UNIT 1 RDBMS TERMINOLOGY

Structure

1.0 INTRODUCTION

A database system is essentially a sophisticated, computerized record keeping system, a repository for a collection of computerized data files. A database system maintains information and makes that information available on demand, for this purpose a database system provides set of facilities to perform such operations. For example, adding new empty files in database, inserting new data in existing files, retrieve data, update data, delete data, query the database etc. Thus we can say that a database system stores data and provides information related to data. Although some people use "data" and "information" synonymously, but some prefer to distinguish between two. Data refers to the values actually stored in the database and information refers to the meaning of those values as understood by user.

The benefits of database system over any traditional file system are very obvious as data in database (at least in large system), is integrated as well as shared, thus, a database eliminates redundancy and unnecessary repetition of data, and also as a consequence, database lets multiple users access the same piece of data.

The most important advantage of the database is to maintain the integrity i.e., it ensures that changes made to the database by the authorized users do not result in a loss of data consistency and guard against accidental damage to the database.

In this unit, we present a brief introduction to the principles of database system. We also compare the pros and cons of database system and file oriented system. As we know that the emphasis of this course is very much on the relational approach, thus, this unit covers the theoretical foundation of that approach– namely, the relational model. Please note that most of our examples are using a general syntax, which is close to SQL, but is not SQL, and at most places self-explanatory.

1.1 OBJECTIVES

At the end of this unit, you should be able to:

- describe database and database management system;
- compare database and file oriented approach;
- describe various features of database like data consistency and security;
- describe relational model; and
- describe relational data integrity.

1.2 SOME DEFINITIONS

You will be revisiting these definitions as they are most commonly referred in any reference notes on RDBMS.

1.2.1 Database

The collection of data, usually referred to as database, contains information about one particular enterprise. It is a collection of related information stored so that it is available to many users for different purpose. It is a kind of electronic file, which has a large number of ways of cross-referencing checks so that the user can reorganise and retrieve data in the way he wants.

1.2.2 Database-management System

A database-management system (DBMS) consists of a collection of interrelated data and a set of programs to access those data. The primary goal of the DBMS is to provide an environment that is both convenient and efficient to use in retrieving and storing database information. Thus we can say that, a database management system is a combination of hardware and software that can be used to set up and monitor a database, and can manage the updating and retrieval of database that has been stored in it.

Most DBMS have the following facilities/capabilities:

- Creating of a file, addition of data, deletion of data, modification of data; creation, addition and deletion of entire files.
- Retrieving data collectively or selectively.
- The data stored can be sorted or indexed at the user's discretion and direction.
- Various reports can be produced from the system. These may be either standardised reports or that may be specifically generated according to specific user definition.
- Mathematical functions can be performed and the data stored in the database can be manipulated with functions to perform the desired calculations.

- To maintain data integrity and database use.
- Data integrity for multiple users.
- Providing form based interface for easy accessibility and data entry.

The DBMS interprets and processes user's requests to retrieve information from a database. From the following figure we can depict that DBMS serves as an interface in several forms.

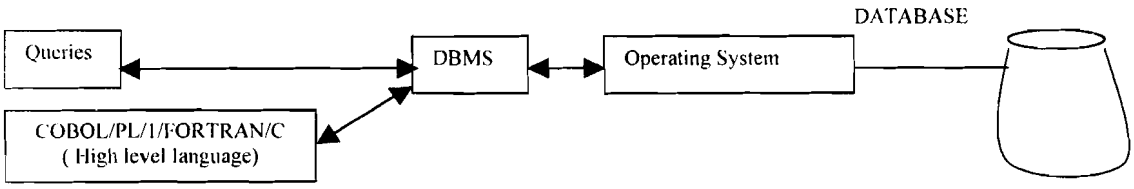


Figure 1: Database Management System

1.2.3 Instances and Schemas

Databases change over time as information is inserted and deleted. The collection of information stored in the database at a particular moment is called an instance of the database.
The overall design of the database is called the database schema. If schemas change at all, they change very infrequently.

A database schema corresponds to the programming-language type definition. A variable of a given type has a particular values at a given instant. Thus, the values of a variable in programming languages corresponds to an instance of a database scheme.

Database systems have general schemas, partitioned according to the levels of abstraction. At the lowest level is the physical schema, at the intermediate level is the logical schema, and at the highest level is a subschema. In general, database systems support one physical schema, one logical schema and several subschemas.

1.2.4 Traditional File Oriented Approach

The traditional file-oriented approach to information processing for each application has a separate master file and its own set of personal files. An organisation needs flow of information across these applications also, and this requires sharing of data, which is significantly lacking in the traditional approach. One major limitation of such a file-based approach is that the programs become dependent on the files and the files become dependent upon the programs.

Although such file-based approaches which came into being with the first commercial applications of computers did provide an increased efficiency in the data processing compared to earlier manual paper record-based systems as the demand for efficiency and speed increased, the computer-based simple file oriented approach to information processing started suffering from the following significant disadvantages.

Data redundancy and inconsistency: Different files may have different formats and the programs may be written in different programming languages as they are developed by different programmers. Moreover, the same information may be duplicated in several places (files). For example, the address and telephone number of a particular customer may appear in a file that consists of saving-account records and in a file that consists of checking-account records. This redundancy leads to higher storage and access cost. It may lead to data inconsistency; that is, the various copies of the same data may no longer agree.

Difficulty in accessing data: Suppose we need to access information about all the customers of a particular scheme. During the initial stages of development of the system this kind of query might not have been known, so no application program would be on hand to meet it. Say we have the application program that generates list of all the customers along with the scheme names. Thus we will have to run the latter program and sort the customers of particular scheme manually, or for each query we will have to write a new application program. Clearly we can see that accessing data is not easy in these cases.

Data isolation: Since data is scattered in various files, which might be in different formats thus, it is difficult to write a new application to retrieve appropriate data.

Integrity Problems: The data values stored in the database must satisfy certain types of consistency constraints. For example, the balance of a bank account may never fall below a prescribed amount (say, Rs. 500). Developers enforce these constraints in the system through hard-coding these conditions. When new constraints are added, it is difficult to change the program to enforce them.

Atomicity problems: Consider the situation in which during some transaction, say transferring money say Rs. 1000 from A's account to B's account, the power fails or the system goes down, leaving the state of database in an inconsistent state. Thus, the transactions should be atomic i.e., they must happen in their entirety or should not happen at all. It is difficult to ensure this property in a conventional file-processing system.

Security problems: Every person should not be allowed to access the database for security purposes. Since application programs are added to the system in an ad-hoc manner, it is difficult to ensure such security constraints.

1.2.5 Benefits of Conventional or Centralised DBMS

Let us discuss some specific advantages of the centralised control of data.

1. **Redundancy can be reduced:** As we know that in nondatabase systems (file oriented systems) each application has its own private files, because of which there exists considerable redundancy in stored data, resulting in wastage of storage space. Although database systems try to eliminate all possible redundancy, but we do not mean to suggest that all redundancy can be eliminated or necessarily should be eliminated. Sometimes there are sound business or technical reasons for maintaining several distinct copies of the same stored data. However, we do mean to suggest that any such redundancy should be carefully controlled i.e., the DBMS should be aware of it, if it exists, and should assume responsibility for "propagating updates".

2. **Inconsistency can be avoided to some extent:** Let us assume that in EMPLOYEE database, there exist two entries for the employee "ABC". This duplication is not only leads to redundancy but also leads to inconsistency as only one value of "ABC" might change during any modification of the data, hence two values in the database will not agree with each other. Clearly a database that is in an inconsistent state will supply incorrect or contradictory information to its users. It should be clearly noted that if the given fact is represented by a single entry, then such inconsistency can be avoided. Alternatively, if the redundancy is not removed but is controlled, then the DBMS could guarantee that the database is never inconsistent.

3. **The data can be shared:** With DBMS it is possible to share data in the database among various applications and also it is possible to satisfy the data requirements of new applications without having to create any additional stored data.

4. **Standards can be enforced:** With central control of database, the DBA (Database Administrator) can ensure that the data represents all applicable standards like corporate, installation, departmental, industry, national and international standards etc.

5. **Security restrictions can be applied:** A DBA can ensure that the database is accessed only through proper channels and by authorised users. For this purpose DBA defines some security rules, which are checked whenever there is an attempt to access the sensitive data.

6. **Integrity can be maintained:** The problem of integrity is the problem of ensuring that the data in the database is accurate and correct. Centralised control of DBMS can help in avoiding the problems of inconsistency as explained in the 2nd point, by permitting the DBA to define integrity rules to be checked whenever any data update, delete or insert operation is attempted.

7. **Conflicting requirements can be balanced:** Knowing the overall requirements of the enterprise – as opposed to the requirements of individual users – the DBA can so structure the system as to provide an overall service that is "best for the enterprise". For example, a representation can be chosen for the data in storage that gives fast access for the most important applications.

1.2.6 Data Independence

Most of the advantages of database systems, we have seen earlier, are fairly obvious. However, apart from these benefits i.e., reduced redundancy, maintaining integrity and consistency, sharing of data and balancing conflicting requirements, one further point, which might not be so obvious needs to be added to the list- namely, the provision of data independence. In fact we could say that rather than being an advantage, it is an objective for the database systems.

Before understanding data-independence, let us try to understand about data dependence.
Data dependence is the way in which the data is organised in secondary storage, and the technique for accessing it, are both dictated by the requirements of the application under consideration, and moreover that knowledge of that data organisation and that access technique is built into the application logic and code.

Let's have a file EMPLOYEE, indexed on field "Emp_No". In an older system, the application will know the fact that the index exists, and also knows the file sequence as defined by that index, and the internal structure of the application will be built around that knowledge. So, clearly such an application is data-dependent because it is impossible to change the storage structure (i.e., how the data is physically stored) or access technique (i.e., how it is accessed) without affecting the application, probably drastically.

In database systems, data dependence is undesirable for two reasons–

1. Different applications will need different views of the same data. Suppose their are two applications, A and B. Let A stores records in decimal and B stores records in binary. If application C using A and B is data dependent then C will not be able to integrate A and B. So, major work of programmers, done in applications A and B will be wasted.

   But, it will still be possible to integrate the two files and to eliminate the redundancy, provided the DBMS is ready and able to perform all necessary conversions between the stored representations chosen and the form in which each application wishes to see it.

2. The DBA must have the freedom to change the storage structure or access technique in response to changing requirements, without having to modify existing applications. If this doesn't happen then for slight change in the way of accessing data would lead to change the program completely. This is wastage of scarce and valuable resource.

Thus provision of data independence is a major objective of database system. Following the definition of data dependence, thus, data independence can be defined as "the immunity of applications to change in storage structure and access techniques"– which clearly implies that applications concerned do not depend on any one particular storage structure or access technique.

We can also say that data independence is the ability to modify a schema definition in one level without affecting a schema in the next higher level.

Let us consider data type conversion on a particular field on each access. In principle, the difference between what the application sees and what is actually stored might be quite considerable, but that difference should not affect the user accessing the fields, hence this difference should be transparent to the user. To amplify this remark, we need to see the aspects of database storage structure that might be subject to variation.

1. **Representation of numeric data:** A numeric data field might be stored in internal arithmetic form (e.g., in packed decimal) or as a character string. Either way, the DBA must choose an appropriate base (e.g., binary or decimal), scale (fixed or floating point), mode (real or complex), and precision (number of digits). Any of these aspects might be changed to improve performance or to conform to a new standard or for many other reasons.

2. **Representation of character data:** A character string field might be stored using any of several distinct coded character sets or "forms-of-use" (e.g., ASCII, EBCDIC).

3. **Units of numeric data:** The units of numeric field might change from inches to centimeters, for example, during a process of metrication.

4. **Data encoding:** We can be able to represent data in coded value, e.g., we can store "Red" as 1, "Blue" as 2 and "Green" as 3, etc., but the user should be able to access them as Red, Blue and Green only.

5. **Structure of stored records:** Two existing records might be combined into one, and alternatively, a single stored record type might be split into two.

6. **Structure of stored files:** A given stored file can be physically implemented in storage in a wide variety of ways. For example, it might be entirely contained within a single storage volume (e.g., a single disk), or it might be spread across several volumes on several different types; it might or might not be sorted based on certain attribute; it might be indexed or it might have one or more embedded pointer chains; it might or might not be accessible via hash-addressing.

None of the above considerations should affect applications in any way. The database should be able to grow without affecting existing applications.

There are two levels of data independence:

1. **Physical data independence** is the ability to modify the physical schema without causing application programs to be rewritten. Modifications at the physical level are occasionally necessary to improve performance.

2. **Logical data independence** is the ability to modify the logical schema without causing application programs to be rewritten. Modifications at the logical level are necessary whenever the logical structure of the database is altered. It is more difficult to achieve.

   The concept of data independence can be called as similar to abstract data types in modern programming languages. Both hide implementation's details from the users, to allow users to concentrate on the general structure, rather than two-level implementation details.

1.2.7 Data Dictionary

Database management system provide a facility known as data definition language (DDL), which can be used to define the conceptual scheme and also give some details about how to implement this scheme in the physical devices used to store the data. This definition includes:

- All entity sets and their associated attribute as well as the relationships among the entity sets.
- Any constraints that have to be maintained, including the constraints on the value that can be assigned to a given attribute and the constraints on the value assigned to different attributes in the same or different records.

These definitions, which can be described as metadata about the data in the database, are expressed in the DDL of the DBMS and maintained in a compiled form (usually as a set of tables). The compiled form of the definition is known as a **data dictionary**, **directory**, or **system catalog**. The data dictionary contains information on the data stored in the database and is consulted by the DBMS before any data manipulation operation.

Thus information pertaining to the structure and usage of data contained in the database, the metadata, is maintained in the data dictionary. The data dictionary is the database itself, documents the data. Each database user can consult the data dictionary to learn what each piece of data and the various synonyms of the data fields mean.

In an integrated system (i.e., in a system where the data dictionary is a part of the DBMS) the data dictionary stores information concerning the external, conceptual, and internal levels of the database. It contains the source of each data-field value, the frequency of its use, and an audit trail concerning updates, including the "who" and "when" for each update.

Currently data dictionary systems are available as add-on to the DBMS. Standards have yet to be evolved for integrating the data dictionary facility with the DBMS so that the two databases, one for metadata and the other for data, can be manipulated using a unified DDL/DML.

1.2.8 Database Security

Generally the terms "security" and "integrity" are considered to be same, but these two concepts are quite distinct. **Security** refers to the protection of data against unauthorised disclosure, alteration or destruction; **integrity** refers to the accuracy or validity of data. In other words,

- Security involves ensuring that users are allowed to do the things they are trying to do;
- Integrity involves ensuring that the things they are trying to do are correct.

Now the Question arises that why do we need Security?

The very obvious reason for the security is to prevent the misuse of the database by intentional (malicious) and accidental causes.

Accidental loss of data consistency may result from–

- Crashes during transaction processing;
- Anomalies caused by concurrent access to the database;
- Anomalies caused by the distribution of data over several computers;
- Logical errors that violate the assumption that transactions preserve the database consistency constraints.

The loss of data due to malicious access to data results from following:

- Unauthorised reading of data (theft of information);

- Unauthorised modification of data ;
- Unauthorised destruction of data.

The various aspects to the security problem are:

1. **Legal, social and ethical aspects** (for example, does the person making the request, say for withdrawing money from a bank, has legal rights for making the requested transaction?)
2. **Physical controls** (for example, is the computer or terminal room always locked or otherwise guarded.)
3. **Policy Questions** (for example, how does the enterprise owning the system decide who should be allowed access and to what?)
4. **Operational problems** (for example, is a password scheme is used, how are the passwords themselves kept secret? how often are they changed?)
5. **Hardware controls** (for example, does the processing unit provide any security features, such as the storage protection keys or a privileged operation mode?)
6. **Operating system security** (for example, does the underlying operating system erase the contents of storage and data files when they are finished with?)

The unit of data or "data object" that might need to be protected can range all the way from an entire database or collection of relations on the one hand, to the specific data item at a specific attribute position within a specific tuple within a specific relation on the other.

There are two approaches to security and control, discretionary and mandatory control.

In **discretionary control**, a given user will typically have different access rights on different objects. Different users will typically have different rights on the same object. Discretionary schemes are, thus, flexible. Discretionary control can be compared with the access control in the UNIX operating system. In UNIX file system, access permissions of user, group and others (ugo) are associated with each file. All three (ugo) has the combination of read, write and execute permission on the file.

In **mandatory control**, by contrast, each data object is tagged or labeled with a certain classification level, and each user is given a certain clearance level. A given data object can then be accessed only by users with the appropriate, clearance. Mandatory schemes are, thus, comparatively rigid.

Most DBMS support either discretionary control or mandatory control or both. In fact most of the DBMS support discretionary control, and some systems support mandatory control as well, thus, discretionary control is, thus, more likely to be encountered in practice.

Audit Trail

The DBMS has certain routines that maintain audit trail or a journal. An audit trail or a journal is a record of an update operation made on the database. Audit trails may be used to trace the occurrence of an incorrect activity. The audit trail records–

Who (user or the application program and a transaction number),

When (time and date),

(from) **where** (location of the user and/or the terminal), and

What (identification of the data affected, as well as before-and-after image of that portion of the database that was affected by the update operation).

In addition, a DBMS contains routines that make a backup copy of the data that is modified. The current SQL standards support only the discretionary access control.

1.3 THE DEFINITIONS RELATED TO RELATIONAL MODEL

We define RDBMS as a system in which, at a minimum:

1. The data is perceived by the user as relations (and nothing but relations); and

2. The operators are at the user's disposal, for example, for data retrieval – the operators that generate new relations from old, include at least **SELECT**, **PROJECT**, and **JOIN**.

The Relational model is a way of looking at data– that is, it is a prescription for a way of representing data (namely, by means of tables), and a prescription for a way of manipulating such a representation (namely, by means of operators such as JOIN). More precisely, the relational model is concerned with three aspects of data: **data structure, data integrity,** and **data manipulation**.

The data structure aspect of RDBMS deals with defining the tables; it's various attributes etc. Tables are nothing but the logical structure in a relational system, not the physical structure, we can also say that tables represent an abstraction of the way the data is physically stored – an abstraction in which numerous storage-level details, such as stored record placement, stored record sequence, stored data encoding, stored record prefixes, stored access structures such as indexes, and so forth, are all hidden from user.

The data manipulation deals with the way data is retrieved from the database. We use the SQL query for the purpose of retrieving and manipulating data.

All likelihood, the database is always subject to numerous integrity rules; for example, employee salaries might have to be in the range of Rs. 10,000 to Rs. 40,000.

1.3.1 Domain Definition

For defining the concept of domain, we first need to discuss the term scalar. The individual data value, for example, individual employee name in the **EMP** Table, can be called as the **scalar**, thus we can say that Scalar value represent "the smallest semantic unit of data" in the sense that they are atomic as they are non-decomposable as far as Relational DBMS is concerned.

We define a domain to be a set of scalar values, all of the same type necessarily. For example, the domain of Employee Names is the set of all the employees in an organisation listed on the basis of their names.

Thus, domain is the pool of values, from which actual attributes are drawn. More precisely, each attribute must be "defined on" exactly one underlying domain, meaning that values of that attribute must be taken from a domain.

Most of the today's relational systems do not really support domains, even though they are actually one of the most fundamental ingredients of the entire relational model.

One question which **naturally** should come in our mind is that, what is the significance of the domains?

The importance of domains is in domain-constrained comparisons.

Let us consider three tables:

S (S#, SNAME, STATUS, CITY) which describes about Supplier
P (P#, PNAME, COLOUR, WEIGHT, CITY) which describe about Parts
SP (S#, P#, QTY) which describes about shipments.

Now let us consider two queries,

```
SELECT      .....
FROM  P, SP
WHERE       P.P# = SP.P#;
```

```
SELECT      .....
FROM  P, SP
WHERE       P.COLOUR = SP.QTY;
```

Of these two queries, the one on the left makes sense as it is the comparison of two attributes P.P# and SP.P# which are defined on the same domain. On the contrary the comparison on the left is comparison of two attributes P.COLOUR and SP.QTY, which are not defined on same domains, hence it is not of much sense. Hence operators based on comparison like join, union etc. will fail for the query on the right.

The above example is described using SQL to describe the concept of domain, although concept of domain is primarily conceptual. Sometimes the SQL query might be OK yet conceptually wrong for example, consider the SQL,

```
SELECT      .....
FROM  P, SP
```

WHERE P.WEIGHT = SP.QTY; Here since both WEIGHT and QTY are real numbers, and they could be equated, but logically comparison of these two entities might not be useful.

In broad sense we can say that a domain is nothing more or less than a data type, which could be built-in or user-defined but has logical implications.

1.3.2 A Relation

Till now we are considering relations analogical to tables itself. We might use them interchangeably but conceptually there is slight difference between both of above two. For that let us first define Relation based on domains.

A relation, R, say on a collection of domains D1, D2, ...Dn – not necessarily all distinct – consists of two parts, a heading and a body. In elaborate:

1. The **heading** consists of a fixed set attributes, or more precisely <attribute-name:domain-name>pairs, { <A1:D1>, <A2:D2>, ..., <An:Dn> }, such that each attribute Aj corresponds to exactly one of the underlying domains Dj (j = 1,2,...., n). The attribute names A1, A2, ..., An are all distinct.

2. The **body** consists of a set of **tuples**. Each tuple in turn consists of a set of <attribute-name:attribute-value> Pairs

{<A1:vi1>, <A2:vi2>, ..., <An:vin> } i = 1, 2, ... m, where m is the number of tuples in the set.

The value 'm' i.e., number of tuples is called the cardinality and 'n' i.e. the number of attributes is called degree, of relation R.

Now try to differentiate between the concepts of relation and table.

So on analysing the above definition we can say that relation is an abstract kind of object and table is a reasonable concrete picture of such an abstract data type. But there are certain cases, which suggest that tabular representation or tables suggests something, which are not true with relations. For example, it clearly suggests that the rows of the table (i.e., tuples of the relation) are in a certain top-to-bottom order, but they are actually not.

Properties of Relations

- They do not contain duplicate tuples.
- Tuples in relations are unordered;
- Attributes in relations are unordered;
- All attribute values in a relation are atomic.

Kinds of Relations

Named relation: A named relation is a relation that has been defined to the DBMS. In SQL syntax we define relations by means of CREATE BASE RELATION or CREATE VIEW or CREATE SNAPSHOT etc.

Base Relation: A base relation is a named relation that is not a derived relation i.e., they are autonomous. We can manipulate the conceptual or "physical" relations stored in the database. By "physical" we mean that the relation corresponds to some stored data. This data may not be stored as a table and may actually be split horizontally or vertically and reside on one or more storage devices (at one or more sites). Such conceptual relations are referred as base relations.

Derived Relation: A derived relation is a relation that is defined by means of some relational expressions, in terms of other named relations. Ultimately they are derived from the base relations only. For example, view is a derived relation.

Views

For security and other concerns it is undesirable that all users should be able to see the entire relation, thus, it would be beneficial if we could create useful relations for different group of users, rather allowing all users to manipulate the base relation. Therefore, relation that is not a part of the physical database, i.e., a virtual relation, is made available to the users as view. To be useful, we assign the view a name and relate it to the query expression as

Create view <view name> **as** <query expression>

Example

Let EMPLOYEE be the base relation. We create the table for EMPLOYEE as follows:

Create table EMPLOYEE

```
(Emp_No      integer not null,
 Name        char(25),
 Skill       char(20),
 Sal_Rate    decimal(10,2),
 Dob         date,
 Address     char(100) )
```

For reasons of confidentiality, not all users are permitted to see the Sal_Rate of an employee. For such users the DBA can create a view, for example, EMP_VIEW defined as:

Create view EMP_VIEW **as**
(**select** Emp_No, Name, Skill, Dob, Address
from EMPLOYEE)

Limitations of Views

Although views are a useful tool for queries, they represent 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 must be translated to a modification to the actual relations in the logical model of the database.

Since views represent only the selected attributes from the various relations which are abstracted as per our requirement. If we perform any operation (insert, delete or update) in the view, then we will perform the same operation in the base relations also maintaining both the integrity constraints. This task is very difficult to achieve, thus modifications are generally not permitted on view relations, except in limited cases. The general problem of database modification through views has been the subject of substantial research.

Views and Security

Views can be used to provide security mechanism. An example using SQL statements is:

CREATE VIEW SECVIEW AS
SELECT S.S#, S.SNAME, S.CITY
FROM S;

The view defines the data over authorisation is to be granted. The granting itself is done by means of the GRANT statement, e.g.,

GRANT SELECT ON SECVIEW TO Saurabh, Anita, Bharat;

Thus, allowing only limited portion available to the users. These grant permissions can also be revoked using REVOKE command.

Check Your Progress 1

Question 1. What are the advantages of using a database system?

.....

.....

.....

Question 2. What are the disadvantages of using a database system?

.....

.....

.....

Question 3. Consider a database PART_DETAIL:

MAJOR_PART#	MINOR_PART#	QUANTITY
P1	P2	2
P1	P3	4
P2	P3	1
P2	P4	3
P3	P5	9
P4	P5	8
P5	P6	3

(a) Write a CREATE BASE RELATION and appropriate set of CRETAE DOMAINS the above relation.

(b) Assuming this relation is included in some other database, say supplier-and-part, show the updates which system make to the catalog (data dictionary) to reflect your answer to part (a).

(c) Write an appropriate set of DESTROY statements to cause the catalog updates of part (b) to be undone.

.....

.....

Question 4. A relation is defined to have a set of attributes and a set of tuples. Now, in mathematics the empty set is a perfectly respectable set; Indeed, it is usually desirable that results, theorems, etc., that hold true for a set of n elements, should also hold true for a set of n = 0. Can a relation have an empty set of tuples? Or an empty set of attributes?

.....

.....

Question 5. Base relation STUDENT looks like:

STUDENT (STUDENT_ID, SEX, PART_TIME_JOB, STIPEND, COURSE, FEES)
PRIMARY KEY (STUDENT_ID)

Write security rules in the following statements using some pseudo-query approach.

(i) User Anu RETRIEVE privileges over the entire relation.
(ii) User Smita INSERT and DELETE privileges over the entire relation.
(iii) Each user RETRIEVE privileges over the user's own tuple (only).
(iv) User Nath RETRIEVE privileges over the entire relation and UPDATE privileges over the STIPEND and FEES attribute (only).
(v) User Ram RETRIEVE privileges over the STUDENT_ID, STIPEND and FEES attributes only.
(vi) User John RETRIEVE privileges as for Ram and UPDATE privilege over the SALARY and FEES attributes (only).

.....

.....

1.4 RELATIONAL DATA INTEGRITY

What do we mean by database consistency?

At any given time, any given database contains some particular configuration of data values, and of course that configuration is supposed to "reflect reality ".

We should know first that what is integrity? Integrity simply means to maintain the consistency of data, thus Integrity constraints provide a means of ensuring that changes made to the database by authorised users do not result in a loss of data consistency. Thus, integrity constraints guard against damage to the database.

The integrity constraints are of two types:

Key declarations: The stipulation that certain attributes form a candidate key for a given entity set. The set of legal insertions and updates is constrained to those that do not create two entities with the same value on a candidate key.

Form of a relationship: Many-to-many, one-to-many, one-to-one. A one-to-one or one-to-many relationship restricts the set of legal relationships among entities of a collection of entity sets.

1.4.1 Candidate Keys

Let R be a relation. Then a candidate key for R is a subset of the set of attributes of R, say K, such that:

Uniqueness property:	No two distinct tuples of R have the same value for K
Irreducible property:	No proper subset of K has the uniqueness property.

Every relation does have at least one candidate key, because relations do not contain duplicate tuples. A candidate key could be combination of many attributes also, if necessary. For example, if we say that the attribute combination {S#, P#} is a candidate key for base relation SP, we do not mean merely that no two tuples in the relation that happens to be the current value of SP have the same value for that combination; rather, we mean that all possible relations R that can be values of SP are such that no two distinct tuples of R have the same value for that combination.

Properties of a candidate key

1. They are defined on set of attributes. For example, {S#} is a candidate key for relation S etc.

2. A candidate that involves more than one attribute is said to be composite. A candidate key that involves exactly one attribute is said to be simple.

3. The requirement for irreducibility needs a little elaboration. Basically the point is that if we were to specify a "candidate key" that was not irreducible, the system would not be aware of true state of affairs, and thus would not be able to enforce the associated integrity constraint properly. For example, suppose we were to define the combination {S#, CITY} instead of S# alone, as a candidate key for the base relation S. Then the system will not enforce the constraint that supplier numbers are "globally" unique: instead, it will enforce only the weaker constraint that supplier numbers are "locally" unique within city. Thus, true candidate keys do not include any attributes that are irrelevant for unique identification purposes.

4. Finally, logical notion of candidate key should not be confused with the physical notion of a "unique index".

Candidate keys are important because they provide the basic tuple-level addressing mechanism in a relational system. For example if S# is a candidate key for the relation S then the expression, S WHERE S# = 'DEL0001' is guaranteed to yield at most one tuple. If SNAME is also the candidate key for the relation S, then the expression S WHERE SNAME = 'RAM' will also yield at most one tuple.

By contrast if we query on non-candidate key attribute then we end up yielding unpredictable number of tuples. For example, the expression S WHERE CITY = "DELHI" will yield unpredictable number of tuples.

1.4.2 Primary Key

The primary key is the kind of candidate key that is chosen by the database designer as the principal means of identifying entities within an entity set. The remainder, if any, is then called **alternate keys**.

1.4.3 Foreign Keys

Let R2 be a base relation, then a foreign key in R2, is a subset of the set of attributes of R2, say FK, such that:

1. There exists a base relation R1 (R1 and R2 not necessarily distinct) with a candidate key CK, and

2. For all time, each value of FK in the current value of R2 is identical to the value of CK in some tuple in the current value of R1.

Foreign Keys are also defined on set of attributes. For example, the foreign keys in relation SP in the suppliers-and-parts database are {S#}(here the base relation R1 is relation S) and {P#}(here the base relation R1 is relation P). Every value of a given foreign key is required to appear as a value of the matching candidate key. There is no requirement that a foreign key be a component of the primary key-or of some candidate key- of its containing relation.

Example

An employee may perform different roles in different projects, say RAM is doing coding in one project and designing in other. So, let's have relations EMPLOYEE and PROJECT. Let ROLE be the relationship describing the different roles required in the any project.

The relational scheme for the above three relations are:

EMPLOYEE (EMP#, Name, Designation)
PROJECT (PROJ#,Proj_Name, Details)
ROLE (ROLE#, Role_descrption)

In the above relations EMP# and PROJ# are unique and not null, respectively. As we can clearly see that we can identify the complete instance of the entity set employee through the attribute EMP#, thus EMP# is the primary key of the relation EMPLOYEE, similarly PROJ# is the primary key for the relation PROJECT.

Let ASSIGNMENT is a relationship between entities EMPLOYEE and PROJECT and ROLE, describing that which employee is working on which project and what is the role of the employee in the respective project. A possible representation of this relationship is by using the entities involved in the relationship.

ASSIGNMENT (Employee, Project, Role)

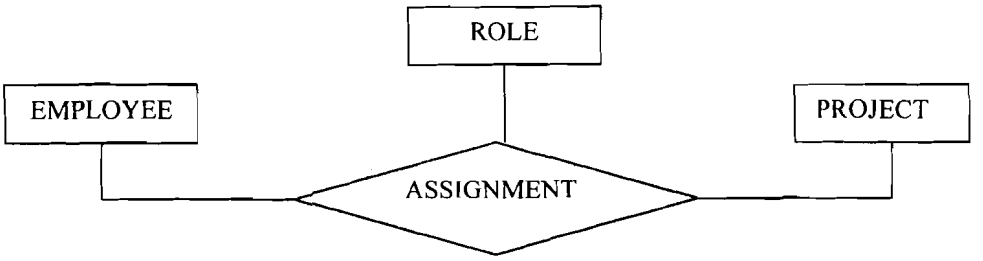


Figure 2: E-R diagram for employee role in development team

Let us consider sample relations:

EMPLOYEE

Emp#	Name	Designation
101	RAM	Analyst
102	SITA	Receptionist
103	ARVIND	Manager

PROJECT

PROJ#	Proj_name	Details
TCS	Traffic control System	For traffic shaping.
LG	Load Generator	To simulate load for input in TCS.
B++1	B++_TREE	ISS/R turbo sys

ROLE

ROLE#	Role_decription
1000	Design
2000	Coding
3000	Marketing

ASSIGNMENT

EMP#	PROJ#	ROLE#
101	TCS	1000
101	LG	2000
102	B++1	3000

Using the unique identification properties of keys, we can replace the Employee, Project and role entities in ASSIGNMENT by their keys. The keys act as surrogates for their respective entities. Thus, we can define relational scheme for the relation ASSIGNMENT as follows:

ASSIGNMENT (EMP#, PROJ#, ROLE#)

It is clear that ASSIGNMENT is a relation that establishes a relationship among three "owner" relations. Such a relation may be thought of as an associative relation. The key of the associative relation is always the union of the key attributes of the owner relation. Thus, the key of the relation ASSIGNMENT is the combination of the attributes EMP#, PROJ# and ROLE#.

The attributes EMP#, PROJ# and ROLE# in the relation ASSIGNMENT are foreign keys.

Now after knowing the concept of foreign key, we can proceed to know about the actual integrity constraints namely Referential Integrity and Entity Integrity.

1.4.4 Referential Integrity

The database must not contain any unmatched foreign key values

The term "unmatched foreign key value" here means a foreign key value for which there does not exist a matching value of the relevant candidate key in the relevant target relation. In other words, the rule simply says: If B references A, then A must exist.

Database modifications can cause violations of referential integrity. We list here the test we must make for each type of database modification to preserve the following referential-integrity constraint:

$\pi_\alpha (r_2) \subseteq \pi_k (r_1)$, here α is the foreign key (FK) and k is the key of relation R1(CK). π is the projection operator, r_1 and r_2 are the instance of relation R1 and R2 respectively.

Delete

In SQL query **delete** statement takes the form

DELETE target;

The target is a relational expression: all tuples in a target relation are deleted. Here is an example:

DELETE S WHERE STATUS < 20 ;

The target will often be just a restriction of some named relation.

A very obvious question arises in our mind that what should happen on an attempt to delete the target of a foreign key reference?

For example, an attempt to delete a supplier for which there exists at least one matching shipment? In general there are at least two possibilities:

RESTRICTED – The delete operation is "restricted" to the case where there are no such matching shipments (it is rejected otherwise)

CASCADE – The delete operation "cascades" to delete those matching shipments also.

So, if a tuple t_1 is deleted from r_1, the system must compute the set of tuples in r_2 that reference t_1

$\sigma_{\alpha = t1[k]} (r_2)$

If this set is not empty, either the delete command is rejected as an error, or the tuples that reference t_1 must themselves be deleted. The latter solution may lead to cascading deletions, since tuples may reference tuples that reference, t_1 and so on.

Insert

The **insert** statement takes the form

INSERT source INTO target;

Where source and target are relational expressions that evaluate to type-compatible relations (in practice, the target will usually be just a named relation). The source is evaluated and all tuples of the result are inserted into the target. Here is an example:

INSERT (S WHERE CITY = "London") INTO TEMP;

TEMP here is assumed to be type-compatible with relation S.

If a tuple t_2 is inserted into r_2, the system must ensure that there is a tuple t1 in r1 such that $t_1[k] = t_2[\alpha]$. That is,

$t_2[\alpha] \in \pi_k(r1)$

Update

The **update** statement takes the form

UPDATE target assignment-commalist ;

Where each assignment is of the form

Attribute := scalar-expression

The target is a relational expression, and each attribute is an attribute of the relation that results from evaluating that expression. All tuples in the target relation are updated in accordance with the specified assignments. Here is an example:

UPDATE P WHERE COLOR = 'Red'

CITY := 'Paris' ;

In practice the target will often be just a restriction of some named relation.

But, what should happen on an attempt to update a candidate key that is the target of a foreign key reference? – For example, an attempt to update the supplier number for a supplier for which there exists at least one matching shipment? In general there are the same possibilities as for DELETE:

RESTRICTED – the update operation is "restricted" to the case where there are no matching shipments (it is rejected otherwise).

CASCADES – The update operation "cascades" to update the foreign key in those matching shipments. Also we must consider two cases for update: updates to the referencing relation (r_2), and updates to the referenced relation (r_1).

If a tuple t_2 is updated in relation r_2, and the update modifies values for the foreign key α, then a test is made which should be similar to test in insert case. The system must ensure that $t_2[\alpha] \in \pi_K(r_1)$

If a tuple t_1 is updated in relation r_1, and the update modifies values for the primary key [K], then a test is made which should be similar to delete case. The system must compute $\sigma_{\alpha=t1[k]}(r_2)$ using the old value of t_1 (the value before the update is applied). If this set is not empty, the update is rejected as an error, or the update is cascaded in a manner similar to delete.

Now before understanding second type of Integrity constraint viz., Entity Integrity, we should be familiar to the concept of **NULL.**

Basically, nulls are intended as a basis for dealing with the problem of missing information. This kind of situation is frequently encountered in the real world. For example, historical records such as "Date of birth unknown", or police records may include the entry "Present whereabouts unknown." Hence it is necessary to have some way of dealing with such situation in our formal database systems, thus, Codd proposed an approach to this issue that makes use of special markers called nulls to represent such missing information.

A given attribute in the table might or might not be allowed to contain nulls – it will depend on the definition of the attribute in question, at least for base relations. If a given attribute is allowed to contain nulls, and a tuple is inserted into the relation and no value is provided to that attribute, the system will automatically place a null in that position. If a given attribute is not allowed to contain nulls, the system will reject any attempt to introduce a null into that position.

1.4.5 Candidate Keys and Nulls

The relational model has historically required that exactly one candidate key be chosen as the primary key for the relation in question, the remaining candidate keys, if any, are then alternate keys. So along with the primary key concept, the model has included the following rule known as **Entity Integrity Rule**, which states, no component of the primary key of a base relation is allowed to accept nulls. In other words, the definition of every attribute involved in the primary key of any base relation must explicitly or implicitly include the specifications of NULLS NOT ALLOWED.

Foreign Keys and Nulls
Let us consider the relation:

DEPT		
DEPT#	DNAME	BUDGET
D1	Marketing	10M
D2	Development	12M
D3	Research	5M

EMP			
EMP#	ENAME	DEPT#	SALARY
E1	Rahul	D1	40K
E2	Aparna	D1	42K
E3	Ankit	D2	30K
E4	Sangeeta		35K

Suppose that Sangeeta is not assigned any Department, thus, in the EMP tuple corresponding to Sangeeta, therefore, there is no genuine department number that can serve as the appropriate value for the DEPT# foreign key, thus, one cannot determine DNAME and BUDGET. This may be a real situation where the person has newly joined and is undergoing training and will be allocated to a department only on completion of the training. Thus, nulls in foreign key values may not be logical error.

So, based on Nulls we redefine Foreign Keys. Let R2 be a base relation, then a foreign key in R2 is a subset of the set of attributes of R2, say FK, such that:

- There exists a base relation R1 (R1 and R2 not necessarily distinct) with a candidate key CK, and
- For all time, each value of FK in the current value of R2 **either is null** or is identical to the value of CK in some tuple in the current value of R1.

Here referential integrity rule – "The database must not contain any unmatched foreign key values"– remains unchanged, but the meaning of the term "unmatched foreign value" is extended to refer to nonnull foreign key value for which there does not exist key value of the relevant candidate key in the relevant target relation

Check Your Progress 2

Question 1. Consider supplier-part-project database;

S

S#	SNAME	CITY
S1	Smita	Delhi
S2	Jim	Pune
S3	Ballav	Pune

P

P#	PNAME	COLOUR	CITY
P1	Nut	Red	Delhi
P2	Bolt	Blue	Pune
P3	Screw	White	Bombay

S

S#	SNAME	CITY
S4	Sita	Delhi
S5	Anand	Agra

P

P#	PNAME	COLOUR	CITY
P4	Screw	Blue	Delhi
P5	Cam	Brown	Pune
P6	Cog	Grey	Delhi

J

J#	JNAME	CITY
J1	Sorter	Pune
J2	Display	Bombay
J3	OCR	Agra
J4	Console	Agra
J5	RAID	Delhi
J6	EDS	Udaipur
J7	Tape	Delhi

SPJ

S#	P#	J#	QUANTITY
S1	P1	J1	200
S1	P1	J4	700
S2	P3	J2	400
S2	P2	J7	200
S2	P3	J3	500
S3	P3	J5	400
S3	P4	J3	500
S3	P5	J3	600
S3	P6	J4	800
S4	P6	J2	900
S4	P6	J1	100
S4	P5	J7	200
S5	P5	J5	300
S5	P4	J6	400

Using the sample data values in above table, tell the effect of each of the following operation:

1. UPDATE project J7, setting CITY to Nagpur.
2. UPDATE part P5, setting P# to P4.
3. UPDATE supplier S5, setting S# to S8, if the relevant update rule is restricted.
4. DELETE supplier S3, if the relevant rule is CASCADES.
5. DELETE part P2, if the relevant delete rule is RESTRICTED.
6. DELETE project J4, if the relevant delete rule is CASCADES.
7. UPDATE shipment S1-P1-J1, setting S# to S2. (shipment S1-P1-J1 means that in SPJ table the value for attributes S#, P# and J# is S1, P1 and J1 respectively)
8. UPDATE shipment S5-P5-J5, setting J# to J7.
9. UPDATE shipment S5-P5-J5, setting J# to J8
10. INSERT shipment S5-P6-J7
11. INSERT shipment S4-P7-J6
12. INSERT shipment S1-P2-jjj (where jjj stands for a default project number).

.....

.....

.....

.....

Question 2. The following two relations represent a database containing information about departments and employees:

DEPT (DEPT#, ... , MGR_EMP#, ...)
EMP (EMP#, ..., DEPT#, ...)

Every department has a manager (MGR_EMP#); every employee has a department (DEPT#). Try to show the relationship diagrammatically and write a suitable database definition for this database.

.....

.....

.....

Question 3. The following two relations represent a database containing information about employees and programmers.

EMP (EMP#, ..., JOB, ...)
PGMR (EMP#, ..., LANG, ...)

Every programmer is an employee, but the converse is not true. Write a suitable database definition for this database.

.....

.....

.....

1.5 DATA DICTIONARY CHECKLIST

This section is just for reference. You must visit this section again after you have gained experience with commercial database management systems.

Data dictionaries are used as the notation for representing structure of data items. So, we can say that data dictionaries are the data about the data. In data dictionaries we need to express;

Composition (sequence?) – how an item is made up of simpler ones (its attributes).

Repetition – items which are repeated in (e.g.) lists, arrays, etc.

Selection – values for items chosen from alternatives.

Optionality – items, which are not always present.

Symbols used in DD Notation names a meaningful name for every composite or basic data item

Symbols	Meanings in DD Notation
=	'is defined as', or 'is made up of'
+	'and'
{}	Zero or more of whatever's inside i.e., repetition
n{}m	between n and m (inclusive)
[||]	one of the listed attributes is present
()	item inside is optional
" "	enclose literals (actual values)
* *	enclose comments - define meaning of data, informally.

Let us take an example of the Tutorial List in a university put up on the Notice Board.

The data on this consists of:

- a Title
- a Version Number
- a Date
- a day and time and room and a list of students' names
- another day and time and room and list of students' names
- (yet) another day and time and room and list of students' names.

In this example, we have composition and repetition. But we need to know that how much repetition is required for the day, time and room of the tutorial.

Tutorial List Example

TutorialList = Title + VersionNumber + Date + {TutorialDetails}

TutorialDetails = DayOfWeek + TimeSlot + Room + StudentList

StudentList = {FamilyName + FirstName}

or.....

TutorialList = Title + VersionNumber + Date +
{DayOfWeek + TimeSlot + Room + {FamilyName + FirstName}}

VersionNumber = Digit + "."+ Digit

Digit = ["1"|"2"|"3"|"4"|...]

TimeSlot = StartHour + "-" + EndHour

StartHour = ["9"|"10"|"11"|"12"...]

SDD Course Plan

CoursePlan = DateOfPlan + VersionNumber + Titles + {WeekDetails}

WeekDetails = WeekNo. + StartDate + [TeachingWeek |NonTeachingWeek]

NonTeachingWeek = ["admin week"| "induction week"| "student centred learning"|..]

TeachingWeek = 2{LectureDetails}2 + (TutorialDetails) + (PracticalWork)

LectureDetails = *Description of Lecture Content*

StartDate = Date

DateOfPlan = Date

Date = *date in format "dd-mmm-yy"*

Data Dictionary - Conclusions

- Simple way of describing syntax of composite data.

- Meaning or Semantics captured informally, by: 'Meaningful' names for data.
- Comments explaining constraints and usage.
- Wide range of uses - any situation in which composite data is found:
  - particularly data 'on the move'.
  - particularly data crossing the system boundary.
  - customers' source documents.
  - input data on data entry forms.
  - output data on reports/screens, etc.
  - Internal data needs more attention.....

Check List for Data Dictionary Review

Since these days more and more projects are implemented on Relational Database Management System (RDBMS) platform, thus we must have a checklist for the review of Data Dictionary on such a platform.

By adding few more questions, this checklist can be used to evaluate a Data Dictionary.

In the Data Dictionary, entity maps to a Database table and the attributes correspond to columns (field) of the table.

1. Each entity should have a primary key, which uniquely identifies a record in the table. This primary key can either be system-generated sequence or user defined unique value.
2. Each attribute should belong to a particular datatype (char, integer, date, number etc.).
3. Each attribute should have a precision defined (char (30), number (10,2) etc.).
4. Mandatory (required) or not mandatory characteristic of a attribute should be specified.
5. Default format (e.g.. mm/dd/yyyy) of the attribute may be specified. This is useful if some CASE tool is used to generate the code.
6. The Data Dictionary (ER diagram) should display the relationships between different entities (One to Many, One to One, Many to One etc.)
7. Many to Many (M: M) relationship between entities should be broken down into One to Many (1:M) and Many to One (M: 1).
8. Foreign Keys should be defined to create relationships.
9. Foreign key in a detail table should be a Primary Key or a unique key of the Master table.
10. Constraints should be present on the columns and tables to enforce referential integrity.
11. Database Triggers, which get executed when changes (insert, update, delete) occur on the underlying tables, should be defined.
12. Approximate number of records in each table should be calculated.

13. Percentage increase in the number of records over a period of time (may be a year).
14. Database space allocated to each table.
15. Description of each entity and its attributes should be present.
16. Any procedures or functions operating on the tables should be defined.
17. Access privileges (select, insert, update, delete, all, etc.) for the entities should be specified.
18. Structure and attributes of the tables should satisfy the User Requirements.
19. Owner of the table should be specified.
20. Naming convention for the table and column names should follow the enterprise wide standards.

Additional points which should be checked to evaluate a Data Dictionary.

1. All Classes should be defined as per user requirements.
2. All the methods needed for each class should be defined in the Data Dictionary.
3. All the data members for each class should be defined.
4. Scope of the data members should be specified (Public or Private).
5. Pre and Post conditions should be defined for each method.
6. Parameter types passed to each method and the return type should be specified.

An Example Dictionary: Oracle Data Dictionary

There are many times when a DBA finds the need to generate a custom SQL report based on data dictionary and dynamic performance tables. There is a wealth of information contained in the DBA_, V$, and X$ series of tables and views, but to be of any use, the purpose of each table and its relationship to other tables in the Data Dictionary must first be understood.

The Oracle Data Dictionary is a collection of C constructs, Oracle tables and Oracle views. At the lowest level are the "hidden" C structures known as the X$ tables. To even see their contents a DBA has to jump through a few hoops, and once you are there, they are not well documented, and have such logical attributes as "mglwmp" or "tabsrpr". It is suggested that administrators leave these to Oracle.

The next higher layer of the Oracle Data Dictionary is the $ tables. These are more readable descriptions of the X$ tables and have such names as COL$ or TAB$. While being a step above the X$ structures, it is still suggested that the DBA only use them when it is really needed.

The third layer of the dictionary is the V$ views and their cousins the V_$ tables (actually twins, since for nearly all of the V$ views there is a corresponding V_$ view). These are the workhorses of the Data Dictionary, and what most DBA scripts should deal with.

The next layer of the dictionary is the DBA_ series of views. These views are made from the V$ and $ sets of views and tables and provide a more useful interface for viewing the insides of the Oracle Data Dictionary. USER_ and ALL_ views are based on the DBA_ views.

The USER_ and ALL_ views provide an important function since not all users are granted access to the DBA_ level views. The USER_ views provide the user with information on all of the objects they personally own while the ALL_ views provide information on all of the objects that they are allowed to access.

The following figure is a pseudo entity relationship diagram of the USER_ views layer of the Oracle Data Dictionary. It is particularly handy as a printed reference.

One-to-One Relationship Must have a related entry in the table. They may have a related entry in the table.

The USER_Views – Descriptions

The following is a description of Oracle provided USER_ views. These views are derived from the DBA_ level data dictionary views and contain information for the current user.

USER_ADMINCATALOG	Information about the current user's asynchronous administration requests.
USER_ADMINMASTERS	N-way replication information about the current user.
USER_ADMINOBJECTS	Replication information about the current user's objects.
USER_ADMINREGSNAPS	All snapshots in the database.
USER_ADMINRESOLUTION	Description of all conflict resolutions in the database.
USER_ADMINSCHEMAS	Replication information about the current user.
USER_ARGUMENTS	Arguments in object accessible to the user.
USER_AUDIT_OBJECT	Audit trail records for objects accessible to the user.
USER_AUDIT_SESSION	Records for the userÕs audit trail records regarding CONNECT and DISCONNECT.
USER_AUDIT_STATEMENT	Records for the userÕs audit trails involving statements (GRANT, REVOKE, AUDIT, NOAUDIT,and ALTER SYSTEM commands).
USER_AUDIT_TRAIL	Audit trail entries relevant to the user.
USER_CATALOG	Tables, Views, Synonyms and Sequences owned by the user.
USER_CLUSTERS	Descriptions of user's own clusters.
USER_CLUSTER_HASH_EXPRESSIONS	Hash functions for the user's hash clusters.
USER_CLU_COLUMNS	Mapping of table columns to cluster columns.
USER_COL_COMMENTS	Comments on columns of user's tables and views.
USER_COL_PRIVS	Grants on columns for which the user is the owner, grantor or grantee.
USER_COL_PRIVS_MADE	All grants on columns of objects owned by the user.
USER_COL_PRIVS_RECD	Grants on columns for which the user is the grantee.
USER_CONSTRAINTS	Constraint definitions on user's own tables.
USER_CONS_COLUMNS	Information about accessible columns in constraint definitions.
USER_DB_LINKS	Database links owned by the user.
USER_DEPENDENCIES	Dependencies to and from users objects.
USER_ERRORS	Current errors on stored objects owned by the user.
USER_EXTENTS	Extents comprising segments owned by the user.
USER_FREE_SPACE	Free extents in tablespaces accessible to the user.
USER_INDEXES	Description of the user's own indexes
USER_IND_COLUMNS	COLUMNs comprising user's INDEXes or on user's TABLES.

USER_JOBS	All jobs owned by this user.
USER_OBJECTS	Objects owned by the user.
USER_OBJECT_SIZE	Sizes, in bytes, of various pl/sql objects.
USER_OBJ_AUDIT_OPTS	Auditing options for user's own tables and views.
USER_REFRESH	All the refresh groups.
USER_REFRESH_CHILDREN	All the objects in refresh groups, where the user owns the refresh group.
USER_REPAUDIT_ATTRIBUTE	Information about attributes automatically maintained for replication
USER_REPAUDIT_COLUMN	Information about columns in all shadow tables for user's replicated tables.
USER_REPCAT	Replication information about the current user.
USER_REPCATLOG	Information about asynchronous administration requests.
USER_REPCOLUMN_GROUP	Information about the current user's asynchronous administration requests
USER_REPDDL	Arguments that do not fit in a single repcat log record.
USER_REPGENERATED	Objects generated for the current user to support replication
USER_REPGROUPED_COLUMN	Columns in the all column groups of user's replicated tables.
USER_REPKEY_COLUMNS	Primary columns for a table using column-level replication.
USER_REPOBJECT	Replication information about the current user's objects.
USER_REPPARAMETER_COLUMN	All columns used for resolving conflicts in user's replicated tables
USER_REPPRIORITY	Values and their corresponding priorities in all priority groups, which are accessible to the user.
USER_REPPRIORITY_GROUP	Information about all priority groups which are accessible to the user.
USER_REPPROP	Propagation information about the current user's objects.
USER_REPRESOLUTION	Description of all conflict resolutions for user's replicated tables.
USER_REPRESOLUTION_METHOD	All conflict resolution methods accessible to the user.
USER_REPRESOLUTION_STATISTICS	Statistics for conflict resolutions for user's replicated tables.
USER_REPRESOL_STATS_CONTROL	Information about statistics collection for conflict resolutions for user's replicated tables.
USER_REPSCHEMA	N-way replication information about the current user.
USER_RESOURCE_LIMITS	Display resource limit of the user.
USER_ROLE_PRIVS	Roles granted to current user.
USER_SEGMENTS	Storage allocated for all database segments.
USER_SEQUENCES	Description of the user's own SEQUENCEs.
USER_SNAPSHOTS	Snapshots at which the user can look.
USER_SNAPSHOT_LOGS	All snapshot logs owned by the user.
USER_SOURCE	Source of stored objects accessible to the user.
USER_SYNONYMS	The user's private synonyms.
USER_SYS_PRIVS	System privileges granted to current user.
USER_TABLES	Description of the user's own tables
USER_TABLESPACES	Description of accessible tablespaces.

USER_TAB_COLUMNS	Columns of user's tables, views and clusters.
USER_TAB_COMMENTS	Comments on the tables and views owned by the user.
USER_TAB_PRIVS	Grants on objects for which the user is the owner, grantor or grantee.
USER_TAB_PRIVS_MADE	All grants on objects owned by the user.
USER_TAB_PRIVS_RECD	Grants on objects for which the user is the grantee.
USER_TRIGGERS	Triggers owned by the user.
USER_TRIGGER_COLS	Column usage in user's triggers.
USER_TS_QUOTAS	Tablespace quotas for the user.
USER_USERS	Information about the current user.
USER_VIEWS	Text of views owned by the user.

1.6 SUMMARY

A Database Management System (DBMS) consists of a collection of interrelated data and a collection of programs to access that data. Data refers to the values actually stored in the database and information refers to the meaning of those values as understood by the user. The data describes one particular enterprise. The primary goal of a DBMS is to provide convenient and efficient environment for people to use in retrieving and storing information.

Although file based systems were first commercial applications that provided increased efficiency in data processing compared to earlier manual paper record-based systems, but still these file based systems suffered from certain disadvantages, like, data redundancy and inconsistency, difficulty in accessing data, data isolation, integrity problems, atomicity problems and security problems.

Database systems are designed to store large bodies of information. The management of data involves both definition of structures for the storage of information and provision of mechanism for the manipulation of information.

DBMS provides a facility called **DDL** i.e., **data definition language**, which specifies the database scheme. DDL is expressed by set of definitions. These definitions are compiled into a set of tables that is stored in a special file called the **data dictionary**, thus, data dictionary contains metadata i.e., data about data.

The relational data model is based on the collection of tables. Tables are formed of attributes and tuples. The user of the database system may query these tables, insert new tuples, delete tuples, and update (modify) tuples.

Domains are the pools of values, from which actual attributes are drawn, and Relation on collection of domains consists of two parts, a heading and a body. The heading consists of a fixed set attributes, and the body consists of a set of tuples. Views are "virtual relations" defined by a query expression. Views are useful mechanism for simplifying database queries, but modification of the database through views may have potentially disadvantageous consequences. Therefore, database systems severely restrict updates through views.

Integrity constraints ensure that changes made to the database by authorised users do not result in a loss of data consistency. Domain constraints specify the set of possible values that may be associated with an attribute. Such constraints may also prohibit the use of null values for particular attributes..

Referential-integrity constraints ensure that a value that appears in one relation for given sets of attributes also appears for a certain set of attributes in another relation.

1.7 MODEL ANSWERS

Check Your Progress 1

Question 1.

- Reduced Redundancy
- Sharing
- Enforcement of standards
- Security
- Integrity of Data
- Fine tuning of database
- Data independence

Question 2. Some of the disadvantages are as follows:

- Security might be compromised (without good controls)
- Integrity might be compromised (without good controls)
- Additional hardware might be required
- Performance overhead might be significant
- Successful operation is crucial (the enterprise might be highly vulnerable to failure)
- The system is likely to be complex (though such complexity should be concealed from the user).

Question 3. (a) Data Definition (It is not in SQL)

```
CREATE DOMAIN P# CHAR (6);
CREATE DOMAIN QTY NUMERIC (9);
CREATE BASE RELATION PARTR_DETAIL
(MAJOR_PART# DOMAIN ( P# ),
MINOR_PART# DOMAIN ( P# ),
QUANTITY DOMAIN ( QTY ) )
PRIMARY KEY (MAJOR_PART#, MINOR_PART#);
```

Note: If the PART_DETAIL relation were a part of supplier-and-part database, we would probably need additional specifications in the relation definition.

```
FOREIGN KEY (RENAME MAJOR_PART# AS P#) REFERENCES P;
FOREIGN KEY (RENAME MINOR_PART# AS P#) REFERENCES P;
```

(b) We show the new catalog entries.

DOMAINS

DOMNAME	DATATYPE	.....
P#	CHAR (6)	.....
QTY	NUMERIC (9)	.....

RELATIONS

RELNAME	DEGREE	CARDINALITY	RELTYPE	.....
PART_DETAIL	3	0	Base	.....

ATTRIBUTES

RELNAME	ATTRNAME	DOMNAME	.....
PART_DETAIL	MAJOR_PART#	P#	.....
PART_DETAIL	MINOR_PART#	P#	.....
PART_DETAIL	QUANTITY	QTY	.....

Note that the CARDINALITY entry in the RELATIONS relation for the RELATIONS relation itself will also need to be incremented by one. Also, we have shown cardinality of PART_DETAIL as 0, not 7 (despite the fact that in Question our relation PART_DETAIL contains 7 tuples), because, of course, the relation will be empty when it is first created.

(c) DESTROY BASE RELATION PART_DETAIL;
DESTROY DOMAIN QTY;
DESTROY DOMAIN P#;

Question 4. A relation with an empty set of Tuples is perfectly reasonable, and indeed common (it is analogous to file with no records). In particular, of course, every base relation has an empty set of tuples when it is first created. It is usual, though a trifle imprecise, to refer to a relation with no tuples as an empty relation.

Question 5. What is perhaps less immediate obvious is that a relation with an empty set of attributes is perfectly reasonable too! In fact, such relations turn out to be of crucial importance - much as empty sets are crucially important I general set theory, or zero is important I ordinary arithmetic.

(i) CREATE SECURITY RULE SEC51
GRANT RETRIEVE ON STUDENT TO Anu;

(ii) CREATE SECURITY RULE SEC52
GRANT INSERT, DELETE ON STUDENT TO Smita;

(iii) CREATE SECURITY RULE SEC53
GRANT RETRIEVE
ON STUDENT WHERE STUDENT.STUDENT_ID = USER ( );
TO ALL;

(iv) CREATE SECURITY RULE SEC54
GRANT RETRIEVE, UPDATE (STIPEND, FEES)
ON STUDENT TO Nath;

(v) CREATE SECURITY RULE SEC55
GRANT RETRIEVE (STUDENT_ID, STIPEND, FEES)
ON STUDENT TO Ram;

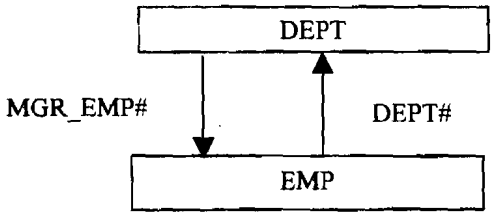
(vi) CREATE SECURITY RULE SEC56
GRANT RETRIEVE (STUDENT_ID, STIPEND, FEES)
UPDATE (STIPEND, FEES)
ON STUDENT TO John;

Check Your Progress 2

Question 1.

1. Accepted
2. Rejected (candidate key uniqueness violation)
3. Rejected (violates RESTRICTED update rule)
4. Accepted (supplier S3 and all shipments for supplier S3 in the relation SPJ are deleted).
5. Rejected (violates RESTRICTED delete rule as P2 is present in relation SPJ).
6. Accepted (project J4 and all shipments for project J4 from the relation SPJ are deleted).
7. Accepted
8. Rejected (candidate key uniqueness violation as tuple S5-P5-J7already exists in relation SPJ).
9. Rejected (referential integrity violation as there exists no tuple for J8 in relation J).
10. Accepted
11. Rejected (referential integrity violation as there exists no tuple for P7 in relation P).
12. Rejected (referential integrity violation – the default project number jjj does not exist in relation J).

Question 2. The diagram could be shown as below:



Since the database involves a referential cycle (there is a referential path from each of two relations to itself), either one of the two foreign keys must be permitted to accept nulls or the integrity checking must be deferred. The data definition (an outline) is:

```
CREATE BASE RELATION DEPT
        (DEPT#....,
                ..... ,
         MGR_EMP# ...)
        PRIMARY KEY (DEPT#) );


CREATE BASE RELATION EMP
        (EMP#....,
                ..... ,
         DEPT# .....)
        PRIMARY KEY (EMP#)
        FORIEGN KEY (DEPT# ) REFERENCES DEPT
                DELETE CASCADES
                UPDATE CASCADES;
```

Question 3. Here also we show just the CREATE BASE RELATION statements

```
CREATE BASE RELATION EMP
        (EMP#....,
                ..... ,
         JOB ...)
        PRIMARY KEY (EMP#) ) ;

CREATE BASE RELATION PGMR
        (EMP#....,
                ..... ,
         LANG .....)
        PRIMARY KEY (EMP#)
        FORIEGN KEY (DEPT# ) REFERENCES EMP
                DELETE CASCADES
                UPDATE CASCADES;
```

This example illustrates the point that a foreign key can in fact also be a candidate key of its containing relation. Relation EMP lists all employees, and relation PGMR lists just those employees that are programmers; thus, every employee number appearing in PGMR is also a foreign key, referring to the primary key of EMP. Note that the two relations can be regarded as representing, respectively, an entity supertype (employees) and an entity subtype (programmers). In fact, the example is typical of the way entity supertypes and subtypes would be represented in a relational database.

Note that there is another integrity constraint that also needs to be maintained in this example – namely, the constraint that a given employee must be represented in relation PGMR and only if the value of EMP.JOB for that employee is "Programmer". This constraint is not a referential constraint.