Definition of Database

Data are facts that can be recorded and have implicit meaning. Data refers to values such as names, telephone, addresses that can be easily stored inside diary, PC or floppy. Data is actually stored in the database and information refers to the meaning of that data as understood by user.

The database is collection of related data. A database has the following implicit properties.

i. A database represents some aspect of the real world, sometimes called the miniworld or the Universe of Discourse (UoD). Changes to the miniworld are reflected in the database.

ii. A database is a logically coherent collection of data with some inherent meaning.

iii. A database is designed built and populated with data for a specific purpose. It has an intended group of users and some applications.

Database can be of any size. Example for Sources of databases are patients in hospital, bank, university, government department etc.

Definition of DBMS

DBMS means database Management System. It is a collection of programs that enables users to create and maintain database as well as enables to store, modify and extract information from the database. DBMS is software for defining, constructing and manipulating databases. It is also called database manager or database server. Example of DBMS are Ms. Access, oracle, MYSQL, Ms. SQL server etc.

Thus the goal of DBMS is to provide an environment that is both convenient and efficient to use in retrieving and storing database information. In DBMS, user issue request for information then DBMS analyzes and some internal processing takes place and then the result is sent back to the user.

Definition of Database System

Database system is computerized record keeping system. e.g. Computerized library system, flight reservation system, automated teller machine etc. Database and the DBMS software collectively known as database system.

The following operations take place in the database system.

i. Adding new / empty files to database.

ii. Inserting, retrieving, updating, deleting data from existing database.

iii. Removing existing files from database.

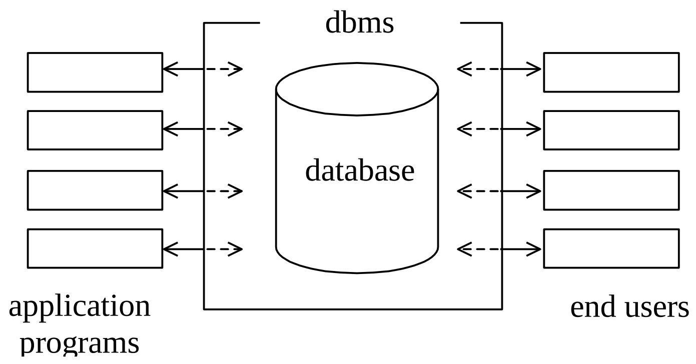


Fig. simplified picture of a database system

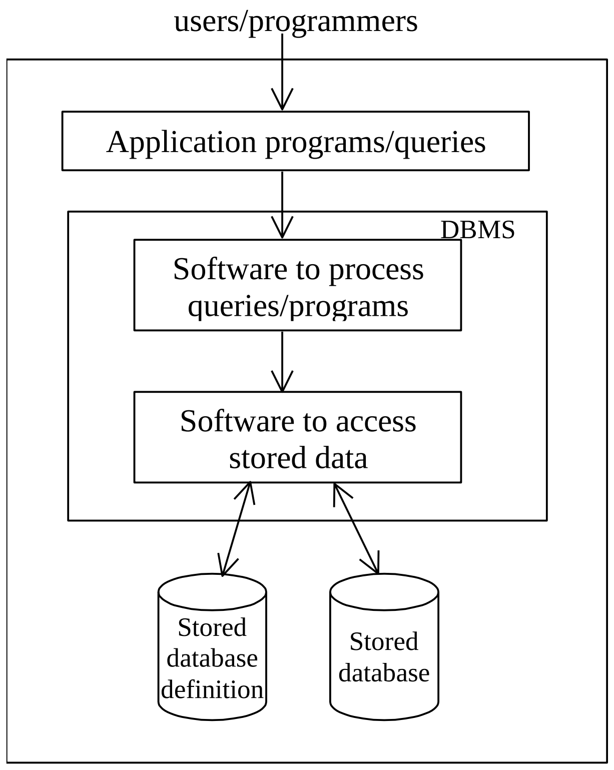

users/programmers
Application programs/queries
DBMS
Software to process queries/programs
Software to access stored data
Stored database definition
Stored database

Fig. A simplified database system environment

Advantages of database system over paper based methods of record keeping are (i) compactness (ii) speed and (iii) accuracy

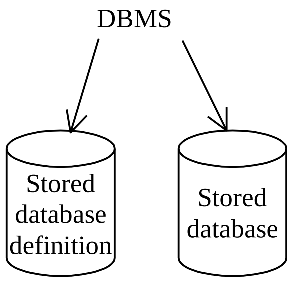
Characteristics of Database Approach

There are a no. of characteristics which distinguish the database from the traditional approach of programming with files. In the traditional approach of programming with files, many users may be using the same data such as student name separately. Thus data is duplicated and leads to wastage of storage space.

Main characteristics of database approach versus the file processing approach are as follows

i) Self describing nature of a database system

The definition or description of the database is stored in the system catalog separately and thus are available to users.



The system catalog stores structure and details of database only and no other data. thus the system catalog inside dbms describes database itself.

ii) Insulation between programs and data, and data abstraction

In traditional file processing, the changes the structure of data file may require changing all programs that access this file but the DBMS changes catalog information only. Thus both the program and data are independent and also called program data independence.

Data abstraction : DBMS provides user with a conceptual representation of data that does not include many of details of how the data is stored or how the operations are being implemented. Suppose the example of car. People don't think of a car as set of tens of thousands of individual parts. They think of it as a well defined object with its own behavior. Similarly data abstraction hides the complexity. Data model is a type of data abstraction.

iii) Support of multiple views of data

A database typically has many users, each of whom may require a different perspective or view of the database. A view may be portion or subset of the database. It is also called virtual table as it may contain virtual data. Users shouldn't be given the whole privilege for security purpose about some users may not be aware of whether the data they refer to is stored or derived. The DBMS supports multiple news view of data in a multi-user DBMS.

iv) Sharing of data and multi-user transaction processing

Many user can select, update data at the same time. So dbms must support concurrency control. for example in applications such as train/bus reservation system, flight reservation system, many users use the system from different locations at the same time and so is sharing of data and multi-user transaction processing.

Advantages and benefits using DBMS

Advantages of using DBMS are as follows

i) Controlling redundancy

In traditional file processing system, each user maintains their own file and so there may be duplication of data. Storing same data multiple times lead to several problems such as wastage of space, duplication effort for entering data, data may become inconsistent.

ii) Restricting unauthorized access (security)

Confidential data should not be available to all users. User accounts with certain restrictions to data may be created for security. Similarly multiple views can be created for database security. In traditional file processing, if own get file gets everything & all data.

iii) Providing persistent storage for program objects and data structure

The values of program variables are discarded once the program terminates as in C, C++ pascal program unless the programmer write them in files. A complex object in C++ can be stored permanently in an object oriented DBMS.

iv) Permitting inferencing and actions using rules

Database system may be deductive or active. Deductive databases have capabilities for defining deduction rules for inferencing new information from stored database. It works like reporting system.

v) Providing multiple user interfaces

DBMS provides variety of interfaces for varying users. DBMS provide query language for casual user, programming languages for application programmers, forms and command for parametric users, menu driven interfaces for stand alone users. Form styles and menu driven interfaces are collectively called GUI (Graphical user interface)

vi) Representing complex relationship among data

Relationships may be created among data using DBMS which helps in managing the data and defining constraints for updating and deleting.

vii) Enforcing integrity constraints

Something that limits data is called constraints in database. For example, the minimum balance should not fall in a bank. It is a constraint. Some of the constraints are primary key, NOT NULL, check.

viii) Providing backup and recovery

DBMS provides facilities for taking backup of the database which can be used for recovery in case of failure of computer system or hardware system.

ix) Easy in accessing data

It becomes very easy and fast while accessing data from database using DBMS. Reports can be used for easy access of data.

x) Concurrent access to database

Many users can share the data at the same time and thus dbms provides users to access the database concurrently.

Database system concepts and architecture

Data Model

Data model is a collection of tools for describing data, data relationship and consistency constraints. It is used to describe the structure of a database, basic operations for specifying retrievals and updates on the database.

Data model is a type of data abstraction. 3 levels of data abstraction are as follows.

i) Physical level : It is also called internal or low level data model. It describes about how data is actually stored in the database.

ii) Logical level : Next higher level is the logical level which describes about what data are stored and its relationship.

iii) View level : It is the highest level and describes about multiple views of data.

Many data models have been proposed.

Categories of data model:

1) Object based logical models

2) Record based logical models

3) Physical models

1. Object based logical models

It is used in describing data at the logical and view levels. There are many different models. Some of them are:

i) Entity-relationship model

ii) Object-oriented model

iii) Semantic data model

iv) Functional data model

i) Entity-relationship model:

The entity relationship (ER) data model is based on a perception of a real world that consists of a collection of basic objects, called entities, and of relationships among these objects.

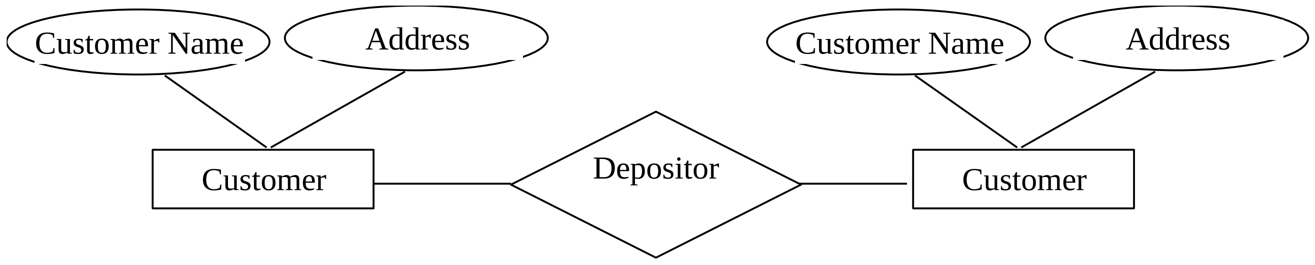


Fig. A sample E-R diagram

ii) Object oriented model

The object oriented model is based on a collection of object. An object contains values stored in instance variables within the object. An object also contains bodies of code that operate on the object. These bodies of code are called methods. Objects that contain the same types of values and the same methods are grouped together into classes.

iii) Semantic data models

It is similar to E-R modeling. It is also called object modeling. It also supports entity, which has properties and relationship.

iv) Functional data model

It is based on functions instead of relations. The functional approach shares certain ideas with object approach. It addresses object, which are functionally related to other.

2. Record based logical models

Record based logical models are used in describing data at the logical and view levels. It is used to specify the overall logical structure of the database.

Record based models are so named because the database is structured in fixed format records of several types. Each record type defines a fixed no. of fields, or attributes, and each field is usually of a fixed length. The three most widely accepted record based data models are the relational, network and hierarchical models.

i) Relational model

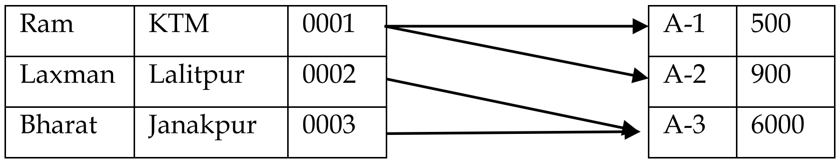
The relational model uses a collection of relations(tables) to represent both data and the relationships among those data. Each table has multiple columns, and each column has a unique name.

Customer Name	Address	Account No.	Account No.	Balance
Ram	KTM	A-1	A-1	500
Laxman	Lalitpur	A-2	A-2	700
Bharat	Jhapa	A-3	A-3	900

Fig. A sample relational database

ii)Network model

Data in the network model are represented by collections of records and relationships among data are represented by links, which can be viewed as pointers.



Ram	KTM	0001
Laxman	Lalitpur	0002
Bharat	Janakpur	0003

A-1	500
A-2	900
A-3	6000

Fig. A sample Network database

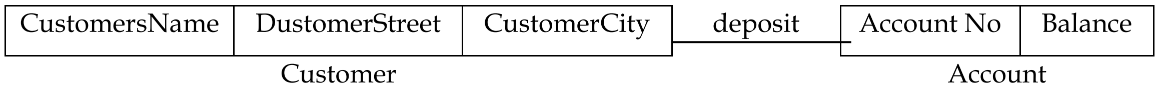


Fig. data structure diagram for network data model

iii) Hierarchical model

It is similar to the network model in the sense that data and relationships among data are represented by records and links, respectively. Records are organized as collections of trees rather than arbitrary graphs.

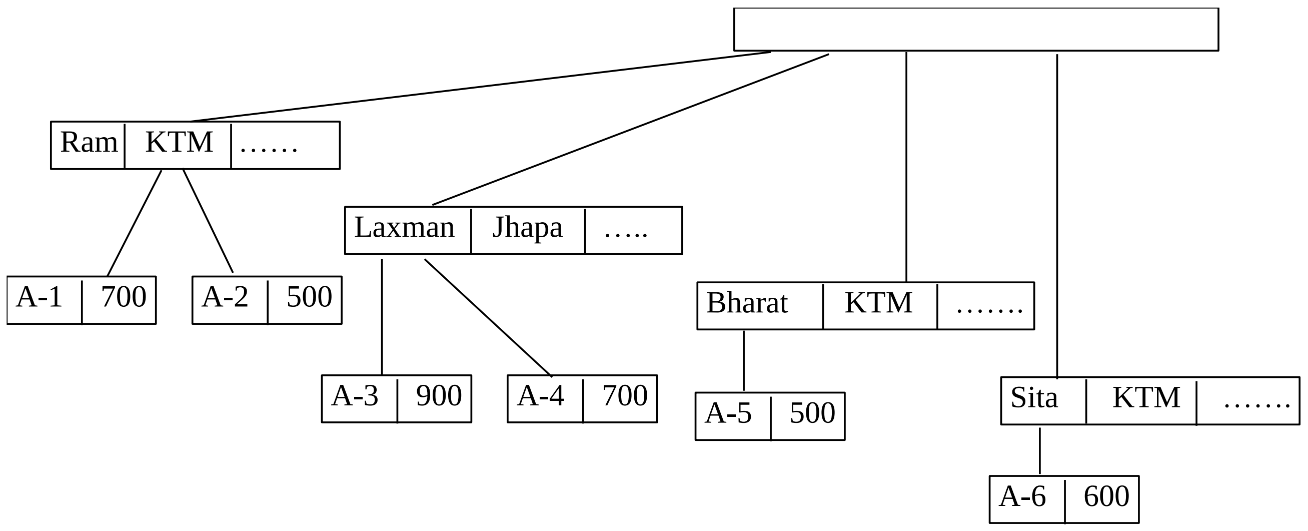


Fig. A sample hierarchical database

Custom erName	CustomerStreet	CustomerCity	Customer

Account No.	Balance		Account

Fig. Tree structure diagram for hierarchical model

Physical models

Physical data models are used to describe data at the lowest level. There are only few physical data modes in use. Two widely known ones are the unifying model and the frame memory model.

Schemas and instances

Database=description of database + database itself

The overall design of the database is called database schema. Database schema is specified during database design and not expected to change frequently.

Student			Course		
Name	Class	Major	Course Name	Duration	Remarks

Fig. Schema Diagram

Database changes over time as information is inserted and deleted. The collection of information stored in the database at a particular moment is called instances in the database. It is also called database state or snapshot or current set of occurrences. When database is designed, the database is in empty state with no data. It is in initial state when database is loaded with data. Thus at any point, database has a current state. When any field is added to database, it is called schema evolution.

DBMS Architecture

The main characteristics of database approach are (i) insulation of programs and data (ii) support of multiple user views (iii) use of a catalog to store the database description. The architecture of the DBMS is proposed to visualise these characteristics and so called the three schema architecture. It is also called ANSI/SPARC (American National standard Institute/Standards planning and requirements committee) Architecture.

Goal of the architecture is to separate the user applications and physical database. In this architecture, schemas can be defined as the following three levels.

i) The internal level

It has an internal schema which describes the physical structure of the database. It describes the complete details of data storage and access paths for the database.

ii) The conceptual level

It has a conceptual schema, which describes the structure of the whole database for a community of users. It hides the details of physical storage structures and concentrates on describing entities, data types, relationships, user operations and constrains.

iii) The external or view level

It includes a number of external schemas or user views. Each external schema describes the past of the database that a particular user group is interested in and hides the rest of the database from that user group.

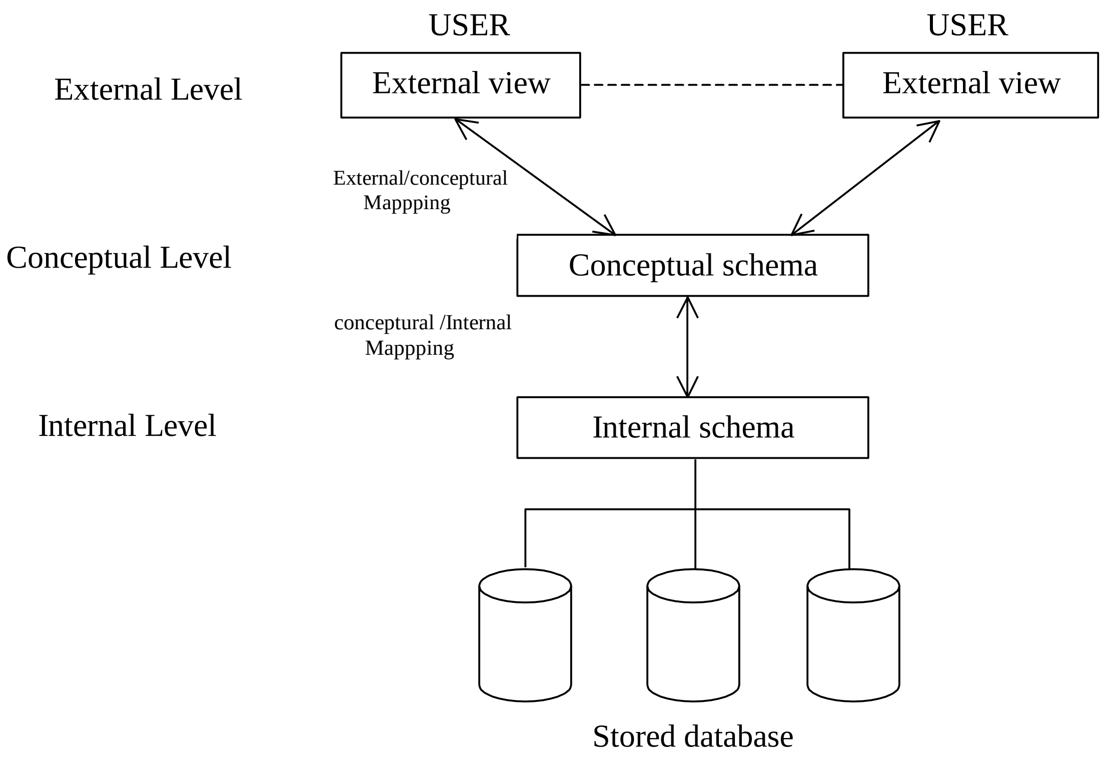


Fig. Three schema Architecture

Three schema architecture is a tool for the user to visualize the schema levels in a database system. Most DBMS don't separate the three levels data actually exists at the physical level. User/groups refer only to its own external schema. So DBMS must transform a request from users into a request against conceptual schema and then into a request on the internal schema for processing over the stored database. If the request is a database retrieval, the data extracted from the stored database must be reformatted to match the user's external view. The process of transforming requests and results between levels are called mappings.

DATA INDEPENDENCE

Data independence is defined as the capacity of DBMS to change the schema at one level of a database system without having to change the schema at the next higher level. We can define two types of data independence.

i) Logical data independence

Logical data independence is the capacity to change the conceptual schema without having to change external schemas or application programs. We may change the conceptual schema to expand the database (by addressing a record type or data item) or to reduce the database (by removing a record type or data item). It results in change in E-R diagram but the application program or external schema is not changed.

ii) Physical data independence

Physical data independence is the capacity to change the internal schema without having to change the conceptual (or external) schemas. Changes to the internal schema may be needed because some physical files had to be reorganized. For example, by creating additional access structures to improve the performance of retrieval or update. If the same data as before remains in the database, we should not have to change the conceptual schema.

In multiple level DBMS, its catalog must be expanded to include information on how to map requests and data among the various levels. In data independence, when the schema is changed at some level, the schema at the next higher level remains unchanged only mappings change.

DATABASE LANGUAGES

A database system provides mainly two different types of languages: one to specify the database schema, called data definition language and the other to express database queries and updates called data manipulation language.

i) Data definition language (DDL):

A database schema is specified by a set of definitions expressed by a special language called a data-definition language. The result of compilation of DDL statements is a set of tables that is stored in a special file called data dictionary or data directory. A data dictionary is a file that contains metadata that is data about data. DBMS will have a DDL complier, which process DDL statements. DDL is used to specify conceptual schema only. Similarly, SDL (storage definition language) is used to specify the internal schema & VDL (view definition language) is used to specify user views and their mappings to the conceptual schema.

iii) Data manipulation language (DML)

A data manipulation language is a language that enables users to access or manipulate data as organized by the appropriate data model.

Data manipulation consists of

- The retrieval of information stored in the database
- The insertion of new information into the database
- The deletion of information from the database

- The modification/update of information stored in the database

DML is of the following 2 types

a) Non-procedural DMLs

The language requires a user to specify what data are needed without specifying how to get those data. It is easier to learn and use. many DBMS allow it either to be entered interactively from a terminal or to be embedded in a general purpose programming language. For example SQL (structured query language). SQL can retrieve many records in a single DML statements and hence it is also called set at a time or set oriented language. It is also called high level language.

b) Procedural (Low Level) DMLs

The language requires a user to specify what data are needed and now to get those data. It is embedded in a general purpose programming language. This type of DML retrieves records one by one and processes each record separately using programming language construct such as looping and hence it is also called record at a time DML.

When DML are embedded in a general purpose-programming language, then that language is called host language and the DML is called the data sub language.

DBMS Interfaces

DBMS provides the following user-friendly interfaces.

i) Menu based interfaces for browsing

These interfaces present the user with lists of options, called menus that lead the user through the formulation of a request. The query is composed step by step by picking optional from a menu that is displayed by the system. Pull down menus is becoming popular technique in window based user interfaces.

ii) Forms based interfaces

A forms-based interface displays a form to each other. Users can fill out all of the form entries to insert new data, or they fill out only certain entries. Forms are usually designed and programmed for naïve users as interfaces to canned transactions.

iii) Graphical user interface

A graphical interface (GUI) typically displays a schema to the user in diagrammatic form. The user can then specify a query by manipulating the diagram. GUIs utilize both menus and forms.

iv) Natural language interfaces

These interfaces accept requests written in English or some other language and attempt to understand them. The natural language interface usually has its own schema, which is similar to database conceptual schema. The natural language interface refers to words in its schema to interpret the request. If the interpretation is successful, the interface generates a high level query corresponding to the natural language request and submits it to the DBMS for processing.

v) Interfaces for parametric users

Parametric users such as bank tellers, often have a small set of operations that they must perform repeatedly. System analysts and programmers design and implement a special interface for a known class of naïve users. For example, function keys in a terminal can be programmed to initiate the various commands. This allows the parametric user to proceed with a minimal number of keystrokes.

vi) Interfaces for DBA

Most database system contains privileged commands that can be used only by the DBA's staff. These include commands for creating accounts, setting system parameters, granting account authorization, changing a schema, and reorganizing the storage structure of database.

Database System Environment

A DBMS is a complex software system. The database system environment consists of DBMS component modules, Database system utilities and tools, application environments and communication facilities. They are explained as follows.

i. DBMS component modules

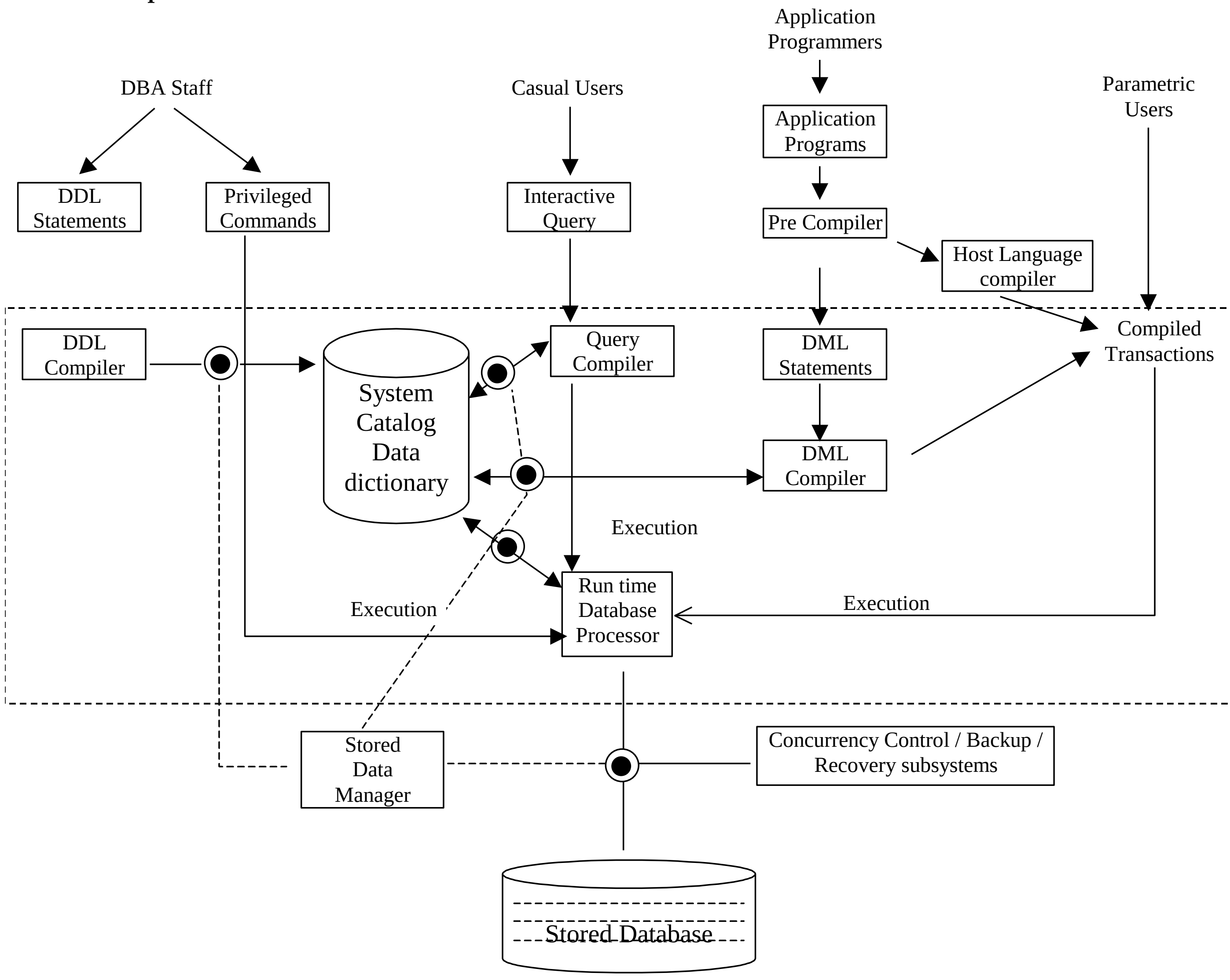


Fig. typical component modules of a DBMS

The database and the DBMS catalog are stored on the disk. There are different modules in the Database system. The stored data manager module of the DBMS controls access to the information of database and catalog. The dotted lines and the circles marked A, B, C, D and E as shown in fig illustrate access that are under the control of this stored data manager.

Similarly DDL compiler processes schema definitions, specified in the DDL and stores description of the schemas in the DBMS catalog. The catalog includes information such as the names of files, data items, storage details of each file, mapping information among schemas and constraints. Run time database processor handles database accesses at run time. It receives retrieval or update operations and carries them out on the database.

The Query compiler handles high level query that are entered interactively. It parses, analyzes and compiles a query by creating database access code and then generates calls to the run time processor for executing the code.

The precompiler extracts DML statements from an applications. Program written in the host programming language. These commands are sent to the DML compiler for compilation into object code for database access. Rest of the program is sent to the host language compiler. Object codes for DML commands and the rest of the Program and linked forming a transaction whose code includes calls to the run time database Processor. If many users share the computer system, the operating system will schedule DBMS disk access requests and DBMS processing along with other processes.

ii. Database system Utilities

Most DBMS have database utilities that help the DBA in managing the database system. common Utilities have the following types of functions.

a. Loading: A loading utility is used to load existing data files such as text files or sequential files into the database. Some vendors are offering products that generate the appropriate loading programs, giving the existing source and target database storage descriptions. Such tools are also called conversion tools.

b. Backup: A backup utility creates a backup copy of the database, usually by dumping the entire database onto tape. The back up can be used to restore the database in the case of failure of computer system or database system. Incremental or differential backup method may be used.

c. File reorganization: This utility can be used to reorganize, reindex a database file to improve the performance.

d. Performance monitoring: This utility monitors database usage and provides statistics to DBA which helps in making decisions to improve performance.

iii. Tools, application environments and Communication facilities:

Tools such as CASE, data dictionary, application developments are often available to database system. The CASE (Computer Aided Software Engineering) tools are used in the design Phase of database system. Data dictionary is used for storing catalog information about schemas and constraints, usage standards, application program description and user information. Such system is also called information repository. Application develop environment provide an environment for developing database design, application, querying and updating. Many commercial database systems have communication package whose function is to allow users at location remote from database site to access the database through communication hardware.

Classification of Database Management System

DBMS is classified on the basis of data model, number of users, number of sites, cost and types of access path.

On the basis of data model, DBMS is classified into

i. Relational data model

ii. Object data model

iii. Hierarchical data model

iv. Network data model

On the basis of numbers of users, DBMS is classified into

i. Single user system – supports only one user at a time

ii. Multi-user system supports multiple users concurrently

On the basis of numbers of sites, DBMS is classified into

i. Centralized – if the data is stored at a single computer site.

ii. Distributed – database and DBMS software distributed over many sites, connected by a computer network.

iii. Homogeneous – Use same DBMS software at multiple sites.

iv. Heterogenous – Participating DBMS are loosely coupled and have a degree of local autonomy. Many DBMS use a client server architecture.

On the basis of cost, DBMS is classified into

i. DBMS packages between $10,000 and $100,000

ii. DBMS packages costing more than $100,000

On the basis of types of access of Path, DBMS is classified into

i. General purpose – Designed for general purpose

ii. Special purpose – Designed and built for specific application such as airlines reservation, telephone directly system such DBMS can't be used for other applications without major change.

Data Dictionary

The data dictionary can be regarded as a system database which contains data about data. This is also called metadata. It contains definitions of other objects in the system instead of raw data. It also stores schemas and mappings details, various security and integrity constraints. It is also called data directory or system catalog or simply catalog or data repository

TABLE		
TABNAME	**COLCOUNT**	**ROWCOUNT**
DEPT	2	2
Emp	3	3

COLUMN	
TABNAME	**COLNAME**
DEPT	Dept No.
DEPT	Dept Name
EMP	Emp Name
EMP	Emp No.

		EMP	Emp Telephone No.

Fig. Catalog for department and Employee database

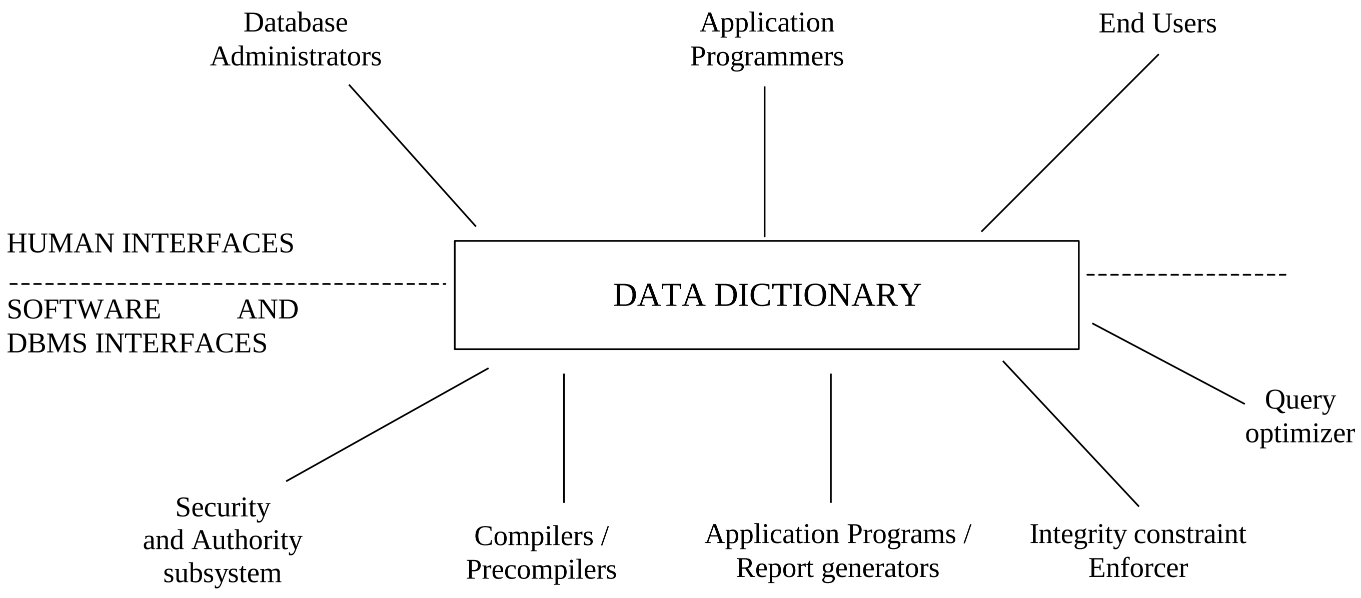

Database Administrators
Application Programmers
End Users
HUMAN INTERFACES
SOFTWARE AND DBMS INTERFACES
DATA DICTIONARY
Query optimizer
Security and Authority subsystem
Compilers / Precompilers
Application Programs / Report generators
Integrity constraint Enforcer

Fig. Human & Software interfaces to a data dictionary

Data dictionary is accessed by various software modules of DBMS itself such as DDL/DML compilers, query optimizer, constraint enforcer. If the data dictionary is used by designers, users and administrators, not by DBMS software it is called a passive data dictionary, otherwise it is called an active data dictionary.

E-R MODEL

E-R model means entity relationship which is a popular high level conceptual data model. ER model describes data as entities, relationship and attributes. ER model is based on a perception of a real world that consists of a set of basic objects called entities, and of relationship among these objects.

Entity types and entity sets: An entity is a thing or object in the real world that is distinguishable from all other objects. For example, each person in an enterprise is an entity. An entity has a set of properties, which may uniquely identify an entity. An entity may be concrete, such as a person or a book, or it may be abstract such as loan or a holiday or a concept.

An entity set is a set of entities of the same type that share the same properties or attributes. The set of all persons who are customers at a given bank for example can be defined as the entity set customer. Similarly, the entity set loan might represent set of all loans awarded by a particular bank.

Attributes: Attributes are descriptive properties possessed by each member of an entity set. Each entity has attributes. For example an employee entity may be described by the employee's name, age, address, salary and job. Possible attributes of the loan entity set are loan number and amount. For each attribute, there is a set of permitted values, called the domain or value set.

Employee Name	**Address**	**Age**	**Salary**	**Job**
Ram	KTM	15	5000	Manager

Loan No.	**Loan Amount**
L – 1	5000

Shyam	KTM	20	3000	Operator
Sita	BRT	17	4000	CEO

Customer

L - 2	3000
L – 3	10,000

Loan

Fig.: Entity sets customer and loan

An attribute, as used in the E-R model, can be characterized by the following attribute types.

i) Simple (Atomic) and composite attributes: Attributes that are not divisible are called simple or atomic attributes. Such as age as shown in fig. is simple attribute. Composite attributes can be divided into smaller attribute. Composite attributes can be divided into smaller subparts, which represent more basic attributes with independent meaning. For example: Employee name could be structured as a composite attribute consisting of first name, middle name and last name.

ii) Single valued and multivalued attributes: Single valued have a single value for a particular entity. For example, the loan number attribute for a specific loan entity refers to only one loan number and so it is single valued. Consider the employee entity set with the attribute dependent name. Any particular employee may have zero, one or more dependents. So, different employee entities within the entity set will have different numbers of values for the dependent name attribute and this type of attribute is said to be multivalued.

iii) Null (missing): A null value is used when an entity does not have a value for an attribute. It is unknown value of not applicable. If a particular employee has ho dependents. The dependent name value for that employee will be null.

iv) Derived attribute: The value for this type of attribute can be derived from the values of other related attributes. For instance, let us say that the customer entity has an attribute loans-held, which represents how many loans a for this attribute by counting the number of loan entities associated with that customer.

v) Complex Attributes: The composite and multivalued attributes be nested in an arbitrary way. We can represent arbitrary resting by grouping components of a composite attribute between Parentheses ( ) and separating the components with commas, and by displaying multivalued attributes between braces { { . Such attributes are called complex attributes. For example, if a person can have more than one residence and each residence can have multiple phones. An attributes Address Phone for a person entity type can be specified as bellows:

{ Addres Phone ({ Phone (AreaCode, PhoneNumber)},

Address (Street Address (Number, street, ApartmentNumber), city, state, up))}

Key attributes of an entity type:

An entity type defines a collection of entities that have the same attributes. The collection of all entities of a particular entity type in the database at any point in time is called an entity set.

Entity Type	Employee	Company
Name	EmpID, Name, Age, Salary	Name, Headquarters, President
Entity Set:	e_1 : (1, Ram, 10, 2000)	C_1 (wlink, Jawalakhel, Dr. Ashish)

	e_2 (2, Shyam, 20, 5000) e_2 (3, Mohan, 25, 1000)	C_2 (Nepasoft, Ratnapark, S.P. Joshi) C_3 (NEA, KTM, Dr. S.R. Malla)

Fig.: Two entity types named employee and company and some of the member entities in the entity set.

It is important to be able to specify how entities within a given entity set and relationships within a given relationship set are distinguished. An entity type usually has an attribute whose values are distinct for each individual entity in the collection. Such an attribute is called a key attribute and its values can be used to identify each entity uniquely. For example, the EmpId attribute is a key of the employee entity type. Some keys are superkey, Candidate key and Primary Key.

Superkey: A super key is a set of one or more attributes that, taken collectively, allows us to identify uniquely an entity in the entity set. For example, EmpId is a super key for the entity set employee.

For example: suppose the attributes of the customer entity set are customer Name, Social security, customer street, customerCity. Then social security is a superkey.

Candidate Key: There may be superkeys for which no proper subset is a superkey. Such minimal superkeys are called candidate keys. {Social – security} and {customerName, CustomerStreet} are candidate keys. Although the attributes social security and customer Name together can distinguish customer entities, their combination does not form a candidate key, since the attribute social security alone is a candidate key.

Primary key: Primary key is a candidate key that is chosen by the database designer as the principal means of identifying entities within an entity set. A key (Primary, candidate and super is a property of the entity set, rather than of the individual entites.

Relationships & Relationship Types

A relationship is an association between entities. Each relationship is identified so that its name is descriptive of the relationship. Verbs such as takes, teaches, and employs make good relationship names. For example, a student takes a class, a professor teaches a class, a department employs a professor and so on.

Rectangles represent entity sets, ellipses represent attributes. Similarly relationships are represented by diamond shaped symbols as shown below in fig. and the lines link attributes to entity sets and entity sets to relationship sets.

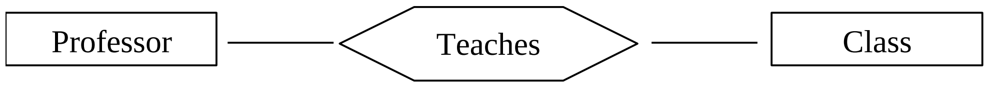


Fig.: An entity relationship

The figure shows a relationship between two entities (also known as participants in the relationship) named professor and class respectively.

A relationships degree indicates the number of associated entities or participants. A unary relationship exists when an association is maintained within a single entity. A binary relationship exists when two entities are associated. A ternary relationship exists when three entities are associated. Although higher degrees exist, they are rare and are not specifically named.

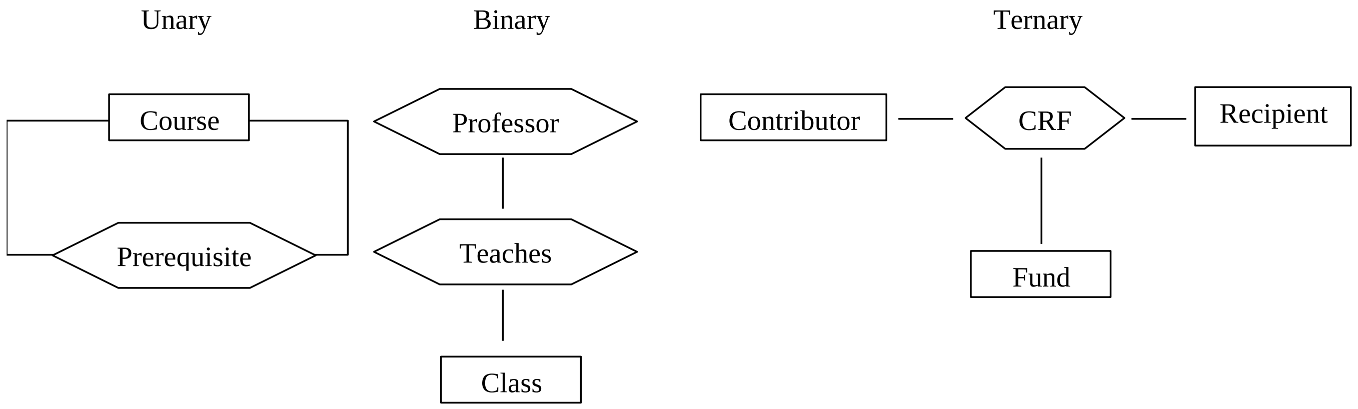


Fig.: Three types of relationships

A course within the course entity is a prerequisite for another course within that entity. The existence of a course prerequisite means that a course requires a course i.e. a course has a relationship with itself. Such a relationship is also called a recursive relationship.

Connectivity: The relationships are all classified as M : N for example. A fund can have many donors. A fund may support many researchers who become the fund receiptants and a researcher may draw support from many funds. Contributors can make donations to many funds.

The term connectivity is used to describe the relationship classification.

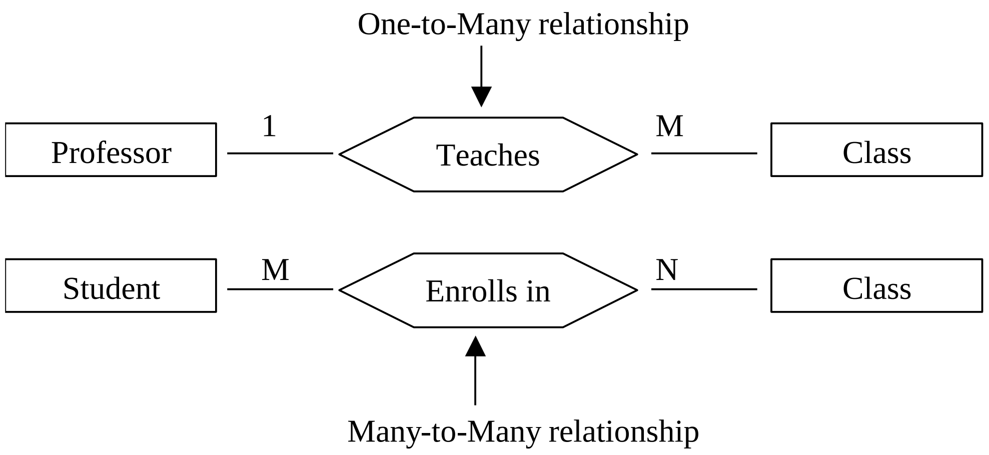


Fig.: Connectivity in an E-R diagram

Cardinality: Cardinality expresses the specific number of entity occurrences associated with one occurrence of the related entity. The actual number of associated entities usually is a function of an organizations policy. For example: For Purbanchal University limits the professor to teaching a maximum of three classes per week. Therefore, the cardinality rule governing the professor – class association is expressed as "one professor teaches upto three classes per week. The cardinality is indicated by placing the appropriate numbers beside the entities as shown in fig.

One to many relationship

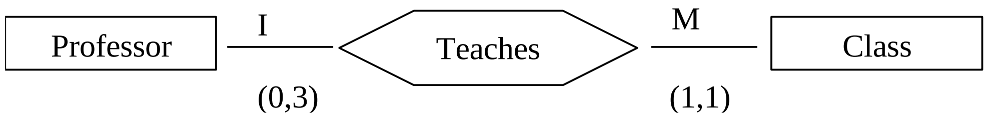


Fig.: Cardinality in an E-R diagram

- The relationship between Professor and class is 1:M
- The cardinality limits are (0,3) for professor indicating that a professor may teach a minimum of zero and a maximum of three class
- The cardinality limits for class entity are (1,1) indicating that the minimum no. of professor required to teach a class is one, as is the maximum number of Professors.

For binary relationship between entity sets A and B, the mapping cardinality must be one of the following.

i) One to one: An entity in A is associated with at most one entity in B, and an entity in B is associated with at most one entity A.

ii) One to many: An entity in A is associated with any number of entities in B and an entity in B however, can be associated with almost one entity in A.

iii) Many to one: An entity in A is associated with at most one entity in B and an entity in B, however can be associated with any number of entities in A.

iv) Many to many: An entity in A is associated with any number of entities in B and an entity in B is associated with any number of entities in A

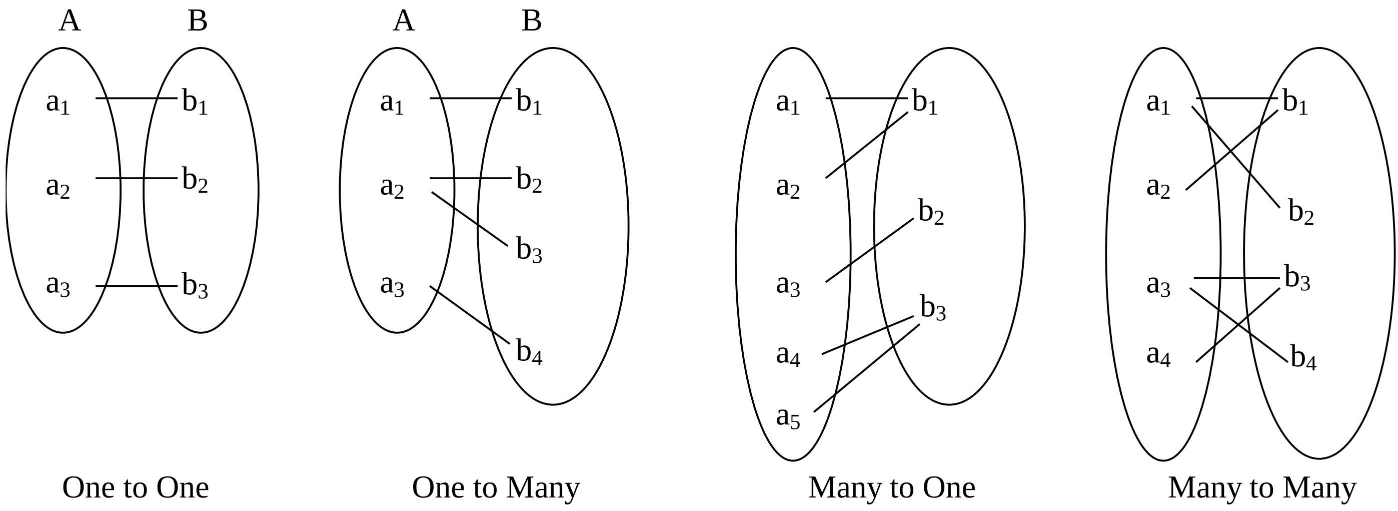


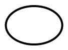
Fig.: Mapping Cardinalities

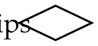
A relationship is an association among several entities.

Entity – Relationship Diagram (E-R diagram)

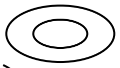
The E-R diagram is used to represent to E-R model. The E-R diagram consists of the following major components.

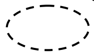
i) Rectangles – to represent entity sets

ii) Ellipses – to represent attributes 

iii) Diamonds – to represent Relationships 

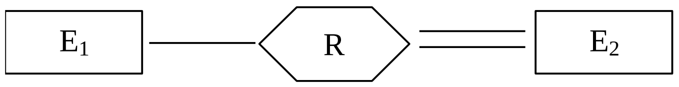
iv) Lines – to link attributes to entity sets and entity sets to relationship sets -

v) Double ellipses – to represent multivalued attributes 

vi) Dashed ellipses – to denote derived attributes – 

vii) Double lines – to indicate total participation of an entity in a relationship set.

Total participation of $\in_2$ in R



For example: Suppose the attributes associated with customer are customerName, SocialSecurity, CustomerStreet and CustomerCity. The attributes associated with loan are loanNumber and amount. The relationship set borrower may be many to many, one to many, many to one and one to one. To distinguish among these types, we know either a directed line (→) or an undirected line (-) between the relationship set and the entity set.

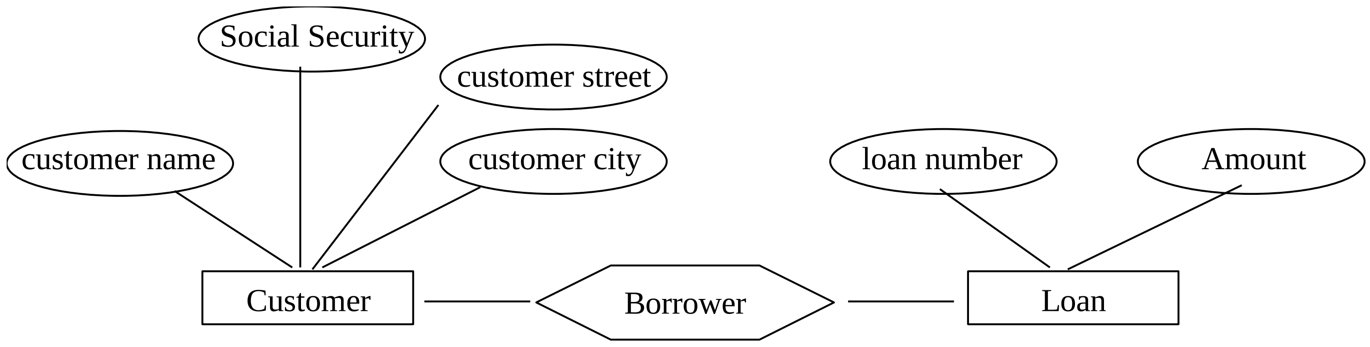


Fig. E-R diagram corresponding to customers and loans

Similarly, we can see another example as follows:

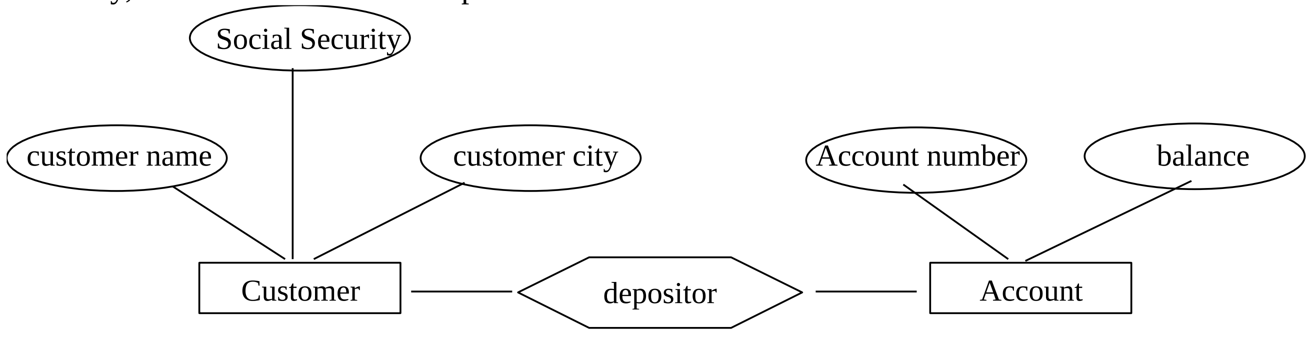


Fig. E-R diagram showing one to many relationship

A directed line from the relationship set depositor to the entity set account specifies that a customer can deposit to many accounts. So is a one to many relationships.

Weak Entity Types

An entity set may not have sufficient attributes to form a primary key. Such an entity set is termed as weak entity set. An entity set that has a primary key is termed as strong entity set. For example: consider the entity set payment, which has three attributes: PaymentNumber, PaymentDate and paymentAmount. Although each payment entity is distinct, payment for different loans may share the same paymentNumber. Thus, this entity set does not have a primary key. Hence it is a weak entity set.

The primary key of weak entity set is formed by the primary key of the strong entity set on which the weak entity set is existence dependent, plus the weak entity set's discriminator.

In this case, the existence of entity payment depends on the existence of entity loan. If loan is deleted, its associated payment entities must be deleted. So, entity set loan is dominant and payment is subordinate. Discriminator of weak entity set is a set of attributes that can uniquely identify weak entities that are related to the same owner entity. For example: The discriminator of the weak entity set payment is the attribute payment Number. Since, for each loan, a paymentNumber uniquely identities one single payment for that loan. Hence, in the case of the entity set payment, its primary key is {loanNumber, PaymentNumber}

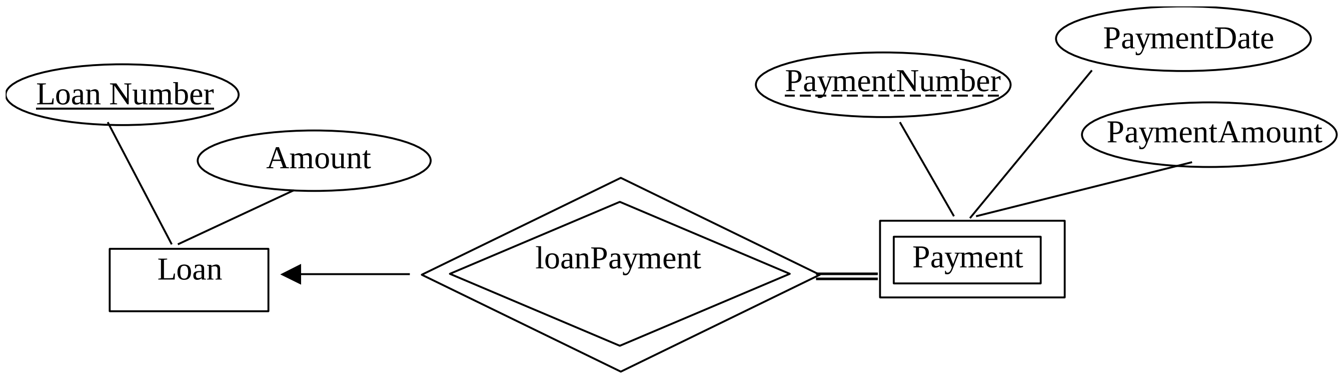


Fig. E-R diagram with a weak entity set.

Roles On Relationships

Each entity type that participates in a relationship type plays a particular role in the relationship. The role name signifies the role that a participating entity from the entity type plays in each relationship instance and helps to explain what the relationship means. For example, in the works_for relationship type, EMPLOYEE plays the role of employee or worker and DEPARTMENT plays the role of department or employer.

Role names are not technically necessary in relationship types where all the participating entity types are distinct, since each entity type name can be used as the role name. However, in some cases the same entity type participates more than once in a relationship type in different roles. In such cases, the role names become essential for distinguishing the meaning of each participations. Such relationships are called recursive relationships.

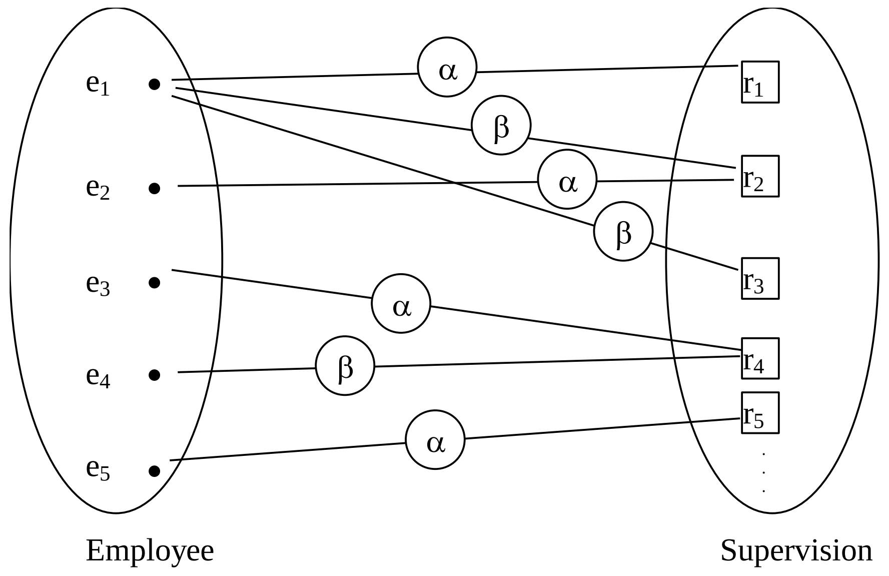


Fig.: Recursive relationship where employee entity type plays two roles

α - supervisee

β - Supervisor

The supervision relationship type relates an employee to a supervisor, where both employee and supervisor entities are members of the same EMPLOYEE entity type.

Structural Constraints On Relationships Types

Relationship types usually have certain constraints that limit the possible combinations of entities that may participate in the corresponding relationship set. These constraints are determined from the miniworld situation that the relationship represent.

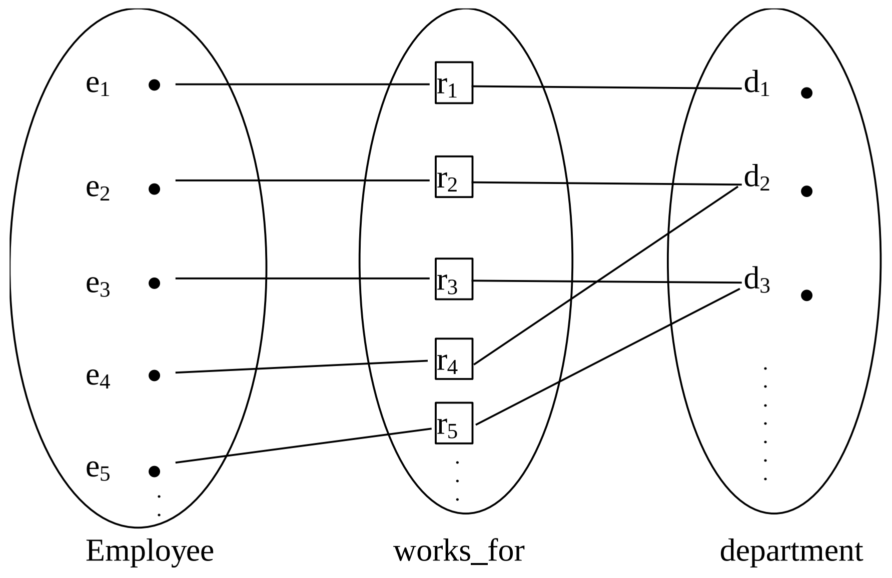


Fig.: Some instances of the works_for relationship between employee and department.

Suppose the company has rule that each employee must work for exactly one department.

There are two types of relationship constraints: Cardinality ratio and participation.

i. Cardinality ratios for binary relationships: The cardinality ratio for a binary relationship specifies the number of relationship instances that an entity can participate in for example: in the works_for binary relationship type, department : employee is of cardinality ratio I:N, meaning that each department can be related to numerous employee but an employee can be related to only one department. The possible cardinality ratios for binary relationships types are 1:1, 1:N, N:1 and M:N.

ii. Participation constraints: The participation constraints specifies whether the existence of an entity depends on its being related to another entity via the relationship types. There are two types of participation constraints total and partial. If a Company policy states that every employee must work for a department, then an employee can exist only if it participants in a works_for relationship instance. Thus, the participation of employee in works_for is called total participation meaning that every entity in the total set of employee entity must be related to a department entity via works for. Total participation is also called existence dependency.

Cardinality ratio and participation constraints, taken together is called the structural constraints.

NAMING CONVENTIONS

The choice of names for entity types, attributes, relationship types and roles is not always straight forward. One should choose names that convey, as much as possible, the meanings attached to different constructs in the schema. We choose to use singular names for entity types, rather than plural ones, because the entity type name applies to each individual entity belonging to that entity type.

In E-R diagrams, we will use the convention that entity type and relationship type names are in uppercase, letters, attribute names are capitalized and roles names are in lowercase letters. Generally the nouns appearing in the narrative tend to give rise to entity type names and the verbs tend to indicate names of relationship types.

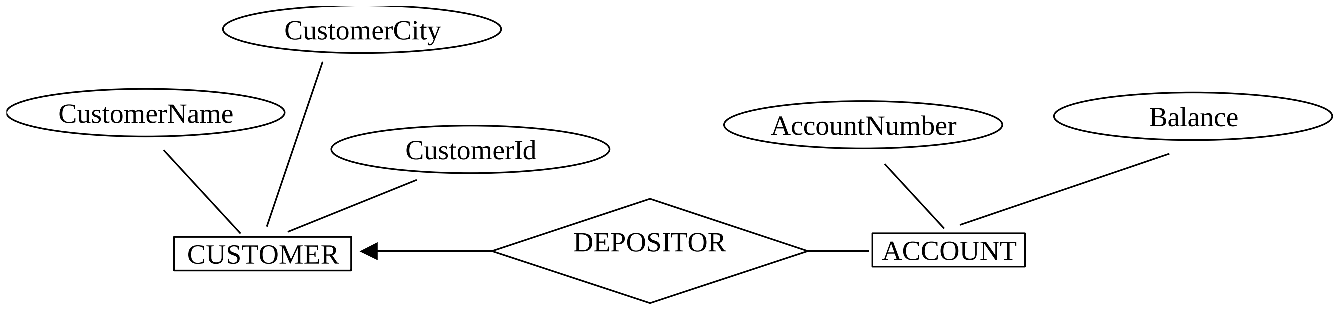


Fig. E-R diagram using Naming conventions

Another naming convention involves choosing relationship names to make the ER diagram of the schema readable from left to right and from top to bottom.

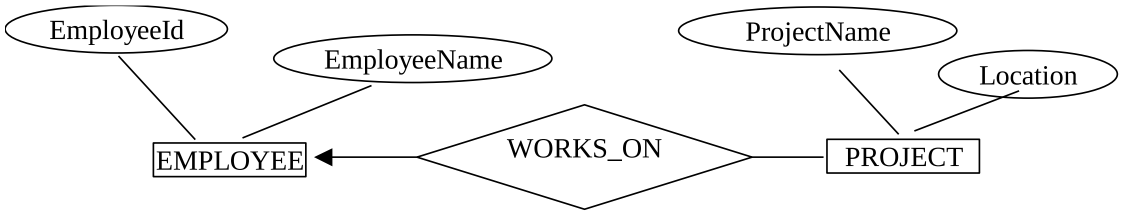


Fig. E-R diagram using convention for relationship Name from Left to right.

Relational Model

Introduction To Relational Databases:

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. The relational model has established itself as the primary data model for commercial data processing applications.

Structure of Relational databases: A relational databases consists of a collection of tables, each of which is assigned a unique name. A row in a table represents a relationship among a set of values. Since a table is a collection of relationships, there is close correspondence between the concept of table and the mathematical concept of relation, from which the relational data model takes its name.

Basic structure: Consider the account table with data as follows.

Branch Name	Account Number	Balance
KTM	A – 1	50,000
Lalitpur	A – 2	75,000
Bhaktapur	A – 3	45,000

Fig. The account relation

For each attribute, there is a set of permitted values, called the domain of that attribute. For the attribute branchName, for example, the domain is the set of all branch Names. Let D_1 denote this set, D_2 denote the set of all account numbers and D_3 denote the set of all balance. Any row of account must consist of a 3-tuple (V_1,V_2,V_3), where V_1 is a branch name i.e. V_1 is in domain D_1, V_2 is an accountNumber i.e. V_2 is in D_2 and V_3 is a balance i.e. V_3 is in domain D_3.

In general, account will contain only a subset of all possible rows. Therefore account is a subset of

$D_1 \times D_2 \times D_3$

Thus, a table of n attributes must be a subset of

$D_1 \times D_2 \times D_3 \times \ldots\ldots \times D_{n-1} \times D_n$

Mathematicians define a relation to be a subset of a cartesian product of a list of domains. This definition corresponds atmost exactly with definition of table. As tables are essentially relations, the mathematical terms relation is used in place of table and tuple is used in place of row.

In the account relation of figure, there are three tuples. Let the tuple variable t refer to the first tuple of the relation. The notation t[branchName] is used to denote the value of t on the branchName attribute. Thus, t[branch Name]="KTM" and t[balance]=50,000. Alternatively t[1] is used to denote the value of tuple t on the first attribute (branchName), t[2] to denote accountNumber and so on. Mathematical notation of $t \in r$ is used to denote that tuple t is in relation r.

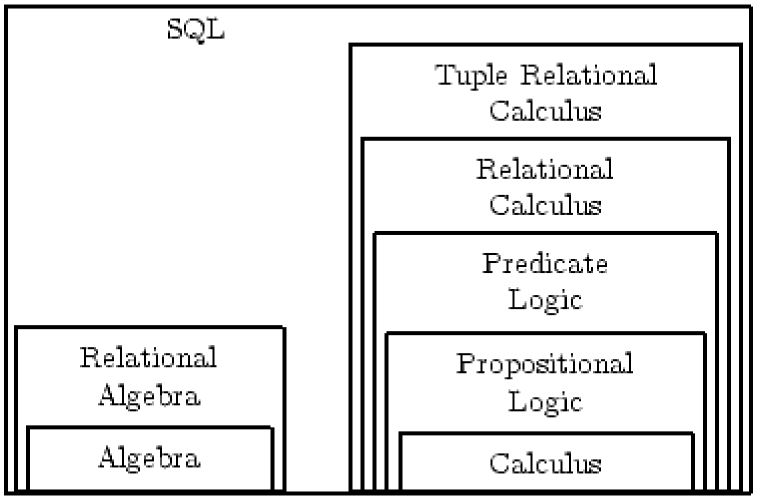
It is possible for several attributes to have the same domain. It is possible that the attributes customerName and employeeName will have the same domain.

Database schema is the logical design of the database. Account schema is used to denote their elation schema for relation account as follows.

Account-schema=(branchName, accountNumber, balance)

Thus, account is a relation on Account-schema by account (Account-schema)

Relational algebra



Relational algebra is a collection of operations to manipulate relations. The fundamental operations in the relational algebra are select, project, union, set difference, Cartesian product and rename. The result of each of these operations is also a relation. Relational algebra is theoretical realization of query language, relational calculus is theoretical representation of query language and sql is actual query language.

Relational algebra is a procedural language. It specifies the operations to be performed on existing relations to derive result relations. Furthermore, it defines the complete scheme for each of the result relations.

Fundamental operations

The fundamental operations of relational algebra are select, project, union, set difference, Cartesian product and rename. The select, project and rename operations are called unary operations as they operate on one relation. Where as union, set difference and Cartesian product are called binary operations as they operate on pairs of relations

Select operation

The select operation selects tuples that satisfy a given predicate. The lowercase Greek letter sigma (σ) is used to denote selection. The predicate appears as a subscript to σ. The argument relation is given in parentheses following the σ.

Consider the following relation.

BranchName	LoanNumber	Amount
Kathmandu	L-1	5,000
Lalitpur	L-2	7,000
Bhaktapur	L-3	50,000
Kathmandu	L-4	3,000
Lalitpur	L-5	55,000
Kathmandu	L-6	40,000

fig Loan relation

- To select the tuples of the loan relation where branch is "Lalitpur", we write

  $\sigma_{\text{branchName = "Lalitpur"}}$ (Loan)
- To find all tuples in which the amount is more than 40,000 we write

  σ_{amount}>40,000 (Loan)
- Comparisons are done using = ≠, <, ≤, >, ≥ in the selection predicate. Several predicates can be combined using the connectives and (^) and or (v).

Thus, to find those tuples pertaining to loans of more than 3,000 made by the Kathmandu branch, we write.

$\sigma_{\text{branchName ="Kathmandu"^amount> 3,000}}$ (loan)

Project Operation

The project operation is a unary operation that returns its argument relation, with certain attributes left out. Suppose we want to list all loan numbers and the amount of the loans, but not the branch name, then the project operation can be used for this purpose. Since a relation is a set, any duplicate rows are eliminated. It is denoted by the Greek letter pi(π). We list those attributes that we wish to appear in the result as a subscript to π The query to list loan numbers and the amount of the loan can be written as

$\pi_{\text{loanNumber, amount}}$ (loan)

Result:

loanNumber	amount
L-1	5,000
L-2	7,000
L-3	50,000
L-4	3,000
L-5	55,000
L-6	40,000

fig. Loan no. and the amount of the loan as a result of π loanNumber, amount(loan)

Composition of relational operations

Suppose, we want to list the name of customers who live in "Kathmandu". Then we write name of customers who live in "Kathmandu". Then we write

$\pi_{\text{customerName}}$ ($\sigma_{\text{customercity="Kathmandu}}$"(customer))

Here, instead of giving the name of a relation as the argument of the project operation, we give an expression that evaluates to a relation.

Union Operation:

CustomerName	Account No.
Ram	A – 1
Manu	A – 2
Sushant	A – 3
Anand	A – 4

Depositor

customerName	Loan No.
Prajesh	L – 1
Manil	L – 2
Anand	L – 3
Ram	L – 4

Borrower

customerName	city
Ram	KTM
Rahul	Lalitpur
Mohan	Jhapa
Radha	Dang

Customer

The union operation is similar to that of union of mathematics. Suppose we have to find the names of all bank customers who have either an account or a loan or both. As the Depositor relation doesn't contain information about loan and Borrower relation does not contain information about depositor customer, it is necessary to get information from both the depositor relation and the borrower relation. This is done using UNION operation.

To find Names of customers with loan, we write $\pi_{\text{customername}}$ (loan)

To find Names of customers with an account, we write $\pi_{\text{customerName}}$(Depositor)

To find all customers that appear in either or both of the two above relations, we write

$\pi_{\text{customerName}}$ (borrower) U $\pi_{\text{customerName}}$ (Depositor)

The operation is denoted by 'U'. Union is binary operation. For a union operator $r \cup s$ to be valid, the following two conditions are necessary.

1. The relations r and s must be of the same arity i.e. they must have the same number of attributes.
2. The domains of the ith attribute of r and the ith attribute of s must be the same for all i.

Set difference operation:

It is denoted by '_'. It is used to find tuples that are in one relation but are not in other. The expression r - s results in a relation containing those tuples in r but not in s.

Suppose, we want to list all customers who have an account but not a loan, then we write

$\pi_{\text{customerName}}$ (depositor) - $\pi_{\text{customername}}$ (borrower)

Result:

customername
Manu
Shusant

Fig. customers with account but not loan.

Cartesian Product Operation:

It is denoted by 'X'. It is used to combine information from any two relations. Cartesian product of r and s is denoted by r×s. Relation is defined to be a subset of a Cartesian product of a set of domains. Since the same attributes name may appear in both r and s, we need to devise a naming schema to distinguish between these attributes. This is done by attaching to an attribute the name of the relation from which the attribute originally came. For example, the relation schema for r = borrower × loan is

(borrower.customerName, borrower. loanNumber, loan.branchName, loan.loanNo, loan.amount)

Suppose there are n_1 tuples in borrower and n_2 tuples in loan. Then there are $n_1 * n_2$ ways of choosing a pair of tuples, one tuple from each relation. So there are $n_1 * n_2$ tuples in r.

Suppose we want to find names of all customers who have a loan at the Kathmandu branch. We need the information in both the loan relation and the borrower relation to do so. Since the Cartesian operation associates every tuple of loan with every tuple of borrowers.

So, if we write

$\sigma_{borrower.loanNumber\ =\ loan.loanNumber}\ (\sigma_{branchName="Kathmandu"}(borrower \times loan))$

results in tuples of borrower × loan that pertain to customers that have a loan at the Kathmandu branch.

Finally, since we want only customerName, we write

$\pi_{customername}\ (\sigma_{borrower.loanNumber\ =loan.loanNumber}\ (\sigma_{branchName="Kathmandu"}(borrower \times loan)))$

Rename operation:

It is denoted by the lower-case Greek letter rho (ρ). Given a relational – algebra expression ∈, the expression $\rho_x(\in)$ returns the result of expression ∈ under the name x.

Suppose we want to find the largest account balance in the bank. For this let us compute a temporary relation consisting of those balances that are not the largest. The temporary relation is created computing the Cartesian product and forming a selection to compare the value of any two balances appearing in one tuple. This is done using the rename operation as follows

$\pi_{account.balance}\ (\sigma_{account.balance<d.balance}\ (account \times \rho_d\ (account)))$

results in a temporary relation with all balances except the largest one. Now to find the largest account balance, we write

$\pi_{balance\ (account)} - \pi_{account.balance}\ (\sigma_{account.balance\ <d.balance\ (account\ \times\ \rho d}\ (account)))$

Similarly, the rename operation can be used to find the customer name who live on the same city.

Additional Operation:

Some additional operations used in relational algebra are the set intersection operation, natural join operation, division operation and assignment operation.

Set-intersection operation: Suppose we want to list customers who have both a loan and an account. Then, using set intersection we can write.

$\pi_{customerName}\ (borrower)\ \cap \pi_{customerName}(Depositor)$

We can rewrite any relational algebra expression using set intersection by replacing the intersection operation with a pair of set difference operations as follows:

$r \cap s = r - (r - s)$

Thus, set intersection is not a fundamental operation and does not add any power to the relational algebra. It is simply more convenient to write $r \cap s$ than to write r-(r-s).

Natural join operation:

Suppose we want to list the names of all customers who have a loan at the bank and find the amount of the loan. We first form the Cartesian product of the borrower and loan relations. Then we select those tuples that pertain to only the same loanNumber followed by the projection of the resulting customerName, loanNumber and amount

$\pi_{customerName}$, loan.loanNumber, amount ($\sigma_{borrower.loanNumber = loan.loanNumber}$(borrower × loan)

The natural join is a binary operation that allows us to combine certain selection and a Cartesian product into one operation. It is denoted by the "join" symbol $\bowtie$. The natural join operation forms a Cartesian product of its two arguments, performs a selection forcing equality on those attributes that appear in both relation schemas and finally removes duplicate attributes.

Consider two relations r(R) and s(S). The natural join of r and s, denoted by r $\bowtie$ s is a relation on schema RUS formally defined as follows:

$r \bowtie s = \pi_{RUS}(\sigma_{r.A1=S.A1 \wedge r.A2=S.A2\wedge \ldots \wedge r.An=S.An}(r\times s))$

Where $R\cap S=\{A_1,A_2, \ldots A_n\}$

Division Operation:

The division operation is denoted by ÷. It is suited to queries that include the phrase "for all" suppose that we wish to find all customers who have an account at all the branches located in Kathmandu.

We write,

$r_1 = \pi_{branchName}(\sigma_{branchcity="Kathmandu"}(branch))$

We can find all (customerName, branchName) pairs for which the customer has an account at a branch – by writing.

$r_2 = \pi_{customer\ Name,\ branchName}$ (depositor $\bowtie$ account)

Now, we need to find customers who appear in r_2 with every branchName in r1. The operation that provides exactly those customers is the divide operation.

We formulate the query by writing

$\pi_{customerName,\ branchName}$(depositor $\bowtie$ account) ÷ $\pi_{brachName}$ ($\sigma_{branchcity = "Kathmandu"}$ (branch))

The result of this expression is a relation.

Formally, let r(R) and s(S) be the relations and let $S \subseteq R$. That is every attribute of schema S is also in Schema R. The relation r÷s is a relation on schema R-S- that is, on the schema containing all attributes of schema R that are not in Schema S.

A tuple t is in r÷s if and only if both of two conditions hold.

1. t is in π_{R-S} (r)
2. For every tuple ts is S, there is a tuple t_r in r satisfying both of the following.
   a. t_r [S] = t_s[S]
   b. t_r [R – S] = t

Assignment operation:

The assignment operation is denoted by ←. It is similar to assignment in a programming language we could write r÷s as

temp1 ← π_{R-S} (r)

temp2 ← π_{R-S} ((temp 1 × s) – r)

result = $temp_1 - temp_2$

The evaluation of an assignment doesn't result in any relation being displayed to the user. Rather, the result of the expression to the right of the ← is assigned to the relation variable on the left of the ←. This relation variable may e used in subsequent expressions.

Relational Calculus

Calculus is descriptive. Algebra provides a collection of explicit operators that specify how to construct some relation from other relations. Calculus states the definition of a relation in terms of other relation through some notation.

Algebra and calculus are logically equivalent. For every algebraic expression, there is an

equivalent calculus expression. For every calculus expression, there is an equivalent algebraic expression.
Relational calculus is of 2 types:
1. Tuple Relational calculus
2. Domain Relational calculus

1. **Tuple relational calculus:** A query in the tuple relational calculus is expressed as

   $$\{ t \mid p(t) \}$$

   i.e. it is the set of all tuples t such that predicate is true for t. t[A] to is used to denote the value of tuple t on attributes A and we use $t \in r$ to denote that tuple t is in relation r.
   Suppose that we want to find the branchName, loanNumber and amount for loans of over 5000.

   $\{ t \mid t \in loan \wedge t[amount] > 1200 \}$

   Suppose that we want to find the loan number for each loan of an amount greater than 5000.

   $\{ t \mid t \ni s \in loan\ (t[loanNumber] = s[loanNumber]\} \wedge s[amount]>5000) \}$

   Where $\ni s \in loan\ (t[loanNumber] = s[loanNumber]\} \wedge s[amount]>5000) \}$ means that there exists a tuple s in relation loan such that the condition is true.
   A tuple relational calculus is of the form

   $$\{ t \mid p(t) \}$$

   where p is a formula. Several tuple variables may appear in a formula.
   Relational algebra is theoretical realization of query language and relational calculus is theoretical representation of query language where as SQL is actual query language.
2. **Domain Relational calculus:** This is the second form of domain relational calculus. This form uses domain variables that take on values from an attribute from an attribute's domain, rather than values for an entire tuple. The domain relational calculus , however, is closely related to the tuple relational calculus.
   An expression in the domain relational calculus is of the form

   $$\{<x_1,x_2,x_3,\ldots\ldots,x_n> \mid p(x_1,x_2,x_3,\ldots\ldots,x_n)\}$$

   where $x_1,x_2,x_3,\ldots\ldots,x_n$ represent domain variables. P represents a formula composed of atoms.
   e.g. find the branchName, loanNumber, and amount for loans of over 5000.

   $$\{<b, l, a> \mid <b, l, a> \in loan \wedge a > 5000 \}$$

   Find all loan numbers for loans with an amount greater than 7000.

   $$\{ < l > \mid \ni b, a\ (<b, l, a> \in loan \wedge a > 5000) \}$$

Views

Any relation that is not part of logical model but is made visible to a user as a virtual relation is called a view. It is possible to support a large number of views on top of any given set of actual relations.

It is not desirable for all users to see the entire logical model. Security considerations may require that certain data be hidden from users. Consider a person who needs to know a customer's loan number but has no need to see the loan amount. The person should see a relation described in the relation algebra by

$\pi_{customerName,\ loanNumber}$ (borrower IXI loan)

A view can be defined using the create view statement. The view must be given a name and should have a query that computes the view. The form of the create view statement is create view viewName as <query expression>

where viewname is any valid name of the view and <query expression> is any logical relational algebra query expression. Consider the view consisting of branches and their customers as follows.

create view viewCustomer as

$\pi_{branchName, customerName}$ (depositor $\bowtie$ account)

$\cup$ $\pi_{branchName, customerName}$ (borrower $\bowtie$ loan)

Once we have defined a view, we can use the view name to refer to the virtual relation. Using this view, we can find the customers of the Kathmandu branch by writing

$\pi_{customerName}$ ($\sigma_{branchName="Kathmandu"}$(viewcustomer))

Certain database systems allow new relations to be stored, such views are called materialized views. Views are a useful tool for queries but they present significant problems if updates, insertions, or deletions are expressed with them. The difficulty is that a modification to the database expressed in terms of a view in the logical model of the database.

Views can also be created using other views.

Domains

In any relation, for each attribute, there is a set of permitted values, called domains of that attribute. For the attribute. For the attribute branch name, domain is the set of all branches.

For all relations r, the domains of all attributes of r be atomic. A domain is atomic if elements of the domain are considered to be individual units. For example, the set of integers is an atomic domain, but the set of all sets of integers is a non-atomic domain. The distinction is that we do not normally consider integers to have subparts, but we consider sets of integers to have subparts namely, the integers comprising the set. The important issue is not what the domain itself is, but rather how we use domain elements in our database. The domain of all integers would be non-atomic if we consider each integer to be an ordered list of digits.

It is possible for several attributes to have the same domain. For example suppose that we have a relation customer that has the three attributes customers name customerStreet, customerCity. Similarly Employee has the attribute employeeName. It is possible that the attributes customerName and employeeName will have the same domain.

Some examples of domains follow:

i. Kathmandu Phone Numbers: The set of 10 digit phone no. valid in KTM.
ii. Names: The set of names of persons
iii. EmployeeAges: Possible ages of employees of a company; each must be a value between 15 and 80 yrs. old.

Relations

Given a collection of n types of domains $T_{i\ (I=1,2,\ldots.,\ n)}$, r is a relation on those types if it consists of two parts, a heading and a body

Where

i. The heading is a set of n attributes of the form Ai: T_i, where Ai(Which all must be distinct) are the attribute names of r and the T_i are the corresponding type names(i=1,2,…n)
ii. The body is a set of m types t , where t_i in turn is a set of components of the form Ai :vi in which vi s_i a value of type t_i…the attribute value for the attribute of tuple t_i (i=1,2,…n)

The values m and n are the called cardinality and the degree respectively of relation r.

A relation is what the definitions says, it is namely a rather abstract kind of object and table is a concrete picture of such an abstract object. Of course, they are very similar and in informal contexts, at least , it is usual and perfectly acceptable to say they are the same.

Properties of relations:

i. There are no. duplicates tuples. There is always a primary key
iii. Tuples are unordered, top to bottom. There is no concept of positional addressing and nextness
iv. attributes are ordered, left to right
v. each tuple contains exactly one value for each attribute. All attribute values are atomic.
vi. Normalized relation : a relation that does not contain repeating groups.
vii. Advantage of normalization : to allow simpler structure and simpler operations.

Kinds of relations

i. base relations
ii. views
iii. snapshots
iv. query results
v. temporary relations

Predicates

Employee

Empno | emp#	Ename | ename#	Deptno | dept#	Salary | money
E1	Ram	D1	7000
E2	Shyam	D2	8000
E3	Hari	D3	9000
E4	Mohan	D4	6000

Fig. Sample Employee relation value with column types shown

Given a relations, the heading of r denotes a certain predicate or truth valued function Each row in the body of r denotes a certain true proposition obtained from the predicate by substituting certain argument values of the appropriate type for the parameters of that predicate.

In fig, the predicate looks something like this:

Employee emp# is name ENAME , works in department DEPT#, and earns salary SALARY.

(The parameters are EMP#, ENAME, DEPT# and SALARY corresponding of course to the four EMP Columns) and the corresponding true propositions are:

Employee E1 is named RAM, works in department D1 and earns salary 7000. (Obtained by substituting the EMP# Value E1, the Name value Ram, the Dept# value D1 and the MONEY value 7000 for appropriate parameters).

Thus, every relation r (or table t) has an associated predicate p (heading or set of attributes), even relations that are derived from others using relational algebra operations.

Integrity Constraints

Integrity constraints guard against accidental damage to database.

Entity Integrity

Entity integrity ensures that ***each row in the table is uniquely identified***. In other words, entity integrity ensures a table does not have any duplicate rows. Example: Two separate customers should not have the same customer number .SQL Server will allow duplicate rows if entity integrity is not enforced. Entity integrity is a key concept in the relational database model. Data in the relational database is independent of physical storage; there is no such thing as the '5th customer row' in a table. Physical independence is achieved by being able to reference each row by a unique value, sometimes referred to as a 'key'. Entity integrity ensures that each row in a table has a unique identifier that allows one row to be distinguished from another. Entity integrity is most often enforced by placing a ***primary key (PK)*** constraint on a specific column (although it can also be enforced with a UNIQUE constraint, a unique index, or the IDENTITY property) .The PK constraint forces each value inserted into a column (or combination of columns) to be unique; if a user attempts to insert a duplicate value into the column(s), the PK constraint will cause the insert to fail
A PK will not allow any Nulls to be inserted into the column(s) (A NULL entry would be disallowed even if it would be the only NULL in the column and therefore unique.) . A PK is referred to as a ' surrogate key' if the column contains no real data other than a uniqueness identifier .If 'real' data can be used as a PK (e.g., a social security number), then it is referred to as an ' intelligent key' .There can be only one PK per table .A ***composite PK*** is a PK that consists of more than one column; it is used when none of the columns in the composite key is unique by itself .Thus, there can be only one PK in a table but the PK can consist of more than one column .If you need to enforce uniqueness on more than one column, use a PK constraint on one column and a UNIQUE constraint or IDENTITY property on any other columns that must not contain duplicates .Example: If the 'customer ID' column is the PK in the 'customers' table and you also want to make sure there are no duplicate customer names, you can place a UNIQUE constraint on the 'customer name' column . Non-PK columns on which uniqueness is enforced are referred to as ***alternative keys*** or AKs; they get their name from the fact that they are 'alternatives' to the PK and as such, make good candidates for indexing or 'joining' on.

Domain Constraint

A domain of possible values must be associated with every attribute SQL allows the domain declaration of an attribute to include the specification "not null" and thus prohibits insertion of a null value for this attribute. Any database modification that would cause a null to be inserted in a not null domain generates an error diagnostic. There are many situations where the prohibition of null values is desirable. A particular case where it is essential to prohibit null values is in the primary key of a relation schema.

The SQL-92 allows us to define domains using a create domain clause, as shown in the following example.

```
create domain personName char (60)
```

We can then use the domain name personName to define the type of an attribute, just like a built-in domain.

Domain constraints are the most elementary form of integrity constraint. They are tested easily by the system whenever a new data item is entered into the database. It is possible for several attributes to have the same domain. The principle behind attribute domains is similar to that behind typing of variables in programming languages.

The check clause in SQL-92 permits the schema designer to specify a predicate that must be satisfied by any value assigned to a variable whose type is the domain. For instance, a check clause can ensure that an hourly wage domain allows only values greater that a specified value (such as minimum wage) as shown below.

create domain hourlywage numeric (5, 2)

Constraint wage, valuetestcheck (value> = 4.00)

The domain hourlywage is declared to be a decimal number with a total of five digits, two of which are placed after the decimal point, and the domain has a constraint that ensures that the hourlywage is equal to or greater that 4.00.

The check clause can also be used to restrict a domain not to contain any null values, as shown below.

e.g.: create domain accountNumber char(10)

constraint accountNumber NullTest check (value not null)

create domain gender char (10)

constraint checkgendercheck (value in ("Male", "Female")

Referential Integrity

It is also required that a value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation. This condition is called referential integrity.

The referential integrity constraint is specified between two relations and is used to maintain the consistency among tuples of the two relations. Informally, the referential integrity constraint states that a tuple in one relation that refers to another relation must refer to an existing tuple in that relation. Consider the two relations EMPLOYEE and DEPARTMENT as follows.

EMPLOYEE

NAME	SSN	Address	Sex	Salary	Dept No.

DEPARTMENT

Dept No.	DeptName	MGRSSN

The attribute dept No. of EMPLOYEE gives the department Number for which each employee works. hence, its value in every EMPLOYEE tuple must match the dept no. value of some tuple in the DEPARTMENT relation. To define referential integrity more formally, we must first define the concept of a foreign key. The conditions for a foreign key between two relation schemas R_1 and R_2 states that a set of attributes FK in relation schema R_1 is a foreign key of R_1 that references relation R_2 if it satisfies the following two rules.

i. The attributes in FK have the same domain as the primary key attributes PK of R_2. The attributes FK are said to reference or refer to the relation R_2.

ii. A value of FK in a tuple t_1 of the current state $r_1(R_1)$ either occurs as a value of PK for some tuple t_2 in the current state r_2 (R_2) or is null. In the former case, we have $t_1[FK] = t_2 [PK]$, and we say that the tuple t_1 references or refers to the tuple t_2. R_1 is called the referencing relation and R_2 is the referenced relation.

In a database of many relations, there are usually many referential integrity constraints. To specify these constraints, we must first have a clear understanding of the meaning or role that each set of attributes plays in the various relation schemas of the database.

In the EMPLOYEE relation the attribute deptNo refers to the department for which employee work hence, we designate deptNo to be a foreign key of EMPLOYEE, referring to the DEPARTMENT relation. This means that a value of deptNo in any tuple t_1 of the EMPLOYEE relation must match a value of the primary key of the department.

We can diagrammatically display referential integrity constraints by drawing a directed arc from each foreign key to the relation it references. For clarity, the arrowhead may point to primary key of the referenced relation.

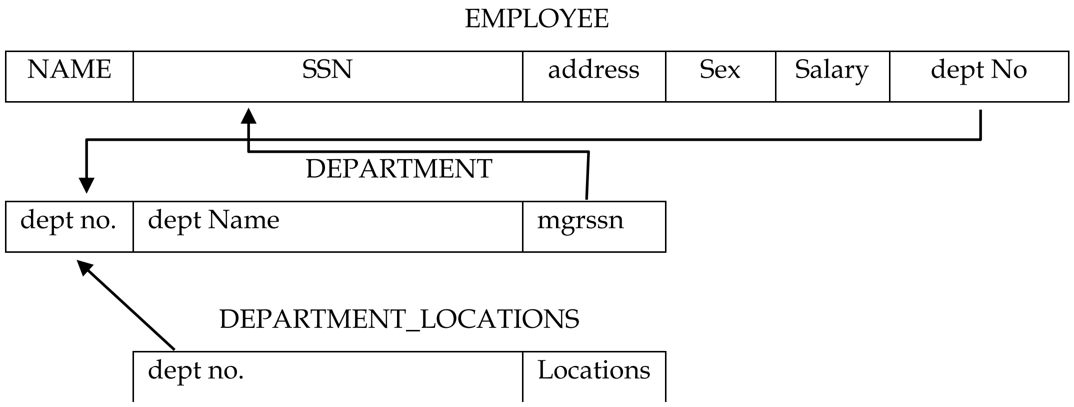


Fig. Referential integrity constraints

Referential integrity in SQL :

Primary and foreign key can be specified as part of the SQL create table statement.

- The primary key clause of create table statement includes a list of attributes that constitute a candidate key.
- The UNIQUE clause of create table statement includes a list of the attributes that constitute a candidate key.
- The FOREIGN KEY clause of create table statement includes both a list of attributes that constitute foreign key and the name of the relation referenced by the foreign key.

Assertion

An assertion is a predicate expressing a condition that we wish the database always to satisfy. Domain constraints and referential integrity constraints are special forms of assertions. However, there are many constraints that we can't express using only these special forms.

For example suppose the constraints are:

i. The sum of all loan amounts for each branch must be less than the sum of all account balances at that branch.

ii. Every loan has at least one customer who maintains an account with a minimum balance of 50,000.

An assertion in SQL-92 takes the form

```
create assertion <assertionName> check <Predicate>
e.g.:
create assertion sumConstraint check
(not exists (select * from branch
where (select sum(amount) from loan
where loan.branchName = branch.branchName)
>= (select sum (amount) from account
where account.branchName = branch.branchName)))
```

When an assertion is created, the system tests it for validity. If the assertion is valid, then any further modification to the database is allowed only if it does not cause that assertion to be violated.

Triggers

A trigger is a statement that is executed automatically by the system as a side effect of a modification to the database. Trigger must contain the following two requirements.

i. specify the conditions under which the trigger is to be executed.

ii. specify the actions to be taken when the trigger executes.

Triggers are useful mechanisms for alerting humans, or for performing certain tasks automatically when certain conditions are met. Triggers are sometimes called rules or active rules. Triggers are written in both front end and backend. If the triggers are written in backend, they are called database triggers.

Types:

i. row level triggers
ii. statement level triggers
iii. before and after triggers
iv. database level triggers

Triggers can be written for events such as insert, update, delete, create, alter, drop etc.

For example suppose we want to store username and the system date into a table logdata. For this purpose, the trigger can be written as follows.

```
CREATE OR REPLACE TRIGGER tg_before_update_user
BEFORE INSERT OR UPDATE
ON Policies
FOR EACH ROW
BEGIN
        INSERT INTO LOGDATA
        VALUES (USER, SYSDATE); COMMIT;
END;
```

In this trigger, if any insert or update is made in the policies table, then the user name & current date is stored in the logdata table.

SQL

Introduction

The history of SQL begins in an IBM laboratory in San Jose, California, where SQL was developed in the late 1970s. The initials stand for Structured Query Language, and the language itself is often referred to as "sequel." It was originally developed for IBM's DB2 product (a relational database management system, or RDBMS, that can still be bought today for various platforms and environments). In fact, SQL makes an RDBMS possible. SQL is a nonprocedural language, in contrast to the procedural or third-generation languages (3GLs) such as COBOL and C that had been created up to that time.

Background

- SQL = **Structured Query Language**
- Created in late 70's at IBM, under the name of SEQUEL
- Went through major standardizations (which contributed to its wide acceptance):

SQL-86 (SQL1)
Queries + some schema definitions & manipulation
SQL-89
Referential integrity
SQL-92 (SQL2)
Revised and expanded
SQL-99 (SQL3)
Archive rules & triggers, recursive operations, aggregate operations, object-oriented features

- Consists of
  - A **Data Definition Language (DDL)** for declaring database schemas. e.g. create table
  - **Data Manipulation Language (DML)** for modifying and querying database instances . It also includes commands to insert tuples into, deletes tuples from, and to modify tuples in the database. e.g.select,insert,update,delete,explain,lock table etc.
  - **Embedded DML :** The embedded form of SQL is designed for use within general-purpose programming languages, such as Cobol, Pascal, Fortran and C.
  - **View Defintion:** The SQL DDL includes commands for defining views.
  - **Authorization:** The SQL DDL includes commands for specifying access rights to relations and views.
  - **Integrity:** The SQL DDL includes commands for specifying integrity constraints that the data stored in the database must satisfy.
  - **Transaction Control:** SQL includes commands for specifying the beginning and ending of transactions. e.g. set transaction, commit, rollback
  - **Session Control:** Manages the properties of user session. e.g. alter session
  - **System Control :** manipulates the properties of database. e.g. alter system

Basic Structure

The basic structure of an sql expression consists of three clauses: select, from and where. The **select** clause corresponds to the projection operation of the relational algebra. It is used to list the attributes desired in the result of query.

The **FROM** CLAUSE corresponds to the Cartesian product operation of the relational algebra. It lists the relations to be scanned in the evaluation of the expression.

The **WHERE** clause corresponds to the selection predicate of the relational algebra. It consists of a predicate involving attributes of the relations that appear in the from clause.

select *attribute-expression*
from *table*
[**where** *condition*]

i.e. A typical SQL query has the form

SELECT A1,A2,A3,….,An
FROM R1,R2,R3,…..,Rm
WHERE P

Each Ai represents an attribute, and each ri a relation. P is a predicate. The query is equivalent to the relational algebra expression
$\Pi_{A1,A2,\ldots An}(\sigma_P(r_1 \times r_2 \times \ldots \times r_m))$

select Name
from Students
where name<'N';

STUDENTS

Name	Number	Sex
Ben	3412	M
Dan	1234	M
Nel	2341	F

Name
Ben
Dan

The result of this select statement is a relation consisting of a single attribute with the heading Name.
We can rewrite the preceding query as
Select distinct name from students if we want duplicates to be removed. SQL allows us to use the keyword ALL to specify explicitly that duplicates are not removed.

Select all name from students.
Since duplicate retention is the default, we will not use all in the query.
The asterisk symbol "*" is used to denote all attributes.
The select clause can also contain arithmetic expressions involving the operators, +,-,*, and /, and operating on constants or attributes of tuples.

Select branchName, loanNumber, loanAmount*100
From Loan

Similarly SQL uses the logical connectives and, or, and not rather than the mathematical symbols in the where clause.

e.g. Select loanNumber from loan where loanAmount<50000 and branchName="Kathmandu";

If we wish to find the loan number of those loans with loan amounts between 100000 and 500000, we can use
Select loanNumber
From loan
Where loanAmount between 100000 and 500000;
Operators in conditions: =, <>, <, >, <=, >=, or, and, not

Renaming Attributes

SQL provides a mechanism for renaming both relations and attributes. It uses **as** clause, taking the form

oldName as NewName

select *attribute-expression* [**as**] *target-attribute*
from *table*
[**where** *condition*]

```
select Name as Names
from Students
where Name < 'N'
```

STUDENTS

Name	Number	Sex
Ben	3412	M
Dan	1234	M
Nel	2341	F

Names
Ben
Dan

Multiple Attributes

```
select Name as Names, Number * 10 as Num
from Students
where Name < 'N'
```

STUDENTS

Name	Number	Sex
Ben	3412	M
Dan	1234	M
Nel	2341	F

Names	Num
Ben	34120
Dan	12340

The full list of attributes may be referenced through the character '*'

```
select *
from Students
where Name < 'N'
```

Multiple Tables

```
select Code
from Students, Classes
where Students.Number = Classes.Num
```

STUDENTS

Name	Number	Sex
Ben	3412	M
Dan	1234	M
Nel	2341	F

CLASSES

Code	Num
670	1234
680	1234

Code
670
680

680	4123

- The dot operator is available for distinguishing attributes of different tables
- When no ambiguity arises, the identifying tables are not needed.

```
select Classes.Code
from Students, Classes
where Number = Num
```

Aliases for Tables

from *table* [**as**] *alias*,....

select S1.FN, S1.LN
from Students as S1, Students as S2
where S1.LN = S2.LN and S1.FN <> S2.FN

STUDENTS

FN	LN
Ben	Smith
Dan	McLean
Nel	Smith

FN	LN
Ben	Smith
Nel	Smith

Duplicate Tuples

- In general, SQL tables are multisets, allowing duplicated rows in the tables
- Some SQL tables (e.g., with key attributes) are forced to be sets
- Requests to remove the extra entries can be made with the '`distinct`' keyword, following the '`select`' keyword.

```
select S1.LN
from Students as S1,
     Students as S2
where S1.LN = S2.LN
  and S1.FN <> S2.FN
```

```
select distinct S1.LN
from Students as S1,
     Students as S2
where S1.LN = S2.LN
  and S1.FN <> S2.FN
```

LN
Smith
Smith

LN
Smith

Set Operations

The set operations include UNION, INTERSECT AND EXCEPT operations on relations.
select ... <union | intersect | except> [all] select ...

a. The UNION Operation:

To find all customers having a loan, an account, or both at the bank we write

```
SELECT  customerName  from depositor)
UNION
SELECT customerName from borrower)
```

If we want to retain all duplicates , we must write UNION ALL in place of union as follows.

```
     (SELECT  customerName  from depositor)

     UNION ALL

     (SELECT customerName from borrower)
```

```
select FN as Name from Students
    union
select LN as Name from Students
```

STUDENTS

FN	LN
Ben	Smith
Dan	McLean
Nel	Smith

Name
Ben
Dan
Nel
Smith
McLean

- The 'all' requests to retain duplicates. The default is to eliminate them
- After aliasing, the tables involved in the operations must agree on their attributes, and the ordering of the attributes

b. The INTERSECT operation: To find all customers who have both a loan and an account at the bank, we write

(select customerName from depositor)
intersect
(select distinct customerName from borrower)

The intersect operation also automatically eliminates duplicates . If we want to retain all duplicates, we must write INTERSECT ALL in place of intersect.

(select customerName from depositor)
intersect all
(select distinct customerName from borrower)

c. The EXCEPT operation: To find all customers who have an account but no loan at the bank, we write

(select distinct customerName from depositor)
except
(select customerName from borrower)

If we want to retain all duplicates , we must write EXCEPT ALL in place of except as follows.

(select distinct customerName from depositor)
except all
(select customerName from borrower)

Ordering

Ordering may be imposed on rows of tables, based on values of attributes.

order by *attribute* [**asc** |**desc**],**...**

The default assumes ascending order

select Name,Number
from Students order by Name

String Comparisons

- **=, !=,...** standard operations
- **like**, binary operator for comparing string patterns
- %, wild card for strings . The % character matches any substring.
- _, wild card for characters . The – character matches any character.
- `(name` **like** `'%a_')` is true for all names having 'a' as second letter from the end.
- **`Like`** `"ab\%cd%"  escape "\" matches all strings beginning with "ab%cd"`

suppose "Find the names of all customers whose street address includes the substring 'PUR'. " This query can be written as

Select customerName
From customer
Where customerStreet like "%PUR%";

Similarly,
like "ab\%cd%" escape "\" matches all strings beginning with "ab%cd".

Null Values

An attribute can be checked whether it 'is null' or it 'is not null'.

SQL allows the use of null values to indicate absence of information about the value of an attribute.
The result of an arithmetic expressions (involving for example +,-,* or /) is null if any of the input values is null. The result of any comparison involving a null value can be thought of as being false.

select ...
from ...
where (x is null) and (y is not null)

Aggregate Queries

Aggregate functions are functions that take a collection (a set or multiset) of values as input and return a single value. SQL offers five built-in aggregate function.

- Average: avg
- Minimum: min
- Maximum: max
- Total: sum
- Count: count

count(*), count ([distinct] attributes)
Counts the number of tuples

```
select count(*)
from Students as S1,
     Students as S2
where S1.LN = S2.LN
```

STUDENTS

FN	LN	Number
Ben	Smith	3412
Dan	McLean	1234
Nel	Smith	2341

Count
2

```
        and S1.FN <> S2.FN
```

sum ([distinct] ***attributes*****)**

max ([distinct] ***attributes*****)**

min ([distinct] ***attributes*****)**

avg ([distinct] ***attributes*****)**

```
     select min(Number), max(Number), avg(Number)
from Students
```

There are circumstances where we would like to apply aggregate function not only to a single set of tuples, but also to a group of sets of tuples; we specify this in SQL using **group by** clause. The attribute or attributes given in the group by clause are used to form groups.

e.g. "Find the average account balance at each branch". We write this query as follows. Select branchName, avg(balance)

From account

Group by branchName;

Group Clauses

Tables might be partitioned to subsets of rows which agree on their entries for given attributes. The attributes are to be specified within a 'group by' clause, and are the only ones allowed in the projection specified by the 'select' component.

select Sex, sum(Number)
from Students
group by Sex

STUDENTS

Name	Number	Sex
Ben	3412	M
Dan	1234	M
Nel	2341	F

Sex	sum_Number
M	4646
F	2341

Group predicates through 'having' clause may be added, to exclude subgroups which don't satisfy desirable conditions.

select Sex, sum(Number)
from Students
group by Sex
having sum(Number) < 3000

STUDENTS

Name	Number	Sex
Ben	3412	M
Dan	1234	M
Nel	2341	F

Sex	sum_Number
F	2341

Nested Subqueries

A common use of subqueries is to perform tests for set membership, set comparisons, and set cardinality.

a. Set Membership

SQL draws on the relational calculus for operations that allow testing tuples for membership in a relation. The IN connectives tests for set membership , where the set is a collection of values produced by a select clause. The NOT IN connective tests for the absence of set membership. We begin by finding all account holders, and we write the subquery

```
(Select CustomerName
From depositor)
```

We then need to find those customers who are borrowers from the bank and who appear in the list of account holders obtained in the subquery. We do so by nesting the subquery in an outer SELECT. The resulting query is

```
Select distinct  customerName
From borrower
Where customerName in (Select customerName from depositor)
```

Similarly to "find all the customers who have both an account and a loan at the kathmandu branch" , The query is

```
Select distinct customerName
From borrower,loan
Where borrower.loanNumber=loan.loanNumber and branchName="kathmandu" and
(branchName,customerName) IN (Select branchName,customerName from depositor,account
where depositor.accountNumber=account.accountNumber)
```

Similarly "find all customers who do have a loan at the bank who are others than "Ram", "Mohan", "Barsha". The query is

```
Select distinct customerName
From borrower
Where customerName NOT IN("Ram","Mohan"," barsha")
```

b. Set Comparison

Some, any, all are used for set comparison.
Consider the query "Find the names of all branches that have assets greater than those of at least one branch located in Kathmandu". The query is

```
Select distinct T.branchName
From branch as T, branch as S
Where T.assets > S.assets and S.branchCity="Kathmandu"
```

The alternative style for writing the preceding query is using SOME. The phrase "greater than at least one " is represented in SQL by >SOME as follows.

```
        Select branchName
From branch
Where assets > some (select assets
                    from branch where branchCity="Kathmandu")
```

SQL also allows <some, <=some, >=some, =some, <>some comparisons.

Similary "Find the names of all branches that have assets greater than that of each branch in kathmandu" . The query is

```
Select branchName
From branch
Where assets > all (select assets
                    from branch
                    where branchCity="Kathmandu")
```

```
select Name
from Students
where Number = all (select Number
            from Same)
```

STUDENTS

Name	Number
Ben	3412
Dan	1234
Nel	2341

SAME

Number
3412
3412

Name
Ben

any

```
select Name
from Students
where Number = any (select Number
            from Diff)
```

STUDENTS

Name	Number
Ben	3412
Dan	1234
Nel	2341

DIFF

Number
3412
2341

Name
Ben
Nel

```
select Name
from Students
where (Name,Number) not in
  (select Name,Number
   from Student
   where Number <> 1234)
```

STUDENTS

Name	Number
Ben	3412
Dan	1234
Nel	2341

Name
Dan

c. Test for empty Relations

SQL includes a feature for testing whether a subquery has any tuples in its result The EXISTS construct returns the value TRUE if the argument subquery is nonempty. Using the EXISTS construct , we can write the query "Find all customers who have both an account and a loan at the bank" in another way as follows.

```
Select customerName
From borrower
Where exists (select * from depositor
              where depositor.customerName=borrower.customerName)
```

d. Test for the absence of duplicate tuples

SQL includes a feature for testing whether a subquery has any duplicate tuples in its result. The UNIQUE construct returns the value true if the argument subquery contains no duplicate tuples. Using the unique construct , we can write the query " Find all customers who have only one account at the Kathmandu branch," as follows.

```
Select T. CustomerName
From depositor as T
Where unique(select R.customerName
                From account,depositor as R
                Where T.customerName=R.customerName and
 R.accountNumber=account.accountNumber and
                 Account.branchName="Kathmandu")
```

Similary we can test for the existence of duplicates tuples in a subquery by using the not unique construct. Consider the query "Find all customers who have at least two accounts at the Kathmandu branch" can be written as

```
Select distinct T. CustomerName
From depositor as T
Where not unique(select R.customerName
                From account, depositor as R
                Where T.customerName=R.customerName and
                 R.accountNumber=account.accountNumber and
                 Account.branchName="Kathmandu")
```

Views

Views are virtual tables whose contents depend on other tables. Views are defined in SQL using CREATE VIEW command.

Create or replace view viewName as <sql expression>

e.g. create or replace view v_employee as
select emid, empname from employee

```
create view Males (Nm,Num)
select Name,Number
from Students
where  Sex = 'M'
```

STUDENTS

Name	Number	Sex
Ben	3412	M
Dan	1234	M
Nel	2341	F

MALES

Nm	Num
Ben	3412
Dan	1234

We can use the view forcefully with compilation errors as follows.

Create force view viewname as <sql expression>

View can be created with compilation errors and later on the error can be fixed and compiled using

Alter view viewname compile;

View can be created read only and with check option constraint. Create view with an optional WITH READ ONLY specifies that the view will be read only. Similary WITH CHECK OPTION specifies that inserts and updates done through the view should satisfy the where clause of the view.

e.g. Create or replace view v_employee(empid,empname) as select employeeid,empname from employee where salary>2999 with check option constraint top_emp_sal;

Joined Relation

Instead of providing simple tables, combinations of them may be specified using the **inner**, **left outer**, **right outer**, and **full outer** operations.

table-1 join-op table-2 **on** *condition*

```
select Name,Code as Course
from Students inner join Classes
on Student.Number = Classes.Number
```

STUDENTS

Name	Number	Sex
Ben	3412	M
Dan	1234	M
Nel	2341	F

CLASSES

Code	Number
670	1234
680	1234
680	4123

Name	Course
Dan	670
Dan	680

```
select Name,Code as Course
from Students natural inner join Classes
```

STUDENTS

Name	Number	Sex
Ben	3412	M
Dan	1234	M
Nel	2341	F

CLASSES

Code	Number
670	1234
680	1234
680	4123

Name	Course
Dan	670
Dan	680

For natural joins, the 'natural' keyword can be specified before the operator instead of providing the condition.

Attributes names might be renamed, to facilitate the natural join operation.

```
select Name,Code as Course
from Students natural inner join
   Classes  as C(Code,number)
```

STUDENTS

Name	Number	Sex
Ben	3412	M
Dan	1234	M
Nel	2341	F

CLASSES

Code	Num
670	1234
680	1234
680	4123

Name	Course
Dan	670
Dan	680

The keyword NATURAL appears before the join type. The meaning of the join condition natural, in terms of tuples from two relations match, is straightforward. The ordering of the attributes in the result of a natural join is as follows. The join attributes appear first, in the order in which they appear in the order in the left hand side relation. Next come all nonjoin attributes of the left hand side relation and finally all nonjoin attributes of the right hand side relation.

Consider the following two tables with data.

LOAN (branchname, loannumber, amount)

downtown	L-170	3000
redwood	L-230	4000
perryridge	L-260	1700

BORROWER(customername, loannumber)

jones	L-170
smith	L-230
Hayes	L-155

branchname,	loannumber,	amount,	customername,	loannumber
downtown	L-170	3000	jones	L-170
redwood	L-230	4000	smith	L-230

fig. Result of loan inner join borrower on loan.loannumber=borrower.loannumber

branchname,	loannumber,	amount,	customername,	loannumber
downtown	L-170	3000	jones	L-170
redwood	L-230	4000	smith	L-230
perryridge	L-260	1700	null	null

fig. Result of loan left outer join borrower on loan.loannumber=borrower.loannumber

branchname,	loannumber,	amount,	customername
downtown	L-170	3000	jones
redwood	L-230	4000	smith
null	L-155	1700	hayes

fig. Result of loan natural right outer join borrower

branchname,	loannumber,	amount,	customername
downtown	L-170	3000	jones
redwood	L-230	4000	smith
perryridge	L-260	1700	null
null	L-155	null	hayes

fig. Result of loan full outer join borrower using(loannumber)

Data Definition Language (DDL) in SQL

Table Definition

create table *name* **(***attributes***)**

```
create table Student(
 Name              varchar2 (5),
 registrationNo    number(4),
class              char(1),
 gender            char(1),
 Joining_dt        date
)
```

Default Values

default *value* **| user | null**

```
     create table Student(
 Name              varchar2 (5),
 registrationNo    number(4),
class              char(1),
 gender            char(1) default 'M',
 Joining_dt        date
)
```

Constraints on Attributes

not null, unique, primary key ,foreign key, check

```
     create table Student(
 Name              varchar2 (5),
 registrationNo    number(4) constraint pk_registrationNo Primary key,
class              char(1) constraint uniq_class UNIQUE,
 gender            char(1) constraint chk_gender check(gender
in('M','F')),
 Joining_dt        date
)
```

Referential Triggers

Foreign key:

```
create table bank(
   bank_code number(10),
   bank_name varchar2(60),
   constraint pk_bank_code primary key(bank_code)
)
```

Create table branch(
branch_code number(10) constraint pk_branch_code primary key,
bank_code number(10),
branch_name varchar2(60),
constraint fk_bank_code foreign key(bank_code) references bank(bank_code)
on delete set cascade
);

When violating referential constraints

- In the default case, requested updates are rejected
- Alternative actions may be requested

on <delete | update> <cascade | set null | no action>

	updates (external table/master/parent)	**/deletes (external tables)**
cascade	Updates/ deletes child record automatically	
set null	change to null in the internal table(child or detail table)	
no action	reject action	

User-defined Data Types

create domain [*name*] **as** *known-domain* [*default-value*] [*constraints*]
create domain person_name varchar2(60);

Data Manipulation Language(DML) in SQL

Inserting Rows

insert into *table* [**(attributes)**]
<**values (***values***)** | *SQL-query* >

insert into Students (Name,Number,Sex)
values ('Don',4123,'F')

STUDENTS

Name	Number	Sex
Ben	3412	M
Dan	1234	M
Nel	2341	F

Name	Number	Sex
Ben	3412	M
Dan	1234	M
Nel	2341	F
Don	4123	F

The previous example inserts a single record, the following incorporates information from an alternative table.

```
insert into  Students
        (select Name,Number,Sex
         from Applicants
         where State = 'OH')
```

STUDENTS

Name	Number	Sex
Ben	3412	M
Dan	1234	M

Nel	2341	F

APPLICANTS

Name	Number	Sex	State
Don	4123	F	OH
Pam	3421	F	MI

STUDENTS

Name	Number	Sex
Ben	3412	M
Dan	1234	M
Nel	2341	F
Don	4123	F

Incomplete insertions are similar to

```
insert into Students (Name,Number)
        (select Name,Number
          from Applicants
          where State = 'OH')
```

Deleting Rows

delete from *table*
[**where** *condition*]

delete from Students
where Number < 2000

STUDENTS

Name	Number	Sex
Ben	3412	M
Dan	1234	M
Nel	2341	F

Name	Number	Sex
Ben	3412	M
Nel	2341	F

Updating Attributes

update *table*
set *attribute* **=** <*expr* **|** *SQL-query* **| null | default** >**,...**
[**where** *condition*]

update Students
set Name = 'Tom', Number = Number + 5
where Name = 'Dan'

STUDENTS

Name	Number	Sex
Ben	3412	M
Dan	1234	M
Nel	2341	F

Name	Number	Sex
Ben	3412	M
Tom	1239	M
Nel	2341	F

Updating Table Definitions

alter table *name*
<
add constraint *def* **|**
drop constraint *constraint* **|**
add column *def* **|**
drop column *name* **|**
alter column *name* < **set default** *value* **| drop default** >
>

Names can be assigned to constraints by a prefix of the form **constraint** *name*.

```
     create table Student(
   Name   varchar2 (5) not null,
   Number numeric(4) primary key,
   constraint foo  primary key(Number)
)
alter table Student
     add column BirthDate date
```

alter table bank
add constraint pk_bankcode primary key(bank_code);

Removing Components

drop < **table | view | assertion** > *name* [**restrict | cascade**]

- `restrict` asks the action to take place only if the component is empty
- `cascade` removes the component and its dependents

Data Control Language(DCL) in SQL

- To manipulate data, use the Data Control Language (DCL). With DCL, you can perform the following:
- CALL: Execute an SQL procedure.
- RETURN: Return a value from an SQL function.
- SET assignment: Assign a value to an SQL variable.
- SIGNAL: Raise an SQLState exception.
- VALUES: Invoke an SQL routine.
- CALL
- CALL procedure_name([argument_list])
- The CALL statement executes an SQL routine that is a procedure.
- Syntax

■ CALL	■ The CALL keyword is required in a CALL statement.
■ *procedure_name*	■ The *procedure_name* is the name of the procedure which is executed. No results are returned.
■ argument_list	■ The optional *argument_list* clause specifies values for the CALL statement.

	■ NOTE: Only constants can be used. You cannot use new or old row values.

- Examples

CALL PROC2(`abc');

- RETURN
- RETURN *SQL_expression*
- The Return statement returns a scalar value from an SQL expression. It can only be used within an SQL function.
- Syntax

■ RETURN	■ The RETURN keyword is required as the first word in a RETURN statement.
■ *SQL_expression*	■ The *SQL_expression* can be a constant, an SQL routine invocation, one of the SQL Scalar functions, an SQL Cast functions, or an SQL Special Register.

- Examples
- RETURN `abc';
- RETURN gestlastcount ( );
- RETURN `Happy New Year' ;
- RETURN NULL;
- RETURN;

- SET assignment
- SET assignment_target = assignment_source
- Examples
- SET newrow.selldate = CURRENT_DATE;

- SIGNAL
- SIGNAL `*sqlstate_message*'
- With the SIGNAL statement, you can use it to raise an SQLSTATE exception. This statement can only be used within a *trigger_body* or within the body of an SQL routine, whose language type is SQL. This statement will cause an SQLSTATE exception to be thrown and propagated back to your program.
- SIGNAL `The oranges inventory is empty';
- SIGNAL `The salary of an employee would have been higher than the salary of his/her Manager';

- VALUES
- VALUES (*SQL_expression* [{ , *SQL_expression* } ...])
- Examples
- VALUES (addnewfruit(`apple'));
- VALUES (increaseorders(200));

- VALUES (CURENT_DATE);
- Similary the commands that come under DCL are

CREATE/DROP USER/ROLE

Normalization

Pitfalls of Relational Model

A bad database design may have repetition of information and inability to represent certain information. Let us consider the lending relation with data as follows.

Lending Schema = (branchName, branchCity, assets, customerName, loanNumber, amount)

branchName	branchCity	Assets	customerName	loanNumber	Amount
RBB	Kathmandu	1000000	Ram	L – 1	1000
RBB	Lalitpur	100000	Mohan	L – 2	2000
RBB	Bhaktapur	900000	Ram	L – 3	3000
NBL	Kathmandu	1000000	Shyam	L – 4	1500
NBL	Lalitpur	100000	Ram	L – 5	2500

Fig. Sample Lending Relation

The fig. above shows an instance of the relation lending (lending schema). A tuple t in the lending relation has the following intuitive meaning.

t [assets] is the asset figure for the branch named t[branchName]

t [branchCity] is the city in which the branch named t[branchName] is located.

t [loanNumber] is the number assigned to a loan made by the branch named t[branchName] to the customer named t[customerName]

t [amount] is the amount of loan whose number is t[loanNumber]

Suppose we wish to add a new loan to our database. Let the loan is made by RBB of Kathmandu to Sohan in the amount of 9000. So, in our design, we need a tuple with values on all the attribute of lending-schema. Thus, we must repeat the asset and city data for the RBB branch and must add the tuple with data as follows.

(RBB, Kathmandu, 1000000, Sohan, L – 6, 9000)

In general, the asset and city data for a branch must appear once for each loan made by that branch.

Repetition of information wastes space as well as complicates updating the database. Suppose the RBB branch at bhaktapur moves from bhaktapur to Kathmandu, then many related tuples of the lending relation need to the changed.

Another problem with the lending schema design is that we can not represent directly the information concerning a branch (branchName, branchcity, assets) unless there exists at least one loan at that branch. The problem is that tuples in the lending relation requires values for loanNumber, amount and customerName. Clearly, this situation is undesirable. So, the lending schema may be decomposed into the following two schemas:

branch CustomerSchema= (branchName, branchCity, assets, customerName)

CustomerLoan Schema = (customerName, loanNumber, amount).

We know that a bank branch is located in exactly one city. In other words, the functional dependency

branch Name $\rightarrow$ branchcity

holds on lending schema

There are four informal measures of quality for relation schema design. They are as follows.

i. Semantics of the attribute

ii. Reducing the redundant values in tuples

iii. Reducing the null values in tuples

iv. Disallowing the possibility of generating Spurious tuples.

Each relation can be interpreted as a set of facts or statements. This meaning or semantics, specifies how to interpret the attribute values stored in a tuple of relation. In other words, how the attribute values in a tuple relate to one another.

Our goal of Schema design is to minimize the storage space that the base relations occupy. Grouping attributes into relation schema has a significant effect on storage space. If there are redundant information in tuples, there may be update anomalies, insertion anomalies and deletion anomalies.

If many of the attributes don't apply to all tuples in the relation, we end up with many nulls in those tuples. This can waste space at the storage level and may also lead to problems with understanding the meaning of the attributes.

The tuples with wrong information, which is not valid, is called supurious tuples. Such tuples are generally generated as a result of applying join operations.

Functional Dependencies

A functional dependency is a constraint between two sets of attributes from the database. A functional dependency, denoted by $X \rightarrow Y$, between two sets of attributes X and Y that are subsets of R specifies a constraint on the possible tuples that can form a relation state r of R. The constraint is that, for any two tuples t1 and t2 in r that have t_1 [x] = t2[x], we must also have $t_1[y] = t_2[y]$. This means that the values of the Y component of a tuple in r depend on, or are determined by, the values of the X component or alternatively, the values of the X component of a tuple uniquely (or functionally) determine the values of the X component of a tuple uniquely (or functionally), determine the values of the Y component. We also say that there is a functional dependency from x to y or that y is functionally dependent on x. The abbreviation for functional dependency is FD or f.d. The set of attributes X is called the left hand side of the FD, and Y is called the right hand side.

Thus X functionally determines Y in a relation schema R if and only if, whenever two tuples of r(R) on their X-value, they must necessarily agree on their Y-value.

A functional dependency is a property of the semantics or meaning of the attributes. The database designers will use their understanding of the attributes. The database designers will use their understanding of the semantics of the attributes of R i.e. how they relate to one another to specify the functional dependencies that should hold on all relation states r of R. consider the following relations with functional dependencies.

Emp_DEPT

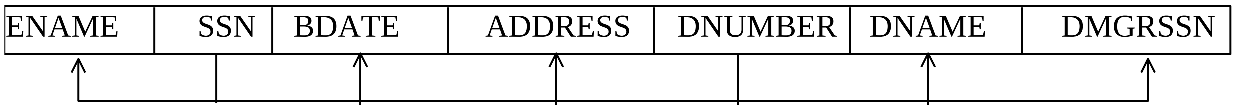


Fig. EMP_DEPT Relation Schema with their functional dependencies

EMP_PROJ

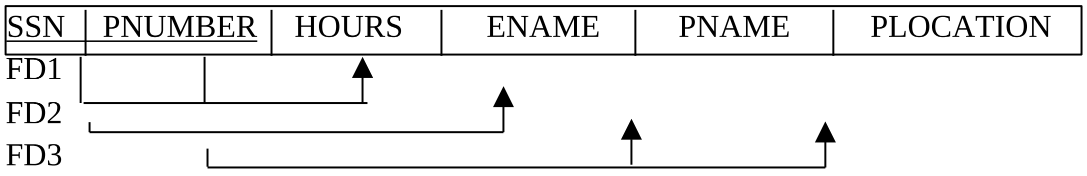


Fig. EMP_PROJ relation schema with their functional dependencies in EMP_PROJ, from the semantics of the attributes, we know that the following functional dependencies should hold:

i. SSN → ENAME

ii. PNUNBER →{ PNAME, PLOCATIONS}

iii. {SSN, PNUMBER} → HOURS

The functional dependency SSN → ENAME specify that the value of an employee's social security number(SSN) uniquely determines the employee name (ENAME)

Each FD is displayed as a horizontal line. The left hand side attributes of the FD are connected by vertical lines to the line representing the FD, while the right hand side attributes are connected by arrows pointing of the relation schema R, not of a particular legal relation state (extension) r of R.

Inference Rules For Functional Dependency

We denote by F the set of functional dependencies that are specified one relation schema R. Typically, the Schema designer specifies the functional dependencies that are semantically obvious, usually, however, numerous other functional dependencies hold in all legal relation instances that satisfy the dependencies in F. Those other dependencies can be inferred or deduced from the FDs in F. The set of all such dependencies is called the closure of F and is denoted by F^+. For example, suppose that we specify the following set F of obvious functional dependency on the relation schema EMP_DEPT.

F = {SSN→ ENAME, DATE, ADDRESS, PNUMBER},

DNUMBER → { DNAME, DMGRSSN}}

We can infer the following additional functional dependency from F;

SSN → { DNAME, DMGRSSN},

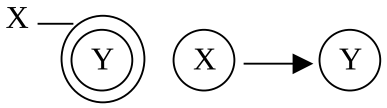
SSN → SSN

DNUMBER → DNAME

An FD X → Y is inferred from a set of dependencies F specified on R if X → Y holds in every relation state r that is a legal extension of R. i.e. whenever r satisfies all the dependencies in F, X→Y holds in r. There are a set of inference rules that can be used to infer new dependencies from a given set of dependencies.

We use the notation F⊧X → Y to denote that the functional dependency X → Y is inferred from the set of functional dependencies F. The following six rules are well known inference rules for functional dependencies.

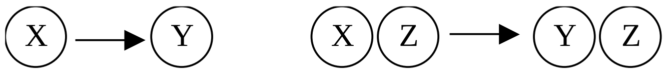
Reflexive Rule (IR1): If X ≥ Y, then X → Y



Augmentation rule (IR2): {X → Y} ⊧ X2 →Y2

$\{X_1, \ldots, X_m\} \to \{Y_1, \ldots, Y_m\}$ implies $\{X_1, \ldots X_m, Z\} \to \{Y_1, \ldots Y_n, Z\}$

e.g. {Name, sex} → { Name} implies { Name, age, sex} →{ Name, age}



Transitive Rule (IR3) :- {X → Y, y→Z} ⊧ X → 2

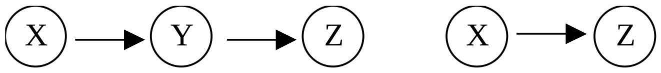
$\{X_1, \ldots X_m\} \to \{Y_1, \ldots Y_n\}$,

$\{Y_1, \ldots Y_n\} \to \{Z_1, \ldots Z_{sw}\}$ imply $\{X_1, \ldots X_m\} \to \{Z_1, \ldots Z_s\}$

e.g. {Number} →{Name}

{Name} →{Sex}

imply {Number} →{sex}



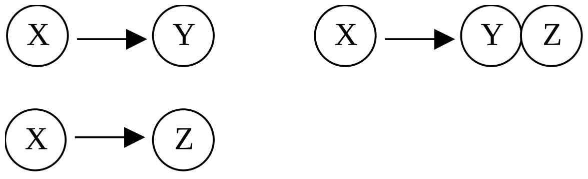
Decomposition or projective rule (IR4):- {X →YZ} ⊧ X →Y

$\{X_1, \ldots\ldots X_m\} \to \{Y_1, \ldots\ldots, Y_n, Z\}$ implies $\{X_1, \ldots\ldots, X_m\} \to \{Y_1, \ldots\ldots Y_n\}$ and $\{X_1, \ldots\ldots X_m\} \to Z$

e.g. Number → {Name, Age} implies number → Name

Union or Additive rule (IR5):→{X→Y, X→Z}⊧ X→YZ

$\{X_1, \ldots\ldots, X_m\} \to \{Y_1, \ldots\ldots, Y_n\}$ and $\{X_1, \ldots\ldots, X_m\} \to 2$ implies $\{X_1, \ldots\ldots, X_m\} \to \{Y_1, \ldots\ldots Y_n, Z\}$



Pseudo transitive rule (IR6): {X →Y, WY →Z} ⊧ W × →Z

The reflexive rule states that a set of attributes always determines itself or any of its subsets, which is obvious. Because IR1 generates dependencies that are always true, such dependencies are called trivial. Formally, a functional dependency X → Y is trivial if X ≥ Y; otherwise it is nontrivial.

Normalization

Normalization is the process of separating elements of data (such as names, addresses or skills) into affinity groups and defining the normal or right relationships between them. Normalization means putting things right and making them normal. The origin of the term is the Latin word norma, which was a carpenter's square that was used for assuming a right angle. In geometry, when a line is at a right angle to another line, it is said to be "normal" to it.

The normalization proceeds in a top down fashion by evaluating each relation against the criteria for normal forms and decomposing relations as necessary can thus be considered as relational design by analysis.

Normalization of data can be looked upon as a process of analyzing the given schemas based on their FDS and primary keys to achieve the desirable properties of minimizing redundancy and minimizing the insertion, deletion and update anomalies.

Initially, codd proposed three normal forms, which he called first, second and third normal forms. Later, a fourth normal form (4NF) and a fifth normal form (5Nf) were proposed. The normal form of a relation refers to the highest normal form condition that it meets, and hence indicates the degree to which it has been normalized. Sometimes the database designers need not normalize to the highest possible normal form. Relations may be left in a lower normalization status for performance reasons. The process of storing the join of higher normal form relations as a base relation – which is in a lower normal form, is called as denormalization.

Types of Normal Forms:

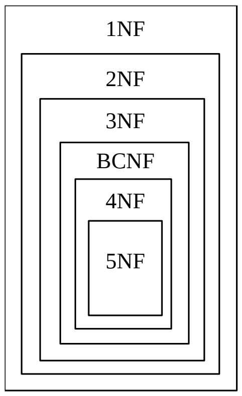
- Functional dependencies over primary keys:

- 1st normal form (1NF)

- 2nd normal form (2NF)

- 3rd normal form (3NF)

- Boyce-codd normal form (BCNF)



- Multivalued dependencies

- Fourth Normal form (4NF)

- Fifth Normal form (5NF)

First Normal Form (1NF)

A relation is in first normal form if there are only atomic values in attributes. This is done by moving data side separate relations (tables), where the data in each relation is of a similar type, and giving each table a primary key i.e. a unique label or identifier. A relation in 1NF is defined to disallow multivalued attributer, composite attributes, set of values.

Let us consider the department relation with data as follows:

DEPARTMENT

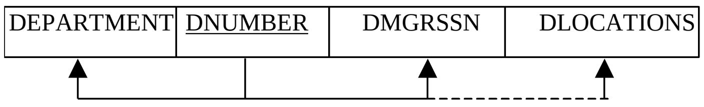


Fig.: Relation Department which is not in 1NF

DEPARTMENT

DNAME	DNUMBER	DMGRSSN	DLOCATIONS
IT	1	11111	{Baluwater, thapathali}
Finance	2	22222	{Baluwater}
Research	3	33333	{sanepa}

Fig.: Department instance

DEPARTMENT

DNAME	DNUMBER	DMGRSSN	DLOCATIONS
It	1	11111	Baluwater
It	1	11111	Thapathali
Finance	2	22222	Baluwater
Research	3	33333	Sanepa

Fig.: Department in 1NF with redundancy

There are three main techniques to achieve first normal form for the relation 'department'.

i) As Dlocation contains set of values and hence is nonatomic. Remove the attribute DLOCATIONS that violates INF and place it in a separate relation DEPT_LOCATIONS along with the primary key DNUMBER of DEPARTMENT. The primary key of this relation is the combination {DNUMBER, DLOCATIONS} as shown fig.

DEPARTMENT

DNAME	DNUMBER	DMGRSSN
It	1	11111
Finance	2	22222
Research	3	33333

Fig.: Relation in 1NF

DEPT_LOCATIONS

DNUMBER	DMGRSSN
1	Baluwater
1	Thapathali
2	Baluwater
3	Sanepa

ii) Expand the key so that there will be separate tuple in the original department relation for each location of a department as shown above in department in 1NF with redundancy. This solution has the disadvantage of introducing redundancy in the relation.

iii) If a maximum no. of values is known for the attribute, for example, if it is known that at most three locations can exist for a department, then replace the DLOCATIONS attribute by three atomic attributes: DLOCATION1, DLOCATION2 and DLOCATIONS3. This solution has the disadvantage of introducing null values if most departments have fewer than three locations. First solution is superior because it does not suffer from redundancy.

The 1NF also disallows multivalued attributes that are themselves composite. These are called nested relations because each tuple can have a relation within it as shown in fig.

EMP_PROJ

SSN	**ENAME**	**PROJS**	
		PNUMBER	HOURS

→

EMP_PROJ1

SSN	ENAME	

11111	Priya	1	10
		2	5.5
22222	Supriya	1	5
		3	10

EMP_PROJ2

SSN	PNUMBER	HOURS

Fig. decomposing EMP_PROJ into INF relations EMP_PROJ1 and EMP_PROJ2 with their primary keys

Such procedure can be applied recursively to a relation with multiple level nesting to unnest the relation into a set of 1NF relations.

Second Normal Form (2NF)

A relation schema R is in second normal form if every nonprime attribute A in R is not partially dependent on any key of R. An attribute that is not part of any candidate key is called non prime attribute.

2NF is based on the concept of full functional dependency. A functional dependency X → Y is a full functional dependency if removal of any attribute A from X means that the dependency does not hold any more i.e. for any attribute A ∈ X, (X – {A}) does not functionally determine Y. A functional dependency X → Y is a partial dependency if some attribute A ∈ X, (X –{A}) →Y. If we consider that EMP_PROJ relation, {SSN, PNUMBER} → HOURS is a full dependency because neither SSN→Hours nor PNUMBER →Hours holds. But the dependency {SSN, PNUMBER}→ENAME is partial because SSN→ENAME holds.

A relation schema R is in 2NF if every nonprime attributes A in R is fully functionally dependent on the primary key of R.

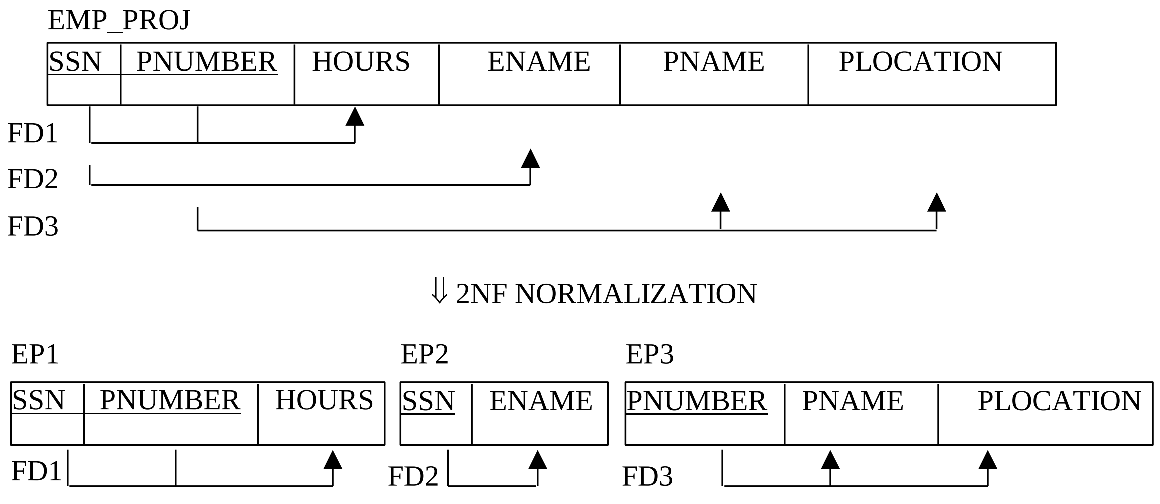


Fig. Normalizing EMP_PROJ into 2NF relations

The EMP_PROJ relation shown above is IN INF but is not in 2NF. The nonprime attribute ENAME violates 2NF because of FD2, as do the non prime attribute PNAME and PLOCATION because of FD3. The functional dependencies FD2 and FD3 make ENAE, PNAME and PLOCATION partially dependent the 2NF test. If a relation schema is not in 2NF, it can be second normalized into a number of 2NF relations in which nor prime attributes are associated only with the part of the primary key on which they are fully functionally dependent. The functional dependencies FD1, FD2 and FD3 lead to the decomposition of EMP_PROJ into three relation schemas EP1, EP2 and EP3 each of which is in 2NF.

Third Normal Form (3NF)

A relation schema R is in 3NF if it satisfies 2NF and no nonprime attribute of R is transitively dependent on the primary key. 3NF is based on the concept of transitive dependency. A functional dependency $X \rightarrow Y$ in a relation schema R is a transitive dependency if there is a set of attributes Z that is neither a candidate key nor a subset of any key of R_1 and both $X \rightarrow Z$ and $Z \rightarrow Y$ hold. Let us consider the following relation.

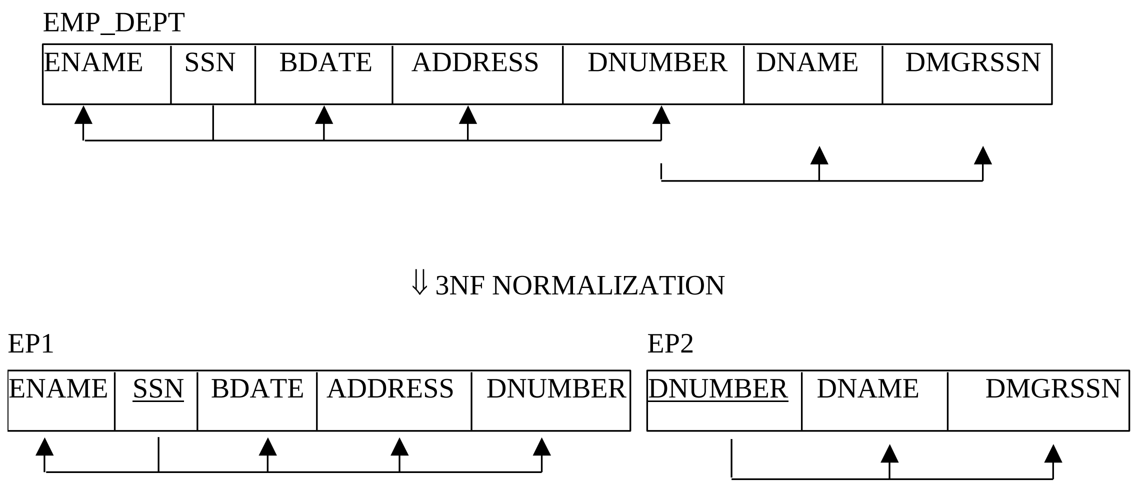
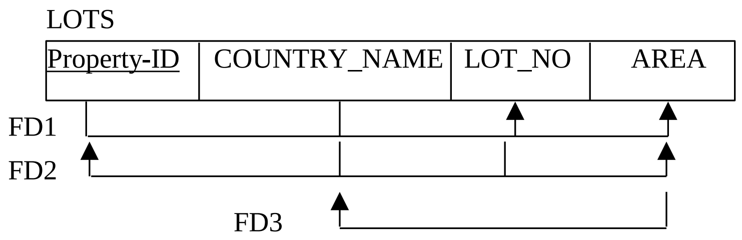


Fig. Normalizing EMP_DEPT INTO 3NF relations

The dependency SSN $\rightarrow$ DMGRSSN is transitive through DNUMBER because both the dependencies SSN $\rightarrow$ DNUMBER and DNUMBER $\rightarrow$ DMGRSSN hold and DNUMBER is neither a key itself nor a subset of the key of EMP_DEPT. Thus, we see that the dependency of DMGRSSN on DNUMBER is undesirable in EMP_DEPT since DNUMBER is not a key of EMP_DEPT. EMP_DEPT is in 2NF, since no partial dependencies on a key exist. However, EMP_DEPT is not in 3NF because of the transitive dependency of DMGRSSN (and also DNAME) on SSN via DNUMBER. We can normalize EMP_DEPT by decomposing it into the two 3NF relation schemas ED1 and ED2 as shown in fig. above.

Boyce – Codd Normal Form (BCNF)

A relation schema R is in BCNF if whenever a nontrivial functional dependency $X \rightarrow A$ holds in R, z then X is a super key of R. The main difference between BCNF and 3NF is that 3NF allows A to be Prime. BCNF is a stronger definition of 3NF which was proposed later by Boyce and Codd. Every relation in BCNF is also in 3NF, However, a relation in 3NF is not necessarily in BCNF. Let us consider the LOTS relation as follows.



⇓ BCNF NORMALIZATION

LOTS A

Property-ID	AREA	LOT_NO

LOTS B

AREA	COUNTRY_NAME

Fig. BCNF normalization with the dependency of FD2 being lost in the decompostion

Suppose there are thousands of lots from only two countries and the lost sizes in one country are 1, 2, 3, 4 and 5 ropanies and in other country are 6, 7, 8, 9 and 10 ropanies. The relation LOTS shown above is in 3NF because COUNTRY_NAME is a prime attribute. As the candidate keys are PROPERTY_ID, {COUNTRY_NAME, LOT_NO}.

The area of a lot determines the country as specified in FD3, can be represented by 10 tuples in a separate relation R(AREA, COUNTRY_NAME) since there are only 10 possible AREA values. This representation reduces the redundancy of repeating the same information in the thousands of LOTS tuples. BCNF is a stronger normal form that would disallow LOTS and suggest the need for decomposing it.

FD3 violates BCNF in LOTS because AREA is not a superkey of LOTS. FD3 satisfies 3NF in LOTS because COUNTRY_NAME is a prime attribute but this condition does not exist in the definition of BCNF. In practice, most relation schemas that are in 3NF are also in BCNF.

Multivalued Dependencies

Whenever two independent 1:N relationships A:B and A:C are mixed in the same relation, an multivalued dependencies (MVD) may arise. MVD are a consequence of 1NF which disallowed an attribute in a tuple to have a set of values i.e. multivalued attributes.

Let us consider the relation EMP as follows

EMP

ENAME	PNAME	DNAME
RAM	X	Laxman
RAM	Y	bharat
RAM	X	bharat
RAM	Y	Laxman

Fig. EMP relation with two MVDs:-
ENAME $\twoheadrightarrow$ PNAME and ENAME

A multivalued dependency (MVD) X $\twoheadrightarrow$ Y specified on relation schema R, where X and Y are both subsets of R, specifies the following constraint on any relation state r of R: if two tuples t1 and t2 exist in r such that t_1 [x]=t_2[x], then two tuples t_3 and t_4 should also exist in r with the following properties, where we use Z to denote (R-(X-Y))

- $t_3[x] = t_4[x] = t_1[x] = t_2 [x]$
- $t_3 [y] = t_1 [y]$ and $t_4[y] = t_2[y]$
- $t_3[z] = t_2[z]$ and $t_4[z] = t_1[z]$

Whenever $X \to \to Y$ holds, we say that X multi determines Y. Because of the symmetry in definition, whenever $X \to\to Y$ holds in R, so does $X \to \to Z$, hence $X\to\to Y$ implies $X \to\to Z$, and therefore it is sometimes written as $X\to\to Y | Z$.

In the fig. the MVDs ENAME$\twoheadrightarrow$PNAME and ENAME $\twoheadrightarrow$DNAME (or ENAME $\twoheadrightarrow$PNAME|DNAME) hold in EMP relation. The employee with ENAME 'RAM' works on projects with PNAME 'X' and 'Y' and has two dependents with DNAME 'Bharat' and 'Laxman'.

An MVD $X\twoheadrightarrow Y$ in R is called a trivial MVD if (a) Y is a subset of X, or (b) XUY=R. In fig. EMP has the trivial MVD ENAME $\twoheadrightarrow$PNAME. An MVD that satisfies neither (a) nor (b) is called a non trivial MVD.

Inference rules for multivalued dependencies:

Let us assume that all attributes are included in a "universal" relation schema $R=\{A_1, A_2, \ldots A_n\}$ and that X, Y, Z and W are subsets of R.

- Complementation rule: $\{X \twoheadrightarrow Y\} \models \{X\twoheadrightarrow (R-(XUY))\}$
- Augmentation rule: If $X\twoheadrightarrow Y$ and $W \geq Z$ then $WX\twoheadrightarrow YZ$
- Transitive rule: $\{X\twoheadrightarrow Y, Y\twoheadrightarrow Z\}\models X\twoheadrightarrow (Z-Y)$
- Replication rule for FD to MVD: $\{X\to Y\}\models X\twoheadrightarrow Y$

Fourth Normal Form (4NF)

A relation schema R is in 4NF with respect to a set of dependencies (F) (that includes functional dependencies and multivalued dependencies) if, for every nontrivial multivalued dependency $X\twoheadrightarrow Y$ in F^+, X is a superkey for R or $X \twoheadrightarrow Y$ is a trivial multivalued dependency. Let us consider the relation

EMP

ENAME	PNAME	PNAME
RAM	X	laxman
RAM	Y	bharat
RAM	X	bharat
RAM	Y	laxman

$\Rightarrow$
4NF

EMP_PROJECTS

ENAME	PNAME
RAM	X
RAM	Y

EMP_PROJECTS

ENAME	DNAME
RAM	Laxman
RAM	Bharat

Fig. EMP with ENAME $\twoheadrightarrow$ PNAME and ENAME $\twoheadrightarrow$ DNAME

Fig. Decomposing EMP into two relations in 4NF

The EMP relation is not in 4NF because in the nontrival MVDs ENAME$\twoheadrightarrow$PNAME and ENAME$\twoheadrightarrow$DNAME, ENAME is not a superkey of EMP. After decomposition of EMP into EMP_PROJECTS and EMP_DEPENDENTS, both are in 4NF because MVDs ENAME$\twoheadrightarrow$PNAME in EMP_PROJECTS and ENAME$\twoheadrightarrow$DNAME in EMP_DEPENDENTS are trivial MVDs. No other ontrivial MVDs hold in either EMP_PROJECTS or EMP_DEPENDENTS.

Lossless Join Decomposition into 4NF relations:

Whenever we decompose a relation schema R into R_1=(XUY) and R_2=(R-Y) based on an MVD $X\twoheadrightarrow Y$ that holds in R, the decomposition has the lossless join property. The lossless join (nonadditive join) property ensures that no spurious tuples are generated when a natural join operation is applied to the relations in the decomposition. The word loss in lossless refers to loss of information, not to loss of tuples. If a decomposition does not have the lossless join property, we may get additional spurious tuples after the PROJECT and NATURAL JOIN operations are applied.

Formally, a decomposition D={R_1,R_2, …,R_m) of R has the lossless (nonadditive) join property w.r.t. the set of dependencies F on R if, for every relation state r of R that satisfies F, the following holds, where * is the NATURAL JOIN of all the relations in D.

$*(\pi_{R1}(r_1), \ldots\ldots, \pi_{Rm}(r))=r$

Thus, we can say that the relation schemas R_1 and R_2 form a lossless join decomposition of R if and only if $(R_1 \cap R_2) \twoheadrightarrow (R_1\text{-}R_2)$ (or by symmetry, if and only if $(R_1 \cap R_2) \twoheadrightarrow (R_2 - R_1)$).

Algorithm: Relational decomposition into 4NF with lossless join property

Input: A universal relation R and a set of functional and multivalued dependencies F.

```
i)    set D = {R};
ii)   While there is a relation schema Q in D that is not in 4NF do
      {
      Choose a relation schema Q in D that is not in 4NF;
      Find a nontrivial MVD $X \rightarrow Y$ in Q that violates 4NF;
      Replace Q in D by two relation schemas (Q-Y) and (XUY);
      }
```

Join Dependencies and Fifth Normal Form (5NF)

The lossless join property give the condition for a relation schema R to be decomposed into two schemas R_1 and R_2, where the decomposition has the lossless join property. However, in some cases there may be no lossless join decomposition of R into two relation schemas but there may be lossless join decomposition into more than two relation schemas. We then resort to another dependency called the join dependency and if it is present, carry out a multiway decomposition into fifth normal form (5NF).

A join dependency (JD), denoted by JD(R_1, R_2,…… R_n) specified on relation schema R, specifies a constraint on the states r of R. The constraint states that every legal state r of R should have a lossless join decomposition into R_1, R_2, … R_n; i.e. for every such r we have

$*(\pi_{R1}(r), \pi_{R2}(r), \ldots, \pi_{rn}(r))=r$

It should be noted that an MVD is a special case of a JD where n=2. That is, a JD denoted as JD(R_1, R_2) implies an MVD $(R_1 \cap R_2 \twoheadrightarrow (R_1 - R_2)$. A join dependency JD($R_1$, R_2, … R_n). Specified on relation schema R, is a trivial JD if one of the relation schemas R_i in JD (R_1, R_2, … R_n) is equal to R.

A relation schema R is in fifth normal form (5NF) also called project join normal form (PJNF) with respect to a set F of functional, multivalued and join dependencies if (i) for every non-trivial join dependency JD (R_1, R_2, …, R_n) in F^+ (i.e. implied by F), every R_i is a superkey of R. (ii) *(R_1, R_2, …, R_n) is a trivial join dependency. Let us consider the relation supply with data as follows.

SUPPLY

SNAME	PARTNAME	PROJNAME
RAM	BOLT	PROJx
RAM	NUT	PROJy
SHYAM	BOLT	PROJy
MOHAN	NUT	PROJz
SHYAM	PIPE	PROJx
SHYAM	BOLT	PROJx
RAM	BOLT	PROJy

Fig. Relation supply with no MVDs satisfies 4NF but doesnot satisfy 5NF if the JD (R_1, R_2, R_3) holds

$\Rightarrow$
5NF

R_1

SNAME	PARTNAME
RAM	BOLT
RAM	NUT
SHYAM	BOLT
MOHAN	NUT
SHYAM	PIPE

R_2

SNAME	PROJNAME
RAM	PROJx
RAM	PROJy
SHYAM	PROJy
MOHAN	PROJz
SHYAM	PROJx

R_3

PARTNAME	PROJNAME
BOLT	PROJx
NUT	PROJy
BOLT	PROJy
NUT	PROJz
PIPE	PROJx

Fig. Decomposing Supply into three 5NF relations.

Suppose that in the SUPPLY relation, the constraint always hold is that whenever a supplier s supplies part p, and project j uses part p, and the supplier s supplies at least one part to project j, then suppliers will also be supplying part p to project j. This constraint can be restated in other ways and specifies a join dependency JD (R_1, R_2, R_3) among the three projections R_1 (SNAME, PARTNAME), R_2 (SNAME, PROJNAME), and R_3 (PARTNAME, PROJNAME) of SUPPLY. If this constraint holds, the tuples below the dotted lines in fig. must exist in any legal state of the SUPPLY relation that also contains the tuples above the dotted line. It should be noted that applying natural join to any two of these relations produce spurious tuples, but applying NATURAL JOIN to all three together does not produce spurious tuples.

Domain Key Normal Form (DKNF)

There is no hard and fast rule about defining normal forms only upto 5NF. Historically the process of normalization was carried through 5NF as a meaningful design activity, but it has been possible to define stricter normal forms that take into account additional types of dependencies and constrains. The idea behind DKNF is to specify the "ULTIMATE NORMAL FORM" that takes into account all possible types of dependencies and constraints.

A relation is said to be in DKNF if all constraints and dependencies that should hold on the relation can be enforced simply by enforcing the domain constraints and key constraints on the relation.

Sub: DBMS Batch : BIT IV Sem.

Database Security

The data stored in the database need to be protected from unauthorized access, malicious destruction and alteration of data. To protect the database, we must take security measures at several levels.

- ❖ Physical : The site or sites containing the computer systems must be physically secured against armed or surreptitious entry by intruders.
- ❖ Human: Users must be authorized carefully to reduce the chance of any such user giving access to an intruder in exchange for a bribe of other favours.
- ❖ Operating System: No matter how secure the database system is, Weakness in operating system security may serve as a means of unauthorized access to the database.
- ❖ Network: Since most all database systems allow remote access through terminals or networks, software level security within the network software is as important as physical security, both the internet and in networks private to an enterprise.
- ❖ Database System: Some database system users may be authorized to access only a limited portion of the database. Other users may be allowed to issue queries, but may be forbidden to modify the data.

SQL Access for database Security

Database Security and the DBA

The database administrator (DBA) is the central authority for managing a database system. The DBA's responsibilities include granting privileges to users who need to Use the system and classifying users and data in accordance with the policy of the organization. The DBA has a DBA account in the DBMS, sometimes called a system or superuser account, which provides powerful capabilities that are not made available to regular database accounts and users. DBA privileged commands include commands for granting and revoking privileges to individual accounts, users, or user groups and for performing the following types of actions:

1. *Account creation:* This action creates a new account and password for a user or a group of users to enable them to access the DBMS.
2. *Privilege granting:* This action permits the DBA to grant certain privileges to certain accounts.
3. *Privilege revocation:* This action permits the DBA to revoke (cancel) certain privileges that were previously given to certain accounts.
4. *Security level assignment:* This action consists of assigning user accounts to the appropriate security classification level.

The DBA is responsible for the overall security of the database system.

GRANT and REVOKE

The view mechanism allows the database to be conceptually divided up into pieces in various ways so that sensitive information can be hidden from unauthorized users. However, it does not allow for the specification of the operations that *authorized* users are allowed to execute against those pieces is performed by the GRANT statement.

Note first that the creator of any object is automatically granted all privileges that make sense for that object. For example, the creator of a base table *T* is automatically granted the SELECT, INSERT, UPDATE, DELETE, and REFERENCES privileges on *T*

The SQL commands used by DBA for security are as follows.

Creating user:

Create user supriya identified by s;

This sql commands creates user supriya whose password is s. The privileges which can be granted to supriya by DBA on any table employee (suppose the table employee is already created) are SELECT,INSERT,UPDATE,DELETE,INDEX,ALTER and REFERENCE.

The privileges can be granted by DBA to user supriya as follows.

Grant SELECT on employee to supriya;

Similarly INSERT,UPDATE,DELETE,INDEX,ALTER and REFERENCE can be granted to any user.

Grant all on employee to supriya;

Grant SELECT,UPDATE on employee to supriya;

The REFERENCE Privilege allows the grantee to create integrity constraints that reference that table.

Similarly the privileges can be revoked using revoke command as follows.

Revoke all on employee from supriya;

Revoke UPDATE on employee from supriya;

The current SQL standard supports discretionary access control only. Two more or less in-dependent SQL features are involved-the view mechanism, which can be used to hide sensitive data from unauthorized users, and the authorization subsystem itself, which allows users having specific privileges selectively and dynamically to grant those privileges to other users, and subsequently to revoke those privileges, if desired. Both features are discussed below.

Views and Security

To illustrate the use of views for security purposes in SQL:

```
CREATE VIEW LS AS
   SELECT S.S#, S.SNAME, S.STATUS S.CITY
   FROM S
   WHERE S,CITY = 'London' ;
```

The view defines the data over which authorization is to be granted. The granting itself is done by means of the GRANT statement--e.g.:

```
GRANT SELECT, UPDATE , DELETE
ON LS
TO Dan, Misha ;
```

ACCESS CONTROL

The access to the database is controlled by defining user to the database, assigning passwords to each user, assigning access privileges such as read, write, delete privileges, by physical access control such as secured entrances, password protected workstations, voice recognition technology etc. and by using DBMS utilities access control such as auditing and log file features. Some used access control methods are Discretionary and Mandatory control.

Discretionary Access Control Based on Granting/Revoking of Privileges

The typical method of enforcing discretionary access control in a database system is based on the granting and revoking of privileges. Let us consider privileges in the context of a relational DBMS.

Informally there are two levels for assigning privileges to use the database system.

1. The account level: At this level, the DBA specifies the particular privileges that each account holds independently of the relations in the database.
2. The relation(or table) level: At this level, we can control the privilege to access each individual relation or view in the database.

The privileges at the account levels include CREATE TABLE to create table, CREATE VIEW to create view, CREATE SYNONYM to create synonym and all the privileges are granted by DBA to individual user or account.

The privileges at the relation levels include SELECT, UPDATE, REFERENCES, DELETE for particular relations and are granted by DBA.

In SQL2, the DBA can assign an owner to a whole, schema by creating the schema and associating the appropriate authorization identifier with that schema using the CREATE SCHEMA command. The owner account holder can pass privileges on any of the owned relations to other users by granting privileges to their accounts. In SQL the following types of privileges can be granted on each individual relation R:

- SELECT (retrieval or read) privilege on R: Gives the account retrieval; privilege In SQL this gives the account the privilege to use the SELECT statement to retrieve, tuples from R.
- MODIFY privileges on R:. This gives the account the capability to modify tuples of R. In SQL this privilege is further divided into UPDATE, DELETE, and INSERT Privilege to apply the corresponding SQL command to R. In addition, both the INSERT and UPDATE privileges can specify that only certain attributes of R can be updated by the account.
- REFERENCES privilege on R: This gives the account the capability to reference relation R when specifying integrity constraints. This privilege can also be restricted to specific attributes of R.

Notice that to create a view the account must have SELECT privilege on all the involved in the view definition.

Specifying Privileges Using Views

The mechanism of views is an important discretionary authorization mechanism in its own right. For example, if the owner A of a relation R wants another account B to be able to retrieve only some fields of R, then A can create a view V of R that includes only that attributes and then grant SELECT on V to B. The same applies to limiting B to retrieving only certain tuples of R; a view V can be created by defining the view by means of a query that selects only those tuples from R that A wants to allow B to access.

Revoking Privileges

In some cases it is desirable to grant some privilege to a user temporarily. For example, the owner of a relation may want to grant the SELECT privilege to a user for a specific task and then revoke that privilege once the task is completed. Hence, a mechanism for revoking privileges is needed. In SQL a REVOKE command is included for the purpose of canceling privileges.

Propagation of Privileges Using the GRANT OPTION

Whenever the owner A of a relation R grants a privilege on R to another account B, the privilege can be given to B *with* or *without* the GRANT OPTION. If the GRANT OPTION is given, this means that B can also grant that privilege on R to other accounts.

Specifying Limits on Propagation of Privileges

Techniques to limit the propagation of privileges have been developed, although they have not yet been implemented in most DBMSs and are not a part of SQL. Limiting horizontal propagation to an integer number i means that an account B given the GRANT OPTION can grant the privilege to at most i other accounts. Vertical propagation is more complicated; it limits the depth of the granting of privileges. Granting a privilege with vertical propagation of zero is equivalent to granting the privilege with no GRANT OPTION. If account A grants a privilege to account B with vertical propagation set to an integer number $j > 0$, this means that the account B has the GRANT OPTION on that privilege, but B can grant the privilege to other accounts only with a vertical propagation less than j. In effect, vertical propagation limits the sequence of grant options that can be given from one account to the next based on a single original grant of the privilege.

Mandatory Access Control for Multilevel Security

The discretionary access control technique of granting and revoking privileges on relations has traditionally been the main security mechanism for relational database systems. This is an all-or-nothing method: a user either has or does not have a certain privilege. In many applications, an additional security policy is needed that classifies data and users based on security classes. This approach-known as mandatory access control-would typically be combined with the discretionary access control mechanisms. It is important to note that most commercial DBMSs currently provide mechanisms only for discretionary access control. However, the need for multilevel security exists in government, military, and intelligence applications, as well as in many industrial and corporate application.

Typical security classes are top secret (TS), secret (S), confidential (C), and unclassified (U), where TS is the highest level and U the lowest Other more complex security classification schemes exist, in which the security classes are organized in a lattice. For simplicity, four security classification levels, where $TS \geq S \geq C \geq U$ are used in the system. The commonly used model for multilevel security known as the Bell-LaPadula model, classifies each subject (user, account, program) and object (relation, tuple, column, view, operation) into one of the security classifications TS, S, C, or U. We will refer to the clearance (classification) of a subject S as class (S) and to the classification of an object O as class (O). Two restrictions are enforced on data access based on the subject/object classifications:

1. A subject S is not allowed read access to an object O unless $class(S) \geq class(O)$.

   This is known as the simple security property.

2. A subject S is not allowed to write an object O unless $class(S) \leq class(O)$. This is, known as the *property (or star property).

The first restriction is intuitive and enforces the obvious rule that no subject can read an object whose security classification is higher than the subject's security clearance. The second restriction is less intuitive. It prohibits a subject from writing an object at a lower security classification than the subject's security clearance. Violation of this rule would allow information to flow from higher to lower classifications, which violates a basic tenet of multilevel security. For example, a user (subject) with TS clearance may make a copy' of an object with classification TS and then write it back as a new object with classification U, thus making it visible throughout the system.

To incorporate multilevel security notions into the relational database model, it is common to consider attribute values and tuples as data objects. Hence, each attribute A is associated with a classification attribute C in the schema, and each attribute value in a tuple is associated with a corresponding security classification,- In addition, in some models, a tuple classification attribute TC is added to the relation attributes to provide a classification for each tuple as a whole. Hence, a multilevel relation schema R with n attributes would be represented as

$R(A_1, C_1, A_2, C_2, ..., A_n, C_n, TC)$

where each C_i represents the classification attribute associated with attribute A_i.

The value of the TC attribute in each tuple t-which is the highest of all attribute classification values within t-provides a general classification for the tuple itself, whereas each c_i provides a finer security classification for each attribute value within the tuple. The apparent key of a multilevel relation is the set of attributes that would have formed the primary key in a regular (single-level) relation. A multilevel relation will appear to contain different data to subjects (users) with different clearance levels. In some cases, it is possible to store a single tuple in the relation at a higher classification level and produce the corresponding tuples at a lower level classification through a process known as filtering. In other cases, it is necessary to store two or more tuples at different classification levels with the same value for the apparent key, This leads to the concept of polyinstantiation, where several tuples can have the same apparent key value but have different attribute values for users at different classification levels.

Assume that the Name attribute is the apparent key, and consider the query SELECT * FROM EMPLOYEE. A user with security clearance S would see the same relation shown below in fig.a, since all tuple classifications are less than or equal to S. However a user with security clearance C would not be allowed to see values for salary of Shyam and job performance for Ram as shown below in fig. b .

a. EMPLOYEE

Name		Salary		JobPerformance		TC
Ram	U	5000	C	Fair	S	S
Shyam	C	5000	S	Good	C	S

Fig. The original Employee tuples

b. EMPLOYEE

Name		Salary		JobPerformance		TC
Ram	U	5000	C	Null	C	C
Shyam	C	null	C	Good	C	C

Fig. Appearance of EMPLOYEE after filtering for classification C users

ENCRYPTION

Access control is only applied to the established avenues of access to the database. Clever people using clever instruments may be able to access the data by circumventing the controlled avenues of access. Also, innocent people who passively stumble upon an avenue of access may be unable to resist the temptation 'to look at and pet misuse the data so acquired. To counteract the possibility that either active or passive intruders obtain unauthorized access to sensitive data, it is desirable to obscure or hide the meaning of the data accessed.

Encryption is any sort of transformation applied to data (or text) prior to transmission or prior to storage, which makes it more difficult to extract information content or meaning. Decryption is method of retrieving the original message(text) from the encrypted message. The word 'cryptography' comes from the Greek meaning 'hidden or secret. Cryptography includes both encryption and decryption.

Encryption techniques complement access controls. Access controls are ineffective if

- A user leaves a listing in the work area or in the trash.
- Passwords are written down and found.
- Offline backup files are stolen.
- Confidential data is left in main memory after a job has completed.
- Someone taps in on a communication line.

When a data system' is geographically dispersed, physical security measures be come less practical and less effective against intrusion because the system is more open and vulnerable to penetration at more points. If the computer system and all the sensitive data are maintained in a single, isolated environment into which a user must be admitted before access to data is permitted, little need for encryption exists. An increased need for encryption comes with the increased tendency for systems to reach out into the using environment and become 'more available to the users.

The basic encryption scheme is shown in Figure . Original *plaintext* is transformed by an *encryption algorithm* using an *encryption key* to produce *ciphertext.* An inverse *decryption algorithm* transforms the ciphertext using the same (or related) key to reconstruct the plaintext.

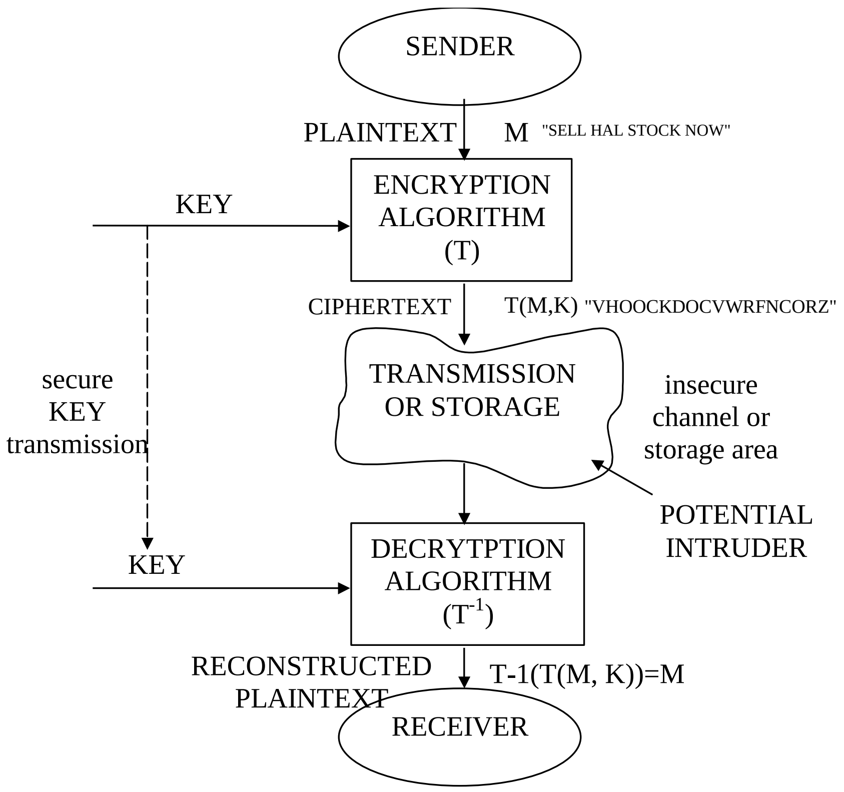


Figure Basic Encryption/ Decryption System.

An encryption algorithm (T). transforms a sender's plaintext message (M) using a key (K) to produce ciphertext. Plaintext'is in a form recognizable by humans (or computers). The encrypted message or data can be transmitted through an insecure channel or stored in an insecure area since a potential intruder would only see a scrambled message. Using the same (or related) key, the decryption algorithm applies an in years transformation (T^{-1}) to the ciphertext to reconstruct the original message (M). Transmission of the key to the decryption must be kept secure, since knowing the decryption key and the encryption algorithm makes it easy to decrypt or decipher a message, thereby disclosing sensitive information.

In a database environment, encryption techniques can be applied to:

- Transmitted data sent over communication lines between computer systems, or to and from remote terminals (1).
- Stored data:
  - remote backup data on removable media (2)
  - active data on secondary storage devices (3)
  - tables and buffers in internal main memory (4)

The figure below indicates the points where encryption can be used to protect data from unauthorized access during transmission between the user and the system. With a remote database, encryption can be used to protect the stored data from unauthorized access.

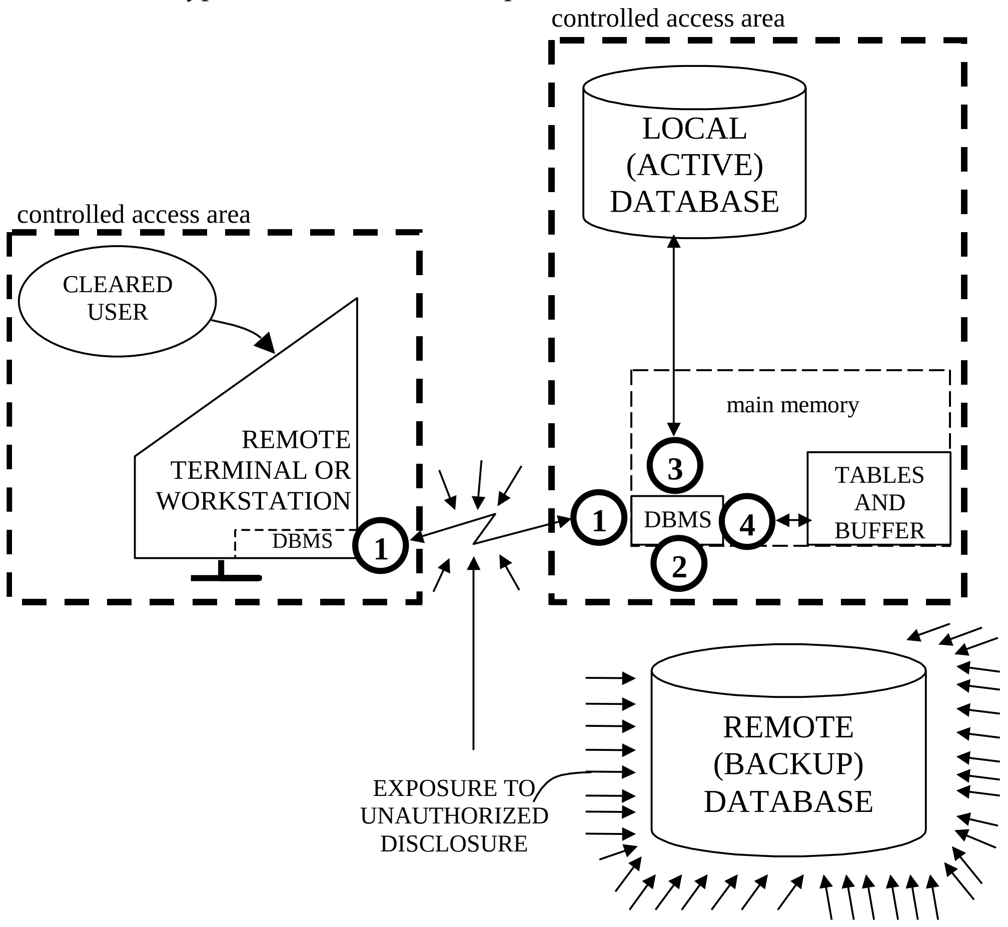


Fig Encryption in a Database Environment.

Encryption techniques can be used in the transmission or storage of data. The stored data may be remote backup data on removable media, active (updated) data on secondary storage devices, or tables and buffers in internal memory. Sensitive data outside of secured area is more exposed to unauthorized disclosure and therefore encryption can contribute significantly to greater security.

Encryption methods can be classified according to:

1. The nature of the algorithm:

- Transposition, or permutation in general.
- Substitution, either mono-alphabetic or polyalphabetic.
- Product, combining permutations and substitutions.

Cryptographic techniques have been widely used in military and government intelligence activities for centuries but due to the secretive nature of the subject.

The increased potential for more sophisticated encryption and cryptanalysis through the use of computers, coupled with the increased concern for data privacy, has generated a great surge of interest in the past decade or two. There has been substantial published literature on the use of encryption in computerized database systems. The U:S. Government has adopted a data encryption standard(DES).

Choosing an encryption method depends upon the cost, what is available, and the level of user need. The object of any particular method is to raise the work factor high enough to discourage anyone from breaking the code. The cost of the chosen method must be commensurate with the level of risk and the degree of security desired. Even fairly modest and simple encryption methods can render transmitted messages and stored data secure from all but the most persistent penetrators.

Traditional Methods: Transposition and Substitution

Historically, transposition and substitution have been the two major classes of encryption techniques. They were applied manually to encrypt streams of text prior to transmission. Permutations and substitutions serve as the basis for some present-day computerized methods.

Transposition techniques permute the ordering of characters in the data stream according to some rule. For example, if the transposition pattern or rule is to transpose each consecutive pair of characters, the phrase

database

would appear as

ADATABES

obviously not very secure. An example of a general permutation would be to form blocks of, say, four characters and permute, 1234 to 3124. The phrase would now appear as:

TDAASBAE

With a correct guess of the length of the permutation block, only a few trials are needed to break the code.

Substitution techniques retain the relative position of the characters in the original plaintext but hide their identity in the ciphertext. By contrast, transposition techniques retain the identity of the original characters but change their position.

A simple example is the *Caesar Cipher* which substitutes the nth letter away from ..the plaintext character in the alphabet. This is applied 'modulo 27' (includes a blank), that is, if you count past the end of the alphabet, cycle back to the beginning. The plaintext message in Fig. was encrypted using a Caesar Cipher. The n is the key indicating the alphabet shift.

A general mono-alphabetic substitution cipher replaces characters in the plaintext with characters from some other alphabet, called a cipher alphabet, which becomes the key. For example:

Plaintext alphabet:	abcdefghijklmnopqrstuvwxyz
Ciphertext alphabet:	GORDNCHALESZYXWVUTQPMKJIFB
Transforms the Plaintext:	database management system
Into the Ciphertext:	DGPGOGQN YGXGHNYNXP QFQPNYQ

Assuming that the plaintext message is written in natural English using 26 letters, mono-alphabetic substutions are susceptible to frequency analysis of single letters, letter pairs, and reversals. The analysis is increasingly accurate for larger messages the letter frequencies in the message would approach the 'characteristic letter frequencies for the language. In English the most frequently occurring letters are E,T,A,O,N,R,I,S,H. Some people can' even read such ciphertext directly Some computers have built-in machine instructions to perform substitutions between two alphabets (needed for A_CII-EBCDI_ code conversion), thus enabling faster exhaustive analysis of mono-alphabetic' substitutions.

Polyalphabetic substitution uses multiple alphabets cyclically according to some rule. Each character of the plaintext is replaced' with a character from a different Ciphertext alphabet, thereby obscuring the frequency characteristics of the characters in the plaintext alphabet.

Sub : DBMS Batch : BIT IV Sem.

Failures and Recovery

A computer system, like any other mechanical or electrical device, is subject to failure. There is a variety of causes of such failure, including disk crash, power failure, software error, a fire in the machine room, or even sabotage. In each of these cases, information may be lost. Therefore, the database system must take actions in advance to ensure that the atomicity and durability properties of transactions. An integral part of a database system is a recovery scheme that is responsible for the restoration of the database to a consistent state that existed prior to the occurrence of the failure.

Failure Classification

There are various types of failure that may occur in a system, each of which needs to be dealt with in a different manner. The simplest type of failure to deal with is one that does not result in the loss of information in the system. The failures that are more difficult to deal with are those that result in loss of information.

- **Transaction Failure(local failure):** This failure is caused by bad input and bugs in the application program.
- **System Failure:** This failure is caused by bugs in database management system code, operating system fault and hardware failure also. This failures occur less frequently and it results in loss of main memory.
- **Media failure:** This failure is caused by software errors, hardware errors, physical errors, head crashes etc. This failures results in loss of parts of secondary storage.
- **Site Failures:** This failure occurs due to failure of local CPU and results in a system crashing.
- **Network Failures:** This failure may occur in communication links.
- **Subotage Failures:** This failure occurs as result of intentional corruption of destruction of data, hardware or software facilities.

Database Recovery

Database recovery is the process of restoring the database to a correct state in the event of a failure. DBMS includes recovery facilities such as a backup mechanism, logging facilities and checkpoint facilities to enable updates to database that are in progress to be permanent.

System recovery

The system must be prepared to recover, not only from purely local failures such as the occurrence of an overflow condition within individual transaction, but also from “global” failures such as a power outage. A local failure, by definition, affects only the transaction in which the failure has actually occurred. A global failure, by contrast, affects all of the transactions in progress at the time of the failure, and hence has significant system-wide implications. Such failures fall into two broad categories:

- **System failures** (e.g., power outage), which affect all transactions currently in progress but do not physically damage the database. A system failure is sometimes called a softcrash.
- **Media failures** (e.g., head crash on the disk); which do cause damage to the database, or to some portion of it, and affect at least those transactions currently using the portion. A media failure is sometimes called a hard crash.

The key point regarding system failure is that the contents of main memory are lost (in particular, the database buffers are lost). The precise state of any transaction that was in progress at the time of the failure is therefore no longer known; such a transaction can therefore never be successfully completed, and so must be undone –i.e. Rolled back when the system restarts.

Furthermore, it might also be necessary to redo certain transactions at restart time that did successfully complete prior to the crash but did not manage to get the their updates transferred from the database buffers to the physical database.

Media recovery

A media failure in a failure such as a disk head crash, or a disk controller failure, in which some portion of the database has been physically destroyed. Recovery from such a failure basically involves reloading (or restoring) the database from a backup copy (or dump), and then using the log-both active and achieve portions, in general-to redo all transactions that completed since that backup copy was taken. There is no need to undo transactions that were still in progress at the time of the failure, since by definition all updates of such transactions have been "undone" (actually lost) anyway.

The need to be able to perform media recovery implies the need for a dump/restore (or unload/reload) utility. The dump portion of that utility is used to make backup copies of the database on demand. Such copies can be kept on tape or other archival storage; it is not necessary that they be on direct access media. After a media failure, the restore portion of the utility is sued to recreate the database from a specified backup copy.

Recovery and Atomicity

Consider simplified banking system and transaction T_i that transfers $50 from account A to account B, with initial values of A and B being $1000 and $2000, respectively. Suppose that a system crash has occurred during the execution of T_i, after output(B_A) has taken place, but before output(B_B) was executed, where B_A and B_B denote the buffer blocks on which A and B reside. Since the memory contents were lost, we do not know the fate of the transaction; thus, we could invoke one of two possible recovery procedures.

- **Reexecute Ti.** Thus procedure will result in the value of A becoming $900, rather than $950. Thus, the system enters an inconsistent state.
- **Do not reexecute Ti.** The current system state is values of $950 and $2000 for A and B, respectively. Thus, the system enters an inconsistent state.

In either case, the database is left in an inconsistent state, and thus this simple recovery scheme does not work. The reason for this difficulty is that we have modified the database without having assurance that the transaction will indeed commit. Our goal is to perform either all or no database modifications made by T_i. However, if T_i performed multiple database modifications, several output operations may be required, and a failure may occur after some of these modifications have been made, but before all of them are made.

Log-Based Recovery

The most widely used structure for recording database modifications is the log. The log is a sequence of log records, and maintains a record of all the update activities in the database. There are several types of log records. An update long record describes a single database write, and has the following fields:

- **Transaction identifier** is the unique identifier of the transaction that performed the write operation.
- **Data-item identifier** is the unique identifier of the data item written. Typically, it is the location on disk of the data item.
- **Old value** is the value of the data item prior to the write.
- **New value** is the value that the data item will have after the write.

Other special log records exist to record significant events during transaction processing, such as the start of a transaction and the commit or abort of a transaction. We denote the various types of log records as follows:

- <T_i start>. Transaction T_i has started.
- <T_i, X_j, V_1, V_2>. Transaction T_i has performed a write on data item X_j. X_j had value V_1 before the write, and will have value V_2 after the write.
- <T_i commit>. Transaction T_i has committed.
- <T_i abort>. Transaction T_i has aborted.

Whenever a transaction performs a write, it is essential that the log record for that write be created before the database is modified. Once a log record exists, we can output the modification to the database if that is desirable. Also, we have the ability to undo a modification that has already been output to the database. We undo it by using the old-value field in log records.

For log records to be useful for recovery from system and disk failures, the log must reside in stable storage. It is assumed that every log record is written to the end of the log on stable storage as soon as it is created. The log contains a complete record of all database activity. The recovery schemes uses two

Log	**Database**
<T_0 start>	
<T_0, A, 1000, 950>	
<T_0, B, 2000, 2050>	
	A = 950
	B = 2050
<T_0 commit>	
<T_1 start>	
<T_1, C, 700, 600>	
	C = 600
<T_1 commit>	

Fig. State of system log and database corresponding to T_0 and T_1.

recovery procedures:

- **undo**(T_i) restores the value of all data items updated by transaction T_i to the old values.

- **redo**(T_i) sets the value of all data items updated by transaction T_i to the new values.

The set of data items updated T_i and their respective old and new values can be found in the log.

The undo and redo operations must be idempotent to guarantee correct behavior even if a failure occurs during the recovery process.

After a failure has occurred, the recovery scheme consults the long to determine which transactions need to be redone, and which need to be undone. This classification of transactions is accomplished as follows:

- Transaction T_i needs to be undone if the log contains the record <T_i **start**>, but does not contain the record <T_i **commit**>.
- Transaction T_i needs to be redone if the log contains both the record <T_i **start**> and the record <T_i **commit**>.

As an illustration, let us return to out banking example, with transaction T_0 and T_1 executed one after the other in the order T_0 followed by T_1. Let us suppose that the system crashes before the completion of the transactions. We shall consider three cases. The state of the logs for each of these cases is shown Fig.

First, let us assume that the crash occurs just after the log record for the step

write(B)

of transaction T_0 has been written to stable storage. When the system comes back up, it finds the record <T_0 start> in the log, but no corresponding <T_0 commit> record. Thus, transaction T_0 must be undone, so an undo(T_0) is

<T_0 start>	<T_0 start>	<T_0 start>
<T_0 A, 1000, 950>	<T_0, A, 1000, 950>	<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>
	<T_0 commit>	<T_0 commit>
	<T_1 start>	<T_1 start>
	<T_1, C, 700, 600>	<T_1, C, 700, 600>
		<T_1 commit>
(a)	(b)	(c)

Figure The same log, shown at three different times.

performed. As a result, the values in accounts A and B (on the disk) are restored to \$1000 and \$2000, respectively.

Next, let us the assume that the crash comes just after the log record for the step

write(C)

of transaction T_1 has been written to stable storage (As shown in fig.b). When the system comes back up, two recovery actions need to be taken. The operation undo(T_1) must be performed, since the record <T_1 start> appears in the log, but there is not record (T_1 commit>. The operation redo (T_0) must be performed, since the log contains both the record <T_0 start> and the record <T_0 commit>. At the end of the entire recovery procedure, the values of accounts A, B, and C are \$950, \$2050, and \$700, respectively. Note that the undo(T_1) operation is performed before the redo (T_0). In this example, the same outcome would result if the order were reversed.

Finally, let us assume that the crash occurs just after the log record.

<T_1 commit>

has been written to stable storage (Figc). When the system comes back up, both T_0 and T_1 need to be redone, since the records <T_0 start> and <T_0 commit> appear in the log as do the records <T_1 start> and <T_1 commit>. After the recovery procedures redo(T_0) and redo(T_1) are performed, the values in accounts A, B, C are \$950, \$2050 and \$600, respectively.

Checkpoints

When a system failure occurs, we must consult the log to determine those transactions that need to be redone and those that need to be undone. In principle, we need to search the entire log to determine this information. There are two major difficulties with this approach:

1. The search process is time-consuming.
2. Most of the transactions that, according to out algorithm, need to be redone have already written their updates into the database. Although redoing them will cause no harm, it will nevertheless cause recovery to take longer.

To reduce these types of overhead checkpoint is introduced. In addition, the system periodically performs checkpoints, which require the following sequence of actions to take place:

1. Output onto stable storage all log records currently residing in main memory.
2. Output to the disk all modified buffer blocks.
3. Output onto stable storage a log record <checkpoint>.

Transactions are not allowed to perform any update actions, such as writing to a buffer block or writing a log record, while a checkpoint is in progress.

The presence of a <checkpoint> record in the log allows the system to streamline its recovery procedure. Consider a transaction T_i that committed prior to the checkpoint the <checkpoint> record. Any database modifications made by T_i must have been written to the database either prior to the checkpoint or as part of the checkpoint itself. Thus, at recovery time, there is no need to perform a redo operations on T_i.

This observation allows us to refine our previous recovery schemes. (We continue to assume that transactions are run serially). After a failure has occurred, the recovery scheme examines the log to determine the most recent transaction T_i that started executing before the most recent checkpoint took place. It can find such a transaction by searching the log backward, from the end of the log, until it finds the first <checkpoint> record (since we are searching backward, the record found is the final<checkpoint> record in the log); then it continues the search backward until it finds the next <T_i start> record. This record identifies a transaction T_i.

Once transaction T_i has been identified, the redo and undo operations need to be applied to only transaction T_i and all the transaction T_j that started executing after transaction T_i. Let us denote these transactions by the set T. The remainder (earlier part) of the log can be ignored, and can be erased whenever desired. The exact recovery operations to be performed depend on whether the immediate-modification technique or the deferred-modification technique is being used. The recovery operations that are required if the immediate-modification technique is employed are as follows:

- For all transactions t_K in T that have no <T_k commit> record in the log, execute undo(T_0).
- For all transactions T_k in T such that the record <T_k commit> appears in the log, execute redo(T_k).

Obviously, the undo operation does not need to be applied when the deferred-modification technique is being employed.

As an illustration, consider the set of transactions {T_0, T_1 … T_{100}} executed in the order of the subscripts. Suppose that the most recent checkpoint took place during the execution of transaction T_{67}. Thus, only transactions T_{67}, T_{68}, …, T_{100} need to be considered during the recovery scheme. Each of them needs to be redone if it has committed; otherwise, it needs to be undone.

Shadow Paging

An alternative to log-based crash-recovery techniques is shadow paging. Under certain circumstances, shadow paging may require fewer disk accesses than do the log-based methods. There are, however, disadvantages to the shadow-paging approach. For example, it is hard to extend shadow paging to allow multiple transactions to execute concurrently.

As before, the database is partitioned into some number of fixed-length blocks, which are referred to as pages. The term page is borrowed form operating systems, since we are using a paging scheme for memory management. Let us assume that there are n pages, numbered 1 through n. (In practice, n may be in the hundreds of thousands.) These pages do not need not to be stored in any particular order on disk. However, there must be a way to find the ith page of the database for any given i. The first entry contains a pointer to the first page of the database, the second entry points to the second page, and so on.

The key idea behind the shadow-paging technique is to maintain two page tables during the life of a transaction: the current page table and the shadow page table. When the transaction starts, both page tables are identical. The shadow page table is never changed over the duration of the transaction. The current page table may be changed when a transaction performs a write operation. All input and output operations use the current page table to locate database pages on disk.

Failures and Recovery

A computer system, like any other mechanical or electrical device, is subject to failure. There is a variety of causes of such failure, including disk crash, power failure, software error, a fire in the machine room, or even sabotage. In each of these cases, information may be lost. Therefore, the database system must take actions in advance to ensure that the atomicity and durability properties of transactions. An integral part of a database system is a recovery scheme that is responsible for the restoration of the database to a consistent state that existed prior to the occurrence of the failure.

Failure Classification

There are various types of failure that may occur in a system, each of which needs to be dealt with in a different manner. The simplest type of failure to deal with is one that does not result in the loss of information in the system. The failures that are more difficult to deal with are those that result in loss of information.

- **Transaction Failure(local failure):** This failure is caused by bad input and bugs in the application program.
- **System Failure:** This failure is caused by bugs in database management system code, operating system fault and hardware failure also. This failures occur less frequently and it results in loss of main memory.
- **Media failure:** This failure is caused by software errors, hardware errors, physical errors, head crashes etc. This failures results in loss of parts of secondary storage.
- **Site Failures:** This failure occurs due to failure of local CPU and results in a system crashing.
- **Network Failures:** This failure may occur in communication links.

- **Subotage Failures:** This failure occurs as result of intentional corruption of destruction of data, hardware or software facilities.

Database Recovery

Database recovery is the process of restoring the database to a correct state in the event of a failure. DBMS includes recovery facilities such as a backup mechanism, logging facilities and checkpoint facilities to enable updates to database that are in progress to be permanent.

System recovery

The system must be prepared to recover, not only from purely local failures such as the occurrence of an overflow condition within individual transaction, but also from "global" failures such as a power outage. A local failure, by definition, affects only the transaction in which the failure has actually occurred. A global failure, by contrast, affects all of the transactions in progress at the time of the failure, and hence has significant system-wide implications. Such failures fall into two broad categories:

- **System failures** (e.g., power outage), which affect all transactions currently in progress but do not physically damage the database. A system failure is sometimes called a softcrash.
- **Media failures** (e.g., head crash on the disk); which do cause damage to the database, or to some portion of it, and affect at least those transactions currently using the portion. A media failure is sometimes called a hard crash.

The key point regarding system failure is that the contents of main memory are lost (in particular, the database buffers are lost). The precise state of any transaction that was in progress at the time of the failure is therefore no longer known; such a transaction can therefore never be successfully completed, and so must be undone –i.e. Rolled back when the system restarts.

Furthermore, it might also be necessary to redo certain transactions at restart time that did successfully complete prior to the crash but did not manage to get the their updates transferred from the database buffers to the physical database.

At certain prescribed in travels-typically whenever some prescribed number of entries have been written to the long-the system automatically takes a checkpoint. Taking a checkpoint involves (a) physically writing ("force-writing") the contents of the database buffers out to the physical database, and (b) physically writing as special checkpoint record out to the physical log. The checkpoint record gives a list of all transactions that were in progress at the time the checkpoint was taken.

A system failure has occurred at time tf.

- The most recent checkpoint prior to time tf was taken at time tc.
- Transactions of type T1 completed (successfully)prior to time tc.
- Transactions of type T2 started prior to time tc and completed (successfully) aftertime tc and before time tf.
- Transactions of type T3 also started prior to time tc but did not complete by time tf.
- Transaction of type T4 started after time tc and completed (successfully) before time tf.
- Finally transactions of type T5 also started after time tc but did not complete by time tf.

Fig 14.3

It should be clear that when the system is restarted, transactions of type T3 and T5 must be undone, and transactions of types T2 and T4 must be redone. However, that transactions of type T1 do not enter into the restart process at all, because their updates were forced to the database at time tc as part of the checkpoint process. Note too that transactions that completed unsuccessfully (i.e., with a rollback) before time tf also do not enter into the restart process at all.

At restart time, therefore, the system first goes through the following procedure in order to identify al transactions of types T2-T5

1.Start with two lists of transactions, the UNDO list and REDO list. Set the UNDO list equal to the list of all transaction given in the most recent checkpoint record: set the REDO list to empty.

2.Search forward through the log, starting from the checkpoint record.

3.If a BEGIN TRANSACTION log entry is found for transaction T, and T to the UNDO list.

4. If a COMMIT log entry is found for transaction T, move T from the UNDO list to the REDO list.

5.When the end of the log is reached, the UNDO and REDO lists identify, respectively, transactions of types T3 and T5 and transactions of types T3 and T5 and transactions of types T2 and T4.

The system now works backward through the log, undoing the transactions in the UNDO list; then it works forward again, redoing the transactions in the REDO list. It should be noted that Restoring the database to a consistent state by undoing work is sometimes called backward recovery. Similarly, restoring it to a consistent state by redoing work is sometimes called forward recovery.

Finally, when all such recovery activity is complete, then (and only then) the system is ready to accept new work.

Media recovery

A media failure in a failure such as a disk head crash, or a disk controller failure, in which some portion of the database has been physically destroyed. Recovery from such a failure basically involves reloading (or restoring) the database from a backup copy (or dump), and then using the log-both active and achieve portions, in general-to redo all transactions that completed since that backup copy was taken. There is no need to undo transactions that were still in progress at the time of the failure, since by definition all updates of such transactions have been "undone" (actually lost) anyway.

The need to be able to perform media recovery implies the need for a dump/restore (or unload/reload) utility. The dump portion of that utility is used to make backup copies of the database on demand. Such copies can be kept on tape or other archival storage; it is not necessary that they be on direct access media. After a media failure, the restore portion of the utility is sued to recreate the database from a specified backup copy.

Two –phase commit

Two-phase commit is important whenever a given transaction can interact with several independent "resource manager", each managing its own set of recoverable resources and maintaining its own recovery log. For example, consider a transaction running on an IBM mainframe that updates both an IMS database and a DB2 database (such a transaction is perfectly legal, by the way). If the transaction completes successfully, the all of its updates, to both IMS data and DB2 data, must be committed; conversely, if it fails, then all of its updates must be rolled back. In other words, it must not be possible for the IMS updates to be committed and the DB2 updates rolled back, or vice versa – for then the transaction would no longer be atomic.

It follows that it does not make sense for the transaction to issue, say, a COMMIT to IMS and a ROLLBACK to DB2; and even if it issued the same instruction to both, the system could still crash in between the two, with unfortunate results. Instead, therefore, the transaction issues a single system-wide COMMIT (or ROLLBACK). That "global" COMMIT or ROLLBACK is handled by a system component called the coordinator, whose task it is to guarantee that both resource managers (i.e., IMS and DB2 in the example) commit or toll back the updates they are responsible for in unison and furthermore to provide that guarantee even if the system fails in the middle of the process. And it is the two-phase commit protocol that enables the coordinator to provide such a guarantee.

Assume for simplicity that the transaction has completed its database processing successfully, so that the system-wide instruction it issues is COMMIT, not ROLLBACK. On receiving the COMMIT request, the coordinator goes through the following two-phase process:

1. First, it instructs all resource managers to get ready to "go either way" on the transaction. In practice, this means that each participant in the process – i.e., each resource manager involved-must force all log entries for local resources used by the transaction out to its own physical log (i.e., out to nonvolatile storage; whatever happens there after, the resource manager will now have a permanent record of the work it did on behalf of the transaction, and so will able to commit its updates or roll them back, as necessary). Assuming the forced write is successful, the resource manager now replies "OK" to the coordinator, other wise it replies "Not OK".

2. When the coordinator has received replies from all participants, it forces an entry to its own physical log, recording its decision regarding the transaction. If all replies were "OK". That decision is "commit", if any reply was "Not OK", the decision is "rollback". Either way, the coordinator then informs each participant of its decision, and each participant must then commit or roll back the transaction locally, as instructed. Note that each participant must do what it is told by the coordinator in Phase 2 – that is the protocol. Note too that it is the appearance of the decision record in the coordinator's physical log that marks the transaction from Phase 1 to Phase 2.

Now, if the system fails at some point during the overall process, the restart procedure will look for the decision record in the coordinator's log. If it find it, then the two-phase commit process can pick up where it left off. It is does not find it then it assumes that the decision was "rollback", and again the process can complete appropriately. Note: it is worth pointing out that if the coordinator and the participants are executing on different machines, as they might be in a distributed system , then a failure on the part of the coordinator might keep some participant waiting a long time for the coordinator's decision and, so long as it is waiting, any updates made by the transaction via that participant must be kept hidden from other transactions i.e., those updates will probably have to be kept locked.

Recovery and Atomicity

Consider simplified banking system and transaction T_i that transfers \$50 from account A to account B, with initial values of A and B being \$1000 and \$2000, respectively. Suppose that a system crash has occurred during the execution of T_i, after output(B_A) has taken place, but before output(B_B) was executed, where B_A and B_B denote the buffer blocks on which A and B reside. Since the memory contents were lost, we do not know the fate of the transaction; thus, we could invoke one of two possible recovery procedures.

- **Reexecute Ti.** Thus procedure will result in the value of A becoming \$900, rather than \$950. Thus, the system enters an inconsistent state.
- **Do not reexecute Ti.** The current system state is values of \$950 and \$2000 for A and B, respectively. Thus, the system enters an inconsistent state.

In either case, the database is left in an inconsistent state, and thus this simple recovery scheme does not work. The reason for this difficulty is that we have modified the database without having assurance that the transaction will indeed commit. Our goal is to perform either all or no database modifications made by T_i. However, if T_i performed multiple database modifications, several output operations may be required, and a failure may occur after some of these modifications have been made, but before all of them are made.

Log-Based Recovery

The most widely used structure for recording database modifications is the log. The log is a sequence of log records, and maintains a record of all the update activities in the database. There are several types of log records. An update long record describes a single database write, and has the following fields:

- **Transaction identifier** is the unique identifier of the transaction that performed the write operation.
- **Data-item identifier** is the unique identifier of the data item written. Typically, it is the location on disk of the data item.
- **Old value** is the value of the data item prior to the write.
- **New value** is the value that the data item will have after the write.

Other special log records exist to record significant events during transaction processing, such as the start of a transaction and the commit or abort of a transaction. We denote the various types of log records as follows:

- <T_i start>. Transaction T_i has started.
- <T_i, X_j, V_1, V_2>. Transaction T_i has performed a write on data item X_j. X_j had value V_1 before the write, and will have value V_2 after the write.
- <T_i commit>. Transaction T_i has committed.
- <T_i abort>. Transaction T_i has aborted.

Whenever a transaction performs a write, it is essential that the log record for that write be created before the database is modified. Once a log record exists, we can output the modification to the database if that is desirable. Also, we have the ability to undo a modification that has already been output to the database. We undo it by using the old-value field in log records.

For log records to be useful for recovery from system and disk failures, the log must reside in stable storage. It is assumed that every log record is written to the end of the log on stable storage as soon as it is created. The log contains a complete record of all database activity. The recovery schemes uses two

Log	**Database**
<T_0 start>	
<T_0, A, 1000, 950>	
<T_0, B, 2000, 2050>	
	A = 950
	B = 2050
<T_0 commit>	
<T_1 start>	
<T_1, C, 700, 600>	
	C = 600
<T_1 commit>	

Fig. State of system log and database corresponding to T_0 and T_1.

recovery procedures:

- **undo**(T_i) restores the value of all data items updated by transaction T_i to the old values.
- **redo**(T_i) sets the value of all data items updated by transaction T_i to the new values.

The set of data items updated T_i and their respective old and new values can be found in the log.

The undo and redo operations must be idempotent to guarantee correct behavior even if a failure occurs during the recovery process.

After a failure has occurred, the recovery scheme consults the long to determine which transactions need to be redone, and which need to be undone. This classification of transactions is accomplished as follows:

- Transaction T_i needs to be undone if the log contains the record <T_i **start**>, but does not contain the record <T_i **commit**>.
- Transaction T_i needs to be redone if the log contains both the record <T_i **start**> and the record <T_i **commit**>.

As an illustration, let us return to out banking example, with transaction T_0 and T_1 executed one after the other in the order T_0 followed by T_1. Let us suppose that the system crashes before the completion of the transactions. We shall consider three cases. The state of the logs for each of these cases is shown Fig.

First, let us assume that the crash occurs just after the log record for the step

write(B)

of transaction T_0 has been written to stable storage. When the system comes back up, it finds the record <T_0 start> in the log, but no corresponding <T_0 commit> record. Thus, transaction T_0 must be undone, so an undo(T_0) is

<T_0 start>	<T_0 start>	<T_0 start>
<T_0 A, 1000, 950>	<T_0, A, 1000, 950>	<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>
	<T_0 commit>	<T_0 commit>
	<T_1 start>	<T_1 start>
	<T_1, C, 700, 600>	<T_1, C, 700, 600>
		<T_1 commit>
(a)	(b)	(c)

Figure The same log, shown at three different times.

performed. As a result, the values in accounts A and B (on the disk) are restored to $1000 and $2000, respectively.

Next, let us the assume that the crash comes just after the log record for the step

write(C)

of transaction T_1 has been written to stable storage (As shown in fig.b). When the system comes back up, two recovery actions need to be taken. The operation undo(T_1) must be performed, since the record <T_1 start> appears in the log, but there is not record (T_1 commit>. The operation redo (T_0) must be performed, since the log contains both the record <T_0 start> and the record <T_0 commit>. At the end of the entire recovery procedure, the values of accounts A, B, and C are $950, $2050, and $700, respectively. Note that the undo(T_1) operation is performed before the redo (T_0). In this example, the same outcome would result if the order were reversed.

Finally, let us assume that the crash occurs just after the log record.

<T_1 commit>

has been written to stable storage (Figc). When the system comes back up, both T_0 and T_1 need to be redone, since the records <T_0 start> and <T_0 commit> appear in the log as do the records <T_1 start> and <T_1 commit>. After the recovery procedures redo(T_0) and redo(T_1) are performed, the values in accounts A, B, C are $950, $2050 and $600, respectively.

Checkpoints

When a system failure occurs, we must consult the log to determine those transactions that need to be redone and those that need to be undone. In principle, we need to search the entire log to determine this information. There are two major difficulties with this approach:

1. The search process is time-consuming.
2. Most of the transactions that, according to out algorithm, need to be redone have already written their updates into the database. Although redoing them will cause no harm, it will nevertheless cause recovery to take longer.

To reduce these types of overhead checkpoint is introduced. In addition, the system periodically performs checkpoints, which require the following sequence of actions to take place:

1. Output onto stable storage all log records currently residing in main memory.
2. Output to the disk all modified buffer blocks.
3. Output onto stable storage a log record <checkpoint>.

Transactions are not allowed to perform any update actions, such as writing to a buffer block or writing a log record, while a checkpoint is in progress.

The presence of a <checkpoint> record in the log allows the system to streamline its recovery procedure. Consider a transaction T_i that committed prior to the checkpoint the <checkpoint> record. Any database modifications made by T_i must have been written to the database either prior to the checkpoint or as part of the checkpoint itself. Thus, at recovery time, there is no need to perform a redo operations on T_i.

This observation allows us to refine our previous recovery schemes. (We continue to assume that transactions are run serially). After a failure has occurred, the recovery scheme examines the log to determine the most recent transaction T_i that started executing before the most recent checkpoint took place. It can find such a transaction by searching the log backward, from the end of the log, until it finds the first <checkpoint> record (since we are searching backward, the record found is the final<checkpoint> record in the log); then it continues the search backward until it finds the next <T_i start> record. This record identifies a transaction T_i.

Once transaction T_i has been identified, the redo and undo operations need to be applied to only transaction T_i and all the transaction T_j that started executing after transaction T_i. Let us denote these transactions by the set T. The remainder (earlier part) of the log can be ignored, and can be erased whenever desired. The exact recovery operations to be performed depend on whether the immediate-modification technique or the deferred-modification technique is being used. The recovery operations that are required if the immediate-modification technique is employed are as follows:

- For all transactions t_K in T that have no <T_k commit> record in the log, execute undo(T_0).
- For all transactions T_k in T such that the record <T_k commit> appears in the log, execute redo(T_k).

Obviously, the undo operation does not need to be applied when the deferred-modification technique is being employed.

As an illustration, consider the set of transactions {T_0, T_1 … T_{100}} executed in the order of the subscripts. Suppose that the most recent checkpoint took place during the execution of transaction T_{67}. Thus, only transactions T_{67}, T_{68}, …, T_{100} need to be considered during the recovery scheme. Each of them needs to be redone if it has committed; otherwise, it needs to be undone.

Shadow Paging

An alternative to log-based crash-recovery techniques is shadow paging. Under certain circumstances, shadow paging may require fewer disk accesses than do the log-based methods. There are, however, disadvantages to the shadow-paging approach. For example, it is hard to extend shadow paging to allow multiple transactions to execute concurrently.

As before, the database is partitioned into some number of fixed-length blocks, which are referred to as pages. The term page is borrowed form operating systems, since we are using a paging scheme for memory management. Let us assume that there are n pages, numbered 1 through n. (In practice, n may be in the hundreds of thousands.) These pages do not need not to be stored in any particular order on disk. However, there must be a way to find the i^{th} page of the database for any given i. The first entry contains a pointer to the first page of the database, the second entry points to the second page, and so on.

The key idea behind the shadow-paging technique is to maintain two page tables during the life of a transaction: the current page table and the shadow page table. When the transaction starts, both page tables are identical. The shadow page table is never changed over the duration of the transaction. The current page table may be changed when a transaction performs a write operation. All input and output operations use the current page table to locate database pages on disk.

Sub : DBMS Batch : BIT IV Sem.

Transaction

Collections of operations that form a single logical unit of work are called transactions. for example, a transfer of funds from a checking account to a savings account is a single operation when viewed from the customer's standpoint; within the database system, however, it comprises several operation. Clearly, it is essential that all these operation occur, or that, in case of a failure, none occur. It would be unacceptable if the checking account were debited, but the savings account were not credited. A database system must ensure proper execution of transactions despite failures- either the entire transaction executes, or none of it does.

A transaction is a unit of program execution that accesses and possible updates various data items. A collection of operations that should be treated as a single logical operation. A transaction usually results from the execution of a user program and the in the program the transaction is of the form **begin transaction** and **end transaction .**

To ensure integrity of data, we require that the database system maintains the following properties (called ACID properties)of the transactions.

ACID (Atomicity, Consistency, Isolation, Durability)

The four states defined by the acronym ACID (atomicity, consistency, isolation, and durability) relate to a successful transaction, usually a database transaction. All of these characteristics must be present, or the transaction must be completely reversed (undone or rolled back).

A: Atomicity: either the entire set of operations happens or none of it does. Atomicity refers to the ability of the DBMS to guarantee that either all of the tasks of a transaction are performed or none of them are. The transfer of funds can be completed or it can fail for a multitude of reasons, but atomicity guarantees that one account won't be debited if the other is not credited as well.

C: Consistency: the set of operations taken together should move the system from one consistent state to another. Consistency refers to the database being in a legal state when the transaction begins and when it ends. This means that a transaction can't break the rules, or *integrity constraints*, of the database. If an integrity constraint states that all accounts must have a positive balance, then any transaction violating this rule will be aborted.

I: Isolation: even though multiple transactions may operate concurrently, there is a total order on all transactions. Stated another way: each transaction perceives the system as if no other transactions were running concurrently. Isolation refers to the ability of the application to make operations in a transaction appear isolated from all other operations. This means that no operation outside the transaction can ever see the data in an intermediate state; a bank manager can see the transferred funds on one account or the other, but never on both -- even if he ran his query while the transfer was still being processed.

D: Durability: even in the face of a crash, once the system has said that a transaction completed, the results of the transaction must be permanent. Durability refers to the guarantee that once the user has been notified of success, the transaction will persist, and not be undone.

This means it will survive system failure, and that the database system has checked the integrity constraints and won't need to abort the transaction. Typically, all transactions are written into a log that can be played back to recreate the system to the its state right before the failure. A transaction can only be deemed committed after it is safely in the log. Implementing the ACID properties correctly is not simple. Processing a transaction often requires a number of small changes to be made, including updating indexes that are used by the system to speed up searches. This sequence of operations is subject to failure for a number of reasons; for instance, the system may have no room left on its disk drives. ACID suggests that the database be able to perform all of these operations at once. In fact this is difficult to arrange. There are two popular families of techniques: Write ahead logging and Shadow paging. In both cases, locks must be acquired on all information that is read and updated. In write ahead logging, atomicity is guaranteed by ensuring that all REDO and UNDO information is written to a log before it is written to the database. In shadowing, updates are applied to a copy of the database, and the new copy is activated when the transaction commits. The copy refers to unchanged parts of the old version of the database, rather than being an entire duplicate.

Terminology

Concurrency control: the method of ensuring that simultaneously running transactions provide isolation.

Recovery: the act of taking a database after an unexpected failure and returning it to a consistent state.

Abort: undo a transaction.

Commit: make a transaction permanent.

Rollback: the act of "undoing" a transaction.

Transaction manager or transaction monitor: the component responsible for maintaining the ACID properties.

Compensating transaction: a transaction used to undo (rollback) another transaction.

Unresolved transaction: a transaction that has neither aborted nor committed.

Terminated transaction: a transaction that has either aborted or committed.

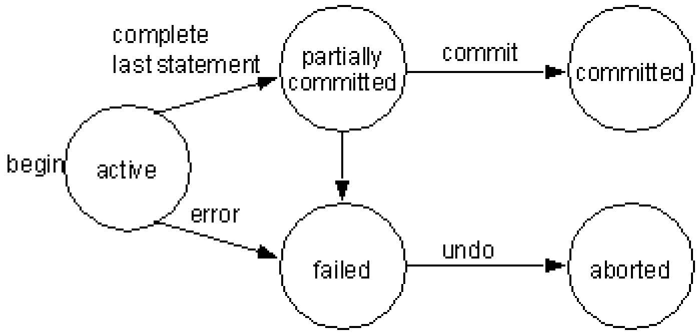
The Life and Times of a Transaction

- Transactions begin by some sort of "transaction begin" operation.
- Transactions end either by being committed or aborted.
- Abort: return the system to the state it was in before the transaction began.
  - System state must be the same as if the transaction had never existed.
  - Must abort any transactions that depend on the outcome of the aborting transaction.
- Commit: declare the transaction permanently complete.
  - If you say you've committed, no actions should be able to move the system to a state not containing the transaction.
  - All operations much be forever persistent in the database.

Transaction States

Active: initial state. Entered upon begin; stays in this state while running.

- Partially Committed: All statements in the transaction have been completed.
- Failed: The transaction has encountered an abnormal or errant condition and cannot continue.
- Committed: The transaction is complete and can never be undone.
- Aborted: The system has been returned to its pre-transaction state.



Three concurrency problems

There are essentially three ways in which things can go wrong three ways, that is, in which a transaction, through correct in itself, can nevertheless produce the wrong answer if some other transaction interferes with it in some way. It should be noted that the interfering transaction might also be correct in itself; it is the uncontrolled interleaving of operations from the two correct transactions that produces the overall incorrect result. The three-concurrency problems are:

- The lost update problem;
- The uncommitted dependency problem and
- The inconsistent analysis problem

The Lost Update Problem

Consider the situation illustrated in Fig. That figure is meant to be read as follows: Transaction A retrieves some tuple to at time t1; transaction B retrieves that same tuple t at

Transaction A	Time	Transaction B
-		-
-		-
RETRIEVE t	t1	-
-		-
-	t2	RETRIEVE t
-		-
UPDATE t	t3	-
-		-
-	t4	UPDATE t
-	↓	-

Transaction A loses an update at time t4

Time t2; transaction A updates the tuple (on the basis of the values seen at time t1) at time t3; and transaction B updates the same tuple (on the basis of the values seen at time t2 which are the same as those seen at time t1) at time t4. Transaction A's update is lost at time t4, because transaction B overwrites it without even looking at it.

The Uncommitted Dependency Problem

The uncommitted dependency problem arises if one transaction is allowed to retrieve or, worse, update – a tuple that has been updated by another transaction but not yet committed by that other transaction. For if it has not yet been committed, there is always a possibility that it never will be committed but will be rolled back instead

In the first example (Fig), transaction A seen an uncommitted update (also called an uncommitted change) at time t2. That update is then undone at time t3. Transaction A is

Transaction A	Time	Transaction B
-		-
-		-
-	t1	UPDATE t
-		-
RETRIEVE t	t2	-
-		-
-	t3	ROLLBACK
-	↓	

Transaction A becomes dependent on an uncommitted change at time t2

Transaction A	Time	Transaction B
-		-
-		-
-	t1	UPDATE t
-		-
UPDATE t	t2	-

-		-
-	t3	ROLLBACK
-	↓	

Transaction A updates an uncommitted change at time t2, and loses that update at update at time t3

therefore operating on a false assumption-namely, the assumption that tuple t has the value seen at time t2, whereas in fact it has whatever value it had prior to time t1. As a result, transaction A might well produce incorrect output. Now, by the way that the rollback of transaction B might be due to no fault of B's –it might, for example be the result of a system crash. (And transaction A might already have terminated by that time, in which case the crash would not cause a rollback to be issued for A also.)

The second example is even worse. Not only does transaction A become dependent on a uncommitted change at time t2, but it actually loses an update at time t3-because the rollback at time r3 cause tuple t to be restored to its value prior to time t1. This is another version of the lost update problem.

The Inconsistent Analysis Problem

Consider Fig , which shows two transactions A and B operating on account (ACC) tuples: Transaction A is summing account balance; transaction B is transferring an amount 10 from account 3 to account 1. The result produced by A 110, is obviously incorrect; if A were to go on the write that result back into the database, it would actually

ACC 1	ACC 2	ACC 3
40	50	30

Transaction A	Time	Transaction B
-		-
-		-
RETRIEVE ACC 1 : sum = 40	t1	-
-		-
		-
RETRIEVE ACC 2 : sum = 90	t2	-
		-
-		-
-	t3	RETRIEVE ACC 3
-		-
-	t4	UPDATE ACC 3 : 30 → 20
-		-
-	t5	RETRIEVE ACC 1
-		-
-	t6	UPDATE ACC 1 40 → 50
-		-
-	t7	COMMIT
-		
RETRIVE ACC 3 : sum = 110, not 120	t8	
-	↓	

Transaction A performs as inconsistent analysis

leave the database in inconsistent state. We say that A has seen an inconsistent state of the database and has therefore performed an inconsistent analysis. Note the difference between this example and the previous one: There is no question here of A being dependent on an uncommitted change, since B commits all of its updates before A see ACC3.

Locking

Concurrency problems can all be solved by means of a concurrency control technique called locking. The basic idea is simple: When a transaction needs an assurance that some object it is interested in typically a database tuple–will not change in some manner while its back is turned (as it were), it acquires a lock on the object. The effect of the lock is to "lock other transaction of the object in question, and thus in particular to prevent them from changing it. The first transaction is therefore able to carry out its processing in the certain knowledge that the object in question will remain in a stable state for al long as the transaction wishes it to.

The working of locks can be explained as follows.

1. First, we assume the system supports two kinds of locks, exclusive locks (X locks) and shared locks (S locks). Note: X and S locks are sometimes called write locks and read locks, respectively.

2. If transaction A holds an exclusive (X) lock on tuple t, then a request from some distinct transaction B for a lock of either type on t will be denied.

3. If transaction A holds a shared (S) lock on tuple t, then:

   - A request from some distinct transaction B for an X lock on t will be denied;
   - A request from some distinct transaction B for an S lock on t will be granted (that is, B will now also hold an S lock on t).

These rules can conveniently be summarized by means of a lock type compatibility matrix (As shown in fig). That matrix is interpreted as follows: Consider some tuple t; suppose transaction A currently holds a lock on t as indicated by the entries in the column headings (dash=no lock); and suppose some distinct transaction B issues a request for a lock on t as indicated by the entries down the left-hand side (for completeness we again include the "no lock" case). An "N" indicates a conflict (B's request cannot be satisfied and B goes into a wait state), a "Y" indicates compatibility (B's request is satisfied). The matrix is obviously symmetric.

	X	S	-
X	N	N	Y
S	N	Y	Y
-	Y	Y	Y

Compatibility matrix for lock types X and S.

data access protocol (or locking protocol) that makes use of X and S locks as just defined to guarantee that problems cannot occur.

1. A transaction that wishes to retrieve a tuple must first acquire an S lock on that tuple.

2. A transaction that wishes to update a tuple must first acquire an X lock on that tuple. Alternatively, if it already holds an S lock on the tuple (as it will in a RETRIEE-UPDATE sequence), then it must promote that S lock to X level.

It should be noted that transaction requests for tuple locks are normally implicit; a "tuple retrieve" request is an implicit request for an S lock, and a "tuple update" request is an implicit request for an X lock, on the relevant tuple. Also, of course (as always), we take the term "update" to include INSERT and DELETEs as well as UPDATEs per se, but the rules requires some minor refinement to take care of INSERTs and DELETEs.

To continue with the protocol:

3. If a lock request from transaction B is denied because it conflicts with a lock already held by transaction A, transaction B goes into a wait state. B will wait until A's lock is released. Note: The system must guarantee that B does not wait forever (a possibility sometimes referred to as livelock). A simple way to provide such a guarantee is to service all lock request on a "first-come, first-served" basis.

4. X locks are held until end-of-transaction (COMMIT or ROLLBACK).

The Three Concurrency Problems Protocols

The Lost Update Problem

As shown in fig, Transaction A's UPDATE at time t3 is not accepted, because it is an implicit request for an X lock on t, and such a request conflicts with the S lock already held by transaction B; so A goes into a wait state. For analogous reasons, B goes into a wait state at time t4. Now both transactions are unable to proceed, so there is no question of any update being lost. Locking thus solves the lost update problem by reducing it to another problem!-but at least it does solve the original problem.

Transaction A	Time	Transaction B
-		-
-		-
RETRIEVE ACC 1 :	t1	-
(acquire s lock on to)		-
-		-
-	t2	RETRIEVE t
-		(acquire S lock on t)
-		-
UPDATE t	t3	-
(request X lock on t)		-
wait		-
wait	t4	UPDATE t
wait		(request X lock on t)
wait		wait
wait		wait
wait	↓	wait

No update is lost, but deadlock occurs at time t4

The Uncommitted Dependency Problem

As shown in fig. Transaction A's operation at time t2 (RETRIEVE and UPDATE in Fig.) is not accepted in either case, because it is an implicit request for a lock on t, and such a request conflicts with the X lock already held by B; so A goes into a wait state. It remains in that wait state until B reaches its termination (either COMMIT or ROLLBACK), when B's lock is released and A is able to proceed; and at that point A sees a committed value (either the pre-B value, if B terminates with rollback, or the post-B value otherwise). Either way, A is no longer dependent on an uncommitted update.

Transaction A	Time	Transaction B
-		-
-		-
-	t1	UPDATE t
-		(acquire X lock on t)
-		-
RETRIEVE t	t2	-
(request S lock on t)		-
wait		-
wait	t3	COMMIT / ROLLBACK
wait		(release X lock on t)
resume : RETRIEVE t	t4	
(acquire S lock on t)		
-	↓	

Transaction A is prevented from seeing an uncommitted change at time t2

Transaction A	Time	Transaction B
-		-
-		-
-	t1	UPDATE t
-		(acquire X lock on t)
-		-
RETRIEVE t	t2	-
(request S lock on t)		-
wait		-
wait	t3	COMMIT / ROLLBACK
wait		(release X lock on t)
resume : RETRIEVE t	t4	
(acquire S lock on t)		
-	↓	

Transaction A is prevented from updating an uncommitted change at time t2

The Inconsistent Analysis Problem

As shown in fig., Transaction B's UPDATE at time t6 is not accepted, because it is an implicit request for an X lock an ACC 1, and such a request conflicts with the S lock already held by A; so B goes into a wait state. Likewise, transaction A's RETREIVE at time t7 is also not accepted, because it is an implicit request for an S lock an ACC 3, and such a request conflicts with the X lock already by B; so A goes into a wait state also. Again, therefore, locking solves the original problem (the inconsistent analysis problem, in this case) by forcing a deadlock.

ACC 1	ACC 2	ACC 3
40	50	30

Transaction A	Time	Transaction B
-		-
-		-
RETRIEVE ACC 1 :	t1	-
(acquire S lock on ACC 1)		-
Sum = 40		-
-		-
RETRIEVE ACC 2 :	t2	-
(acquire S lock on ACC 2)		-
sum = 90		-
-		-
-	t3	RETRIEVE ACC 3
-		(acquire X lock on ACC 3)
-		-
-	t4	UPDATE ACC 3
		(acquire X lock on ACC 3)
-		30 → 20
-		-
-	t5	RETRIEVE ACC 1
-		(acquire S lock on ACC 1)
-		-

-	t6	UPDATE ACC 1
-		(request X lock on ACC 3)
-		wait
RETRIEVE ACC 3 :	t7	wait
(request S lock on ACC 3)		wait
wait		wait
wait	↓	wait

Inconsistent analysis is prevented, but deadlock occurs at time t7

Deadlock Handling

A system is in a deadlock state if there exists a set of transactions such that every transaction in the set is waiting for another transaction in the set. More precisely, there exists a set of waiting transaction {T_O,T_1,…,T_n} such tahat T_O is waiting for a data item that is held by T_1, and T_1 is waiting for a data item that is held by T_2 , and.., and T_{n-1} is waiting for a data item that is held by T_n, and T_n is waiting for a data item that is held by T_O. None of the transactions can make progress in such a situation.

- Occurs when two or more transactions are blocked waiting on each other in such a manner that none will make forward progress.
- Two approaches
  - Prevention: do not let deadlocks occur.
  - Detection and Recovery: allow deadlocks to occur, but detect it and do something about it.

Deadlock Prevention

- Three categories
  - Order lock requests in such a way that deadlock can never occur.
  - Abort a transaction instead of waiting when there is the potential for deadlock.
  - Use time-outs.
- Lock ordering
  - Simple scheme: request all locks atomically at once.

Difficult to know what to lock at the beginning of the transaction.

May hold items locked for along time unnecessarily.

- Preemptive schemes
  - Assign a timestamp to each transaction when it begins.
  - At the time a transaction is about to wait, decide if we allow the transaction to wait, force it to abort, or force someone else to abort.
  - If a transaction aborts, we do not give it a new timestamp.
  - wait-die: wait only on younger transactions; if you encounter an older transaction, abort self.
  - wound-die: if transaction would block on a younger transaction, abort the younger transaction (i.e., wound it).
  - Trade-offs

wait-die: older transactions wait more (but eventually finish).

wait-die: a transaction might get aborted several times before it finally completes.

1. wound-wait: may be fewer rollbacks than wait-die.

  - Both schemes do a lot of rollbacks that may not be necessary.
- Time-out based schemes

- ➲ Allow a transaction to wait for at most a specified time.
- ➲ If the time elapses, the transaction rolls back.

Deadlock Detection and Recovery

- ■ Construct/maintain a waits-for graph.
  - ➲ Vertex set consists of all transactions.
  - ➲ An edge from Vi to V $_j$ indicates that T $_i$ is waiting on a lock held by T $_j$.
- ■ If there is a cycle in the graph, then a deadlock exists.
- ■ Pick a participant in the deadlock and abort it.
  - ➲ May be possible to do partial rollback (i.e., only roll back far enough to break the deadlock instead of to the beginning of the transaction).
  - ➲ In practice, far too complex to maintain the state necessary to do this.
  - ➲ Avoid starvation: do not want the same transaction continually getting rolled back.

Can select a policy that favors transactions previously aborted.

Buffer Management

Programs in a database system make requests on the buffer manager when they need a block from disk. If the block is already in the buffer, the requester is passed the address of the block in main memory. If the block is not in the buffer, the buffer manager first allocates space in the buffer for the block, throwing out some other block, if required, to make space for the new block. The block that is thrown out is written back to disk only if it was modified since the most recent time that it was written to the disk. Then, the buffer manager reads in the block from the disk to the buffer, and passes the address of the block in main memory to the requester. The internal actions of the buffer manager are transparent to the programs that issue disk- block requests.

Buffer manager appears to be nothing more than a virtual memory manager, like those found in most operating systems. One difference is that the size of memory addresses are not sufficient to address all disks blocks. Further, to serve the database system well, the buffer manager must use techniques more sophisticated than typical virtual memory management schemes.

Replacement strategy: When there is no room left in the buffer, a block must be removed from the buffer before a new one can read in. Typically, operating systems use a least recently used(LRU) scheme, in which the block that was referenced least recently is written back to disk and is removed from the buffer. This simple approach can be improved on for database applications.

Pined blocks: For the database system to be able to recover from crashes it is necessary to restrict those times when a block may be written back to disk. A block that is not allowed to be written back to disk is said to be pined. Although many operating systems do not provide support for pinned blocks, such a feature is essential for the implementation of a database system that is resilient to crashes.

Forced output of blocks: There are situations in which it is necessary to write back the block to disk, even thought the buffer space that is occupies is not needed. This write is call the forced output of a block   Main memory contents and thus buffer contents are lost in a crash, whereas data on disk usually survive a crash.

Definition of Database

Data are facts that can be recorded and have implicit meaning. Data refers to values such as names, telephone, addresses that can be easily stored inside diary, PC or floppy. Data is actually stored in the database and information refers to the meaning of that data as understood by user.

The database is collection of related data. A database has the following implicit properties.

i. A database represents some aspect of the real world, sometimes called the miniworld or the Universe of Discourse (UoD). Changes to the miniworld are reflected in the database.

ii. A database is a logically coherent collection of data with some inherent meaning.

iii. A database is designed built and populated with data for a specific purpose. It has an intended group of users and some applications.

Database can be of any size. Example for Sources of databases are patients in hospital, bank, university, government department etc.

Definition of DBMS

DBMS means database Management System. It is a collection of programs that enables users to create and maintain database as well as enables to store, modify and extract information from the database. DBMS is software for defining, constructing and manipulating databases. It is also called database manager or database server. Example of DBMS are Ms. Access, oracle, MYSQL, Ms. SQL server etc.

Thus the goal of DBMS is to provide an environment that is both convenient and efficient to use in retrieving and storing database information. In DBMS, user issue request for information then DBMS analyzes and some internal processing takes place and then the result is sent back to the user.

Definition of Database System

Database system is computerized record keeping system. e.g. Computerized library system, flight reservation system, automated teller machine etc. Database and the DBMS software collectively known as database system.

The following operations take place in the database system.

i. Adding new / empty files to database.

ii. Inserting, retrieving, updating, deleting data from existing database.

iii. Removing existing files from database.

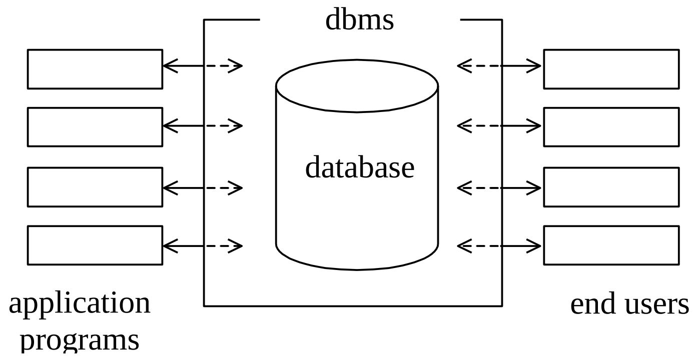


Fig. simplified picture of a database system

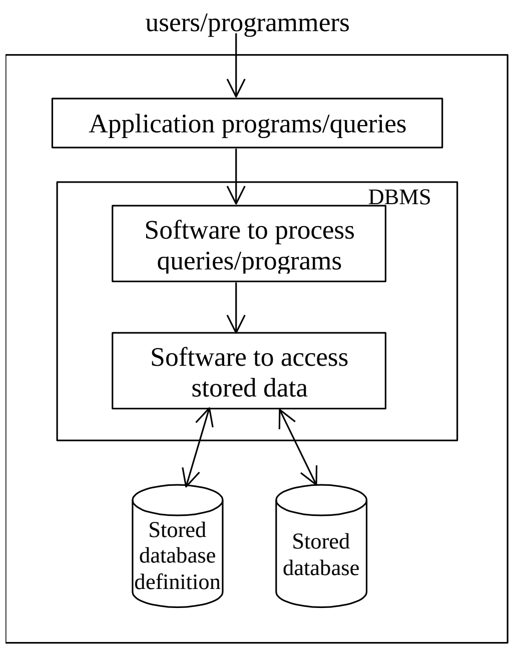

users/programmers
Application programs/queries
DBMS
Software to process queries/programs
Software to access stored data
Stored database definition
Stored database

Fig. A simplified database system environment

Advantages of database system over paper based methods of record keeping are (i) compactness (ii) speed and (iii) accuracy

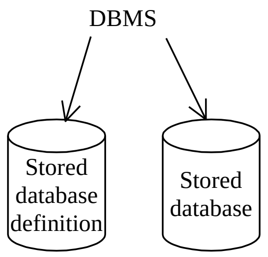
Characteristics of Database Approach

There are a no. of characteristics which distinguish the database from the traditional approach of programming with files. In the traditional approach of programming with files, many users may be using the same data such as student name separately. Thus data is duplicated and leads to wastage of storage space.

Main characteristics of database approach versus the file processing approach are as follows

i) Self describing nature of a database system

The definition or description of the database is stored in the system catalog separately and thus are available to users.



The system catalog stores structure and details of database only and no other data. thus the system catalog inside dbms describes database itself.

ii) Insulation between programs and data, and data abstraction

In traditional file processing, the changes the structure of data file may require changing all programs that access this file but the DBMS changes catalog information only. Thus both the program and data are independent and also called program data independence.

Data abstraction : DBMS provides user with a conceptual representation of data that does not include many of details of how the data is stored or how the operations are being implemented. Suppose the example of car. People don't think of a car as set of tens of thousands of individual parts. They think of it as a well defined object with its own behavior. Similarly data abstraction hides the complexity. Data model is a type of data abstraction.

iii) Support of multiple views of data

A database typically has many users, each of whom may require a different perspective or view of the database. A view may be portion or subset of the database. It is also called virtual table as it may contain virtual data. Users shouldn't be given the whole privilege for security purpose about some users may not be aware of whether the data they refer to is stored or derived. The DBMS supports multiple news view of data in a multi-user DBMS.

iv) Sharing of data and multi-user transaction processing

Many user can select, update data at the same time. So dbms must support concurrency control. for example in applications such as train/bus reservation system, flight reservation system, many users use the system from different locations at the same time and so is sharing of data and multi-user transaction processing.

Advantages and benefits using DBMS

Advantages of using DBMS are as follows

i) Controlling redundancy

In traditional file processing system, each user maintains their own file and so there may be duplication of data. Storing same data multiple times lead to several problems such as wastage of space, duplication effort for entering data, data may become inconsistent.

ii) Restricting unauthorized access (security)

Confidential data should not be available to all users. User accounts with certain restrictions to data may be created for security. Similarly multiple views can be created for database security. In traditional file processing, if own get file gets everything & all data.

iii) Providing persistent storage for program objects and data structure

The values of program variables are discarded once the program terminates as in C, C++ pascal program unless the programmer write them in files. A complex object in C++ can be stored permanently in an object oriented DBMS.

iv) Permitting inferencing and actions using rules

Database system may be deductive or active. Deductive databases have capabilities for defining deduction rules for inference new information from stored database. It works like reporting system.

v) Providing multiple user interfaces

DBMS provides variety of interfaces for varying users. DBMS provide query language for casual user, programming languages for application programmers, forms and command for parametric users, menu driven interfaces for stand alone users. Form styles and menu driven interfaces are collectively called GUI (Graphical user interface)

vi) Representing complex relationship among data

Relationships may be created among data using DBMS which helps in managing the data and defining constraints for updating and deleting.

vii) Enforcing integrity constraints

Something that limits data is called constraints in database. For example, the minimum balance should not fall in a bank. It is a constraint. Some of the constraints are primary key, NOT NULL, check.

viii) Providing backup and recovery

DBMS provides facilities for taking backup of the database which can be used for recovery in case of failure of computer system or hardware system.

ix) Easy in accessing data

It becomes very easy and fast while accessing data from database using DBMS. Reports can be used for easy access of data.

x) Concurrent access to database

Many users can share the data at the same time and thus dbms provides users to access the database concurrently.

Sub : DBMS   Batch : BIT IV Sem.

Database system concepts and architecture

Data Model

Data model is a collection of tools for describing data, data relationship and consistency constraints. It is used to describe the structure of a database, basic operations for specifying retrievals and updates on the database.

Data model is a type of data abstraction. 3 levels of data abstraction are as follows.

i) Physical level : It is also called internal or low level data model. It describes about how data is actually stored in the database.

ii) Logical level : Next higher level is the logical level which describes about what data are stored and its relationship.

iii) View level : It is the highest level and describes about multiple views of data.

Many data models have been proposed.

Categories of data model:

1) Object based logical models

2) Record based logical models

3) Physical models

1. Object based logical models

It is used in describing data at the logical and view levels. There are many different models. Some of them are:

i) Entity-relationship model

ii) Object-oriented model

iii) Semantic data model

iv) Functional data model

i) Entity-relationship model:

The entity relationship (ER) data model is based on a perception of a real world that consists of a collection of basic objects, called entities, and of relationships among these objects.

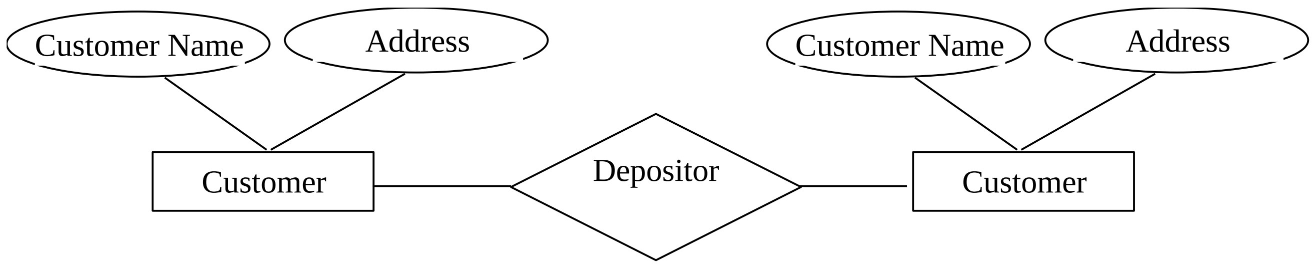


Fig. A sample E-R diagram

ii) Object oriented model

The object oriented model is based on a collection of object. An object contains values stored in instance variables within the object. An object also contains bodies of code that operate on the object. These bodies of code are called methods. Objects that contain the same types of values and the same methods are grouped together into classes.

iii) Semantic data models

It is similar to E-R modeling. It is also called object modeling. It also supports entity, which has properties and relationship.

iv) Functional data model

It is based on functions instead of relations. The functional approach shares certain ideas with object approach. It addresses object, which are functionally related to other.

2. Record based logical models

Record based logical models are used in describing data at the logical and view levels. It is used to specify the overall logical structure of the database.

Record based models are so named because the database is structured in fixed format records of several types. Each record type defines a fixed no. of fields, or attributes, and each field is usually of a fixed length. The three most widely accepted record based data models are the relational, network and hierarchical models.

i) Relational model

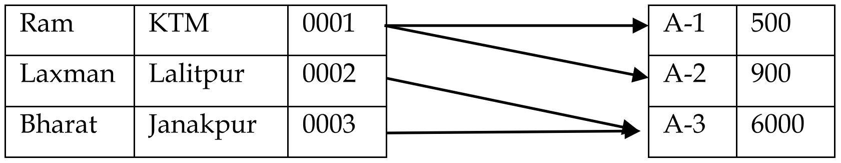
The relational model uses a collection of relations(tables) to represent both data and the relationships among those data. Each table has multiple columns, and each column has a unique name.

Customer Name	Address	Account No.	Account No.	Balance
Ram	KTM	A-1	A-1	500
Laxman	Lalitpur	A-2	A-2	700
Bharat	Jhapa	A-3	A-3	900

Fig. A sample relational database

ii)Network model

Data in the network model are represented by collections of records and relationships among data are represented by links, which can be viewed as pointers.



Ram	KTM	0001
Laxman	Lalitpur	0002
Bharat	Janakpur	0003

A-1	500
A-2	900
A-3	6000

Fig. A sample Network database

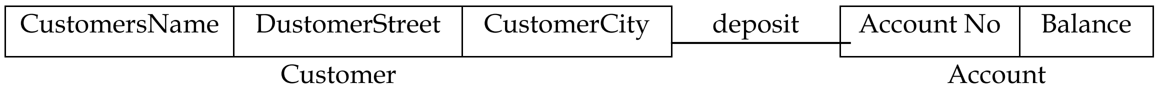


Fig. data structure diagram for network data model

iii) Hierarchical model

It is similar to the network model in the sense that data and relationships among data are represented by records and links, respectively. Records are organized as collections of trees rather than arbitrary graphs.

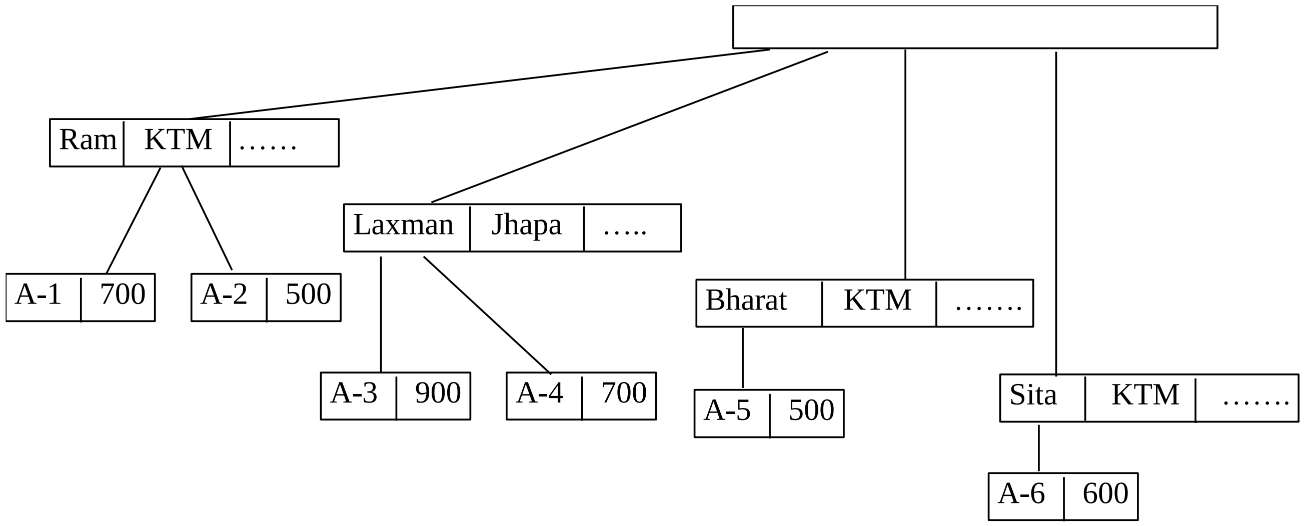


Fig. A sample hierarchical database

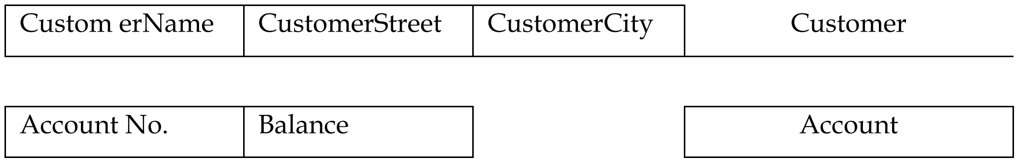


Fig. Tree structure diagram for hierarchical model

Physical models

Physical data models are used to describe data at the lowest level. There are only few physical data modes in use. Two widely known ones are the unifying model and the frame memory model.

Schemas and instances

Database=description of database + database itself

The overall design of the database is called database schema. Database schema is specified during database design and not expected to change frequently.

Student			Course		
Name	Class	Major	Course Name	Duration	Remarks

Fig. Schema Diagram

Database changes over time as information is inserted and deleted. The collection of information stored in the database at a particular moment is called instances in the database. It is also called database state or snapshot or current set of occurrences. When database is designed, the database is in empty state with no data. It is in initial state when database is loaded with data.

Thus at any point, database has a current state. When any field is added to database, it is called schema evolution.

DBMS Architecture

The main characteristics of database approach are (i) insulation of programs and data (ii) support of multiple user views (iii) use of a catalog to store the database description. The architecture of the DBMS is proposed to visualise these characteristics and so called the three schema architecture. It is also called ANSI/SPARC (American National standard Institute/Standards planning and requirements committee) Architecture.

Goal of the architecture is to separate the user applications and physical database. In this architecture, schemas can be defined as the following three levels.

i) The internal level

It has an internal schema which describes the physical structure of the database. It describes the complete details of data storage and access paths for the database.

ii) The conceptual level

It has a conceptual schema, which describes the structure of the whole database for a community of users. It hides the details of physical storage structures and concentrates on describing entities, data types, relationships, user operations and constrains.

iii) The external or view level

It includes a number of external schemas or user views. Each external schema describes the past of the database that a particular user group is interested in and hides the rest of the database from that user group.

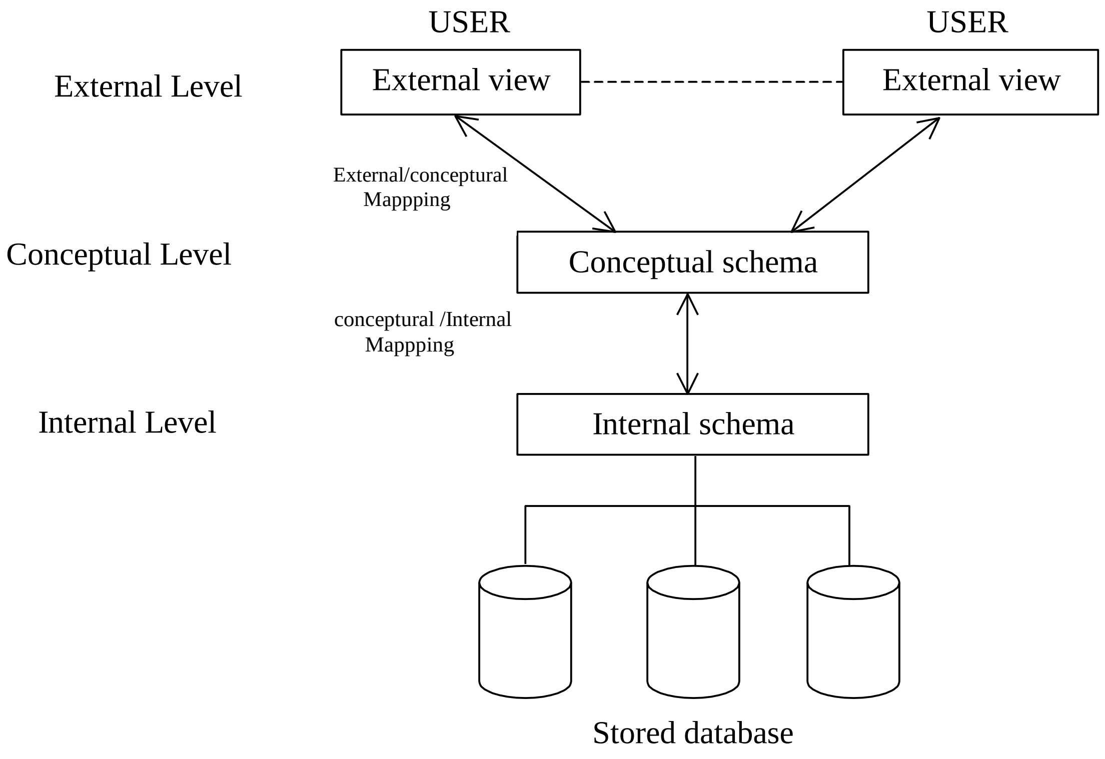


Fig. Three schema Architecture

Three schema architecture is a tool for the user to visualize the schema levels in a database system. Most DBMS don't separate the three levels data actually exists at the physical level. User/groups refer only to its own external schema. So DBMS must transform a request from users into a request against conceptual schema and then into a request on the internal schema for processing over the stored database. If the request is a database retrieval, the data extracted from the stored database must be reformatted to match the user's external view. The process of transforming requests and results between levels are called mappings.

DATA INDEPENDENCE

Data independence is defined as the capacity of DBMS to change the schema at one level of a database system without having to change the schema at the next higher level. We can define two types of data independence.

i) Logical data independence

Logical data independence is the capacity to change the conceptual schema without having to change external schemas or application programs. We may change the conceptual schema to expand the database (by addressing a record type or data item) or to reduce the database (by removing a record type or data item). It results in change in E-R diagram but the application program or external schema is not changed.

ii) Physical data independence

Physical data independence is the capacity to change the internal schema without having to change the conceptual (or external) schemas. Changes to the internal schema may be needed because some physical files had to be reorganized. For example, by creating additional access structures to improve the performance of retrieval or update. If the same data as before remains in the database, we should not have to change the conceptual schema.

In multiple level DBMS, its catalog must be expanded to include information on how to map requests and data among the various levels. In data independence, when the schema is changed at some level, the schema at the next higher level remains unchanged only mappings change.

DATABASE LANGUAGES

A database system provides mainly two different types of languages: one to specify the database schema, called data definition language and the other to express database queries and updates called data manipulation language.

i) Data definition language (DDL):

A database schema is specified by a set of definitions expressed by a special language called a data-definition language. The result of compilation of DDL statements is a set of tables that is stored in a special file called data dictionary or data directory. A data dictionary is a file that contains metadata that is data about data. DBMS will have a DDL complier, which process DDL statements. DDL is used to specify conceptual schema only. Similarly, SDL (storage definition language) is used to specify the internal schema & VDL (view definition language) is used to specify user views and their mappings to the conceptual schema.

iii) Data manipulation language (DML)

A data manipulation language is a language that enables users to access or manipulate data as organized by the appropriate data model.

Data manipulation consists of

- The retrieval of information stored in the database
- The insertion of new information into the database
- The deletion of information from the database
- The modification/update of information stored in the database

DML is of the following 2 types

a) Non-procedural DMLs

The language requires a user to specify what data are needed without specifying how to get those data. It is easier to learn and use. many DBMS allow it either to be entered interactively from a terminal or to be embedded in a general purpose programming language. For example SQL (structured query language). SQL can retrieve many records in a single DML statements and hence it is also called set at a time or set oriented language. It is also called high level language.

b) Procedural (Low Level) DMLs

The language requires a user to specify what data are needed and now to get those data. It is embedded in a general purpose programming language. This type of DML retrieves records one by one and processes each record separately using programming language construct such as looping and hence it is also called record at a time DML.

When DML are embedded in a general purpose-programming language, then that language is called host language and the DML is called the data sub language.

DBMS Interfaces

DBMS provides the following user-friendly interfaces.

i) Menu based interfaces for browsing

These interfaces present the user with lists of options, called menus that lead the user through the formulation of a request. The query is composed step by step by picking optional from a menu that is displayed by the system. Pull down menus is becoming popular technique in window based user interfaces.

ii) Forms based interfaces

A forms-based interface displays a form to each other. Users can fill out all of the form entries to insert new data, or they fill out only certain entries. Forms are usually designed and programmed for naïve users as interfaces to canned transactions.

iii) Graphical user interface

A graphical interface (GUI) typically displays a schema to the user in diagrammatic form. The user can then specify a query by manipulating the diagram. GUIs utilize both menus and forms.

iv) Natural language interfaces

These interfaces accept requests written in English or some other language and attempt to understand them. The natural language interface usually has its own schema, which is similar to database conceptual schema. The natural language interface refers to words in its schema to interpret the request. If the interpretation is successful, the interface generates a high level query corresponding to the natural language request and submits it to the DBMS for processing.

v) Interfaces for parametric users

Parametric users such as bank tellers, often have a small set of operations that they must perform repeatedly. System analysts and programmers design and implement a special interface for a known class of naïve users. For example, function keys in a terminal can be programmed to initiate the various commands. This allows the parametric user to proceed with a minimal number of keystrokes.

vi) Interfaces for DBA

Most database system contains privileged commands that can be used only by the DBA's staff. These include commands for creating accounts, setting system parameters, granting account authorization, changing a schema, and reorganizing the storage structure of database.

Database System Environment

A DBMS is a complex software system. The database system environment consists of DBMS component modules, Database system utilities and tools, application environments and communication facilities. They are explained as follows.

i. DBMS component modules

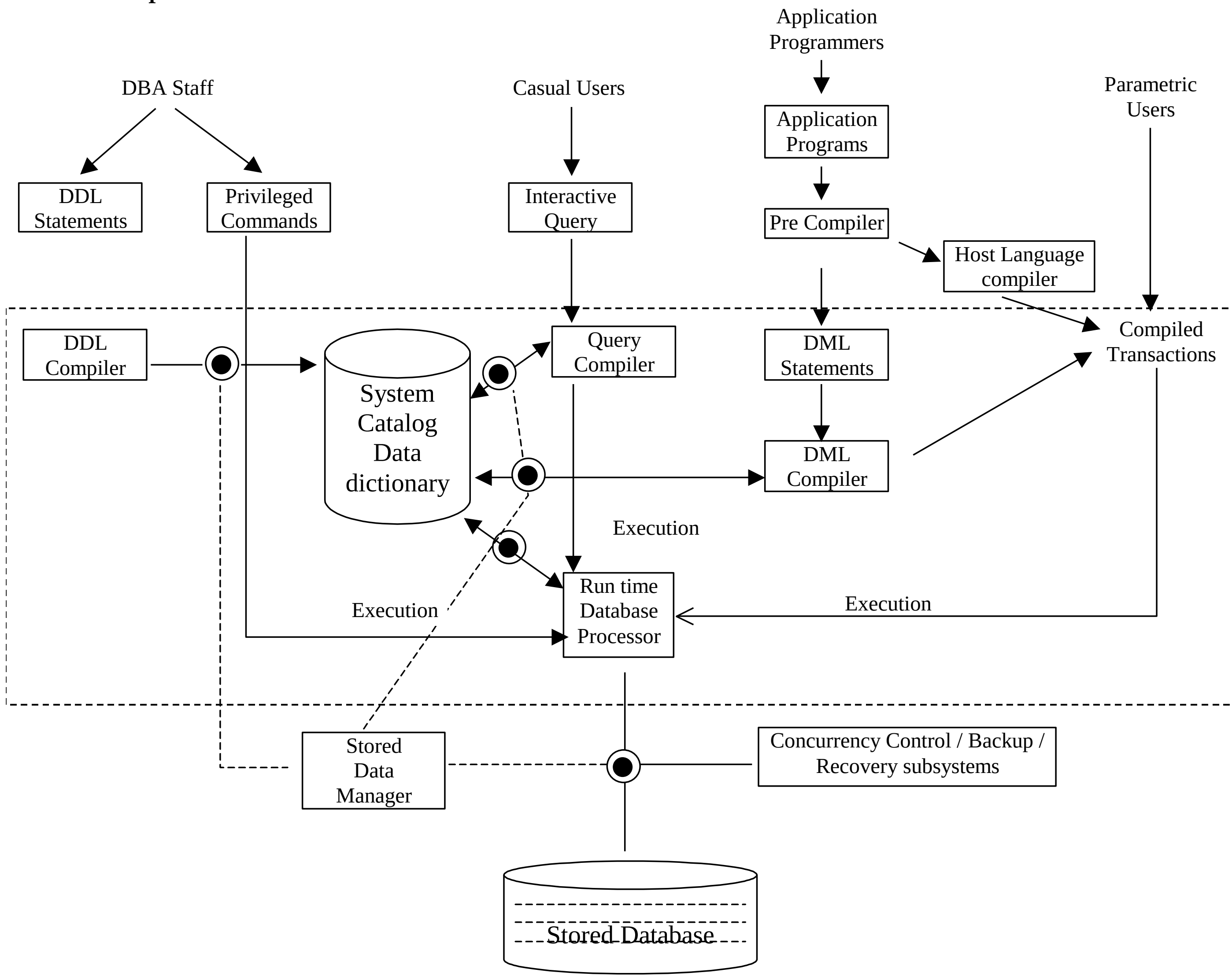


Fig. typical component modules of a DBMS

The database and the DBMS catalog are stored on the disk. There are different modules in the Database system. The stored data manager module of the DBMS controls access to the information of database and catalog. The dotted lines and the circles marked A, B, C, D and E as shown in fig illustrate access that are under the control of this stored data manager.

Similarly DDL compiler processes schema definitions, specified in the DDL and stores description of the schemas in the DBMS catalog. The catalog includes information such as the names of files, data items, storage details of each file, mapping information among schemas and constraints. Run time database processor handles database accesses at run time. It receives retrieval or update operations and carries them out on the database.

The Query compiler handles high level query that are entered interactively. It parses, analyzes and compiles a query by creating database access code and then generates calls to the run time processor for executing the code.

The precompiler extracts DML statements from an applications. Program written in the host programming language. These commands are sent to the DML compiler for compilation into object code for database access. Rest of the program is sent to the host language compiler. Object codes for DML commands and the rest of the Program and linked forming a transaction whose code includes calls to the run time database Processor. If many users share the computer system, the operating system will schedule DBMS disk access requests and DBMS processing along with other processes.

ii. Database system Utilities

Most DBMS have database utilities that help the DBA in managing the database system. common Utilities have the following types of functions.

a. Loading: A loading utility is used to load existing data files such as text files or sequential files into the database. Some vendors are offering products that generate the appropriate loading programs, giving the existing source and target database storage descriptions. Such tools are also called conversion tools.

b. Backup: A backup utility creates a backup copy of the database, usually by dumping the entire database onto tape. The back up can be used to restore the database in the case of failure of computer system or database system. Incremental or differential backup method may be used.

c. File reorganization: This utility can be used to reorganize, reindex a database file to improve the performance.

d. Performance monitoring: This utility monitors database usage and provides statistics to DBA which helps in making decisions to improve performance.

iii. Tools, application environments and Communication facilities:

Tools such as CASE, data dictionary, application developments are often available to database system. The CASE (Computer Aided Software Engineering) tools are used in the design Phase of database system. Data dictionary is used for storing catalog information about schemas and constraints, usage standards, application program description and user information. Such system is also called information repository. Application develop environment provide an environment for developing database design, application, querying and updating. Many commercial database systems have communication package whose function is to allow users at location remote from database site to access the database through communication hardware.

Classification of Database Management System

DBMS is classified on the basis of data model, number of users, number of sites, cost and types of access path.

On the basis of data model, DBMS is classified into

i. Relational data model

ii. Object data model

iii. Hierarchical data model

iv. Network data model

On the basis of numbers of users, DBMS is classified into

i. Single user system – supports only one user at a time

ii. Multi-user system supports multiple users concurrently

On the basis of numbers of sites, DBMS is classified into

i. Centralized – if the data is stored at a single computer site.

ii. Distributed – database and DBMS software distributed over many sites, connected by a computer network.

iii. Homogeneous – Use same DBMS software at multiple sites.

iv. Heterogenous – Participating DBMS are loosely coupled and have a degree of local autonomy. Many DBMS use a client server architecture.

On the basis of cost, DBMS is classified into

i. DBMS packages between $10,000 and $100,000

ii. DBMS packages costing more than $100,000

On the basis of types of access of Path, DBMS is classified into

i. General purpose – Designed for general purpose

ii. Special purpose – Designed and built for specific application such as airlines reservation, telephone directly system such DBMS can't be used for other applications without major change.

Data Dictionary

The data dictionary can be regarded as a system database which contains data about data. This is also called metadata. It contains definitions of other objects in the system instead of raw data. It also stores schemas and mappings details, various security and integrity constraints. It is also called data directory or system catalog or simply catalog or data repository

TABLE		
TABNAME	**COLCOUNT**	**ROWCOUNT**
DEPT	2	2
Emp	3	3

COLUMN	
TABNAME	**COLNAME**
DEPT	Dept No.
DEPT	Dept Name
EMP	Emp Name
EMP	Emp No.

Fig. Catalog for department and Employee database

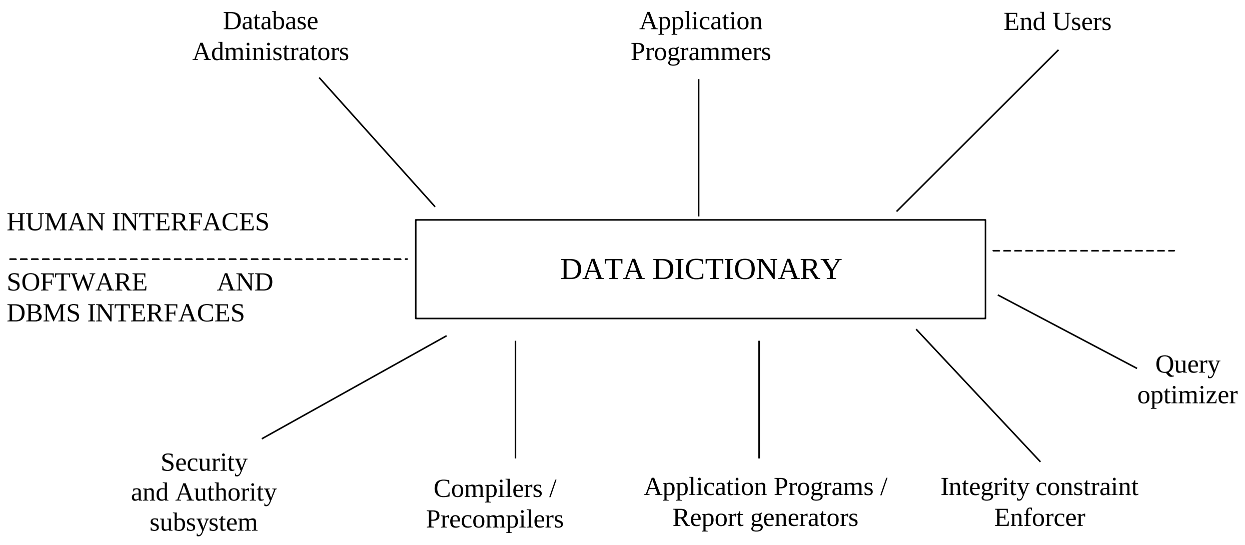

Database Administrators
Application Programmers
End Users
HUMAN INTERFACES
SOFTWARE AND DBMS INTERFACES
DATA DICTIONARY
Query optimizer
Security and Authority subsystem
Compilers / Precompilers
Application Programs / Report generators
Integrity constraint Enforcer

Fig. Human & Software interfaces to a data dictionary

Data dictionary is accessed by various software modules of DBMS itself such as DDL/DML compilers, query optimizer, constraint enforcer. If the data dictionary is used by designers, users and administrators, not by DBMS software it is called a passive data dictionary, otherwise it is called an active data dictionary.

E-R MODEL

E-R model means entity relationship which is a popular high level conceptual data model. ER model describes data as entities, relationship and attributes. ER model is based on a perception of a real world that consists of a set of basic objects called entities, and of relationship among these objects.

Entity types and entity sets: An entity is a thing or object in the real world that is distinguishable from all other objects. For example, each person in an enterprise is an entity. An entity has a set of properties, which may uniquely identify an entity. An entity may be concrete, such as a person or a book, or it may be abstract such as loan or a holiday or a concept.

An entity set is a set of entities of the same type that share the same properties or attributes. The set of all persons who are customers at a given bank for example can be defined as the entity set customer. Similarly, the entity set loan might represent set of all loans awarded by a particular bank.

Attributes: Attributes are descriptive properties possessed by each member of an entity set. Each entity has attributes. For example an employee entity may be described by the employee's name, age, address, salary and job. Possible attributes of the loan entity set are loan number and amount. For each attribute, there is a set of permitted values, called the domain or value set.

Employee Name	**Address**	**Age**	**Salary**	**Job**
Ram	KTM	15	5000	Manager
Shyam	KTM	20	3000	Operator

Loan No.	**Loan Amount**
L – 1	5000
L - 2	3000

Sita	BRT	17	4000	CEO

Customer

L – 3	10,000

Loan

Fig.: Entity sets customer and loan

An attribute, as used in the E-R model, can be characterized by the following attribute types.

i) Simple (Atomic) and composite attributes: Attributes that are not divisible are called simple or atomic attributes. Such as age as shown in fig. is simple attribute. Composite attributes can be divided into smaller attribute. Composite attributes can be divided into smaller subparts, which represent more basic attributes with independent meaning. For example: Employee name could be structured as a composite attribute consisting of first name, middle name and last name.

ii) Single valued and multivalued attributes: Single valued have a single value for a particular entity. For example, the loan number attribute for a specific loan entity refers to only one loan number and so it is single valued. Consider the employee entity set with the attribute dependent name. Any particular employee may have zero, one or more dependents. So, different employee entities within the entity set will have different numbers of values for the dependent name attribute and this type of attribute is said to be multivalued.

iii) Null (missing): A null value is used when an entity does not have a value for an attribute. It is unknown value of not applicable. If a particular employee has ho dependents. The dependent name value for that employee will be null.

iv) Derived attribute: The value for this type of attribute can be derived from the values of other related attributes. For instance, let us say that the customer entity has an attribute loans-held, which represents how many loans a for this attribute by counting the number of loan entities associated with that customer.

v) Complex Attributes: The composite and multivalued attributes be nested in an arbitrary way. We can represent arbitrary resting by grouping components of a composite attribute between Parentheses ( ) and separating the components with commas, and by displaying multivalued attributes between braces { { . Such attributes are called complex attributes. For example, if a person can have more than one residence and each residence can have multiple phones. An attributes Address Phone for a person entity type can be specified as bellows:

{ Addres Phone ({ Phone (AreaCode, PhoneNumber)},

Address (Street Address (Number, street, ApartmentNumber), city, state, up))}

Key attributes of an entity type:

An entity type defines a collection of entities that have the same attributes. The collection of all entities of a particular entity type in the database at any point in time is called an entity set.

Entity Type	Employee	Company
Name	EmpID, Name, Age, Salary	Name, Headquarters, President
Entity Set:	e_1 : (1, Ram, 10, 2000) e_2 (2, Shyam, 20, 5000)	C_1 (wlink, Jawalakhel, Dr. Ashish) C_2 (Nepasoft, Ratnapark, S.P. Joshi)

	e_2 (3, Mohan, 25, 1000)

C_3 (NEA, KTM, Dr. S.R. Malla)

Fig.: Two entity types named employee and company and some of the member entities in the entity set.

It is important to be able to specify how entities within a given entity set and relationships within a given relationship set are distinguished. An entity type usually has an attribute whose values are distinct for each individual entity in the collection. Such an attribute is called a key attribute and its values can be used to identify each entity uniquely. For example, the EmpId attribute is a key of the employee entity type. Some keys are superkey, Candidate key and Primary Key.

Superkey: A super key is a set of one or more attributes that, taken collectively, allows us to identify uniquely an entity in the entity set. For example, EmpId is a super key for the entity set employee.

For example: suppose the attributes of the customer entity set are customer Name, Social security, customer street, customerCity. Then social security is a superkey.

Candidate Key: There may be superkeys for which no proper subset is a superkey. Such minimal superkeys are called candidate keys. {Social – security} and {customerName, CustomerStreet} are candidate keys. Although the attributes social security and customer Name together can distinguish customer entities, their combination does not form a candidate key, since the attribute social security alone is a candidate key.

Primary key: Primary key is a candidate key that is chosen by the database designer as the principal means of identifying entities within an entity set. A key (Primary, candidate and super is a property of the entity set, rather than of the individual entites.

Relationships & Relationship Types

A relationship is an association between entities. Each relationship is identified so that its name is descriptive of the relationship. Verbs such as takes, teaches, and employs make good relationship names. For example, a student takes a class, a professor teaches a class, a department employs a professor and so on.

Rectangles represent entity sets, ellipses represent attributes. Similarly relationships are represented by diamond shaped symbols as shown below in fig. and the lines link attributes to entity sets and entity sets to relationship sets.

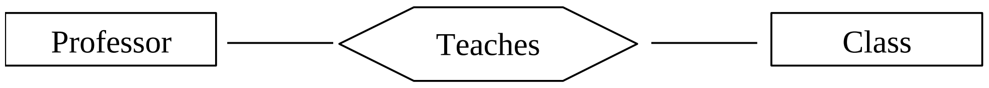


Fig.: An entity relationship

The figure shows a relationship between two entities (also known as participants in the relationship) named professor and class respectively.

A relationships degree indicates the number of associated entities or participants. A unary relationship exists when an association is maintained within a single entity. A binary relationship exists when two entities are associated. A ternary relationship exists when three entities are associated. Although higher degrees exist, they are rare and are not specifically named.

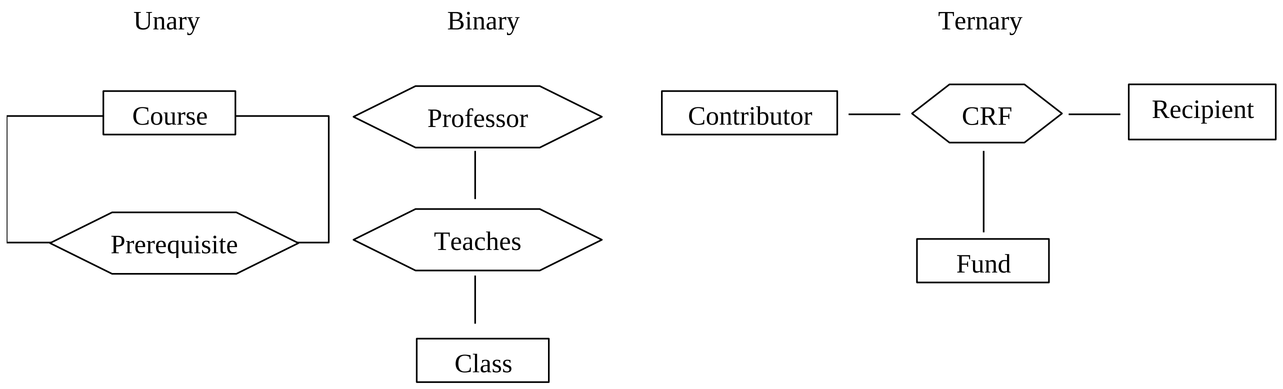


Fig.: Three types of relationships

A course within the course entity is a prerequisite for another course within that entity. The existence of a course prerequisite means that a course requires a course i.e. a course has a relationship with itself. Such a relationship is also called a recursive relationship.

Connectivity: The relationships are all classified as M : N for example. A fund can have many donors. A fund may support many researchers who become the fund receiptants and a researcher may draw support from many funds. Contributors can make donations to many funds.

The term connectivity is used to describe the relationship classification.

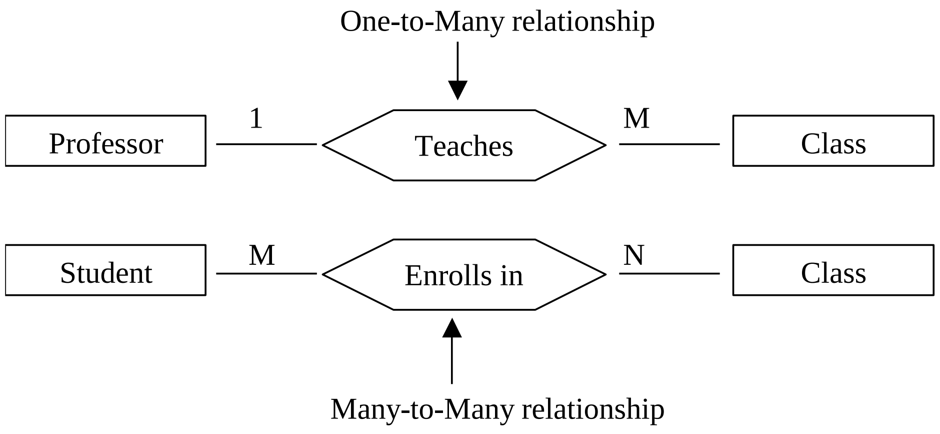


Fig.: Connectivity in an E-R diagram

Cardinality: Cardinality expresses the specific number of entity occurrences associated with one occurrence of the related entity. The actual number of associated entities usually is a function of an organizations policy. For example: For Purbanchal University limits the professor to teaching a maximum of three classes per week. Therefore, the cardinality rule governing the professor – class association is expressed as "one professor teaches upto three classes per week. The cardinality is indicated by placing the appropriate numbers beside the entities as shown in fig.

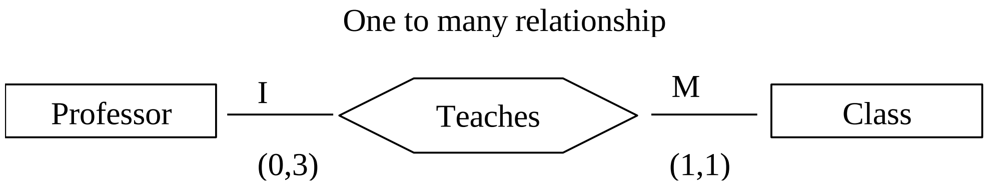


Fig.: Cardinality in an E-R diagram

- The relationship between Professor and class is 1:M
- The cardinality limits are (0,3) for professor indicating that a professor may teach a minimum of zero and a maximum of three class
- The cardinality limits for class entity are (1,1) indicating that the minimum no. of professor required to teach a class is one, as is the maximum number of Professors.

For binary relationship between entity sets A and B, the mapping cardinality must be one of the following.

i) One to one: An entity in A is associated with at most one entity in B, and an entity in B is associated with at most one entity A.

ii) One to many: An entity in A is associated with any number of entities in B and an entity in B however, can be associated with almost one entity in A.

iii) Many to one: An entity in A is associated with at most one entity in B and an entity in B, however can be associated with any number of entities in A.

iv) Many to many: An entity in A is associated with any number of entities in B and an entity in B is associated with any number of entities in A

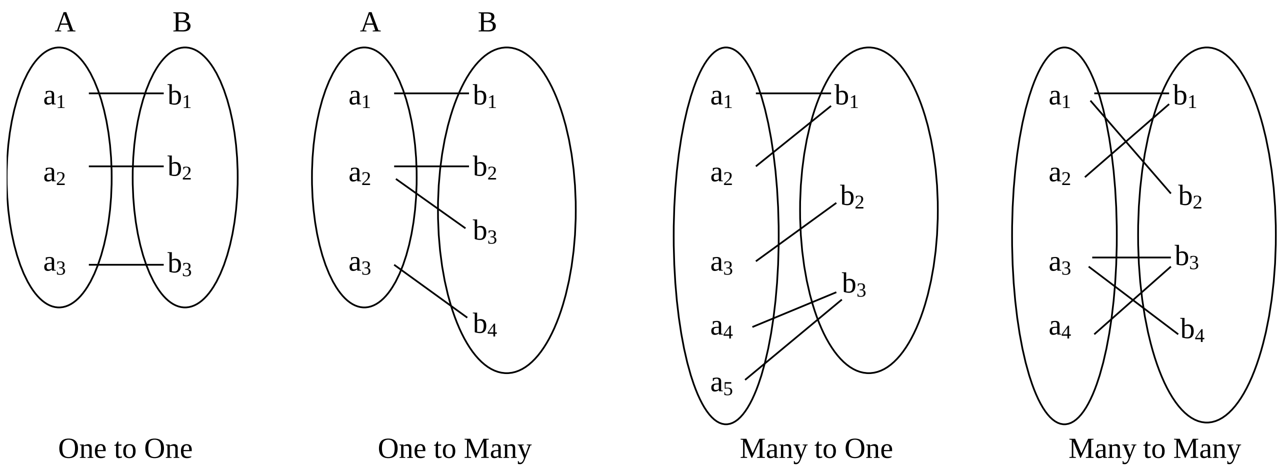
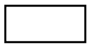


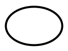
Fig.: Mapping Cardinalities

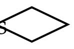
A relationship is an association among several entities.

Entity – Relationship Diagram (E-R diagram)

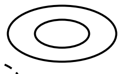
The E-R diagram is used to represent to E-R model. The E-R diagram consists of the following major components.

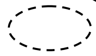
i) Rectangles – to represent entity sets 

ii) Ellipses – to represent attributes 

iii) Diamonds – to represent Relationships 

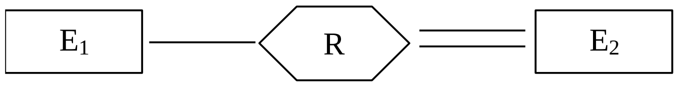
iv) Lines – to link attributes to entity sets and entity sets to relationship sets -

v) Double ellipses – to represent multivalued attributes 

vi) Dashed ellipses – to denote derived attributes – 

vii) Double lines – to indicate total participation of an entity in a relationship set.

Total participation of $\in_2$ in R



For example: Suppose the attributes associated with customer are customerName, SocialSecurity, CustomerStreet and CustomerCity. The attributes associated with loan are loanNumber and amount. The relationship set borrower may be many to many, one to many, many to one and one to one. To distinguish among these types, we know either a directed line (→) or an undirected line (-) between the relationship set and the entity set.

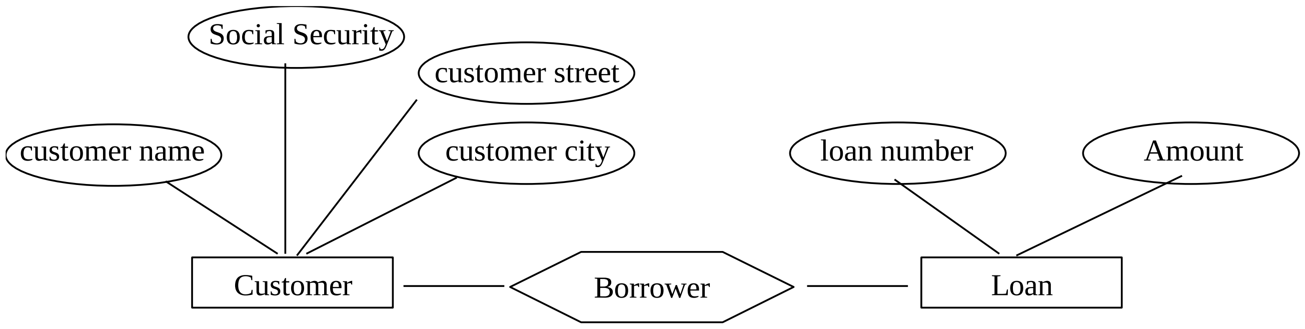


Fig. E-R diagram corresponding to customers and loans

Similarly, we can see another example as follows:

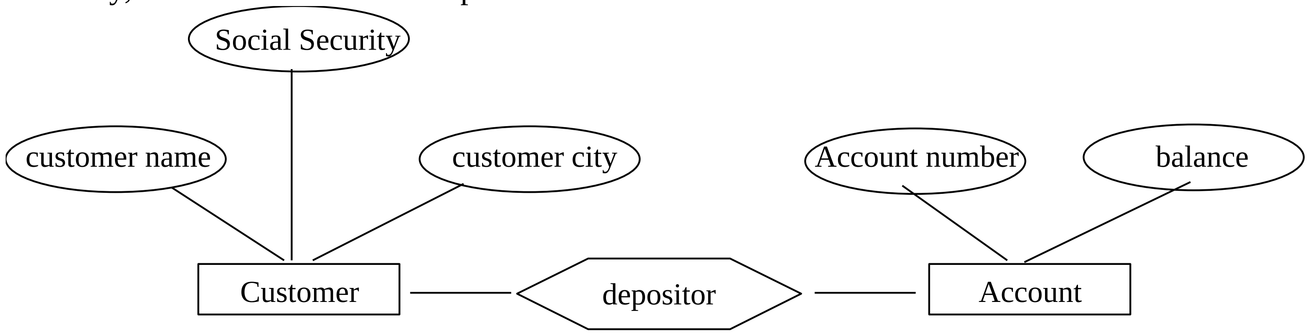


Fig. E-R diagram showing one to many relationship

A directed line from the relationship set depositor to the entity set account specifies that a customer can deposit to many accounts. So is a one to many relationships.

Weak Entity Types

An entity set may not have sufficient attributes to form a primary key. Such an entity set is termed as weak entity set. An entity set that has a primary key is termed as strong entity set. For example: consider the entity set payment, which has three attributes: PaymentNumber, PaymentDate and paymentAmount. Although each payment entity is distinct, payment for different loans may share the same paymentNumber. Thus, this entity set does not have a primary key. Hence it is a weak entity set.

The primary key of weak entity set is formed by the primary key of the strong entity set on which the weak entity set is existence dependent, plus the weak entity set's discriminator.

In this case, the existence of entity payment depends on the existence of entity loan. If loan is deleted, its associated payment entities must be deleted. So, entity set loan is dominant and payment is subordinate. Discriminator of weak entity set is a set of attributes that can uniquely identify weak entities that are related to the same owner entity. For example: The discriminator of the weak entity set payment is the attribute payment Number. Since, for each loan, a paymentNumber uniquely identities one single payment for that loan. Hence, in the case of the entity set payment, its primary key is {loanNumber, PaymentNumber}

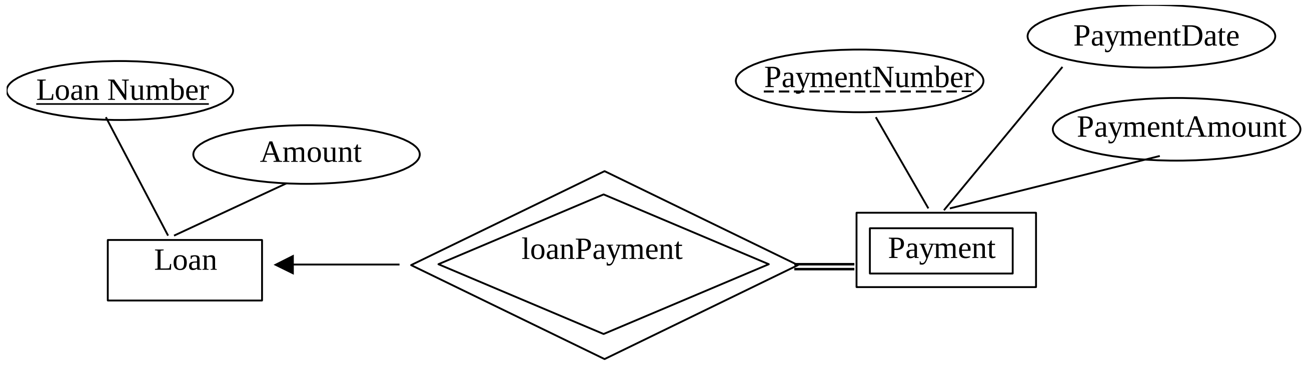


Fig. E-R diagram with a weak entity set.

Roles On Relationships

Each entity type that participates in a relationship type plays a particular role in the relationship. The role name signifies the role that a participating entity from the entity type plays in each relationship instance and helps to explain what the relationship means. For example, in the works_for relationship type, EMPLOYEE plays the role of employee or worker and DEPARTMENT plays the role of department or employer.

Role names are not technically necessary in relationship types where all the participating entity types are distinct, since each entity type name can be used as the role name. However, in some cases the same entity type participates more than once in a relationship type in different roles. In such cases, the role names become essential for distinguishing the meaning of each participations. Such relationships are called recursive relationships.

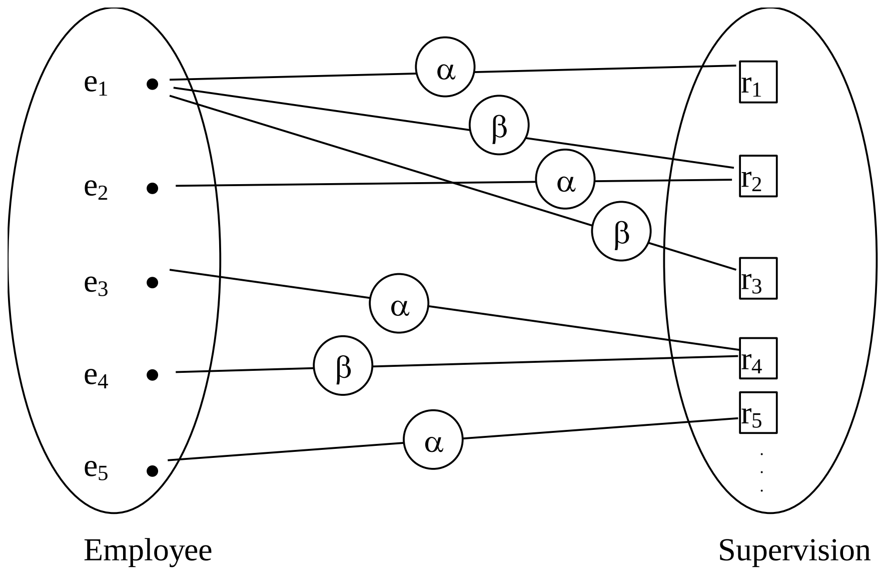


Fig.: Recursive relationship where employee entity type plays two roles

α - supervisee

β - Supervisor

The supervision relationship type relates an employee to a supervisor, where both employee and supervisor entities are members of the same EMPLOYEE entity type.

Structural Constraints On Relationships Types

Relationship types usually have certain constraints that limit the possible combinations of entities that may participate in the corresponding relationship set. These constraints are determined from the miniworld situation that the relationship represent.

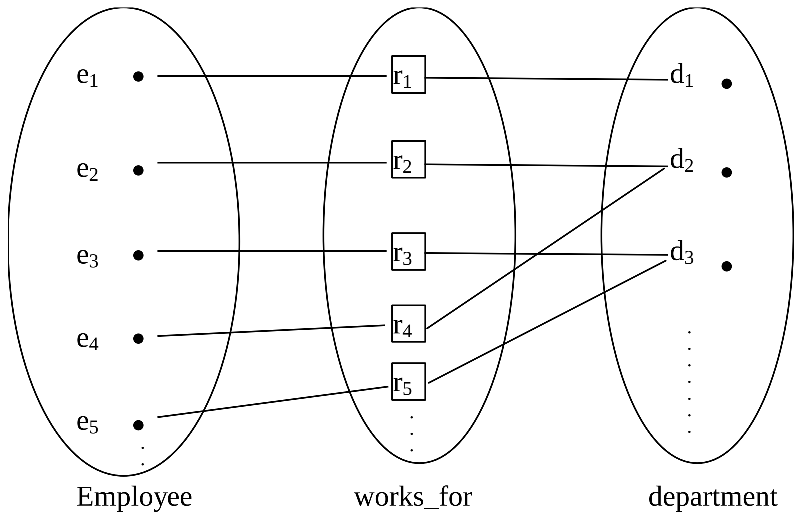


Fig.: Some instances of the works_for relationship between employee and department.

Suppose the company has rule that each employee must work for exactly one department.

There are two types of relationship constraints: Cardinality ratio and participation.

ii. Cardinality ratios for binary relationships: The cardinality ratio for a binary relationship specifies the number of relationship instances that an entity can participate in for example: in the works_for binary relationship type, department : employee is of cardinality ratio I:N, meaning that each department can be related to numerous employee but an employee can be related to only one department. The possible cardinality ratios for binary relationships types are 1:1, 1:N, N:1 and M:N.

ii. Participation constraints: The participation constraints specifies whether the existence of an entity depends on its being related to another entity via the relationship types. There are two types of participation constraints total and partial. If a Company policy states that every employee must work for a department, then an employee can exist only if it participants in a works_for relationship instance. Thus, the participation of employee in works_for is called total participation meaning that every entity in the total set of employee entity must be related to a department entity via works for. Total participation is also called existence dependency.

Cardinality ratio and participation constraints, taken together is called the structural constraints.

NAMING CONVENTIONS

The choice of names for entity types, attributes, relationship types and roles is not always straight forward. One should choose names that convey, as much as possible, the meanings attached to different constructs in the schema. We choose to use singular names for entity types, rather than plural ones, because the entity type name applies to each individual entity belonging to that entity type.

In E-R diagrams, we will use the convention that entity type and relationship type names are in uppercase, letters, attribute names are capitalized and roles names are in lowercase letters. Generally the nouns appearing in the narrative tend to give rise to entity type names and the verbs tend to indicate names of relationship types.

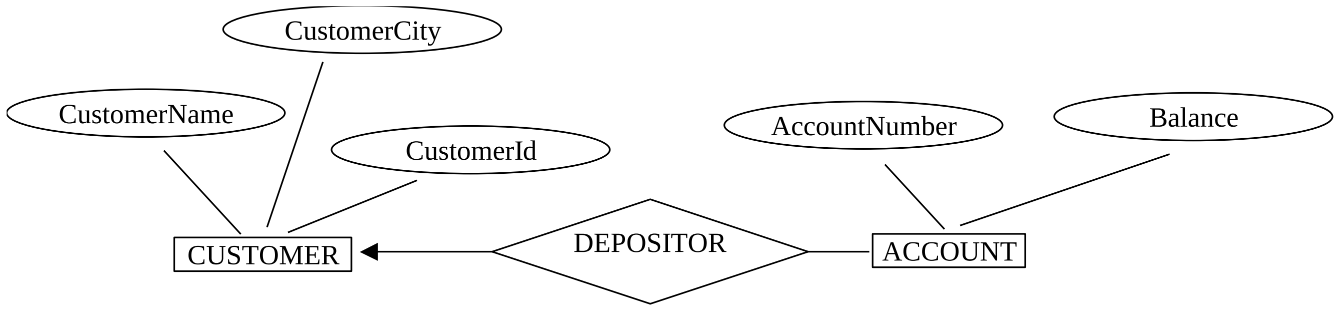


Fig. E-R diagram using Naming conventions

Another naming convention involves choosing relationship names to make the ER diagram of the schema readable from left to right and from top to bottom.

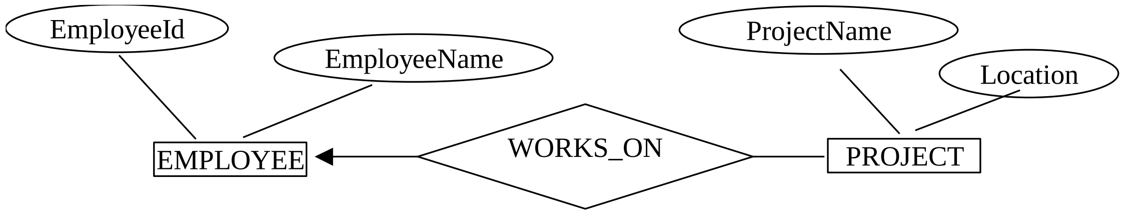


Fig. E-R diagram using convention for relationship Name from Left to right.

Integrity Constraints

Integrity constraints guard against accidental damage to database.

Entity Integrity

Entity integrity ensures that ***each row in the table is uniquely identified***. In other words, entity integrity ensures a table does not have any duplicate rows. Example: Two separate customers should not have the same customer number .SQL Server will allow duplicate rows if entity integrity is not enforced. Entity integrity is a key concept in the relational database model. Data in the relational database is independent of physical storage; there is no such thing as the '5th customer row' in a table. Physical independence is achieved by being able to reference each row by a unique value, sometimes referred to as a 'key'. Entity integrity ensures that each row in a table has a unique identifier that allows one row to be distinguished from another. Entity integrity is most often enforced by placing a ***primary key (PK)*** constraint on a specific column (although it can also be enforced with a UNIQUE constraint, a unique index, or the IDENTITY property) .The PK constraint forces each value inserted into a column (or combination of columns) to be unique; if a user attempts to insert a duplicate value into the column(s), the PK constraint will cause the insert to fail
A PK will not allow any Nulls to be inserted into the column(s) (A NULL entry would be disallowed even if it would be the only NULL in the column and therefore unique.) . A PK is referred to as a ' surrogate key' if the column contains no real data other than a uniqueness identifier .If 'real' data can be used as a PK (e.g., a social security number), then it is referred to as

an ' intelligent key' .There can be only one PK per table .A ***composite PK*** is a PK that consists of more than one column; it is used when none of the columns in the composite key is unique by itself .Thus, there can be only one PK in a table but the PK can consist of more than one column .If you need to enforce uniqueness on more than one column, use a PK constraint on one column and a UNIQUE constraint or IDENTITY property on any other columns that must not contain duplicates .Example: If the 'customer ID' column is the PK in the 'customers' table and you also want to make sure there are no duplicate customer names, you can place a UNIQUE constraint on the 'customer name' column . Non-PK columns on which uniqueness is enforced are referred to as ***alternative keys*** or AKs; they get their name from the fact that they are 'alternatives' to the PK and as such, make good candidates for indexing or 'joining' on.

Domain Constraint

A domain of possible values must be associated with every attribute SQL allows the domain declaration of an attribute to include the specification "not null" and thus prohibits insertion of a null value for this attribute. Any database modification that would cause a null to be inserted in a not null domain generates an error diagnostic. There are many situations where the prohibition of null values is desirable. A particular case where it is essential to prohibit null values is in the primary key of a relation schema.

The SQL-92 allows us to define domains using a create domain clause, as shown in the following example.

Create domain personName char (60)

We can then use the domain name personName to define the type of an attribute, just like a built-in domain.

Domain constraints are the most elementary form of integrity constraint. They are tested easily by the system whenever a new data item is entered into the database. It is possible for several attributes to have the same domain. The principle behind attribute domains is similar to that behind typing of variables in programming languages.

The check clause in SQL-92 permits the schema designer to specify a predicate that must be satisfied by any value assigned to a variable whose type is the domain. For instance, a check clause can ensure that an hourly wage domain allows only values greater that a specified value (such as minimum wage) as shown below.

Create domain hourlywage numeric (5, 2)

Constraint wage, valuetestcheck (value> = 4.00)

The domain hourlywage is declared to be a decimal number with a total of five digits, two of which are placed after the decimal point, and the domain has a constraint that ensures that the hourlywage is equal to or greater that 4.00.

The check clause can also be used to restrict a domain not to contain any null values, as shown below.

e.g.: create domain accountNumber char(10)

constraint accountNumber NullTest check (value not null)

create domain gender char (10)

constraint checkgendercheck (value in ("Male", "Female")

Referential Integrity

It is also required that a value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation. This condition is called referential integrity.

The referential integrity constraint is specified between two relations and is used to maintain the consistency among tuples of the two relations. Informally, the referential integrity constraint states that a tuple in one relation that refers to another relation must refer to an existing tuple in that relation. Consider the two relations EMPLOYEE and DEPARTMENT as follows.

EMPLOYEE

NAME	SSN	Address	Sex	Salary	Dept No.

DEPARTMENT

Dept No.	DeptName	MGRSSN

The attribute dept No. of EMPLOYEE gives the department Number for which each employee works. Hence, its value in every EMPLOYEE tuple must match the dept no. value of some tuple in the DEPARTMENT relation. To define referential integrity more formally, we must first define the concept of a foreign key. The conditions for a foreign key between two relation schemas R_1 and R_2 states that a set of attributes FK in relation schema R_1 is a foreign key of R_1 that references relation R_2 if it satisfies the following two rules.

i. The attributes in FK have the same domain as the primary key attributes PK of R_2. The attributes FK are said to reference or refer to the relation R_2.

ii. A value of FK in a tuple t_1 of the current state r_1 (R_1) either occurs as a value of PK for some tuple t_2 in the current state r_2 (R_2) or is null. In the former case, we have t_1 [FK] = t_2 [PK], and we say that the tuple t_1 references or refers to the tuple t_2. R_1 is called the referencing relation and R_2 is the referenced relation.

In a database of many relations, there are usually many referential integrity constraints. To specify these constraints, we must first have a clear understanding of the meaning or role that each set of attributes plays in the various relation schemas of the database.

In the EMPLOYEE relation the attribute deptNo refers to the department for which employee work hence, we designate deptNo to be a foreign key of EMPLOYEE, referring to the DEPARTMENT relation. This means that a value of deptNo in any tuple t_1 of the EMPLOYEE relation must match a value of the primary key of the department.

We can diagrammatically display referential integrity constraints by drawing a directed arc from each foreign key to the relation it references. For clarity, the arrowhead may point to primary key of the referenced relation.

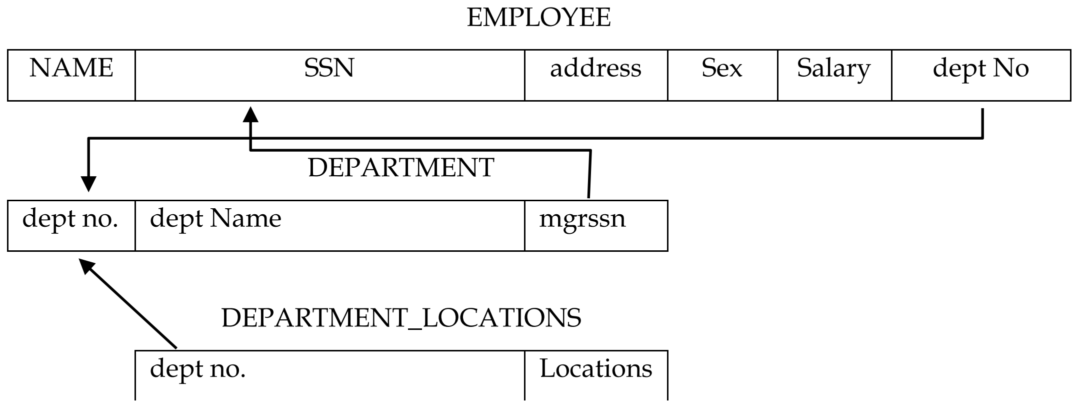


Fig. Referential integrity constraints

Referential integrity in SQL :

Primary and foreign key can be specified as part of the SQL create table statement.

- The primary key clause of create table statement includes a list of attributes that constitute a candidate key.
- The UNIQUE clause of create table statement includes a list of the attributes that constitute a candidate key.
- The FOREIGN KEY clause of create table statement includes both a list of attributes that constitute foreign key and the name of the relation referenced by the foreign key.

Assertion

An assertion is a predicate expressing a condition that we wish the database always to satisfy. Domain constraints and referential integrity constraints are special forms of assertions. However, there are many constraints that we can't express using only these special forms.

For example suppose the constraints are:

i. The sum of all loan amounts for each branch must be less than the sum of all account balances at that branch.

ii. Every loan has at least one customer who maintains an account with a minimum balance of 50,000.

An assertion in SQL-92 takes the form

```
create assertion <assertionName> check <Predicate>
e.g.:
create assertion sumConstraint check
(not exists (select * from branch
where (select sum(amount) from loan
where loan.branchName = branch.branchName)
>= (select sum (amount) from account
where account.branchName = branch.branchName)))
```

When an assertion is created, the system tests it for validity. If the assertion is valid, then any further modification to the database is allowed only if it does not cause that assertion to be violated.

Triggers

A trigger is a statement that is executed automatically by the system as a side effect of a modification to the database. Trigger must contain the following two requirements.

i. specify the conditions under which the trigger is to be executed.

ii. Specify the actions to be taken when the trigger executes.

Triggers are useful mechanisms for alerting humans, or for performing certain tasks automatically when certain conditions are met. Triggers are sometimes called rules or active rules. Triggers are written in both front end and backend. If the triggers are written in backend, they are called database triggers.

Types:

i. row level triggers
ii. statement level triggers
iii. before and after triggers

iv. database level triggers

Triggers can be written for events such as insert, update, delete, create, alter, drop etc.

For example suppose we want to store username and the system date into a table logdata. For this purpose, the trigger can be written as follows.

```
CREATE OR REPLACE TRIGGER tg_before_update_user
BEFORE INSERT OR UPDATE
ON Policies
FOR EACH ROW
BEGIN
          | INSERT INTO LOGDATA
          | VALUES (USER, SYSDATE); COMMIT;
END;
```

In this trigger, if any insert or update is made in the policies table, then the user name & current date is stored in the logdata table.