment in the SQL language both by vendors and by users. It has become part of application architectures such as IBM's system Architecture Application Architecture (SAA) and is the strategic choice of many large and influential organizations. SQL also became a Federal Information Processing standard (FIPS) to which conformance is required for all sales of DBMSs to the US government. The SQL Access Group, a consortium of vendors, defined a set of enhancements to SQL that would support interoperability across disparate systems.

Writing SQL Commands

An SQL statement consists of reserved words and user-defined words. Reserved words are a fixed part of the SQL language and have a fixed meaning. They must be spelled exactly as required and cannot be split across lines. User-defined words are made up by the user and represent the names of various database objects such as tables, columns, views, indexes, and so on.

Most components of an SQL statement are not case-sensitive, which means that letters can be typed in either upper or lowercase. The one important exception to this rule is that literal character data must be typed exactly as it appears in the database.

Although SQL is free-format, an SQL statement or set of statements is more readable if indentation and lineation are used. For example,

* each clause in a statement should begin on a new line

* the beginning of each clause should line up with the beginning of other clauses:

* if a clause has several parts, they should each appear on a separate line and be indented under the start of the clause to show the relationship.

Data Manipulation.

SELECT - to query data in the database.

INSERT - to insert data into a table.

UPDATE - to update data in a table.

DELETE - to delete data from a table.

Owing to the complexity of the SELECT statement and the relative simplicity of the other DML statements, we devote most of this section to SELECT statement and its various formats.

We illustrate the SQL statements using the instance of the DreamHome case study, consist of the following tables.

Branch, Staff, PropertyForRent, Client, private owner, Viewing.

Literals

Literals are constants that are used in SQL statements.

There are different forms of literals for every data type supported by SQL. We can distinguish between literals that are enclosed in single quotes and those are not. All numeric data values must be enclosed in single quotes; all numeric data values must not be enclosed in single quotes. For example, we could use literals to insert data into a table:

```
INSERT INTO PropertyForRent (propertyNo, street, city, postcode,
                type, rooms, rent, ownerNo, staffNo, branchNo)
VALUES ('PA14', '16 Holhead', 'Aberdeen', 'AB7 5SU', 'House',
        6, 650.00, 'CO46', 'SA9', 'B007');
```

The value in column rooms is an integer literal and the value in column rent is decimal number literal: they are not enclosed in single quote. all other columns are character strings and are enclosed in single quotes.

Simple Queries:

The purpose of the SELECT statement is to retrive and display data from one or more database tables. It is an extremely powerful command

capable of performing the equivalent of the relation algebra's selection, projection and join operations in a single statement. SELECT is the most frequently used SQL command and has the following general form;

```
SELET    [ DISTINCT | ALL {* | [columnExpression [AS newName]]
                                                    [,...]}
FROM     TableName [alias] [,...]
[WHERE   condition ]
[GROUP BY  columnList] [HAVING condition]
[ORDER BY  columnList]
```

columnExpression represents a column name or an expression, TableName is the name of an existing database table or view that you have access to, and alias is an optional abbreviation for TableName. The sequence of processing in a SELECT statement is :

FROM → specifies the table or tables to be used

WHERE → filters the row subject to some condition

GROUP → BY forms groups of row with the same column value

HAVING → filters the groups subjects to some condition

SELECT → specifies which column are to appear in the output

ORDER BY → specifies the order of the output.

The order of the clauses in the SELECT statement cannot be changed. The only two mandatory clauses are the first two: SELECT and FROM; the remainder are optional. The SELECT operation is closed;

Retrieve all rows and columns

```
SELECT staffNO, fName, lName, position, sex, DOB, salary,
                                                branchNO.
```

(OR)

```
SELECT *
FROM Staff;
```

The result of above queris are shown.

staffNO	fName	lName	position	sex	DOB	salary	branchNO
SL21	John	White	Manager	M	1 Oct 45	30000.00	B005
SG37	Ann	Beech	Assistant	F	10 Nov 60	12000.00	B003
SG14	David	Ford	Supervisor	M	24-Mar 58	18000.00	B003
SA9	Mary	Howe	Assistant	F	19-Feb 70	9000.00	B007
SG5	Susan	Brand	Manager	F	3-Jun 40	24000.00	B003
SL41	Julie	Lee	Assistant	F	13 Jun 65	9000.00	B005

Retrieve specific columns, all rows.

```
SELECT staffNO, fName, lName, salary FROM Staff;
```

Result:

staffNO	fName	lName	salary.
SL21	John	White	30000.00
SG37	ANN	Beech	12000.00
SG14	David	Ford	18000.00
SA9	Mary	Howe	9000.00
SG5	Susan	Brand	24000.00
SL41	Julie	Lee	9000.00

Use of DISTINCT.

List the property numbers of all properties that have been viewed.

```
SELECT    propertyNo
FROM      viewing;
```

Result:

propertyNo
PA14
PG4
PG4
PA14
PG36.

Notice that these are several duplicates, because unlike the relational algebra Projection operation, SELECT does not eliminate duplicates when it projects over one or more columns. To eliminate the duplicates, we use the DISTINCT keyword. Rewriting the query as:

```
SELECT DISTINCT propertyNo
FROM Viewing;
```

Results:

propertyNo
PA14
PG4
PG36.

Calculated fields.

Produce a list of monthly salaries for all staff, showing the staff number, the first and last names, and the salary details.

```
SELECT staffNO, fName, lName, salary/12
FROM Staff;
```

An SQL expression can involve addition, subtraction, multiplication, and division, and parantheses can be used to build complex expressions. More than one table column can be used in a calculated column; however, the columns referenced in an arithmetic expression must have a numeric type.

The fourth column of this result table has been output as col4. Normally, a column in the result table takes its name from the corresponding column of the database table from which it has been retrived. However, in this case, SQL does not know how to label the column. Some dialects give the column a name corresponding to its position in the table;

Result:

staffNO	fName	lName	col4.
SL21	John	White	2500.00
SG37	Ann	Beech	1000.00
SG14	David	Ford	1500.00
SA9	Mary	Howe	750.00
SG5	Susan	Brand	2000.00
SL41	Julie	Lee	750.00

Some may leave the column name blank or use the expression entered in the SELECT list. The ISO standard allows the column to be named using an AS clause. In the previous example, we could have written:

```
SELECT staffNO, fName, lName, salary/12 AS monthlySalary
FROM staff;
```

In this case, the column heading of the result table would be monthlysalary rather than col4.

Row Selection (WHERE clause)

Keyword WHERE followed by a search condition that specifies the row to be retrieved. The five basic search conditions are as follows:

Comparisons → Compare the values of one expression to the value of another expression.

Range → Test whether the value of an expression falls within a specified range of values.

set membership → Test whether the value of an expression equals one of a set of values.

Pattern match → Test whether a string matches a specified pattern.

Null → Test whether a column has a null value.

Comparison search condition.

List all staff with a salary greater than €10,000

```
SELECT staffNO, fName, lName, position, salary
FROM Staff
WHERE salary >10000;
```

Result:

staffNo	fName	lName	position	salary.
SL21	John	White	Manager	30000.00
SG37	Ann	Beech	Assistant	12000.00
SG14	David	Ford	supervisor	18000,00
SG5	Susan	Brand	Manager	24000.00

In SQL, the following simple comparison operators are available:

=	equals		
<>	is not equal to (ISO standard)	!=	is not equal to (allowed in some dialects)
<	is less than	<=	is less than or equal to
>	is greater than	>=	is greater than or equal to.

More complex predicates can be generated using the logical operators AND, OR, and NOT, with parentheses to show the order of evaluation.

The rules for evaluating a conditional expression are:

* an expression is evaluated left to right.
* subexpression in brackets are evaluated first.
* NOTs are evaluated before ANDs and ORs
* ANDs are evaluated before ORs.

Compound comparison search condition.

List the address of all branch offices in London or Glasgow.

```
SELECT *
FROM Branch
WHERE city = 'London' OR city = 'Glasgow';
```

Results:

branchNo	street	city	postcode
B005	22 Deer Rd	London	SW1 4EH
B003	163 Main St	Glasgow	G11 9QX
B002	56 Clover Dr	London	NW10 6EU

Range search condition. (BETWEEN / NOT BETWEEN)

List all staff with a salary between £20,000 and £30000.

```
SELECT staffNO, fName, lName, position, salary
FROM Staff
WHERE salary BETWEEN 20000 AND 30000;
```

Result :

staffNO	fName	lName	position	salary.
SL21	John	white	Manager	30000.00
SG5	Susan	Brand	Manager	24000.00.

There is also a negated version of the range test that checks for values outside the range. The BETWEEN test does not add much to the expressive power of SQL, because it can be expressed equally well using two comparison tests. We could have expressed the previous query as:

```
SELECT staffNO, fName, lName, position, salary
FROM Staff
WHERE salary >= 20000 AND salary <= 30000;
```

However, the BETWEEN test is a simple way to express a search condition when considering a range of values.

Set membership search condition (IN / NOT IN)

List all managers and supervisors.

```
SELECT staffNO, fName, lName, position
FROM Staff
WHERE position IN ('Manager', 'Supervisor');
```

Results:

staffNo	fName	lName	position
SL21	John	white	Manager
SG14	David	Ford	Supervisor
SG5	Susan	Brand	Manager

NULL search condition (IS NULL / IS NOT NULL)

List the details of all viewings on property PG4 where a comment has been not supplied.

```
SELECT clientNo, viewDate
FROM viewing
WHERE propertyNo = 'PG4' AND comment IS NULL;
```

Result:

clientNo	viewDate
CR56	26-May-13.

Sorting Results (ORDER BY clause)

single-column ordering.

Produce a list of salaries for all staff, arranged in descending order of salary.

```
SELECT staffNo, fName, lName, salary
FROM staff
ORDER BY salary DESC;
```

Result:

staffNo	fName	lName	salary.
SL21	John	White	30000.00
SG5	Susan	Brand	24000.00
SG14	David	Ford	18000.00
SG37	Ann	Beech	12000.00
SA9	Mary	Howe	9000.00
SL41	Julie	Lee	9000.00.

Multiple column ordering:

Produce an abbreviated list of properties arranged in order of property type.

```
SELECT propertyNo, type, rooms, rent
FROM PropertyForRent
ORDER BY type;
```

The system arranges rows in any order it chooses. To arrange the properties in order of rent, we specify a minor order, as follows,

```
SELECT propertyNo, type, rooms, rent
FROM propertyForRent
ORDER BY type, rent DESC;
```

Result:

With one sort key

propertyNo	type	rooms	rent
PL94	Flat	4	400
PG4	Flat	3	350
PG36	Flat	3	375
PG16	Flat	4	450
PA14	House	6	650
PG21	House	5	600.

Now, the result is ordered first by property type, in ascending alphabetic order, and within property type, in descending order of rent.

Results:

with two sort key

propertyNo	type	rooms	rent.
PG16	Flat	4	450
PL94	Flat	4	400
PG36	Flat	3	375
PG4	Flat	3	350
PA14	House	6	650
PG21	House	5	600.

Using the SQL Aggregate function.

COUNT → returns the number of values in a specified column.

SUM → returns the sum of the values in a specified column.

AVG → returns the average of the values in a specified column.

MIN → returns the smallest value in a specified column.

MAX → returns the largest value in a specified column.

```
SELECT staffNo, COUNT(salary)
FROM staff;
```

Use of COUNT (*)

How many properties cost more than £350 per month to rent?

```
SELECT COUNT(*) AS myCount
FROM PropertyForRent
WHERE rent > 350
```

Result:

mycount
5

Use of COUNT (DISTINCT)

How many different properties were viewed in May 2013 ?

```
SELECT COUNT (DISTINCT propertyNo) AS myCount
FROM Viewing
WHERE viewDate BETWEEN '1-May-13' AND '31 May13'.
```

Result:

myCount
2

Use of Count and SUM.

find the total number of Managers and the sum of their salaries.

```
SELECT COUNT (staffNo) AS myCount , SUM (salary) AS
                                                mySum
FROM Staff
WHERE position = 'Manager';
```

Result:

myCount	mySum
2	54000.00.

Use of MIN, MAX, AVG.

find the minimum, maximum and average staff salary.

```
SELECT MIN (salary) AS myMin , MAX (salary) AS
myMax , AVG (salary) AS myAvg FROM Staff ;
```

Results:

myMin	myMax	myAvg
9000.00	30000.00	17000.00.

Grouping Results (GROUP BY clause)

A query that includes the GROUP BY clause is called a grouped query, because it groups the data from the SELECT tables and produces a single summary row for each group. The column named in the GROUP BY clause are called the grouping columns. The ISO standard requires the SELECT clause and the GROUP BY clause to be closely integrated. When GROUP BY is used, each item in the SELECT list must be single-valued per group. In addition, the SELECT clause may contain only:

* column names;
* aggregate functions;
* constants;
* an expression involving combinations of these elements.

The ISO standard considers two nulls to be equal for purposes of the GROUP By clause. If two rows have nulls in the same grouping columns and identical values in all the nonnull grouping columns, they are combined into the same group.

Use GROUP BY.

Find the number of staff working in each branch and the sum of their salaries.

```
SELECT branchNO , COUNT (staffNO) AS myCount , SUM (Salary)
                                        AS mySum.

FROM staff
GROUP BY BranchNo
ORDER BY branch NO;
```

Results:

branchNo	myCount	mysum.
B003	3	54000.00
B005	2	39000,00
B007	1	9000.00.

Restricting grouping (HAVING clause)

The HAVING clause is designed for use with the GROUP BY clause to restrict the groups that appear in the final result table.

eg: for each branch office with more than one member of staff, find the number of staff working in each branch and the sum of their salaries.

```
SELECT branchNo , COUNT (staffNO) AS myCount , SUM (salary)
                                        AS mySum

FROM staff
GROUP BY branchNO
HAVING COUNT (staffNo) > 1
ORDER BY branchNO;
```

Results:

branchNo	myCount	mySum.
B003	3	54000.00
B005	2	39000.00.

Subqueries.

Use of a complete SELECT statement embedded within another SELECT statement. The results of this inner SELECT statement are used in the outer statement to help determine the contents of the final result. A sub-select can be used in the WHERE and HAVING clauses of an outer SELECT statement, where it is called a subquery or nested query. Sub Selects may also appear in INSERT, UPDATE and DELETE statements. There are three types of subquery:

* A Scalar Subquery returns a single column and a single row, that is, a single value. In principle, a scalar subquery can be used whenever a single value is needed.

* A row subquery returns multiple columns, but only a single row. A row subquery can be used whatever a row value

constructor is needed, typically in predicates.

* A table subquery returns one or more columns and multiple rows. A table subquery can be used whenever a table is needed, for example, as an operand for the IN predicate.

Using a subquery with equality.

List the staff who work in the branch at '163 Main St'

```
SELECT staffNo, fName, lName, position
FROM Staff
WHERE branchNo = (SELECT branchNo
                  FROM Branch
                  WHERE street = '163 Main St');
```

Result:

staffNo	fName	lName	position
SG37	Ann	Beech	Assistant
SG14	David	Ford	Supervisor
SG5	Susan	Brand	Manager.

Using a subquery with an aggregate function.

List all staff whose salary is greater than the average salary, and show by how much

their salary is greater than the average.

```
SELECT staffNo, fName, lName, position
    salary - (SELECT AVG (salary) FROM Staff) AS saldif.
FROM Staff
WHERE salary > (SELECT AVG (salary) FROM staff);
```

Scalar sub queries

```
SELECT staffNO, fName, lName, position, salary -
                                    17000 AS salDiff
FROM Staff
WHERE salary >17000;
```

Results:

staffNo	fName	lName	position	salDiff
SL21	John	White	Manager	13000.00
SG14	David	Ford	Supervisor	1000.00
SG5	Susan	Brand	Manager	7000.00.

Multi-table queries.

To combine columns from several tables into a result table, we need to use a join operation.

The SQL join operation combines information from two tables by forming pairs of related rows from the two tables. The row pairs that make up the joined table are those where the matching columns in each of the two tables have the same values.

simple join.

List the names of all clients who have viewed a property, along with any comments supplied.

```
SELECT  c.clientNO, fName, lName, propertyNO, comment
FROM    client c, viewing v
WHERE   c.clientNO = v.clientNO;
```

Results:

clientNO	fName	lName	propertyNO	comment
CR56	Aline	Stewart	PG36	
CR56	Aline	Stewart	PA14	too small
CR56	Aline	Stewart	PG4	
CR62	Mary	Tregear	PA14	no dining room
CRTC	John	Kay	PG4	too remote

The SQL standard provides the following alternative ways to specify this join.

```
FROM Client c JOIN Viewing v ON c.clientNo = v.clientNo
FROM client JOIN viewing USING clientNo
FROM client NATURAL JOIN Viewing.
```

In each case, the from (FROM) clause replaces the original FROM and WHERE clauses. However, the first alternative produces a table with two identical clientNo columns; the remaining two produces a table with a single clientNo column.

Sorting a Join.

For each branch office, list the staff numbers and names of staff who manages properties and the properties that they manage.

```
SELECT s.branchNo, s.staffNo, fName, lName, propertyNo
FROM Staff S, PropertyForRent P
WHERE s.staffNo = p.staffNo
ORDER BY s.branchNo, s.staffNo, propertyNo.
```

Results:

branchNo	staffNo	fName	lName	propertyNo.
B003	SG14	David	Ford	PG16
B003	SG37	Ann	Beech	PG21
B003	SG37	Ann	Beech	PG36
B005	SL41	Julie	Lee	PL94
B007	SA9	Mary	Howe	PA14

Other type of Joins: Three-table Join, Multiple grouping columns, Computing a join, Outer joins, Left outer join, Right outer join, Full outer join

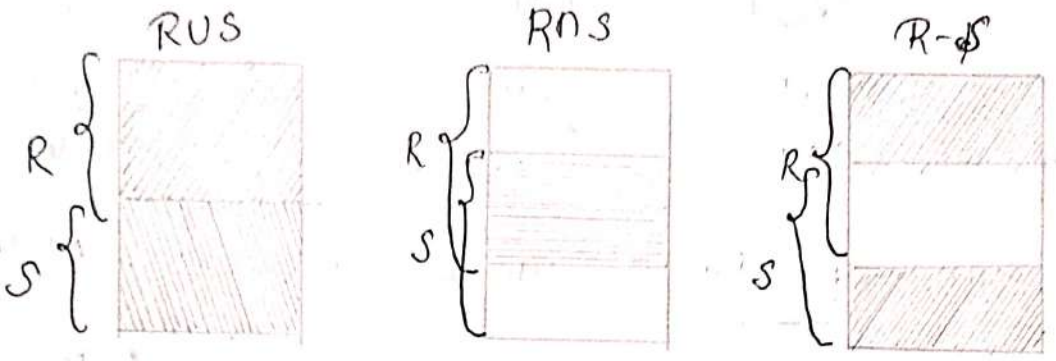
Combining Result tables:

In SQL, we can use the normal set operations of Union, Intersection and Difference to combine the results of two or more queries into a single table:

* The Union of two tables, A and B, is a table containing all rows that are in either the first table A or second table B or both.

* The intersection of two tables, A and B, is a table containing all rows that are common in both A and B.

* The Difference of two tables, A and B, is a table containing all rows that are in table A but are not in table B.



a) Union b) intersection c) Difference

Database Updates:

SQL is a complete data manipulation language that can be used for modifying the data in the database as well as querying the database. The commands for modifying the database are not as complex as the SELECT statement.

INSERT → adds new rows of data to a table

UPDATE → modifies existing data in a table

DELETE → removes rows of data from a table..

Data Definitions.

ISO SQL Data Types.

SQL identifiers are used to identify objects in the database, such as table names, view names, and columns. The characters that can be used in a user-defined SQL identifier must appear in a character set. The ISO standard provids a default character set, which consists of the uppercase letters A ... Z, the lowercase letters a ... z, the digits 0 ... 9, and the underscore (_) character. It is possible to specify an alternative character set. The following restrictions are imposed on an identifier.

* an identifier can be no longer than 128 characters
* an identifier must starts with a letter,
* an identifier cannot contain spaces.

SQL scalar Data Types.

Sometimes, for manipulation and conversion purposes, the data types character and bit are collectively referred to as string data types and exact numeric and approprate numric are referred to as numeric data types as they share similar properties.

Boolean data → TRUE or FALSE.

The value of TRUE is greater than FALSE, and any comparision involving the NULL value or an UNKNOWN truth value returns an UNKNOWN result.

Character Data → character data consists of a sequence of characters from an implementation defined character set, that is defind by the vendor of the particular SQL dialect.

CHARACTER [VARYING] [length]

CHARACTER can be abbreviated to CHAR and CHARACTER VARYING to VARCHAR.

eg: branchNo CHAR (4).
address VARCHAR (30)

Bit data :→ used to define bit strings, that is, a sequence of binary digits (bits), each having either the value 0 or 1.

BIT [VARYING] [length]

eg: bitString BIT (4).

Exact Numeric Data → used to define numbers in exact representation. The number consists of digits, an optional decimal point, and an optional sign.

Integrity Enhancement feature.

Integrity control consists of constrain that we wish to impose in order to protect the database from becoming inconsistent. five types of integrity constraints,

required data
domain constraints
entity integrity
referential integrity
general constraints.

These constraints can be defined in the CREATE and ALTER TABLE statements,

Required Data → some columns must contain a valid value; they are not allowed to contain nulls. A null is distinct from blank or zero, and is used to represent data that is either not available, missing, or not applicable. For example, every member of staff must have an associated job position. The ISO standard provides the NOT NULL column specifier in the CREATE and ALTER TABLE statements to provide this type of constraint.

When not null is specified, the system rejects any attempts to insert a null in the column. If NULL is specified, the system accepts nulls. The ISO defaults is NULL. For example, to specify that the column position of the Staff table cannot be null, we define the column as:

position VARCHAR (10) NOT NULL

Domain constraints → The ISO standard provides two mechanisms for specifying domains in the CREATE and ALTER statements. The first is the CHECK clause, which allows a constraints to be defined on a column or the entire table. The format of the CHECK clause is:

CHECK (search Condition)

The ISO standard allows domains to be defined more explicitly using the CREATE DOMAIN statement:

CREATE DOMAIN DomainName [AS] dataType
[DEFAULT default Option]
[CHECK (search Condition)]

Entity Integrity:- The ISO standard supports entity integrity with the PRIMARY KEY clause in the CREATE and ALTER TABLE statements. For example, to define the primary key of the PropertyForRent table, we include:

PRIMARY KEY (propertyNo)

Referential Integrity:- if the foreign key contains a value, that value must refer to an existing, valid row in the parent table.

The ISO standard supports the definition of foreign keys with the FOREIGN KEY clause in the CREATE and ALTER TABLE statements. For example, to define the foreign key branchNo of the PropertyForRent table, we include the clause:

FOREIGN KEY (branchNo) REFERENCES Branch.

General constraints:- The ISO standard allows general constraints to be specified using the CHECK and UNIQUE clauses of the CREATE and ALTER TABLE statements and the CREATE ASSERTION statement.

CREATE ASSERTION assertionName

CHECK (searchCondition)

Data Definition:

The ISO standard also allows the creation of character sets, collations, and translations. The main SQL data definition language statements are:

CREATE SCHEMA		DROP SCHEMA
CREATE DOMAIN	ALTER DOMAIN	DROP DOMAIN
CREATE TABLE	ALTER TABLE	DROP TABLE
CREATE VIEW		DROP VIEW

These statements are used to create, change, and destroy the ~~state~~ structures that make up the conceptual schema. Although not covered ~~not~~ by the SQL standard, the following two statements are provided by many DBMSs.

CREATE INDEX DROP INDEX.

Additional commands are available to the DBA to specify the physical details of data storage;

Creating a Database.

According to ISO standard, relations and other database objects exist in an environment. Amongother things, each environment consists of one or more catalogs, and each catalog consists of a set of schemas

The schema definition statement has the following form:

CREATE SCHEMA [Name | AUTHORIZATION creatorIdentifier]

Creating a Table (CREATE TABLE)

Basic syntax:

```
CREATE TABLE TableName
    {columnName dataType [NOT NULL] [UNIQUE]
    [DEFAULT defaultOption] [CHECK (searchCondition) [,...]}
    [PRIMARY KEY (listofColumns),]
    {[UNIQUE (listofColumns)] [,...]}
    {[FOREIGN KEY (listofForeignkeyColumns)
    REFERENCES ParentTableName [(listof CandidatekeyColumns)]
        [MARCH {PARTIAL | FULL}
        [ON UPDATE referentialAction]
        [ON DELETE referentialAction]][,...]}
    {[CHECK (searchCondition)] [,...]}].
```

The remaining clauses are known as table constraints and can optionally be preceded with the clause.

CONSTRAINT ConstraintName.

which allows the constraint to be dropped by name using the Alter Table statement.

Changing a Table Definition (ALTER TABLE)

The ISO standard provides an ALTER TABLE statement for changing the structure of a table once it has been created. The definition of the alter Table statement in the ISO standard consists of SIX options.

* add a new column to a table.
* drop a column from a table.
* add a new table constraint.
* drop a table constraint.
* set a default for a column
* drop a default for a column.

Removing a Table (DROP TABLE)

Redundant table from the database using the DROP TABLE statement,

DROP TABLE TableName [RESTRICT | CASCADE]

RESTRICT : The DROP operation is rejected if there are any other objects that depend for their existence upon the continued existence of the table to be dropped.

CASCADE : The DROP operation proceeds and SQL automatically drops all dependent objects.

Creating an Index (CREATE INDEX)

Indexes are usually created to satisfy particular search criteria after the table has been in use for some time and has grown in size. The creation of Indexes is not standard SQL. However, most dialects support at least the following capabilities:

```
CREATE [UNIQUE] INDEX IndexName
ON TableName :(columnName [ASC | DESC] [,...])
```

The specified columns constitute the index key and should be listed in ~~major~~ minor order. Indexes can be created only on base tables not on views. If UNIQUE clause is used, uniqueness of the indexed column or combination of columns will be enforced by DBMS.

Removing an Index (DROP INDEX)

If we create an index for a base table and later decide that it is no longer needed we can use the DROP INDEX statement to remove the index from the database DROP INDEX has the following format:

```
DROP INDEX IndexName
```

Views.

The dynamic result of one or more relational operations operating on the base relations to produce another relation. A view is a virtual relation that does not necessarily exist in the database but can be produced upon request by a particular user, at the time of request.

Creating a View :→ The format of the CREATE VIEW statement is:

```
CREATE VIEW viewName [(newColumnName [, ...])]
    AS subselect [WITH [CASCADED | LOCAL] CHECK OPTION]
```

A view is defined by specifying an SQL select statement. A name may optionally be assigned to each column in the view. If a list of column names is specified, it must have the same number of items as the number of columns produced by the subset.

The subset is known as the defining query. If WITH CHECK OPTION is specified, SQL ensures that if a row fails to satisfy the WHERE clause of the defining query of the view, it is not added to the underlying base table of the view.

Removing a view :- A view is removed from the database with the DROP view statement:

DROP VIEW viewName [RESTRICT | CASCADE] ..

View Resolution :-

* The view column names in the SELECT list are translated into their corresponding column names in defining query.

* View names in the FROM clause are replaced with corresponding FROM lists of the def. query.

Restrictions on views: The ISO standard imposes several important restrictions on the creations and use of views, although there is considerable variations among dialects.

* If a column in the view is based on an aggregate function, then the column may appear only in SELECT and ORDER BY clauses of queries that access the view.

* A grouped view may never be joined with a base table or a view.

View updatability :- All updates to a base table are immediately reflected in all views that encompass that base table. Similarly, we may expect that if view is updated, the base table(s) will reflect that change.

Updatable view : For a view to be updatable, the DBMS must be able to trace any row or column back to its row or column in the source table.

Advantages on views	Disadvantages on views
Data independence	Update restriction
currency	Structure restriction
Improved security	Performance.
Reduced complexity	
convenience	
Customization	
Data Integrity	

SQL. Programming.

High level programming language to help develop more complex database applications.

SQL is a declarative language that handles rows of data and SQL and 3GLs use different models to represent data.

Declarations:

```
[DECLARE                    Optional
    - declarations
                            Mandatory
BEGIN
    - executable statements
                                        Optional
[EXCEPTION
    - exception handlers]
                                        Mandatory
END
```

General structure of PL/SQL Block.

Assignments: variable can be assigned in two ways:

1) using normal assignment statement (:=)
2) the result of an SQL SELECT or FETCH statement.

Control Statements: PL/SQL supports the usual conditional, iterative, and sequential flow-of-control mechanisms.

conditional IF statement:

```
IF (condition) THEN
        <SQL statement list>
[ELSE IF <condition> THEN <SQL statement list>]
[ELSE <SQL statement list>]
END IF;
```

conditional CASE

```
CASE (operand)
[WHEN (whenOperandList) | WHEN (searchCondition)
        THEN <SQL statement list>]
[ELSE <SQL statement>]
END CASE;
```

Iteration statement (LOOP)

```
[labelName:]
LOOP
    <SQL statement list>
    EXIT [labelName] [WHEN (condition)]
END LOOP [labelName];
```

Iteration statement [WHILE and REPEAT]

PL/SQL

```
WHILE (condition) LOOP
    <SQL statement list>
END LOOP [labelName];
```

SQL

```
WHILE (condition) DO
    <SQL statement list>
END WHILE [labelName];

REPEAT
    <SQL statement list>
UNTIL (condition)
END REPEAT [labelName];
```

Iteration statement (FOR)

PL/SQL

```
FOR indexVariable
    IN lowerBand ... upperBound LOOP
    <SQL statement list>
END LOOP [labelName];
```

SQL

```
FOR indexVariable
    AS querySpecification DO
    <SQL statement list]
END FOR [labelName];
```

Exceptions in PL/SQL.

An exception is an identifier in PL/SQL raised during the execution of a block that terminates its main body action. A block always terminates when an exception is raised, although the exception handler can perform some final actions.

Cursors in PL/SQL. cursors allows the rows of a query result to be accessed one at a time. It acts as a pointer to a particular row of the query result. The cursor can be advanced by 1 to access the next row. A cursor must be declared and opened before it can be used, and it must be closed to deactivate it after it is no longer required.

Passing parameters to cursors: PL/SQL allows cursors to be parameterized, so that the same cursor definition can be reused with different criteria.

Updating rows through a Cursor It is possible to update and delete a row after it has been fetched through a cursor.

M. Meenakshi

Prepared by.

Subject Expert

1. Dr. R. Saravanan.
2. Dr. K. ANAND
3. Dr. K. Bhuvaneswari

Case Study: Instance of the DreamHome rental database.

Branch

branchNo	street	city	postcode
B005	22 Deer Rd	London	SW1 4EH
B007	16 Argyll St	Aberdeen	AB2 3SU
B003	163 Main St	Glasgow	G11 9QX
B004	32 Manse RD	Bristol	BS99 1NZ
B002	56 Clover Dr	London	NW10 6EU

Staff

staffNO	fName	lName	position	sex	DOB	salary	branchNo.
SL21	John	White	Manager	M	1 Oct 45	30000	B005
SG37	Ann	Beech	Assistant	F	10 Nov 60	12000	B003
SG14	David	Ford	Supervisor	M	24 Mar 58	18000	B003
SA19	Mary	Howe	Assistant	F	19 Feb 70	9000	B007
SG5	Susan	Brand	Manager	F	3 Jun 40	24000	B003
SL41	Julie	Lee	Assistant	F	13 Jun 65	9000	B005

PropertyForRent

propertyNo	street	city	postcode	type	rooms	rent	ownerNO	staffNO	branchNO
PA14	16 Holhead	Aberdeen	AB7 5SU	House	6	650	CO46	SA9	B007
PL94	6 Argyll st	London	NW2	Flat	4	400	CO87	SL41	B005
PG4	6 Lawrance st	Glasgow	G11 9QX	Flat	3	350	CO40		B003
PG36	2 Manor Rd	Glasgow	G32 4QX	Flat	3	375	CO93	SG37	B003
PG21	18 Dale Rd	Glasgow	G12	House	5	600	CO87	SG37	B003
PG16	5 Novar Dr	Glasgow	G12 9AX	Flat	4	450	CO93	SG14	B003.

client

clientNO	fName	lName	telNO	prefType	maxRent	eMail.
CR76	John	Kay	0207-774-5632	Flat	425	john.kay@gmail.com
CR56	Aline	Stewart	0141-848-1825	Flat	350	astewart@hotmail.com
CR74	Mike	Ritchie	01475-392178	House	750	mritchie01@yahoo.com
CR62	Mary	Tregear	01224-196720	Flat	600	maryt@hotmail.com

PrivateOwner

OwnerNo	fName	lName	address	telNO	eMail	password
CO46	Joe	Keogh	2 Fergus Dr, Aberdeen AB2 7SX	01224 86 1212	jkeogh@gmail.com	******
CO87	carol	Farrel	6 Achray St, Glasgow G32 9DX	0141 - 357 - 7419	cfarrel@gmail.com	********
CO40	Tina	Murphy	63 Well St, Glasgow G42	0141 - 943 - 1728	tinam@hotmail.com	********
CO93	Tony	Shaw	12 Park Pl, Glasgow G4 0QR	0141 - 225 - 7025	tony.shaw@ark.com	********

Viewing

clientNo	propertyNo	viewDate	comment
CR56	PA14	24- May - 13	too small
CR76	PG4	20 Apr - 13	too remote
CR56	PG4	26 May 13	
CR62	PA14	14 May 13	no dining room
CR56	PG36	28 Apr 13	

Registration.

clientNo	branchNo	staffNo	dateJoined.
CR76	B005	SL41	2 Jan 13
CR56	B003	SG37	11 Apr 12
CR74	B003	SG37	16 Nov 11
CR62	B007	SA9	7 Mar 12.

Embedded SQL:

SQL Statements can be embedded into general purpose programming Language such as C, Java, .NET, PHP. The programming Language is called host Language.

An embedded SQL Statement is distinguished from programming Language Statements by prefixing it with the keywords Exec SQL, so that preprocessor (or precompiler) can seprate embedded SQL Statement can be terminated by a Semicolon (;) or a matching END EXEC.

Within the 'C' language to embed SQL code, some Special variable are used which are called "shared variable". These variables can be used in the both 'C' program and the embedded SQL Statements. Shared variables are prefixed by a colon (:), when they appear in SQL Statement.

Consider the example which has a C program, to process the company database. we need to declare program variables to match the type of database attributes that the program will process.

Shared variables are declared within

SQL data types INTEGER, SMALLINT, REAL & DOUBLE are mapped into C types long, short, float & double respectively. Fixed-length & varying length strings (CHAR[i], VARCHAR[i]) in SQL can be mapped to arrays of characters (char[i+1], varchar[i+1] in C that are one character long than SQL type.

The variable SQLCODE & SQLSTATE are used to communicate errors and exception conditions between database and program.

After each database command is executed, DBMS returns a value in SQLCODE. '0' value indicates the execution of SQL statement is successful. If SQLCODE > 0, indicates no more records are available. If SQLCODE < 0, indicates some errors has occured.

A value of '000000' in SQLSTATE indicate no error (or) exceptions, other values indicates error (or) exceptions.

The program reads (inputs) a PANO value and then receives) retrieves the Employee tuple with PAN from the database via the embedded SQL command. The INTO clause specifies the shared variable into which the attribute

CONNECT TO <SERVER name> AS <connection name>
Authorization <user account name and password>;

```
int loop;
EXEC SQL BEGIN DECLARE SECTION;
Varchar dname[10], fname[16], lname[16],
        address[31];
char pan[10], gender[2];
float salary, raise;
int dno;
int SQLCODE;
Char SQLSTATE[6];
EXEC SQL END DECLARE SECTION;
loop=1;
while (loop) {
prompt ("Enter PAN Number:", pan);
EXEC SQL
Select Fname, Lname, address, salary into :fname
lname, :address, :salary from Employee where
pan =:pan;
if (SQLCODE==0) printf (fname, lname, address,
                        salary);
else printf ("fname, lname, pan number does
              not exist:" pan);
prompt ("more PAN numbers (Enter for yes,
n for no):", loop);
```

SQLTYPES INTEGER, SMALLINT, REAL & DOUBLE are mapped into C types long, short, float and double.

CHAR[i], VARCHAR[i] in SQL can be mapped into arrays of characters (Char[i+1]), varchar[i+1] in C.

SQLCODE = 0 → Successful SQL execution

SQLCODE > 0 → No more SQL records available

SQLCODE < 0 → Some error occured.

Relational Algebra:

* It is a procedural query language.

It consists of a set of operations which take one or two relations as input and produce a new relation as their result.

The fundamental operations are:

1. Select (σ)
2. Project (Π)
3. Union (∪) | Intersection (∩).
4. Set difference (−)
5. Cartesian product (×)
6. Rename (ρ)

Fundamental operations:

* The select, project and rename operations are called unary operations, because they operate on one relation.

The other three operations operate on pairs of relations and therefore called binary operations.

Select operation (σ)

* It selects tuples that satisfy the given predicate [conditions] from a relation.

Notation → $\sigma_p r$

where,

σ → Stands for selection predicate

r → relations.

p → Prepositional logic formula which may use connection like and, or and not.

These terms may use relational operators like $=, \neq, <, >, \leq, \geq$

Books:

Subject	author	price	year
Database	Ramer	450	2015
CA	Patterson	475	2014
PDS	Allen	550	2005
TPDE	Veera	350	2012
EVS	Gilbert	275	2010

Eg:

$\sigma_{Subject = "database"}(Books)$

O/P:

database Ramer 450 2015.

$\sigma_{Subject = "database" \vee price = "450"}(BOOKS)$

O/P:

No rows selected:

Project Operation (π):

* It Project columns which satisfy a given predicate.

Notation → $\pi_{A1, A2 \ldots An}\, r$

where, A_1, A_2 → attribute names

r → relations

Duplicate rows are automatically eliminated as relation is a set.

Eg:

∏ subject, author (Books)

O/P:

database	Ramez
CA	Patterson
PDS	Allen
TPDE	Veera
EVS	Gilbert.

Subject:

Subcode	Subname	Semester	year:
CSb301	PDS	Third	Second
CSb302	DBMS	Third	Second
CSb401	OS	Fourth	Second
CSb402	DAA	Fourth	Second
CSb501	IP	FIFth	Third
CSb502	OOAD	FIFth	Third
CSb301	PDS	FIFth	Third

Union operation (∪):

The result of this operation, denoted by R ∪ S, is a relation, which includes all the tuples that are either in R or in S or in both R and S.

Duplicate tuples are eliminated

$\Pi_{sub.code}(\sigma_{semester = "Fifth" \wedge year = "Third"}(Subject))$.

O/P:
CS6501
CS6502
CS6301

$\Pi_{sub.code}(\sigma_{semester = "Third" \wedge year = "Second"}(Subject))$

O/P: CS6301
CS6302

$\Pi_{sub.code}(\sigma_{semester = "Fifth" \wedge year = "Third"}(Subject)) \cup \Pi_{sub.code}(\sigma_{semester = "Third" \wedge year = "Second"}(Subject))$

O/P: CS6501
CS6502
CS6301

Duplicate tuple containing CS6301 is

Set Difference (-):

The result of this operation is denoted by R-S, is a relation which includes all tuples that are in R but not in S.

Eg:

$\Pi_{sub.code}(\sigma_{semester = "Fifth" \wedge year = "Third"}(Subject)) - \Pi_{sub.code}(\sigma_{semester = "Third" \wedge year = "second"}(Subject))$

O/P: cs b501
cs b502

Intersection ($\cap$):

The result of this operation, denoted by $R \cap S$, is a relations, which includes all tuples that are in both R and S.

$\Pi_{sub.code}(\sigma_{semester = "Fifth" \wedge year = "Third"}(Subject)) \cap \Pi_{sub.code}(\sigma_{semester = "Third" \wedge year = "second"}(Subject))$.

O/P:

cs b301.

Cartesian product (×):

The cross product (or) cartesian product operation returns all possible combinations of rows in r with rows in s.

In other words, the result is every possible pairing of the rows of r and s.

Notation ⟶ r × s

where r and s are relation and their o/p will be defined as.

$$r \times s = \{ qt \mid q \in r \text{ and } t \in s \}$$

r

A	B
α	1
β	2

s

C	D	E
α	10	a
β	10	a
β	20	b
γ	10	b

r × s

A	B	C	D	E
α	1	α	10	a
α	1	β	10	a
α	1	β	20	b
α	1	γ	10	b
β	2	α	10	a
β	2	β	10	a
β	2	β	20	b

Rename operation: (ρ):

The result of relational algebra are also relations but without any name. The rename operation allows to rename the output relation.

Relation $\longrightarrow \rho_x E$

where, the result of expression E is saved with the name of x.

Oldname = Newname (r)

Oldname ⟹ Name of the attribute

Newname → New Name of the attribute

r → relation.

r

A	B
α	1
α	2
β	1

s

A	B
α	2
β	3

Eg: ρ (myRelation, (r * s))

A	B
α	1
β	1

→ my Relation.

ρ (myRelation (A1 → A2), (r - s))

myRelation:

A_2	B
α	1
β	1

Dynamic SQL:

When the pattern of database access is known in advance the Static SQL is very adequate. Sometimes, in ~~any~~ many applications, we may not known the pattern of database access in advance. It requires an advanced form of Static SQL known as dynamic SQL.

Using host variables, we can achieve a little bit of dynamics in embedded SQL (Static SQL).

Eg:

```
exec SQL Select fname, gender from teacher
                where salary >: sal;
```

Here, the salary will be asked on run time But getting column name (or) table name at run time is not possible with embedded SQL. For having such features, dynamic SQL is needed.

Dynamic SQL concepts:

In dynamic SQL, the SQL statements are not hard coded in the programming language, The text of the SQL statement is asked at runtime.

In dynamic SQL, the SQL statements the are to be executed are not known until run time. So DBMS can't get prepared for executing the statements in advances.

Dynamic statement Execution (Execute Immediate)

The Execute Immediate statements provides the simplest form of dynamic SQL. This statement passes the text of SQL statements to DBMS and asks the DBMS to execute the SQL statements immediately.

For using the statements our program goes through the following steps.

1. The program constructs a SQL statements as a single string of text in one of its data areas (called a buffer).

2. The Program passes the SQL Statements to the DBMS with the EXECUTE IMMEDIATE Statements.

3. The DBMS executes the Statements and sets the SQL CODE/ SQL STATE values to flag the finishing status save like, if the statements had been hard coded using static SQL.

SET operation: Union, INTERSECTION, MINUS

STUDENT

Fn	Ln
Ganesh	Babaj
Ram	Kumar
Arun	Prasad
Alfred	Paul
Sai	Tarun

Instructor

Fname	Lname
John	Smith
Ramez	Elnami
Alfred	Paul
Ram	Kumar

STUDENT - Instructor

Fn	Ln
Ganesh	Babaji
Arun	Prasad
Sai	Tarun

STUDENT ∪ Instructor

Fn	Ln
Ganesh	Babaji
Ram	Kumar
Arun	Prasad
Alfred	Paul
Sai	Tarun
John	Smith
Ramez	Elnami

STUDENT ∩ Instructor

Fn	Ln
Alfred	Paul
Ram	Kumar

Instructor - STUDENT

Fname	Lname
John	Smith
Ramez	Elnami

AD 8401 Database Design and Management.

UNIT - III Relational Database Design & Normalization.

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

Entity Relationship Modeling.

One of the most difficult aspects of database design is the fact that designers, programmers, and end-users tend to view data and its use in different ways. ER Modeling is a top-down approach to database design that begins by identifying the important data called entities and relationships between the data represented in the model. We then add more details, such as the information want to hold about the entities and relationships called attributes and any constraints on the entities, relationships, and attributes.

For diagrammatic notation that uses an increasingly popular object-oriented modeling language called the Unified Modeling Language (UML).

Entity type : A group of objects with the same properties, which are identified by the enterprise as having an independent existence.

The basic concept of the ER model is the entity type, which represents a group of "objects" in the "real world" with the same properties.

Entity occurance : A uniquely identifiable object of an entity type.

Physical existence	
Staff	Part
Property	Supplier
Customer	Product
Conceptual existence	
Viewing	Sale
Inspection	Work experience

Fig: Example of entities with a physical or conceptual existence.

Each uniquely identifiable object of an entity type is referred to simply as an entity occurance.

Relationship Types:-

A set of meaningful associations among entity type.

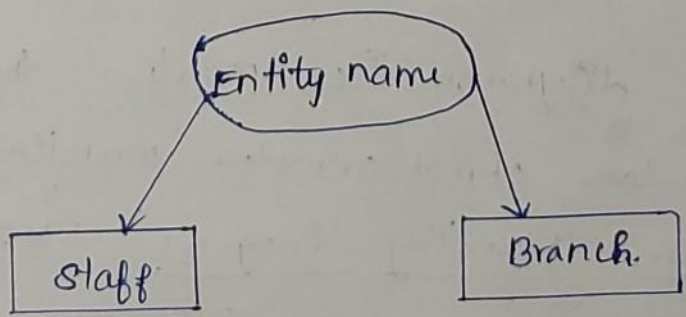


Fig: Diagrammatic representation of the staff and Branch entity types.

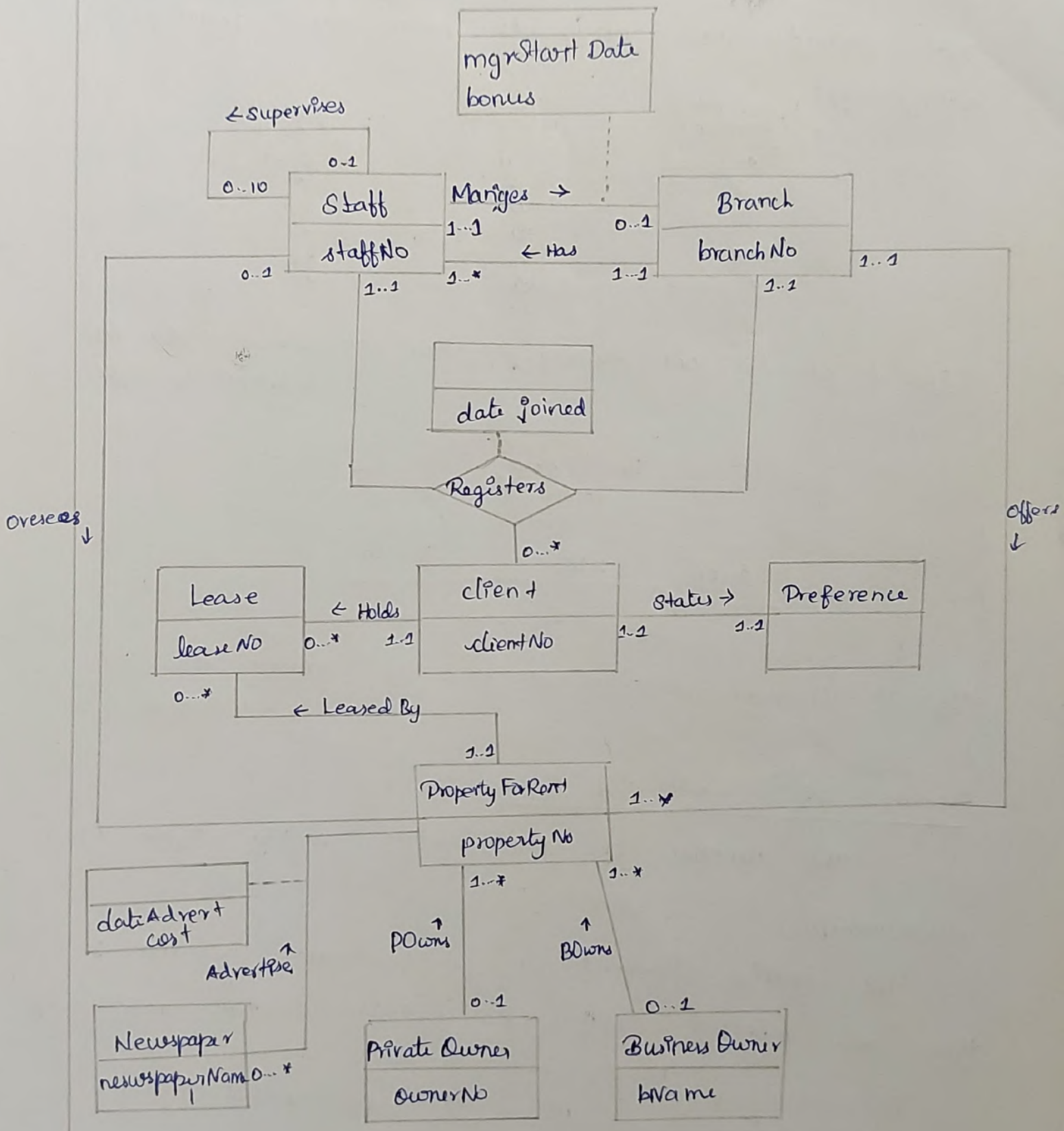


Fig: An Entity - Relationship (ER) diagram of the Branch view of 'Dream Home'

Relationship Occurance : An uniquely identifiable association that includes one occurance from each participating entity type.

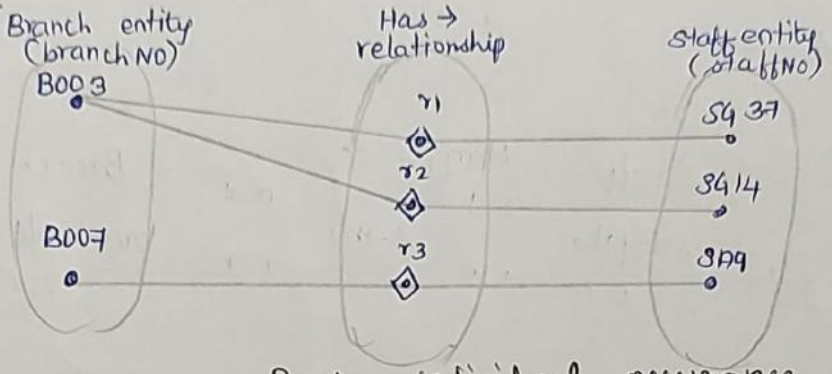


Fig: A sematic net showing individual occurance of the Has relationship type.

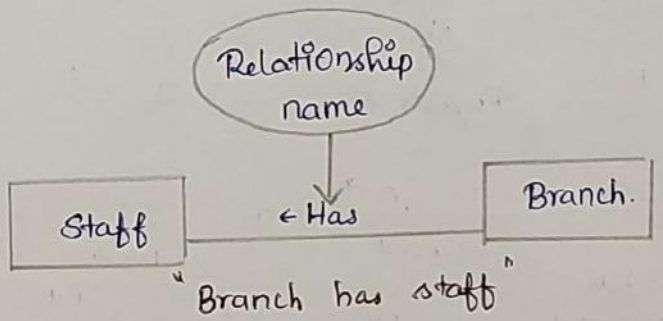


Fig: A diagrammatic representation of Branch Has Staff relationship type.

Degree of Relationship Type.

The number of participating entity types in a relationship.

The entities involved in a particular relationship type are referred to as participants in that relationship.

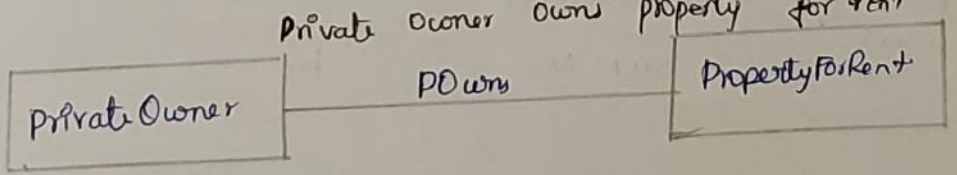


Fig: An example of a binary relationship called POwns

Therefore, the degree of a relationship indicates the number of entity types involved in a relationship.

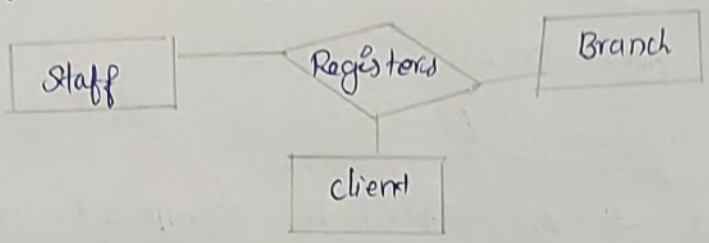


Fig: An example of a ternary relationship called Registers.

A relationship of degree three is called ternary.

Recursive Relationship.

A relationship type in which the same entity type participates more than once in different roles

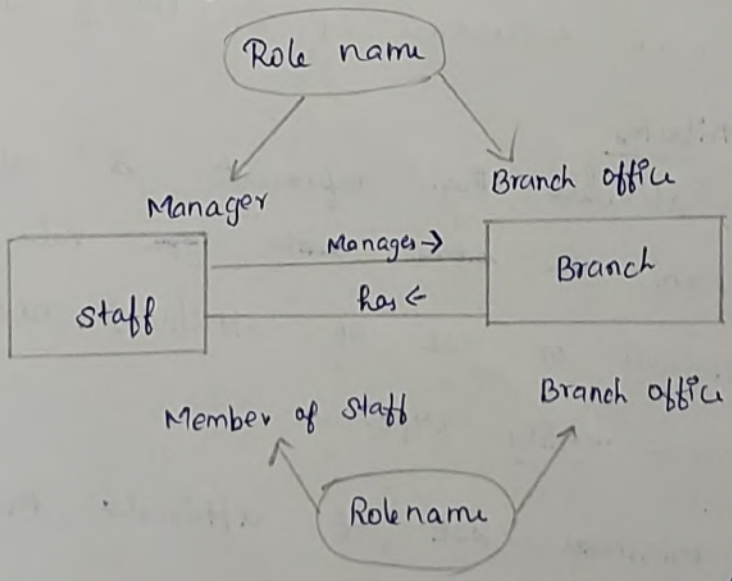


Fig: An example of entities associated through two distinct relationships called Manages and Has with role names.

Attribute: A property of an entity or a relationship typ

Attribute domain: The set of allowable values for one or more attributes.

Simple and composite Attributes

Simple attribute → An attribute composed of a single component with an independent existence.

Composite attribute → An attribute composed of multiple components, each with an independent existence.

Single-valued and Multi-valued Attribute.

Single-valued attribute → An attribute that holds a single value for each occurence of an entity type.

Multi-valued attribute → An attribute that holds multiple values for each occurance of an entity type.

Derived Attributes

An attribute that represents a value that is derivable from the ~~same entity type~~ value of a related attribute or set of attributes, not necessarily in the same entity type.

Key: The minimal set of attributes that uniquely identifies each occurence of a entity type is called candidate Key.

primary Key: The candidate Key that is selected to uniquely identify each occurance of a entity type.

Composite Key: A candidate key that consists of two or more attributes.

primary Key

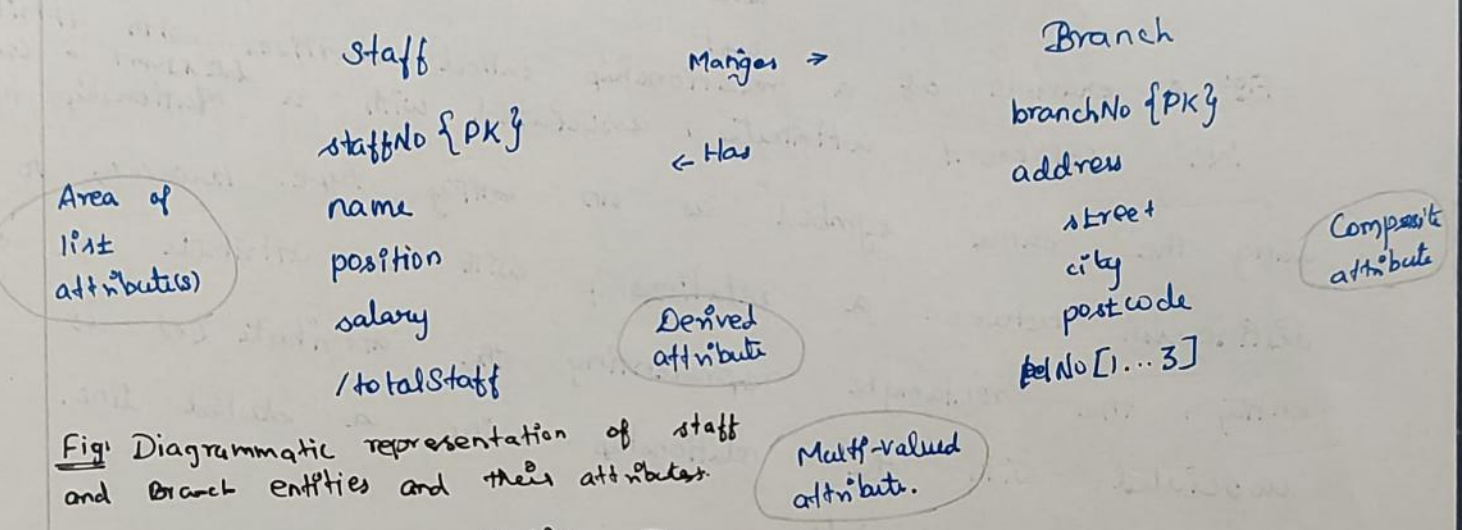


Fig: Diagrammatic representation of staff and Branch entities and their attributes.

Strong and Weak Entity Types.

Strong entity type :- An entity type that is not existence-dependent on some other entity type.

Weak entity type : An entity type that is existence-dependent on some other entity type.

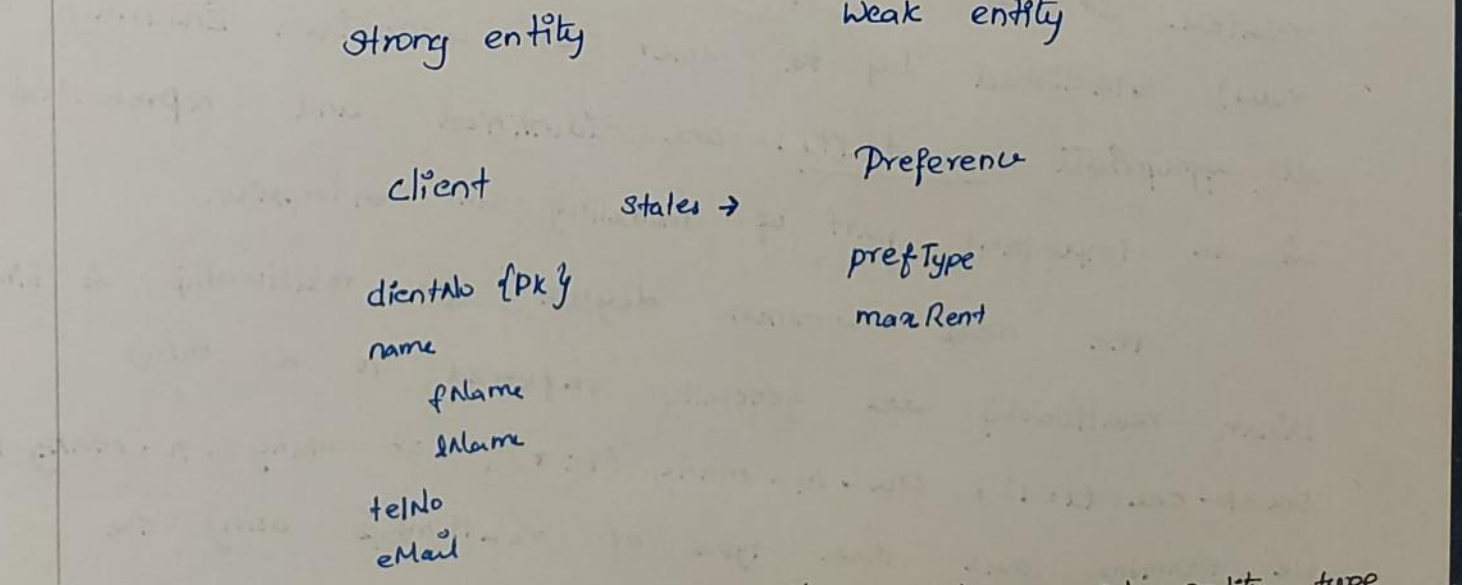
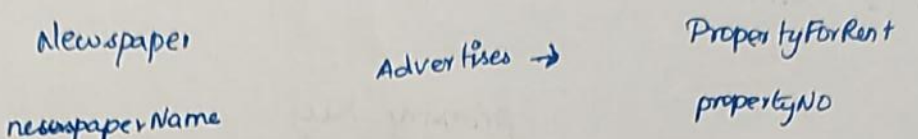
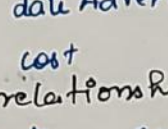


Fig: A strong entity type called Client and a weak entity type called Preference.

Attributes on Relationships.





Eg: A example of a relationship called advertises with attributes date Advert & cost.

We represent attributes associated with a relationship type using the same symbol as an entity type. However, to distinguish between a relationship with an attribute and an entity, the rectangle representing the attribute (s) is associated with the relationship using a dashed line.

Structured Constraints:

Multiplicity :- The number (or range) of possible occurrences of an entity type that may relate to a single occurrence of an associated entity type through a particular relationship.

Multiplicity contrains the way that entities are related. It is a representation of the policies (or business rules) established by the user or enterprise. Ensuring that all appropriate **constraints** are identified and represented is an important part of modeling an enterprise.

The most common degree for relationship is binary. Binary relationship are generally referred to a being one-to-one (1:1), one-to-many (1:*), or many-to-many (*:*). We examine these three types of relationships using the following integrity constraints.

* a member of staff manages a branch (1:1);
* a member of staff oversees properties for rent (1:*);
* newspaper advertise properties for rent (*:*).

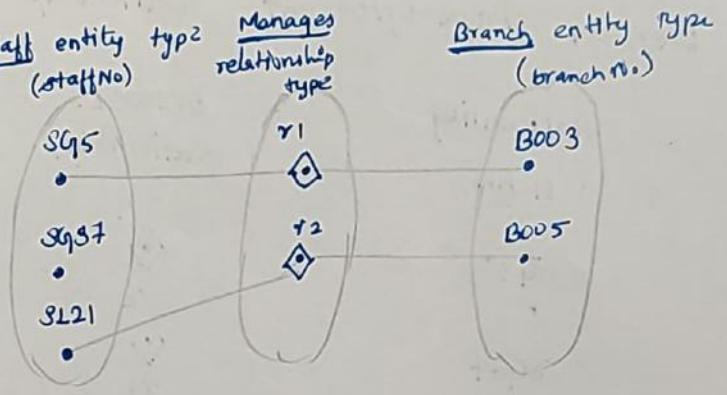
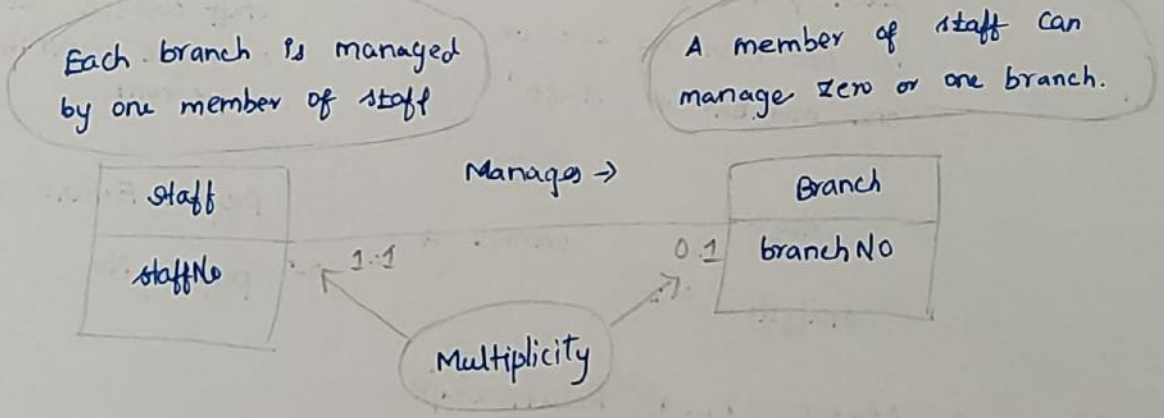


Fig: Semantic net showing two occurance of the staff Manages Branch relationship type.

One-to-One (1:1) Relationship.

As there is a maximum of one branch for each member of staff involved in this relationship and a maximum of one member of staff for each branch.

Fig: The multiplicity of the staff manages Branch one-to-one (1:1) relationship



One-to-Many (1:.*) Relationships.

Considir the relationship Oversees, which relates the staff and PropertyForRent relationship type denoted (r_1, r_2 and r_3) using a semantic net. Each relationship (r_n) represents the association between single staff entity occurence and a single PropertyForRent entity occurence. We represent each entity occurence using the values for the primary key attribute of the)

Staff and PropertyForRent entities, namely staffNo and propertyNo

Fig: Semantic net showing three occurrences of the Staff Oversees PropertyForRent relationship type.

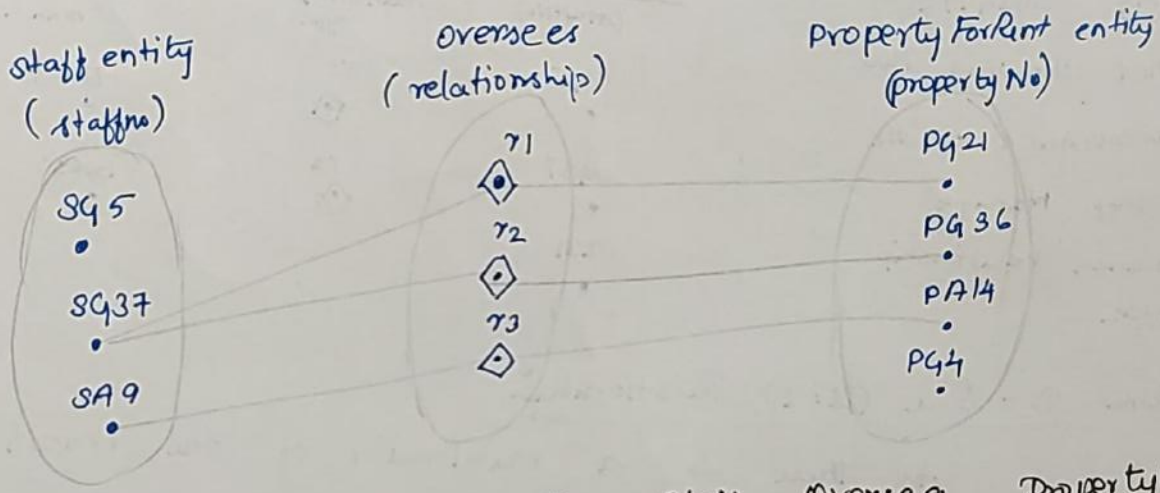
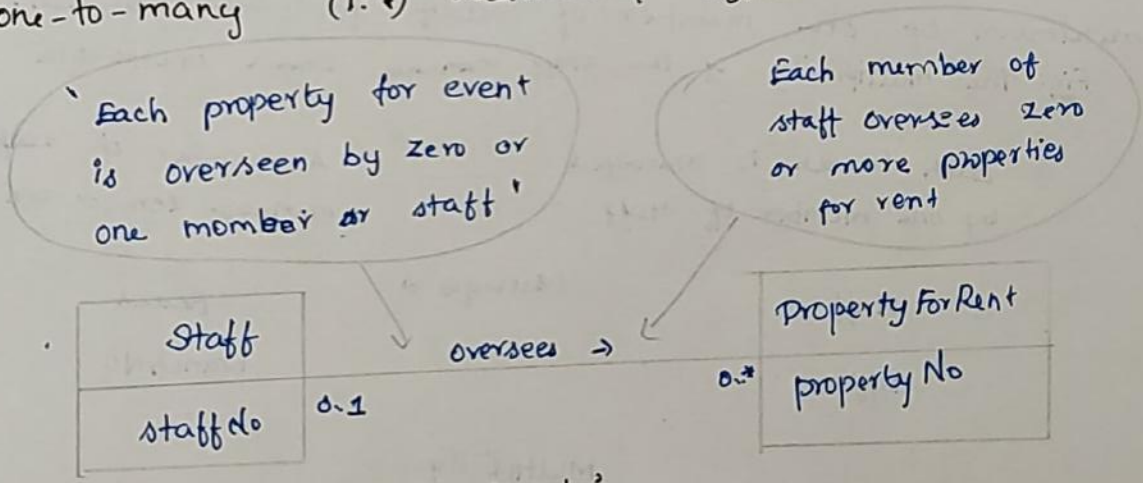


Fig: The multiplicity of the Staff Oversees PropertyForRent one-to-many (1:*) relationship type.



Many-to-Many (*...*) Relationships

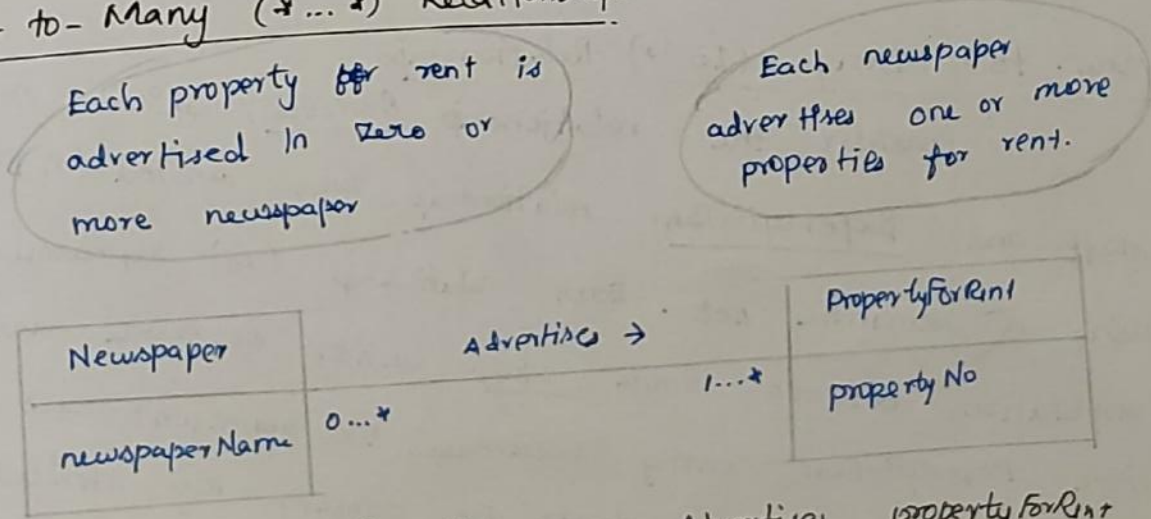


Fig: The multiplicity of the Newspaper Advertises PropertyForRent many-to-many (*..*) relationship

Multiplicity (complex relationship)

The number (or range) of possible occurrences of an entity type in an n-array relationship when the other (n-1) values are fixed.

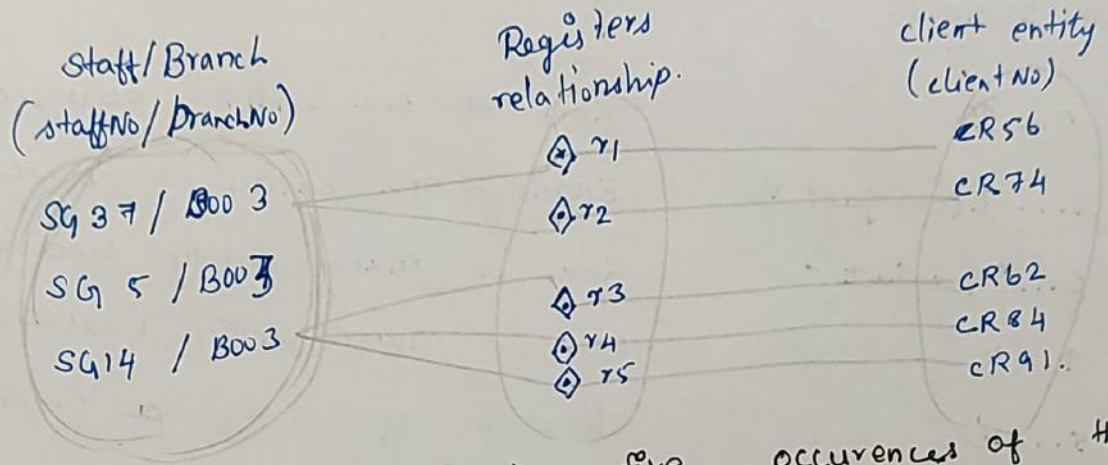


Fig: Semantic net showing five occurences of the ternary Registers relationship with values for staff and Branch entity type fixed.

Cardinality and Participation Constraints.

Multiplicity actually consists of two separate constraints known as cardinality and participation.

Cardinality: Describes the maximum number of possible relationship occurences for an entity participating in a given relationship type.

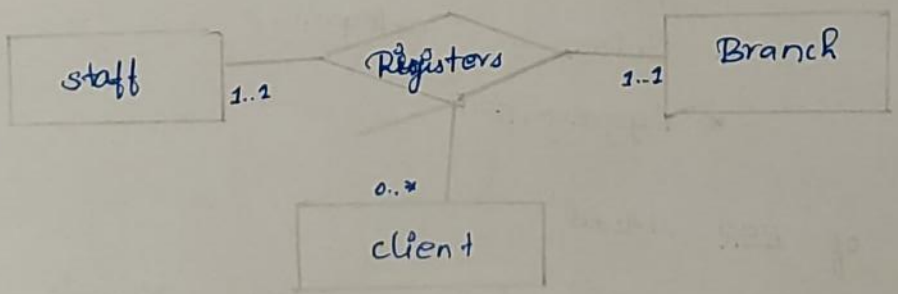


Fig: The multiplicity of the ternary Registers relationship

A Summary ways to represent multiplicity constraints.

Alternative ways to Represent Multiplicity constraints.	Meaning.
0...1	Zero or one entity occurance.
1...1 (or just 1)	Exactly one entity occurence.
0...* (or just *)	Zero or many entity occurence.
1...*	One or many entity occurence.
5...10	Minimum of 5 upto a maximum of 10 entity occurence.
0,3,6-8	Zero or three or six, seven, or eight entity occurences.

Enhanced Entity Relationship Model (EER Model)

EER is a high level data model that incorporates the extensions to the original ER Model. It is a digrammatic technique for displaying the following concepts.

* Sub class and super class
* Specialization and generalization.
* Union or categority
* Aggregation.

Features of EER Model.

* It creates a design more accurate to database schema

* It reflects the data properties and constraints more precisely.

* It includes all modeling concepts of the ER Model.

* Diagrammatic technique helps for displaying the EER schema.

* It includes the concept of specialization and generalization.

* It is used to represent a collection of objects (ii) union of objects of different entity types.

Sub class and Super class.

* Sub class and superclass relationship leads the concept of inheritance.

* The relationship between sub class and super class is denoted with (d) symbol.

Super class:

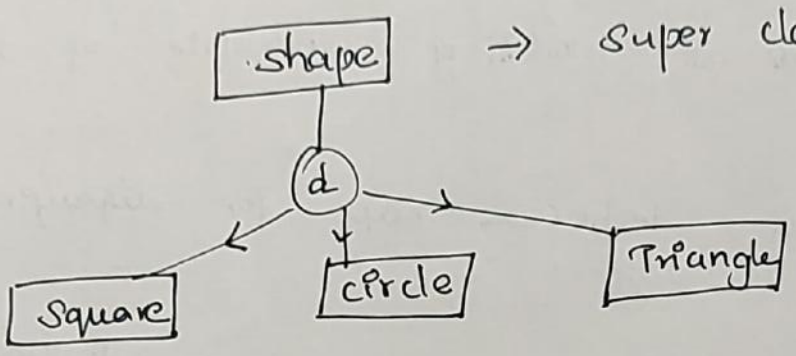
* Super class is an entity type that has a relationship with one or more sub-types.

* An entity cannot exist in database merely by being member of any super class.

Sub class:

* Sub class is a group of entities with attributes.

* It inherits properties and attributes from its super class.



→ super class

→ sub classes.

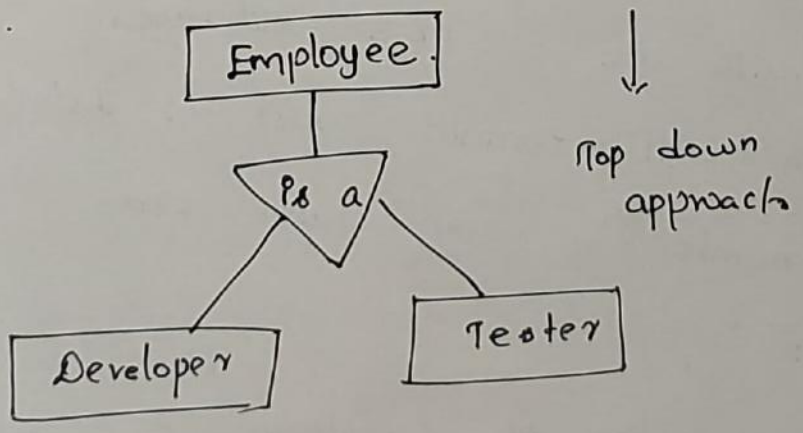
Specialization:

* It is a process which defines a group of entities which is divided into sub groups based on their characteristics.

* It is a top down approach, in which one higher entity can be broken into two lower level entity.

* It maximizes the difference between the members of an entity by identifying the unique characteristic or attribute of each member

* It defines one or more sub class for the superclass and also forms the superclass / sub class relationship.



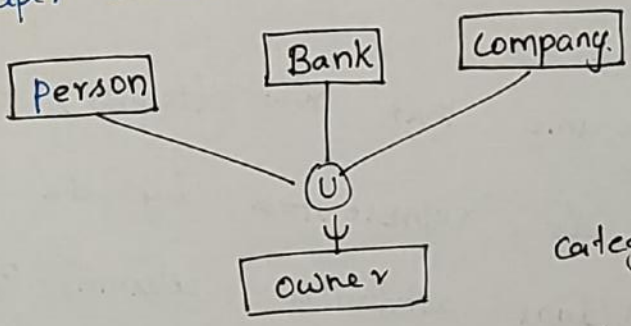
Top down approach

specialization.

Category or Union.

* Category represents a single super class or subclass relationship with more than one super class.
* It can be a total or partial participation.

For example, Car booking : Car owner can be a person, a bank (holds a possession on a car) or a company.
Category (sub class) → Owner is a subset of the union of the three super classes → Company, Bank and person.
A category member must exist in at least one of its super classes.

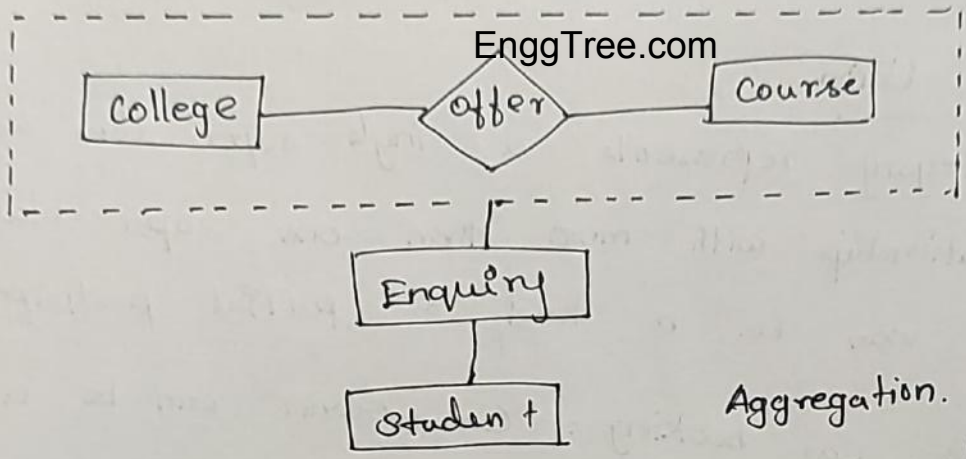


Categories (Union type).

Aggregation.

* It is a process that represents a relationship between a whole object and its component parts.
* It abstracts a relationship between objects and viewing the relationship as an object.
* It is a process when two entity is treated as a single entity.

In the given example, the relation between college and course is acting as an entity in relation with student.



Aggregation.

Functional Dependency.

Functional dependency is a term derived from mathematical theory which states that for every element in the attribute (which appears on some row), there is a unique corresponding element (on the same row).

Let us assume that row (tuples) of a relational table T is represented by the notation $r_1, r_2 \ldots$ and individual attributes (column) of the table is represented by letters A, B, ...

We can say that $A \rightarrow B$, A functionally determines B (or) B is functionally dependent on A. In other words, we can say that, given two rows R_1 and R_2, in table T, if $R_1(A) = R_2(A)$, then $R_1(B) = R_2(B)$.

A can sometimes called as determinant, whereas B is called dependent.

The following example illustrates the concept of functional dependency.

Student.

student ID	semester	subject	Lectures.
1234.	6	AI	Arun
1221	4	DBMS	Rajesh
1234	6	TOC	Peter
1201	2	BEEE	Ravi
1201	2	MII	Ram.

We notice that whenever two rows in this table feature the same student ID, they also necessarily have the same semester values. This basic fact can be expressed by a functional dependency.

Student ID → semester.

If you know the student ID, you can definitely know the semester.

Decomposition.

1. Decomposition is the process of breaking down in parts or elements.

2. It replaces a relation with a collection of smaller relations.

3. It breaks the table into multiple tables in a database.

4. It should always be lossless, because it confirms that the information in the original relation can be accurately reconstructed based on the decomposed relations.

5. If there is no proper decomposition of the relation, then it may lead to problems like loss of information.

6. Decomposition helps in eliminating some of the bad design problems such as redundancies, inconsistencies and anomalies.

There are two types of decomposition:

1) Lossy decomposition.

2) Lossless join decomposition.

Lossy decomposition.

The decomposition of relation R into R_1 and R_2 is lossy, when the join of R_1 and R_2 does not yield the same relation as in R.

Consider the table Student :->

Student.

Roll-No	Sname	Dept
111	Kumar	Computer
222	Kumar	Electrical.

This relation is decomposed into two relations No-name and name-dept.

No_Name

Roll-No	Sname
111	Kumar
222	Kumar

Name-Dept

Sname	Dept
Kumar	Computer
Kumar	Electrical

In lossy decomposition, spurious tuple are generated when a natural join is applied to the relation in the decomposition.

STU_JOINED

Roll-No	Sname	Dept
111	Kumar	Computer
111	Kumar	Electrical
222	Kumar	Computer
222	Kumar	Electrical

The above decomposition is a bad decomposition or lossy decomposition.

Spurious tuples:- It is a record in a database which is created when two tuples are joined badly [without primary key or foreign key]

Lossless join Decomposition [Non-loss decomposition]

The decomposition of a relation R into R_1 and R_2 is lossless, when the join of R_1 and R_2 yield the same relation as in R.

If the student table is decomposed into two relation stu-name and stu-dept.

STU-Name

Roll-No	Sname
111	Kumar
22	Kumar

STU-DEPT

Roll-No	Dept
111	Compulsr
222	Electrical

When these two relations are joined on the common attribute Roll_No [primary key], the resultant relation will look like the original student table.

STU-JOINED.

Roll-No	Sname	Dept
111	Kumar	Compulsr
222	Kumar	Electrical

In lossless decomposition, no spurious tuple are generated when a natural join is applied to the relations.

Dependency Preservation.

If we decompose a relation R into relations R_1 and R_2, All dependencies of R either must be a part of R_1 or R_2 (or) must be derivable from combination of FD's of R_1 and R_2.

For eg: 1

A relation R (A, B, C, D) with FD set $\{A \rightarrow BC\}$ is decomposed into R_1 (ABC) and R_2 (AD) is dependency preserving because FD $\{A \rightarrow BC\}$ is a part of R_1 (ABC).

Eg: 2

R (A, B, C, D) under $F = \{A \rightarrow B, B \rightarrow C\}$

Decomposition is R_1 (AB), R_2 (AC) and R_3(AD)

FD $\{A \rightarrow B\}$ is covered in R_1 (AB), but

FD $\{B \rightarrow C\}$ is uncovered in the decomposition.

∴ * The decomposition is not dependency preserving

If it is decomposed into R_1 (AB), R_2 (BC) & R_3(AD) then,

FD $\{A \rightarrow B\}$ is covered (preserved) in R_1 (AB)

FD $\{B \rightarrow C\}$ is coverd in R_2 (BC).

Hence the decomposition is dependency preserving.

Need for Normalization

Anomalies in DBMS.

There are 3 types of anomalies that occur when the database is not normalized. These are insertion, updation and deletion anomalies.

Eg: Consider the following relation.

Employee.

Emp-id	Emp-name	Emp-address	Emp-dept
101	Raj	Delhi	D001
101	Raj	Delhi	D002
123	Ravi	Agra	D890
166	Kumar	Chennai	D900
166	Kumar	Chennai	D004.

The above table is not normalized, the following problems exist when a table is not normalized.

Update Anomaly:

In the above table, we have two rows, for employee Raj, as he belongs to two departments. If we want to update the data in both rows, or the data will become inconsistent.

If somehow the correct address gets updated in one department, but not in other department, then as per the database, Raj would have two different address, which is incorrect and would lead to inconsistent data.

Insert anomaly:

Suppose a new employee joins the company, who is currently under training and not assigned to any department, then we would not be able to insert the data into the attribute emp-dept, since null values cannot be allowed.

Delete anomaly:

Suppose, if at a point of time, the company closes the department D890, then deleting the rows that have emp-dept as D890 would also delete the information of employee Ravi, since he is assigned only to this department.

To overcome these anomalies, we need to normalize the data.

Normalization:

It is the process of removing all kinds of anomalies from database

Various normalization forms are

1. First Normal form (1NF)
2. Second Normal form (2NF)
3. Third Normal form (3NF)
4. Boyce codd Normal form (BCNF)
5. Fourth Normal form (4NF)
6. Fifth Normal form (5NF)

First Normal Form (1NF)

"A relation R is in 1NF, if it does not have any composite attribute, multivalued attribute or their combination." In other words,

"All attributes (column) in the entity (table) must be single valued".

Repeating or multi-valued attributes are moved into a separate entity (table) and a relationship is established between the two tables or entities.

Customer

Cid	Name	Address		Contact-No.
		Street	city	
C01	aaa	ABC colony	Chennai	{12345678}
C02	bbb	xyz colony	Delhi	{23, 333, 456}
C03	ccc	Hoyls street	Blore	

The above table is not normalized. Solution for composite attribute.

Insert separate attributes in a relation for each sub attribute of a composite attribute.

Customer

Cid	Name	Street	city	Contact-No
C01	aaa	ABC colony	Chennai	{12345678}
C02	bbb	xyz colony	Delhi	{123, 333, 456}
C03	ccc	holly street	B'lore	

Solution for multi-valued attribute.

Remove the multi-valued attribute that violates INF and place it in a separate relation along with the primary key of given original relation.

Customer.

Cid	Name	Street	city.
C01	aaa	ABC colony	chenai
C02	bbb	xyz colony	Delhi
C03	ccc	kolojh Street	B'lore

Customer - Contact.

Cid	Contact No.
C01	1234567789
C02	123
C02	333
C02	456

The above tables are now in normalized form (1NF)

Second Normal Form (2NF)

A table is said to be in 2NF, if it holds the following two conditions.

1) Table should be in INF
2) No non-prime attribute is dependent on the proper subset of any candidate key of the table [dependent on part of primary key]

Consider the following relation:-

Teacher.

Teacher-id	Subject	Teacher_age
111	Maths	38
111	Physics	38
222	Biology	38
333	Physics	40
333	chemistry	40

Candidate Keys : {teacher-id, subject}

Non-prime attribute : teacher-age

The above table is in INF, because no multivalued attributes are present. All the attributes contains only one value [atomic value]

However, it is not in 2NF, because non-prime attribute teacher-age is dependent on teacher-id alone, which is a proper subset of candidate key. This violates the rule for 2NF, or the rule says "no non-prime attribute is dependent on the proper subset of any candidate key".

To make the table satisfies 2NF, the relation is split into two tables like this:

Teacher-Details

Teacher-id	Teacher-age
111	38
222	38
333	40

Teacher-Subject

Teacher-id	Subject
111	maths
111	Physics
222	Biology
333	Physics
333	chemistry

Now the table is in

The non-prime attribute teacher-age is fully dependent on primary key teacher-id, and no subset of candidate key.

Third Normal Form (3NF)

A table is said to be in 3NF, if it contains the following condition:

1. It should be in 2NF.
2. Transitive functional dependency should be removed.

[Every non-prime attribute of a table must be dependent only on primary key. In other words, a non-prime attribute should not be dependent on another non-prime attribute]

Transitive functional dependency.

A functional dependency is said to be transitive, if it is indirectly formed by two functional dependencies.

For eg: $x \to z$ is a transitive dependency if the following three functional dependies hold true:

1) $x \to y$
2) y does not $\to x$
3) $y \to z$

Eg:

Book	Author	Age
DBMS	Elmasri	66
CA	Mano	49
Java	Herbert	66

{Book} → {Author} [If we know the book, we know the author name]

{Author} does not → {Book}

{Author} → {Age}

∴ As per the rule of transitive dependency, {Book} → {Age}. If we the book name we can know the authors age.

3NF:

Consider the following table:

Student-Details.

Student-id	Student-Name	DOB	Street	City	Zip.

In this table Student-id is the primary key, Non-prime attribute student_name, DOB depends on Stu-id. but street and city depends on zip [non-prime]. The dependency between zip and other fields is called transitive dependency. Hence to apply 3NF, we need to move street, city to new table with zip as primary key.

Student Details.

Student-id	Student-name	DOB	Zip.

Address.

Zip	Street	city

Now the relations are in 3NF.

Minimal Cover

A Minimal cover is a simplified and reduced version of the given set of functional dependencies.

Since it is a reduced version, it is also called as irreducible set.

It is also called as canonical cover.

Steps to find Minimal Cover.

1) Split the right-hand attributes of all FDs.

Example:

$A \rightarrow XY \Rightarrow A \rightarrow X$, $A \rightarrow Y$

2) Remove all redundant FDs.

Example:

$\{A \rightarrow B, B \rightarrow C, A \rightarrow C\}$

Here $A \rightarrow C$ is redundant since it can already be achieved using the Transitivity Property.

3) Find the Extraneous attribute and remove it.

Example

$AB \rightarrow C$, either A or B or none can be extraneous

If A closure contains B then B is extraneous and it can be removed.

If B closure contains A then A is extraneous and it can be removed.

Example :

Minimize $\{A \to C, AC \to D, E \to H, E \to AD\}$

Step 1 $\{A \to C, AC \to D, E \to H, E \to A, E \to D\}$

Step 2 $\{A \to C, AC \to D, E \to H, E \to A\}$

Here Redundant FD : $\{E \to D\}$

Step 3 $\{AC \to D\}$

$\{D\}+ = \{A, C\}$

Therefore C is extraneous and is removed. $\{A \to D\}$

Minimal cover = $\{A \to C, A \to D, E \to H, E \to A\}$.

Example : 2

Minimize $\{AB \to C, D \to E, AB \to E, E \to C\}$

Step 1 $\{AB \to C, D \to E, AB \to E, E \to C\}$

Step 2 $\{D \to E, AB \to E, E \to C\}$ Here Redundant FD = $\{AB \to C\}$

Step 3 $\{AB \to E\}$

$\{A\}+ = \{A\}$

$\{B\}+ = \{B\}$ There is no extraneous attribute.

Therefore minimal cover = $\{D \to E, AB \to E, E \to C\}$.

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 schema $D = \{R_1, R_2, \ldots R_n\}$ relational database schema, where 'D' is called the Decomposition of R. The attribute in R will appear in at least one relation schema R_i in the decomposition (ii) no attribute is lost. This is called attribute preservation condition of decomposition.

2. Dependency Preservation:

If each functional dependency $X \to Y$ specified in F appears directly in one of the relation schemas R_i in the decomposition D or could be inferred from the dependencies that appear in some R_i.

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

$R = \{A, B, C\}$

$F = \{A \to B, B \to C\}$

Key = $\{A\}$

R is not in BCNF

Decomposition $R_1 = (A, B)$, $R_2 = (B, C)$.

R_1 and R_2 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 appears in some relation R_i. It is sufficient that the union of dependencies on all the relations R_i 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 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 of $R_1, R_2, \ldots R_n$ of a relation R is called a lossless decomposition for R if the natural join of $R_1, R_2 \ldots R_n$ produces exactly the relation R.

A decomposition is lossless if we can recover:

* R (A, B, C) → Decompose → R_1(A, B) R_2(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.

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 inferene 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 (IR$_1$).

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 (IR_2)

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 (IR_3)

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

if $X \rightarrow Y$ and $Y \rightarrow Z$ then $X \rightarrow Z$.

4) Union Rule (IR_4).

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 IR_2 on 1 by augmentation with X. where $XX = X$)
4. $XY \rightarrow YZ$ (using IR_2 on 2 by augmentation with Y)
5. $X \rightarrow YZ$ (using IR_3 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 $X \rightarrow YZ$ then $X \rightarrow Y$ and $X \rightarrow Z$

Proof.

1. $X \rightarrow YZ$ (given)
2. $YZ \rightarrow Y$ (using IR. Rule)
3. $X \rightarrow Y$ (using IR_3 on 1 and 2)

6. Pseudo transitive Rule (IR_6).

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

If $X \rightarrow Y$ and $YZ \rightarrow W$ then $XZ \rightarrow W$.

Proof:

1. $X \rightarrow Y$ (given)
2. $WY \rightarrow Z$ (given)
3. $WX \rightarrow WY$ (using IR_2 on 1 by augmenting with W)
4. $WX \rightarrow Z$ (using IR_3 on 3 and 2)

Boyce Codd Normal form (BCNF) - 3.5 NF.

The official qualifications for BCNF are:

1) A table is already in 3NF

2) All determinants must be superkeys.

All determinants that are not superkeys are removed to place in another table.

A relational schema R is considered to be Boyce-Codd Normal form (BCNF) if, for every one of its dependencies $X \rightarrow Y$, one of the following conditions holds true:

* $X \rightarrow Y$ is a trivial functional dependencies (i.e Y is a subset of X)
* X is a superkey for schema R

Example

Let's take a look at this table, with some typical data. The table is not in BCNF.

Author	Nationality	Booktitle	Genre	Number of pages.
William Shakespeare	English	The comedy of Errors	comedy	100
Markus Winand	Austrian	SQL Performance Explained.	Textbook	200
Jeffrey Ullman	American	A first course in Database system.	Textbook	500
Jennifer Widow	American	A first course in Database system.	Textbook	500

The nontrivial functional dependencies in the table are

author → nationality.

book title → genre, no. of pages.

We can easily see that the only key is the set.

{author, book title}.

The same data can be stored in a BCNF schema.

However, the time we would need three tables.

Author	Nationality
William Shakespeare	English
Markus Winand	Austrian
Jeffrey Ullman	American
Jennifer Widow	American

Book title	Genre	Number of pages
The Comedy of Errors	comedy.	100
SQL performance Explained	Textbook	200
A first course in Database systems.	Textbook	300.

Author	Book title.
William Shakespeare	The Comedy of Errors
Markus Winand	SQL performance explained
Jeffrey Ullman	A first course in Database systems
Jennifer Widow	A first course in Database systems.

The functional dependencies for this schema are the same as before:

author → nationality.

book title → genre, number of pages.

The Key of the first table is {author}. The Key of the second table is {book title}. The key of the third table is {author, book title}. There are no functional dependencies violating the BCNF rules, so the schema is in Boyce-Codd Normal form.

AD8401 - Database Design and Management.

Unit - IV Transaction Management.

Transaction concepts - Properties - Schedules - Serializability - Concurrency Control - Two phase locking Techniques.

Single-User versus Multiuser Systems

One criterion for classifying a database system is according to the number of users who can use the system concurrently. A DBMS is single-user if at most one user at a time can use the system, and it is multiuser if many users can use the system - and hence access the database - concurrently.

Single-user DBMSs are mostly restricted to personal computer systems; most other DBMSs are multiuser. For example, airline reservations systems and banking systems are many multiuser systems. Multiple users can access database - and use computer systems - simultaneously because of the concept of multiprogramming, which allows the operating system of the computer to

execute multiple programs or processes at the same time. A single central processing unit (CPU) can only execute at most one process at a time. However, multiprogramming operating systems execute some commands from the next process, and so on.

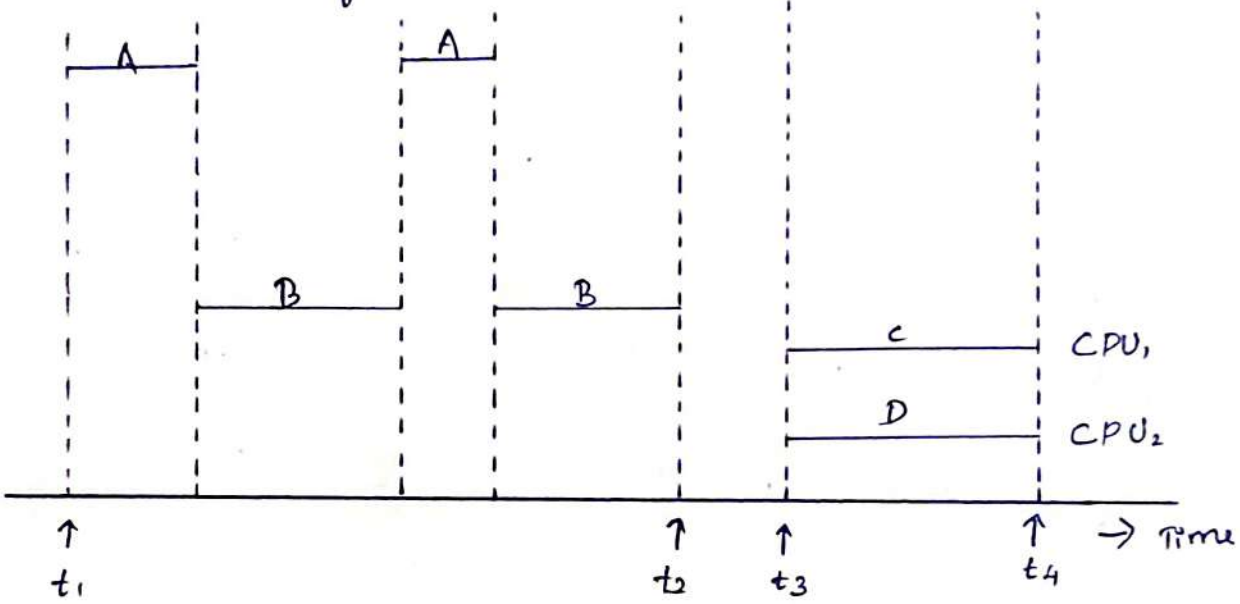


Fig: 1: Interleaved processing versus parallel processing of concurrent transactions.

A process is resumed at the point where it was suspended whenever it gets its turn to use the CPU again. Hence, concurrent execution of processes is actually interleaved, as illustrated in the above figure 1, which shows two processes, A and B, executing concurrently in an interleaved fashion. Interleaving keeps the CPU

busy when a process requires an input or output (I/O) operation, such as reading a block from disk. The CPU is switched to execute another process rather than remaining idle during I/O time. The CPU is switched to execute another process rather than remaining idle. Interleaving also prevents a long process from delaying other processes.

If the computer system has multiple hardware processors (CPUs), parallel processing of multiple processes is possible, as illustrated by processes C and D in figure 1. Most of the theory concerning concurrency control in databases is developed in terms of interleaved concurrency. So for the remainder of this chapter. In a multiuser DBMS, the stored data items are the primary resources that may be accessed concurrently by interactive users or application programs, which are constantly retrieving information from and modifying the database.

Transactions.

A transaction is an executing program which forms a logical unit of database processing.

A transaction includes one or more database operations which includes, insertion, deletion, modification or retrieval operations.

One way of specifying the transaction boundaries is by specifying explicit statements.

Begin transaction
&
End transaction

T_1 :
read (A)
A := A - 50
write (A)
read (B)
B := B + 50
write (B)

In an application program, All database operations between these two boundaries are considered as a single transaction. A single application program may contain more than one transaction, if it contains several transaction boundaries

Consider the basic database access operations that a transaction can include as follows:-

read - item (x) : Reads a database item named 'x' into a program variable 'x'.

write - item (x) : Writes the value of program variable 'x' into the database item named 'x'.

Block → The basic unit of data transfer from disk to main memory is one block.

Executing a read - item (x) command includes the following steps:

1) find the address of the disk block which contains item 'x'.

2) Copy that disk block into a main memory buffer [If that disk block is not already in some main memory buffer]

3) Copy item x from the buffer to the program variable named x.

Executing a write - item (x) command includes the following steps:

1. find the address of the disk block which contains item x.

2. copy that disk block into a buffer in main memory (If that block is not already in some main memory buffer).

3. Copy item x from program variable x into its correct location in the buffer.

4. Store the updated block from the buffer back to disk.

Step 4 actually updates the database on disk. In some cases, the buffer is not immediately stored to disk, in case additional changes are to be made to the buffer.

A transaction is a logical unit of work on a database.

Eg:

```
Begin transaction
    read_item (x)
    write_item (x)
End transaction.
```

Transaction and system concepts.

Transaction states and Additional Operations.

A transaction is an automic unit of work that should either be completed in its entirety or not done at all. For recovery purposes, the system needs to keep track of when each transaction starts, terminates, and commits, or aborts. Therefore, the recovery manager of the DBMS needs to keep track of the following operations:

BEGIN-TRANSACTION: This marks the beginning of transaction execution.

READ or WRITE: These specify read or write operations on the database items that are executed as part of a transaction.

END-TRANSACTION: This specifies that READ and WRITE transaction operations have ended and marks the end of transaction execution. However, at this point it may be necessary to check whether the

Changes introduced by the transaction can be permanently applied to the database (committed) or whether the transaction has to be aborted because it violates serializability or for some other reason.

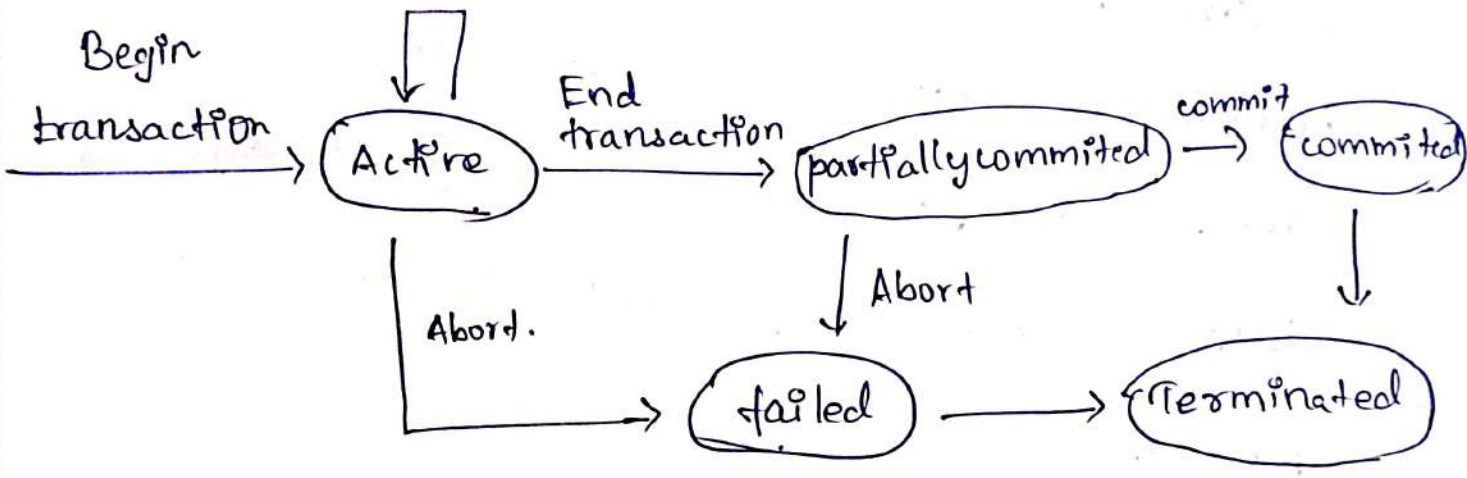


Fig 2 State transition diagram illustrating the states for transaction execution.

COMMIT - TRANSACTION : This signals a succesful end of the tramaction so that any changes (updates) executed by the transaction can be safely commited to the database and will not be undone.

ROLLBACK: or (ABORT) → This signals that the transaction has ended unsuccessfully, so that any changes or effects that the transaction may have applied to the database must be undone.

The figure 2 shows the state transition diagram that illustrates how a transaction moves through its execution states. A transaction goes into an active state immediately after it starts execution, where it can execute its READ and WRITE operations. When the transaction ends, it moves to the partially committed state. At this point, some types of concurrency control protocols may do additional checks to see if the transaction can be committed or not. Also, some recovery protocol need to ensure that a system failure will not result in an inability to record the changes of the transaction permanently. If these checks are successful, the transaction is said to have reached its commit point and enters the committed state. ~~Commit points~~ When a

transaction is committed, it has concluded its execution successfully and its changes must be recorded pemanently in the database, even if a system failure occurs.

However, a transaction can go to the failed state if one of the checks fails or if the transaction is aborted during its active state. The transaction may then have to be rolled back to undo the effect of its WRITE operations on the database. The terminated state corresponds to the transaction leaving the system. The transaction information that is maintained in system tables while the transaction has been running is removed when the transaction terminates. Failed or aborted transactions may be restarted later - either automatically or after being resubmitted by the user - as brand new transactions.

The System Log :-

To be able to recover from failures that affect transactions, the system maintains a log to keep track of all transaction operations that affect the values of database items, as well as other transaction information that may be needed to permit recovery from failures. The log is a sequential, append-only file that is kept on disk, so it is not affected by any type of failure except for disk or catastrophic failure. Typically, one more main memory buffers, called the log buffers, hold the last part of the log file, so that log entries are first added to the log main memory buffer. When the log buffer is filled, or when certain other conditions occur, the log buffer is appended to the end of the log file on disk.

Desirable Properties of Transactions.

Transactions should possess several properties, often called the ACID properties they should be enforced by the concurrency control and recovery methods of the DBMS. The following are the ACID properties:

* **Atomicity:** A transaction is an atomic unit of processing; it should either be performed in its entirely or not performed at all

* **Consistency preservation:** A transaction should be consistency preserving, meaning that if it is completely executed from beginning to end without interference from other transactions, it should take the database from one consistent state to another.

* **Isolation:** A transaction should appear as though it is being executed in isolation from transactions, even though many transactions are executing concurrently. That is, the execution of a transaction should not be interfered with by any other transactions executing concurrently.

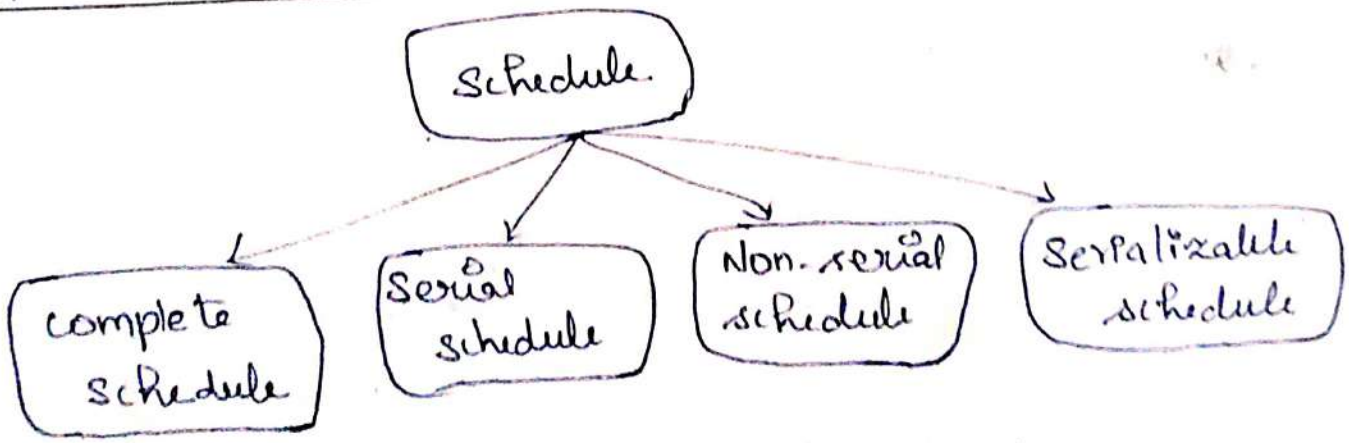
* Durability or permanency:

The changes applied to the database by a committed transaction must persist in the database. These changes must not be lost because of any failure.

Schedules (Scheduling of transactions)

A chronological execution sequence (predefined order) of a transaction is called a schedule. A schedule can have many transactions in it, each comprising of a number of instructions / tasks.

Types of schedules:



1) Complete schedule A schedule that contains either an abort or commit for each transaction, whose actions are listed in it is called a complete schedule. Either the transactions should

be fully completed (or) not fully completed.

2) Serial Schedule

It is a schedule in which, after the completion of one transaction, second transaction takes place. For example, consider a schedule of transactions T_1 & T_2. Assume that A = 1000, B = 2000.

S_1

T_1	T_2
A = A + 100	
B = B - 100	
	A = A * 0.1
	B = B * 0.1

When transaction T_1 completes its execution, then the transaction T_2 takes place. This is called serial schedule.

3. Non-serial schedule:

It is a schedule, in which the operations from a set of transactions will be executed in an interleaved manner. It is called non-serial schedule.

S_2

T_1	T_2
A = A + 100	
	A = A * 0.1
B = B - 100	
	B = B * 0.1

4) Serializable schedule

A schedule is said to be seriable, if

i) It is a non-serial schedule.

ii) It produces the same result as that of its equivalent serial schedule.

Eg:

S_1

T_1	T_2
A = A + 100	
B = B - 100	
	A = A * 0.1
	B = B * 0.1

S_2

T_1	T_2
A = A + 100	
	A = A * 0.1
B = B - 100	
	B = B * 0.1

Assume

A = 1000

B = 2000

The output of both the schedules S_1 & S_2 for A = 110 and B = 190 are same. Hence, the non-serial schedule S_2 is serializable with its equivalent serial schedule S_1.

$S_2 = S_1$ [serializable]

Serializability:

A Schedule 'S' of 'n' transactions is serializable, if it is equivalent to some serial schedule of the 'n' transactions. This property is called serializability.

Example of Serializable schedule or not?

consider schedule S.

T_1	T_2
W (A)	
	R (A)
	R (B)
W (B)	

Read A after updation

Read B before updation

Let us consider schedules S_1, S_2 & S_3 and check whether they are serializable with S or not.

S_1

T_1	T_2
W (A)	
W (B)	
	R (A)
	R (B)

Read A after updation

Read B after updation ✗

$S \neq S_1$ [Not serializable]

S_2

T_1	T_2
	R(A)
	R(B)
W(A)	
W(B)	

∴ Read A before updation ✗
Read B before updation
$S \neq S_2$ [Not serializable]

S_3

T_1	T_2
W(A)	
	R(B)
	R(A)
W(B)	

⇒ It is a non-serial schedule
Hence $S \neq S_3$ [Not serializable]

Conflict equivalence & Conflict serializability.

A pair of operations are said to be in conflict, if they satisfy the following conditions:

1) Both operations belong to different transactions
2) They access the same data.
3) At least one of the operations is write operation.

Eg:

1. $T_1 : R(A)$ $T_2 : R(A)$

T_1	T_2
R(A)	
	R(A)

All the above conditions are not satisfied. only the first two conditions are satisfied. So Non-conflict pair.

2. $T_1 : R(A)$ $T_2 : W(A)$

T_1	T_2
$R(A)$	$W(A)$

All the 3 conditions are satisfied. Hence, conflict pair.

3. $R_1(A)$; $W_2(B)$ } → Not conflict pairs. Because they
4. $W_1(A)$: $R_2(B)$ } access different data.

Swapping of operations is possible if the pair of operations are non-conflict pair

Eg:

S_1

T_1	T_2
$R(A)$	
	$R(B)$

Swapped ⇒

S_2

T_1	T_2
$R(B)$	
	$R(A)$

$S_1 = S_2$

Non-conflict pairs

T_1	T_2
$R(A)$	
	$W(A)$

Swapped ⇒

T_1	T_2
	$W(A)$
$R(A)$	

$S_1 \neq S_2$

conflict pairs

When are 2 schedules equivalent?

There are 3 types of schedule equivalences:

1) Result equivalence
2) conflict equivalence
3) View equivalence.

Based on the types of equivalence, different types of serializability are defined. They are

1) Result serializability
2) Conflict serializability
3) View serializability

Result equivalence & Result serializability.

In result equivalence, the end result of schedules heavily depend on input of schedules. The final values are calculated from both schedules (given and serial) and check whether they are equal. Result serializability is not generally used because of lengthy process.

S_1	S_2
$x = 100$	$x = 100$
$R(x)$	$R(x)$
$x = x + 10$	$x = x * 1.1$
$w(x)$	$w(x)$
$= 110$	$= 110$

For $x = 100$, $S_1 \overset{R}{=} S_2$

If $x = 200$, they are not equal.

$S_1 \overset{R}{\neq} S_2$

Conflict equivalence

Schedules are conflict equivalence, if they can be transformed one into another, by interchanging a sequence of non-conflicting adjacent operations.

Eg: check whether the schedules S_1 & S_2 are conflict equivalent or not.

S_1 : $\frac{1}{R_1(A)}$; $\frac{2}{R_2(B)}$; $\frac{3}{W_1(A)}$.

S_2 : $\frac{4}{R_1(A)}$; $\frac{5}{W_1(A)}$; $\frac{6}{R_2(B)}$.

Solution

S_1

T_1	T_2
1 $R_1(A)$	
	$R_2(B)$ 2
3 $W_1(A)$	

S_2

T_1	T_2
1 $R_1(A)$	
2 $W_1(A)$	
	$R_2(B)$ 3

In S_2, [2] & [3] are interchangeable, ∵ they are non-conflict pairs.

Thus $S_1 = S_2$

S_1 is in conflict equivalence with S_2.

Eg: check whether the schedules S_1 & S_2 are conflict equivalent or not.

$S_1 : \frac{1}{R_2(A)} ; \frac{2}{W_2(A)} ; \frac{3}{R_3(C)} ; \frac{4}{W_2(B)} ; \frac{5}{W_3(A)} ; \frac{6}{W_3(C)} ;$ (21)

$\frac{7}{R_1(A)} ; \frac{8}{R_1(B)} ; \frac{9}{W_1(B)} ; \frac{10}{W_1(C)}.$

$S_2 : \frac{1}{R_3(C)} ; \frac{2}{R_2(A)} ; \frac{3}{W_2(A)} ; \frac{4}{W_2(B)} ; \frac{5}{W_3(A)} ; \frac{6}{R_1(A)} ;$

$\frac{7}{R_1(B)} ; \frac{8}{W_1(B)} ; \frac{9}{W_1(C)} ; \frac{10}{W_1(C)}$

S_1

T_1	T_2	T_3
	$R_2(A)$	
	$W_2(A)$	
		$R_3(C)$
	$W_2(B)$	
		$W_3(A)$
		$W_3(C)$
$R_1(A)$		
$R_1(B)$		
$W_1(B)$		
$W_1(C)$		

S_2

T_1	T_2	T_3.
		$R_3(C)$ 1
2	$R_2(A)$	
3	$W_2(A)$	
4	$W_2(B)$	
		$W_3(A)$ 5
$R_1(A)$		
$R_1(B)$		
$W_1(B)$		
$W_1(C)$		
		$W_3(C)$ 10

In S_2,

[1] & [2] can be interchanged, than after that

[2] & [3] can be interchanged.

[6, 7, 8, 9] & [10] can be interchanged with each other.

After completing, the swapping of non-conflict pairs of S_2, finally S_1 & S_2 are conflict equivalind.

$S_1 \overset{c}{=} S_2$

Conflict Serializable schedule.

A schedule is conflict serializable, if it is conflict equivalent to any of serializable schedule.

Testing of conflict serializability

Method - 1:

1. First write the given schedule in a linear way
2. Find the conflict pairs (RW, WR, WW) on same variable by different transaction.
3. Whenever conflict pairs occur, write the dependency relation like $T_i \rightarrow T_j$. If conflict pair is from T_i to T_j. Eg: $(W_1(A), R_2(A)) = T_1 \rightarrow T_2$
4. check to see if there is a cycle formed.
   - * If yes, then not conflict serializable
   - * if no, then conflict serializable

Example:

check whether the schedule is conflict serializable or not?

$S_1 : R_1(A) ; W_1(A) ; R_2(A) ; R_1(B) ; W_1(B) ; R_2(B)$

Solution

	T_1	T_2
1	$R_1(A)$	
2	$W_1(A)$	
3		$R_2(A)$
4	$R_1(B)$	
5	$W_1(B)$	
6		$R_2(B)$

conflict pairs:

(2,3) $W_1(A) : R_2(A)$

(5,6) $W_1(B) : R_2(B)$

$W_1(A) : R_2(A) \Rightarrow T_1 \rightarrow T_2$

$W_1(B) : R_2(B) \Rightarrow T_1 \rightarrow T_2$

No cycle is formed, ∴ the given schedule is conflict serializable schedule.

Method 2:

To test the conflict serializability, we can draw a $G = (V, E)$ where.

V = vertices = no. of transactions.

E = Edges = for conflicting pair.

Steps:

1. create node for each transaction.
2. find the conflict pairs.
3. Draw edge for the conflict pair.

Eg: $W_2(B) : R_1(A)$ is a conflict pair, draw edge from T_2 to T_1.

4. Testing conditions for conflict serializability.

* If precedence graph is cyclic, the given schedule is not conflict serializable.

* If precedence graph is acyclic, the given schedule is conflict serializable

Example:

check whether the schedule is conflict serializable or not ?

S: $R_1(A)$; $R_2(A)$; $R_1(B)$; $R_2(B)$; $R_3(B)$; $W_1(A)$; $W_2(B)$

1 2 3 4 5 6 7

Step 1:

create a node for each transaction.

T_1 T_2 T_3

Step 2: find the conflict pairs.

T_1	T_2	T_3
1 $R_1(A)$	$R_2(A)$ 2	
3 $R_1(B)$	$R_2(B)$ 4	$R_3(B)$ 5
6 $W_1(A)$	$W_2(B)$ 7	

conflict pairs.

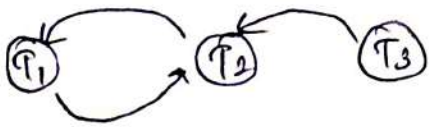
(5,7) $R_3(B) : W_2(B)$

(2,6) $R_2(A) : W_1(A)$

(3,7) $R_1(B) : W_2(B)$

Step 3

Draw a edge for each conflict pairs.



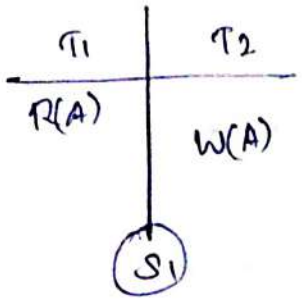
The precedence graph is cyclic, hence the given schedule is not conflict serializable.

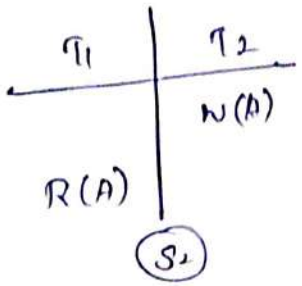
View equivalent schedule

consider two schedules S_1 & S_2, they are said to be view equivalent, if following conditions are true

Condition 1:

Initial read must be same





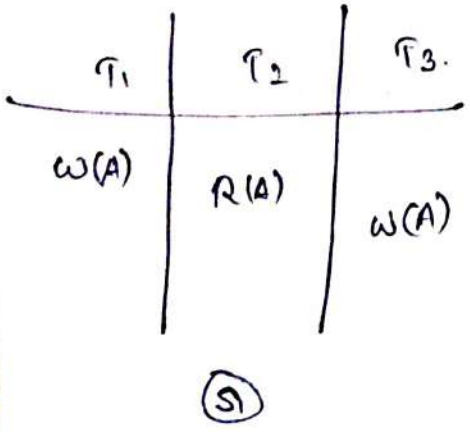
S_1 : T_1 reads A from database

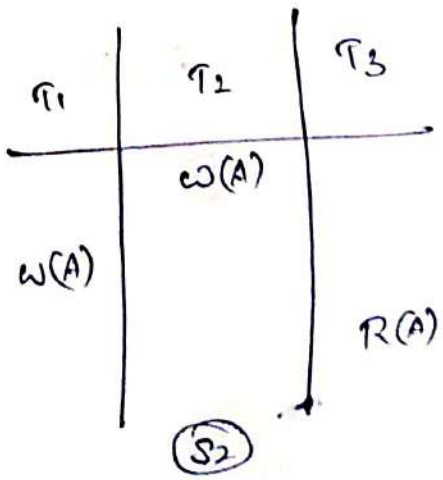
S_2 : T_1 reads A from T_2

$\therefore S_1 \neq S_2$.

Condition 2:

If there are two transactions T_i and T_j, the schedules S_1 and S_2 are view equivalent. If in schedule S_1, T_i reads A and then updated by T_j (i.e) (R W) sequence, then in schedule S_2, T_i must read A, which should be updated by T_j. (i.e) Read-write (RW) sequence must be same between S_1 & S_2.





S_1: Read-write sequence is $R_2(A)$ $W_3(A)$

S_2 : No read-write sequence.

$S_1 \neq S_2$

Condition 3:

Find write operations should be same in S_1 & S_2

T_1	T_2	T_3
$W_1(A)$	$R_2(A)$	$W_3(A)$

S_1

T_1	T_2	T_3
	$W_2(A)$	
		$W_3(A)$
	$R_2(A)$	
$W_1(A)$		

S_2

S_1 : final update of A is done in T_3

S_2 : final update of A is done in T_1

$\therefore S_1 \neq S_2$.

View serializable schedule

A schedule is view serializable, if it is view equivalent to any serial schedule.

Eg: check whether the schedule is view serializable or not!

S : $R_2(B)$; $R_1(A)$; $R_3(A)$; $W_1(B)$; $W_2(B)$; $W_3(B)$.

solution: with 3 transactions, total number of schedules possible are = 3! = 6.

< T_1 T_2 , T_3 >

< T_1 T_3 T_2 >

< T_2 T_1 T_3 >

< T_2 T_3 T_1 >

< T_3 T_1 T_2 >

< T_3 T_2 T_1 >

T_1	T_2	T_3
	$R_2(B)$ 1	
2 $R_1(A)$		$R_3(A)$ 3
4 $W_1(B)$	$W_2(B)$ 5	$W_3(B)$ 6.

Step 1: Final update on data item A (or) B.

In the given schedule, the final update is done on data B in T_3. Hence out of the 6 serial schedules $< T_2\ T_1\ T_3 >$ & $< T_1\ T_2\ T_3 >$ are the schedules which made final update by T_3.

Hence $< T_1\ T_2\ T_3 >$

$< T_2\ T_1\ T_3 >$

Step 2: Initial read on data item A or B:

Out of the above given schedule, the initial read on data B is done in T_2 $[R_2(B)]$. So, from the available serial schedule, initial read should be done by $< T_2\ T_1\ T_3 >$

Step 3: Read write sequence on data item A or B.

In the given schedule, the read write sequence is $R_2(B) : W_1(B)$ i.e. $T_2 \rightarrow T_2$.

Hence in the available only equivalent serial schedule also has the flow as $T_2 \rightarrow T_1 \rightarrow T_3$

Hence the given schedule is view serializable.

Concurrency Control

When multiple transactions are trying to access the same shared resource, then could arise many problems. If the access control is not done properly. There are some important mechanisms to which access control can be maintained. The concurrency control is implemented theoretically. using serializability practically, it can be implemented by 2 mechanisms namely.

1) Lock based protocols.
2) Time stamping protocols.

Lock based protocols → user, is responsible to write consistent concurrent transaction to implement concurrent control.

Time stamp protocols → System itself tries to detect possible inconsistency during concurrent execution and either the inconsistency is recovered (or) avoided.

Classification of concurrency control protocol.

Lock based protocols.

1) Binary Locks
2) Shared / Exclusive (or) Read / write locks
3) 2 phase locking protocol.

Time stamp protocol

1. Time stamp ordering protocol.

Locking.

1. A lock variable is associated with each data item which is used to identify the status of the data item. (whether the data is in use or not)

2. When a transaction, intends to access the data item, it must first examine its associated lock.

3. If no other transaction holds the lock, the scheduler lock the data item for T.

4. If another transaction T_i wants to access the same data item, then the transaction T_i, has to wait until the previous transaction release the lock.

Eg:

T_1

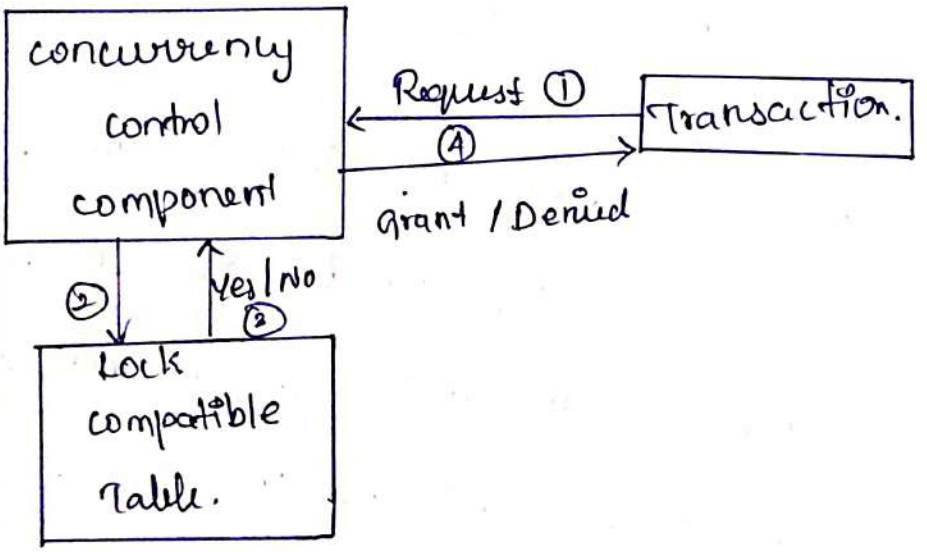
L(A) → locks the data item A

R(A) → Reads the data item A

W(A) → Writes the data item A

U(A) → unlocks the data item A.

L(B) → Locks the data item B.



All locking details.

Binary locks.

A binary lock can have 2 states (or) values.

* Locked (or 1) and
* Unlocked (or 0).

The current state (or value) of the lock associated with data item X as Lock (X).

Operations used with Binary locking.

1. **lock_item**: A transaction requests access to an item by first issuing a lock_item(x) operation.

   * If Lock(x) = 1 or L(x): the transaction is forced to wait.

   * If Lock(x) = 0 or U(x): it to set to 1 (the transaction locks the item) to access it.

2. **Unlock_item**: After the completion of access with the data item, the transaction issues an operation unlock(x), which sets the operation. Lock(x) = 0. (ie) unlock the data item x, so that it can be accessed by other transactions.

Rules.

1. A transaction T must issue the lock(x) operation before any read(x) or write(x) operations in T.
2. A transaction T must issue the unlock(x) operation after all read(x) and write(x) operations in T.
3. If a transaction T already holds the lock on item x, then T will not issue a lock(x) operation.
4. If a transaction T does not hold a lock on item x, then, T will not issue an unlock(x) operation.

Example:

T_1	T_2
l(A)	
w(A)	
U(A)	
	l(A)
	R(A)
	U(A)
	l(B)
	R(B)
	U(B)
l(B)	
R(B)	
w(B)	
U(B)	

Implementation of Binary locks

It is implemented using 3 fields plus a queue for transactions. They are.

1) Data - item - name
2) Lock
3) Locking - transaction.

Shared / Exclusive (or) Read / write lock

The binary lock is too restrictive for data items because at most one transaction can hold on a given item, whether the transaction is reading or writing. To improve [illegible] Shared / Exclusive lock,

in which more than one transaction can access the same data item for reading purpose.

(ii) the read operations on same data item by different transactions are not conflicting.

Two kinds of lock are supported:

* Exclusive (or) write locks
* Shared (or) Read Locks.

Shared locks

If a transaction T_i has locked the data item A in shared mode, then a request from another transaction T_j on A for:

* W (A) → denied, T_j has to wait until T_i unlocks A
* R (A) → Allowed.

Exclusive locks.

If a transaction T_i has locked the data item A in exclusive mode, then a request from another transaction T_j on A for:

* W (A) → denied
* R (A) → denied.

Compatible table for S/E locks.

Transaction T_i holds A on:

Txn T_j requests item A for data on i	S	X
R	Yes	No
W	No	No

S → shared mode
X → Exclusive mode
R → Read
W → write

Lock compatible table holds the status of a data item, whether it is locked or not.

Implementation of S/E locks.

It is implemented using 4 fields:

1) Data-item-name
2) Locks
3) No. of. records
4) Locking-transactions.

Data items that are not in the lock table are considered to be unlocked. The system maintains only those records for data items that currently locked.

Value of lock (A) read_locked or write_locked

* If Lock (A) = write_locked : The value of locking transaction is a single transaction that holds the exclusive lock on A.

* If Lock (A) = read_locked : The value of locking transaction is a list of one (or) more transactions that hold the shared lock on A.

Rules:

1. A txn T must issue the operation S(A) or read_lock (A) or X (A) or write_lock (A) before any read (A) operation is performed in T.

2. A txn T must issue the operation X(A) or write_lock (A), before any write (A) operation is performed in T.

3. After completion of all read (A) and write (A) operations in T, a txn T must issue an unlock(A) operation.

4. If a txn already holds a shared lock (or) exclusive lock on item (A), then txn T will not issue an unlock (A) operation.

5. A txn that already holds a lock on item A, is allowed to convert the lock from one locked state to another under certain conditions.

* Upgrading the lock by issuing a write_lock(A) operation (or) conversion of read_lock() to write_lock().
* Downgrading the lock by issuing a read_lock(A) (or) conversion of write_lock() to read_lock().

2 phase locking.

Binary locks or S/E locks does not guarantee serializability. To ensure serializability 2 phase locking (2PL) is used.

In this scheme, each txn makes locks & unlocks request in 2 phase.

1. Growing phase (Locking phase): In this phase, new locks on the derived data item can be acquired but none can be released.
2. Shrinking phase (unlocking phase): In this phase, existing locks can be released, but no new locks can be acquired.

T

L(A)
L(B)
L(C)
} → Growing (or) locking phase

⋮ ——————→ Lock point [Last lock position (or) first unlock position]

U(B)
U(A)
U(C)
} → shrinking (or) unlocking phase.

A 2PL always results serializable schedule but it does not permit all possible serializable schedules. (ie) some serializable schedules will be prohibited by the protocol.

A txn T does not allow to request any lock if T has already performed some unlock operation and every equivalent serial schedule is based on the order of the lock point.

T_1	T_2	T_3
		*
*		
	*	

order of lockpoint = $T_3 \rightarrow T_1 \rightarrow T_2$.

Points about 2 phase Locking.

1) Every non serializable schedule failed to execute 2pl.

2) 2pl always results in serializable schedule.

3) Equivalent serial schedule is based on the order of lock point.

4) If a schedule is allowed to execute using 2pl, then the schedule is conflict serializable, but not vice versa.

5) If a schedule is non conflict serializable, then the schedule is not allowed to execute using 2pl.

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-

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.

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

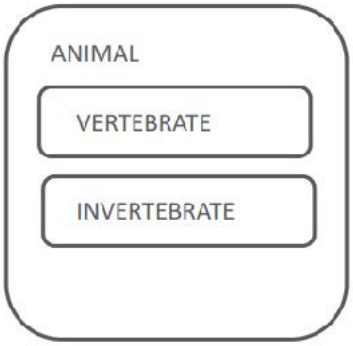
Inherits all relationships of the supertype

Usually has its own attributes or relationships

Is drawn within the supertype

Never exists alone

May have subtypes of its own



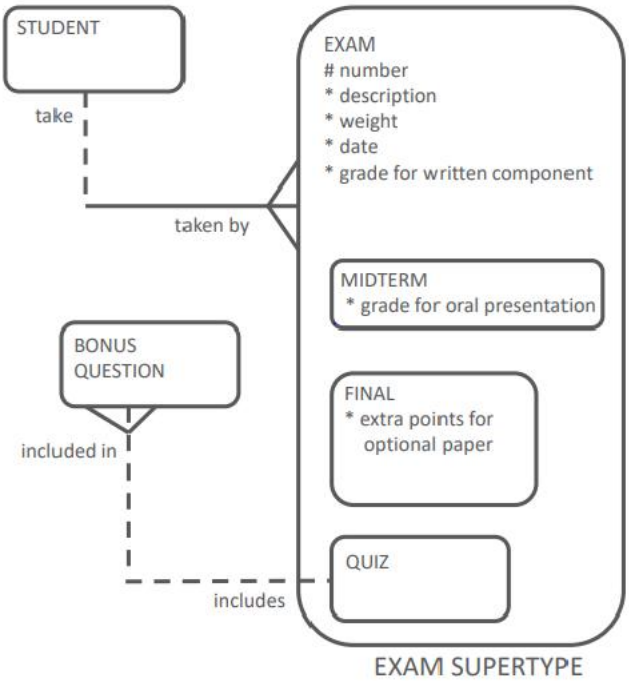
ANIMAL SUPERTYPE

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.



EXAM SUPERTYPE

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.

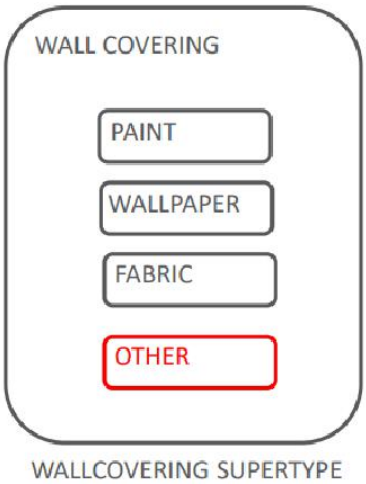
A single subtype is exactly the same as the supertype.

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.

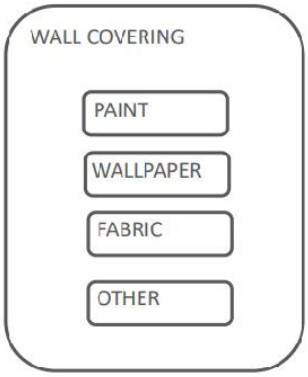


Subtypes Always Exist

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	Bounded	Integer	Always dense	Either in PL/SQL	Yes

(Varray)				block or at schema level	

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

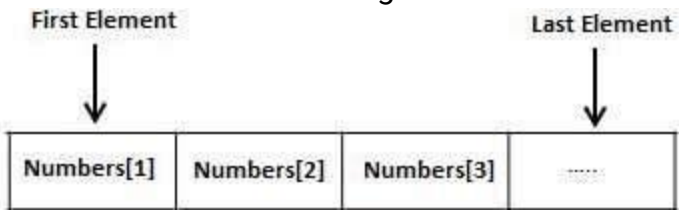
Customer(6): Komal

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.

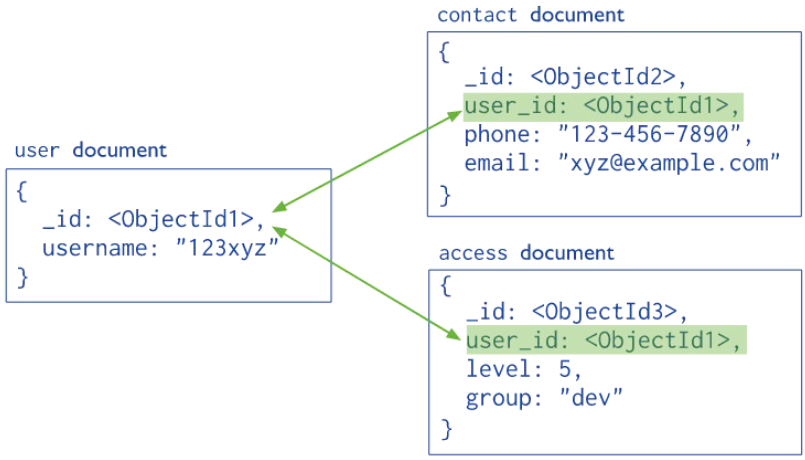
Embedded Data Model and Document Size Limit

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.

```
db.users.deleteMany(          <-- collection
   { status: "reject" }       <-- delete filter
)
```

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

Column Families

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**

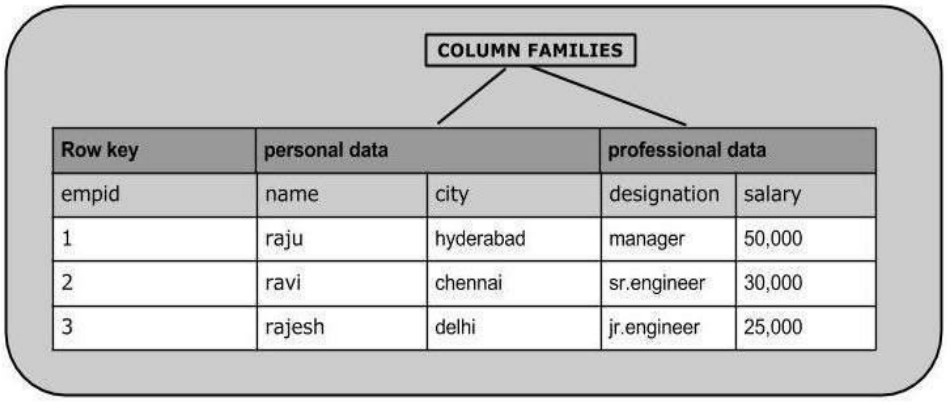
Inserting Data using HBase Shell- to create data in an HBase table. To create data in an HBase table, the following commands and methods are used:

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 asparameters . 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 rowids, 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*