Q.2 a. Discuss the responsibilities of the DBA and Database Designers? (4)

Answer:

Database Administrator

In any organization where many people use the same resources, there is a need for a chief administrator to oversee and manage these resources. In a database environment, the primary resource is the database itself, and the secondary resource is the DBMS and related software. Administering these resources is the responsibility of the **database administrator (DBA)**. The DBA is responsible for authorizing access to the database, coordinating and monitoring its use and acquiring software and hardware resources as needed. The DBA is accountable for problems such as security breaches and poor system response time. In large organizations, the DBA is assisted by a staff that carries out these functions.

Database Designers

Database designers are responsible for identifying the data to be stored in the database and for choosing appropriate structures to represent and store this data. These tasks are mostly undertaken before the database is actually implemented and populated with data. It is the responsibility of database designers to communicate with all prospective database users in order to understand their requirements and to create a design that meets these requirements. In many cases, the designers are on the staff of the DBA and may be assigned other staff responsibilities after the database design is completed. Database designers typically interact with each potential group of users and develop **views** of the database that meet the data and processing requirements of these groups. Each view is then analyzed and *integrated* with the views of other user groups. The final database design must be capable of supporting the requirements of all user groups.

b. With the help of a diagram, explain the component modules of a DBMS and their interactions? (9)

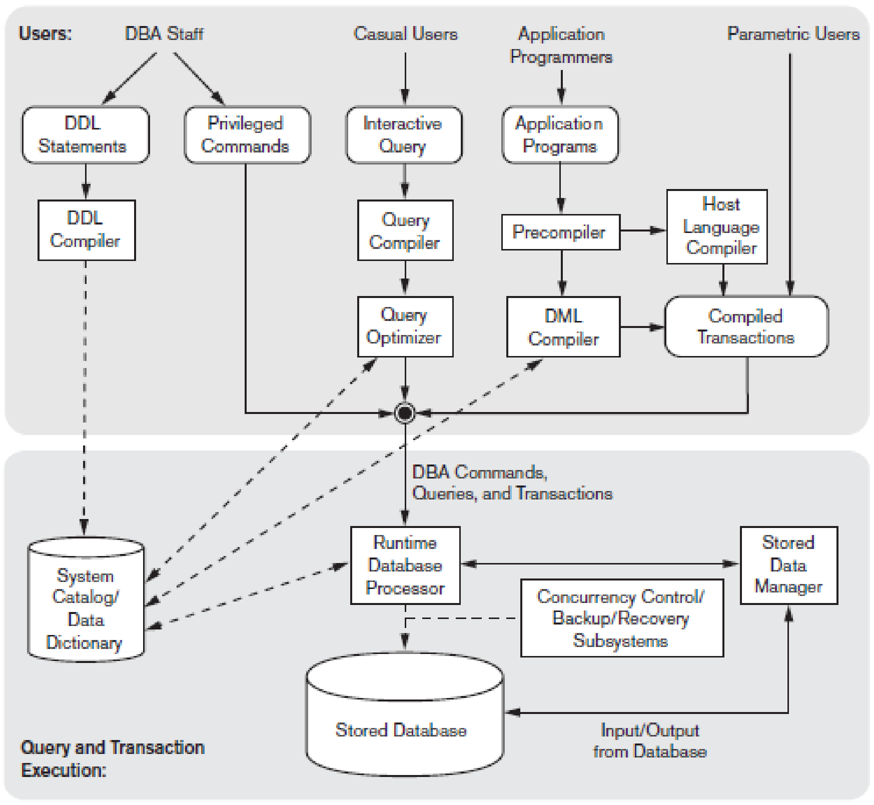
Answer:

The following Figure illustrates, in a simplified form, the typical DBMS components. The figure is divided into two parts. The top part of the figure refers to the various users of the database environment and their interfaces. The lower part shows the internals of the DBMS responsible for storage of data and processing of transactions.

The database and the DBMS catalog are usually stored on disk. Access to the disk is controlled primarily by the **operating system** (**OS**), which schedules disk read/write. Many DBMSs have their own **buffer management** module to schedule disk read/write, because this has a considerable effect on performance. Reducing disk read/write improves performance considerably. A higher-level **stored data manager** module of the DBMS controls access to DBMS information that is stored on disk, whether it is part of the database or the catalog.

The top part of figure shows interfaces for the DBA staff, casual users who work with interactive interfaces to formulate queries, application programmers who create programs using some host programming languages, and parametric users who do data entry work by supplying parameters to predefined transactions. The DBA staff works on defining the database and tuning it by making changes to its definition using the DDL and other privileged commands.

The DDL compiler processes schema definitions, specified in the DDL, and stores descriptions of the schemas (meta-data) in the DBMS catalog. The catalog includes information such as the names and sizes of files, names and data types of data items, storage details of each file, mapping information among schemas, and constraints. In addition, the catalog stores many other types of information that are needed by the DBMS modules, which can then look up the catalog information as needed.



Component modules of a DBMS and their interactions

Casual users and persons with occasional need for information from the database interact using some form of interface, which we call the **interactive query** interface. These queries are parsed and validated for correctness of the query syntax, the names of files and data elements, and so on by a **query compiler** that compiles them into an internal form. This internal query is subjected to query optimization. Among other things, the **query optimizer** is concerned with the rearrangement and possible reordering of operations, elimination of redundancies, and use of correct algorithms and indexes during execution. It consults the system catalog for statistical and other physical information about the stored data and generates executable code that performs the necessary operations for the query and makes calls on the runtime processor.

Application programmers write programs in host languages such as Java, C, or C++ that are submitted to a precompiler. The **precompiler** extracts DML commands from an application program written in a host programming language. These commands are sent to the DML compiler for compilation into object code for database access. The rest of the program is sent to the host language compiler. The object codes for the DML commands and the rest of the program are linked, forming a canned transaction whose executable code includes calls to the runtime database processor. Canned transactions are executed repeatedly by parametric users, who simply supply the parameters to the transactions. Each execution is considered to be a separate transaction. An example is a bank withdrawal transaction where the account number and the amount may be supplied as parameters.

In the lower part of figure, the **runtime database processor** executes (1) the privileged commands, (2) the executable query plans, and (3) the canned transactions with runtime parameters. It works with the **system catalog** and may update it with statistics. It also works with the **stored data manager**, which in turn uses basic operating system services for carrying out low-level input/output (read/write) operations between the disk and main memory. The runtime

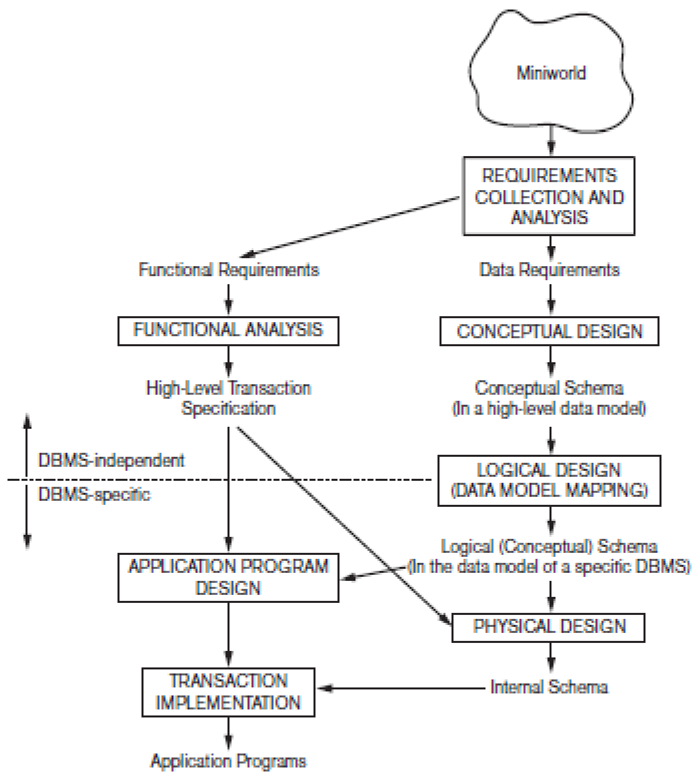
database processor handles other aspects of data transfer, such as management of buffers in the main memory. Some DBMSs have their own buffer management module while others depend on the OS for buffer management. We have shown **concurrency control** and **backup and recovery systems** separately as a module in this figure. They are integrated into the working of the runtime database processor for purposes of transaction management.
It is now common to have the **client program** that accesses the DBMS running on a separate computer from the computer on which the database resides. The former is called the **client computer** running DBMS client software and the latter is called the **database server**. In some cases, the client accesses a middle computer, called the **application server**, which in turn accesses the database server.

c. What does defining, manipulating and protecting of a database mean? (3)

Answer:
Defining a database involves specifying the data types, structures, and constraints of the data to be stored in the database. The database definition or descriptive information is also stored by the DBMS in the form of a database catalog or dictionary; it is called **meta-data**.
Constructing the database is the process of storing the data on some storage medium that is controlled by the DBMS.
Protection includes *system protection* against hardware or software malfunction (or crashes) and *security protection* against unauthorized or malicious access.

Q.3 a. Discuss the role of a high-level data model in the database design process? (7)

Answer:
Following figure shows a simplified overview of the database design process. The first step shown is **requirements collection and analysis**. During this step, the database designers interview prospective database users to understand and document their **data requirements**. The result of this step is a concisely written set of users' requirements. These requirements should be specified in as detailed and complete a form as possible. In parallel with specifying the data requirements, it is useful to specify the known **functional requirements** of the application. These consist of the user defined **operations** (or **transactions**) that will be applied to the database, including both retrievals and updates. In software design, it is common to use *data flow diagrams*, *sequence diagrams*, *scenarios*, and other techniques to specify functional requirements.


Miniworld
REQUIREMENTS COLLECTION AND ANALYSIS
Functional Requirements
Data Requirements
FUNCTIONAL ANALYSIS
CONCEPTUAL DESIGN
High-Level Transaction Specification
Conceptual Schema (In a high-level data model)
DBMS-independent
DBMS-specific
LOGICAL DESIGN (DATA MODEL MAPPING)
Logical (Conceptual) Schema (In the data model of a specific DBMS)
APPLICATION PROGRAM DESIGN
PHYSICAL DESIGN
TRANSACTION IMPLEMENTATION
Internal Schema
Application Programs

A simplified diagram to illustrate the main phases of database design

Once the requirements have been collected and analyzed, the next step is to create a **conceptual schema** for the database, using a high-level conceptual data model. This step is called **conceptual design**. The conceptual schema is a concise description of the data requirements of the users and includes detailed descriptions of the entity types, relationships, and constraints; these are expressed using the concepts provided by the high-level data model. Because these concepts do not include implementation details, they are usually easier to understand and can be used to communicate with nontechnical users. The high-level conceptual schema can also be used as a reference to ensure that all users' data requirements are met and that the requirements do not conflict. This approach enables database designers to concentrate on specifying the properties of the data, without being concerned with storage and implementation details. This makes it is easier to create a good conceptual database design.

During or after the conceptual schema design, the basic data model operations can be used to specify the high-level user queries and operations identified during functional analysis. This also serves to confirm that the conceptual schema meets all the identified functional requirements. Modifications to the conceptual schema can be introduced if some functional requirements cannot be specified using the initial schema.

The next step in database design is the actual implementation of the database, using a commercial DBMS. Most current commercial DBMSs use an implementation data model—such as the relational or the object-relational database model—so the conceptual schema is transformed from the high-level data model into the implementation data model. This step is called **logical design** or **data model mapping**; its result is a database

schema in the implementation data model of the DBMS. Data model mapping is often automated or semi-automated within the database design tools.
The last step is the **physical design** phase, during which the internal storage structures, file organizations, indexes, access paths, and physical design parameters for the database files are specified. In parallel with these activities, application programs are designed and implemented as database transactions corresponding to the high-level transaction specifications.

b. What do you mean by Entity and Relationship in ER model? Explain how a relationship set is defined? (4)

Answer:
Entity: The basic object that ER Model represents is an entity, which is a thing in the real world within independent existences. An entity may be an object with a physical existence such as a particular person, car, house etc.
For example, Entity Name: Employee
Attribute Name: Name, Age, Sex, Salary
Relationship: A relationship is an association among several entities.

Relationship set: A relationship set is a set of relationships of same type. Basically, it is a mathematical relation on n >= 2 entity sets. If E_1, E_2, ………….., E_n are entity sets, then a relationship set R is a subset of

$\{(e_1, e_2, \ldots\ldots, e_n) \mid e_1 \varepsilon E_1, e_2 \varepsilon E_2, \ldots\ldots, e_n \varepsilon E_n\}$
where $(e_1, e_2, \ldots\ldots, e_n)$ is a relationship.

c. Discuss the characteristics of relations that make them different from ordinary tables and files? (5)

Answer:
The characteristics that make a relation different from ordinary tables and files are as follows:

- **Ordering of Tuples in a Relation.** A relation is defined as a *set* of tuples. Mathematically, elements of a set have *no order* among them; hence, tuples in a relation do not have any particular order. In other words, a relation is not sensitive to the ordering of tuples. But, in a file, records are physically stored on disk (or in memory), so there always is an order among the records. This ordering indicates first, second, *i*th, and last records in the file. Similarly, when we display a relation as a table, the rows are displayed in a certain order.
- **Ordering of Values within a Tuple and an Alternative Definition of a Relation.** According to the preceding definition of a relation, an *n*-tuple is an *ordered list* of *n* values, so the ordering of values in a tuple—and hence of attributes in a relation schema—is important. However, at a more abstract level, the order of attributes and their values is *not* that important as long as the correspondence between attributes and values is maintained.
- **Values and NULLs in the Tuples.** Each value in a tuple is an **atomic** value; that is, it is not divisible into components within the framework of the basic relational model. Hence, composite and multivalued attributes are not allowed. This model is sometimes called the **flat relational model**. Much of the theory behind the relational model was developed with this assumption in mind, which is called the **first normal form** assumption. Hence, multivalued attributes must be represented by separate relations, and composite attributes are represented only by their simple component attributes in the basic relational model.

An important concept is that of NULL values, which are used to represent the values of attributes that may be unknown or may not apply to a tuple. A special value, called NULL, is used in these cases.

- **Interpretation of a Relation.** The relation schema can be interpreted as a declaration or a type of **assertion**. Each tuple in the relation can then be interpreted as a **fact** or a particular instance of the assertion. Some relations may represent facts about *entities,* whereas other relations may represent facts about *relationships.*

Q.4 a. With the help of a suitable example, explain the following Relational Algebra operations with their notations:
(i) SELECT
(ii) PROJECT
(iii) INTERSECTION (3×3)

Answer:

(i) SELECT

The SELECT operation $\sigma_P(r)$ selects those tuples from relation r, which satisfy a given predicate P. The predicate P appears as a subscript to σ. The argument relation is given in parentheses following the σ. Thus, to select those tuples of the loan relation where the branch name is "XYZ", the query will be

$\sigma_{\text{branch-name} = \text{"XYZ"}}$ (loan)

(ii) PROJECT

The PROJECT operation $\pi_S(r)$ projects attributes list S from r(R). Any duplicates tuples in the result are automatically eliminated. Thus, the query to list all loan numbers and the amount of the loan can be written as

$\pi_{\text{loan-number, amount}}$(loan)

(iii) INTERSECTION

The INTERSECTION operation r $\cap$ s, between two relation 'r' and 's', produces a relation with tuples which are there in relation 'r' as well as in relation 's'. For the operation r $\cap$ s to be feasible, the relations 'r' and 's' must be compatible.

The query to get the names of those customers who have an account as well as loan in the bank is

$\pi_{\text{cust-name}}$(deposit) $\cap$ $\pi_{\text{cust-name}}$(loan)

This will return the name of all customers with both an account and a loan at the bank.

b. Outline the steps to convert the basic ER model to relational database schema. (7)

Answer:

Step 1: Mapping of Regular Entity Types. For each regular (strong) entity type *E* in the ER schema, create a relation *R* that includes all the simple attributes of *E*. Include only the simple component attributes of a composite attribute. Choose one of the key attributes of *E* as the primary key for *R*. If the chosen key of *E* is a composite, then the set of simple attributes that form it will together form the primary key of *R*.

If multiple keys were identified for *E* during the conceptual design, the information describing the attributes that form each additional key is kept in order to specify secondary (unique) keys of relation *R*. Knowledge about keys is also kept for indexing purposes and other types of analyses.

Step 2: Mapping of Weak Entity Types. For each weak entity type *W* in the ER schema with owner entity type *E*, create a relation *R* and include all simple attributes (or simple components of composite attributes) of *W* as attributes of *R*. In addition, include as foreign key attributes of *R*, the primary key attribute(s) of the relation(s) that correspond to the owner entity type(s); this takes care of mapping the identifying relationship type of *W*. The primary key of *R* is the combination of the primary key(s) of the owner(s) and the partial key of the weak entity type *W*, if any.
If there is a weak entity type E_2 whose owner is also a weak entity type E_1, then E_1 should be mapped before E_2 to determine its primary key first.
Step 3: Mapping of Binary 1:1 Relationship Types. For each binary 1:1 relationship type *R* in the ER schema, identify the relations *S* and *T* that correspond to the entity types participating in *R*. There are three possible approaches: (1) the foreign key approach, (2) the merged relationship approach, and (3) the cross-reference or relationship relation approach.
Step 4: Mapping of Binary 1:N Relationship Types. For each regular binary 1:N relationship type *R*, identify the relation *S* that represents the participating entity type at the *N-side* of the relationship type. Include as foreign key in *S* the primary key of the relation *T* that represents the other entity type participating in *R*; we do this because each entity instance on the N-side is related to at most one entity instance on the 1-side of the relationship type. Include any simple attributes (or simple components of composite attributes) of the 1:N relationship type as attributes of *S*.
Step 5: Mapping of Binary M:N Relationship Types. For each binary M:N relationship type *R*, create a new relation *S* to represent *R*. Include as foreign key attributes in *S* the primary keys of the relations that represent the participating entity types; their *combination* will form the primary key of *S*. Also include any simple attributes of the M:N relationship type (or simple components of composite attributes) as attributes of *S*. Notice that we cannot represent an M:N relationship type by a single foreign key attribute in one of the participating relations because of the M:N cardinality ratio; we must create a separate *relationship relation S*.
Step 6: Mapping of Multivalued Attributes. For each multivalued attribute *A*, create a new relation *R*. This relation *R* will include an attribute corresponding to *A*, plus the primary key attribute *K*—as a foreign key in *R*—of the relation that represents the entity type or relationship type that has *A* as a multivalued attribute. The primary key of *R* is the combination of *A* and *K*. If the multivalued attribute is composite, we include its simple components.
Step 7: Mapping of *N*-ary Relationship Types. For each *n*-ary relationship type *R*, where $n > 2$, create a new relation *S* to represent *R*. Include as foreign key attributes in *S* the primary keys of the relations that represent the participating entity types. Also include any simple attributes of the *n*-ary relationship type (or simple components of composite attributes) as attributes of *S*. The primary key of *S* is usually a combination of all the foreign keys that reference the relations representing the participating entity types. However, if the cardinality constraints on any of the entity types *E* participating in *R* is 1, then the primary key of *S* should not include the foreign key attribute that references the relation E' corresponding to *E*.

Q.5 a. What are the basic data types available for attributes in SQL? (8)
Answer:
The basic **data types** available for attributes include numeric, character string, bit string, Boolean, date, and time.

- **Numeric** data types include integer numbers of various sizes (INTEGER or INT, and SMALLINT) and floating-point (real) numbers of various precision (FLOAT or REAL, and DOUBLE PRECISION). Formatted numbers can be declared by using DECIMAL(*i*,*j*) or DEC(*i*,*j*) or NUMERIC(*i*,*j*) where *i*, the *precision*, is the total number of decimal digits and *j*, the *scale*, is the number of digits after the decimal point. The default for scale is zero, and the default for precision is implementation-defined.

- **Character-string** data types are either fixed length—CHAR(*n*) or CHARACTER(*n*), where *n* is the number of characters or varying length VARCHAR(*n*) or CHAR VARYING(*n*) or CHARACTER VARYING(*n*), where *n* is the maximum number of characters. When specifying a literal string value, it is placed between single quotation marks (apostrophes), and it is *case sensitive*. For fixed length strings, a shorter string is padded with blank characters to the right. For example, if the value 'Smith' is for an attribute of type CHAR(10), it is padded with five blank characters to become 'Smith ' if needed. Padded blanks are generally ignored when strings are compared. For comparison purposes, strings are considered ordered in alphabetic (or lexicographic) order; if a string *str1* appears before another string *str2* in alphabetic order, then *str1* is considered to be less than *str2*. There is also a concatenation operator denoted by || (double vertical bar) that can concatenate two strings in SQL. For example, 'abc' || 'XYZ' results in a single string 'abcXYZ'. Another variable-length string data type called CHARACTER LARGE OBJECT or CLOB is also available to specify columns that have large text values, such as documents. The CLOB maximum length can be specified in kilobytes (K), megabytes (M), or gigabytes (G). For example, CLOB(20M) specifies a maximum length of 20 megabytes.
- **Bit-string** data types are either of fixed length *n,* BIT(*n*) or varying length, BIT VARYING(*n*), where *n* is the maximum number of bits. The default for *n*, the length of a character string or bit string, is 1. Literal bit strings are placed between single quotes but preceded by a B to distinguish them from character strings; for example, B'10101'. Another variable-length bitstring data type called BINARY LARGE OBJECT or BLOB is also available to specify columns that have large binary values, such as images. As for CLOB, the maximum length of a BLOB can be specified in kilobits (K), megabits (M), or gigabits (G). For example, BLOB(30G) specifies a maximum length of 30 gigabits.
- A **Boolean** data type has the traditional values of TRUE or FALSE. In SQL, because of the presence of NULL values, a three-valued logic is used, so a third possible value for a Boolean data type is UNKNOWN.
- The **DATE** data type has ten positions, and its components are YEAR, MONTH, and DAY in the form YYYY-MM-DD. The TIME data type has at least eight positions, with the components HOUR, MINUTE, and SECOND in the form HH:MM:SS. Only valid dates and times should be allowed by the SQL implementation. This implies that months should be between 1 and 12 and dates must be between 1 and 31; furthermore, a date should be a valid date for the corresponding month. The < (less than) comparison can be used with dates or times—an *earlier* date is considered to be smaller than a later date, and similarly with time. Literal values are represented by single-quoted strings preceded by the keyword DATE or TIME; for example, DATE '2008-09-27' or TIME '09:12:47'. In addition, a data type TIME(*i*), where *i* is called *time fractional seconds precision*, specifies *i* + 1 additional positions for TIME—one position for an additional period (.) separator character, and *i* positions for specifying decimal fractions of a second. A TIME WITH TIME ZONE data type includes an additional six positions for specifying the *displacement* from the standard universal time zone, which is in the range +13:00 to –12:59 in units of HOURS:MINUTES. If WITH TIME ZONE is not included, the default is the local time zone for the SQL session.

  A **timestamp** data type (TIMESTAMP) includes the DATE and TIME fields, plus a minimum of six positions for decimal fractions of seconds and an optional WITH TIME ZONE qualifier. Literal values are represented by single quoted strings preceded by the keyword TIMESTAMP, with a blank space between data and time; for example, TIMESTAMP '2008-09-27 09:12:47.648302'.

- Another data type related to DATE, TIME, and TIMESTAMP is the INTERVAL data type. This specifies an **interval**—a *relative value* that can be used to increment or decrement an absolute value of a date, time, or timestamp. Intervals are qualified to be either YEAR/MONTH intervals or DAY/TIME intervals.

b. List the three main approaches for database programming? What are the advantages and disadvantages of each approach? (8)

Answer:

Several techniques exist for including database interactions in application programs. The main approaches for database programming are the following:

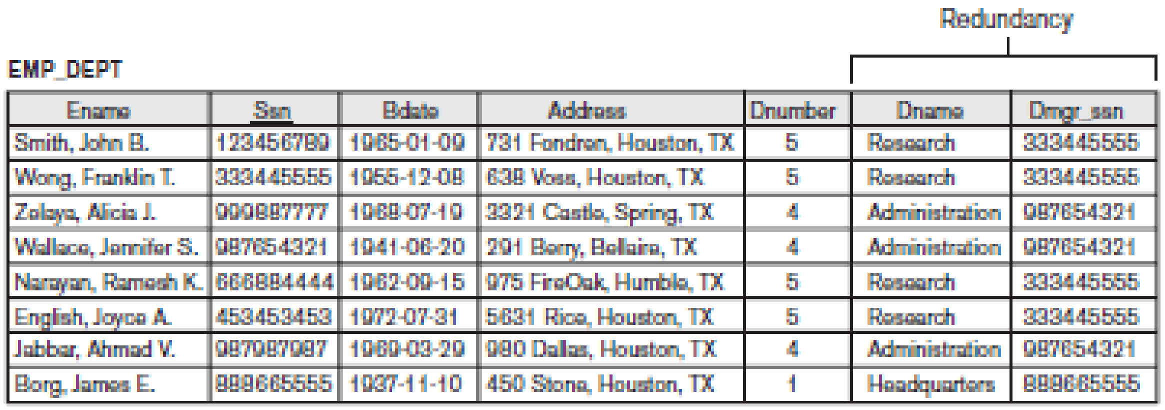
- **Embedding database commands in a general-purpose programming language.**
  In this approach, database statements are **embedded** into the host programming language, but they are identified by a special prefix. For example, the prefix for embedded SQL is the string EXEC SQL, which precedes all SQL commands in a host language program. A **precompiler** or **preproccessor** scans the source program code to identify database statements and extract them for processing by the DBMS. They are replaced in the program by function calls to the DBMS-generated code. This technique is generally referred to as **embedded SQL**.
- **Using a library of database functions.**
  A **library of functions** is made available to the host programming language for database calls. For example, there could be functions to connect to a database, execute a query, execute an update, and so on. The actual database query and update commands and any other necessary information are included as parameters in the function calls. This approach provides what is known as an **application programming interface** (**API**) for accessing a database from application programs.
- **Designing a brand-new language.**
  A **database programming language** is designed from scratch to be compatible with the database model and query language. Additional programming structures such as loops and conditional statements are added to the database language to convert it into a full fledged programming language. An example of this approach is Oracle's PL/SQL.

In practice, the first two approaches are more common, since many applications are already written in general-purpose programming languages but require some database access. The third approach is more appropriate for applications that have intensive database interaction. One of the main problems with the first two approaches is *impedance mismatch*, which does not occur in the third approach.

Q.6 a. Discuss insertion, deletion and modification anomalies. Why are they undesirable? Illustrate with examples? (8)

Answer:

Consider the following EMP_DEPT relation:


Redundancy

EMP_DEPT

Ename	Ssn	Bdate	Address	Dnumber	Dname	Dmgr_ssn
Smith, John B.	123456789	1965-01-09	731 Fondren, Houston, TX	5	Research	333445555
Wong, Franklin T.	333445555	1955-12-08	638 Voss, Houston, TX	5	Research	333445555
Zelaya, Alicia J.	999887777	1968-07-19	3321 Castle, Spring, TX	4	Administration	987654321
Wallace, Jennifer S.	987654321	1941-06-20	291 Berry, Bellaire, TX	4	Administration	987654321
Narayan, Ramesh K.	666884444	1962-09-15	975 FireOak, Humble, TX	5	Research	333445555
English, Joyce A.	453453453	1972-07-31	5631 Rice, Houston, TX	5	Research	333445555
Jabbar, Ahmad V.	987987987	1969-03-29	980 Dallas, Houston, TX	4	Administration	987654321
Borg, James E.	888665555	1937-11-10	450 Stone, Houston, TX	1	Headquarters	888665555

Insertion Anomalies. Insertion anomalies can be differentiated into two types, illustrated by the following examples based on the EMP_DEPT relation:

- To insert a new employee tuple into EMP_DEPT, we must include either the attribute values for the department that the employee works for, or NULLs (if the employee does not work for a department as yet). For example, to insert a new tuple for an employee who works in department number 5, we must enter all the attribute values of department 5 correctly so that they are *consistent* with the corresponding values for department 5 in other tuples in EMP_DEPT.
- It is difficult to insert a new department that has no employees as yet in the EMP_DEPT relation. The only way to do this is to place NULL values in the attributes for employee. This violates the entity integrity for EMP_DEPT because Ssn is its primary key. Moreover, when the first employee is assigned to that department, we do not need this tuple with NULL values any more.

Deletion Anomalies. The problem of deletion anomalies is related to the second insertion anomaly situation just discussed. If we delete from EMP_DEPT an employee tuple that happens to represent the last employee working for a particular department, the information concerning that department is lost from the database.

Modification Anomalies. In EMP_DEPT, if we change the value of one of the attributes of a particular department—say, the manager of department 5—we must update the tuples of *all* employees who work in that department; otherwise, the database will become inconsistent. If we fail to update some tuples, the same department will be shown to have two different values for manager in different employee tuples, which would be wrong.

It is easy to see that these three anomalies are undesirable and cause difficulties to maintain consistency of data as well as require unnecessary updates that can be avoided.

b. Define minimal cover of a set? Give an algorithm for finding a Minimal Cover F for a Set of Functional Dependencies E. (2+6)

Answer:

A minimal cover of a set of functional dependencies E is a minimal set of dependencies that is equivalent to E. We can always find at least one minimal cover F for any set of dependencies E using the following algorithm:

Algorithm for finding a Minimal Cover F for a set of functional dependencies E

1. Set F:= E.

2. Replace each functional dependency $X \rightarrow \{A_1, A_2, \ldots\ldots\ldots, A_n\}$ in F by the n functional dependencies $X \rightarrow A_1, X \rightarrow A_2, \ldots\ldots\ldots, X \rightarrow A_n$.
3. For each functional dependencies $X \rightarrow A$ in F

```
        for each attribute B that is an element of X
        if {{F – {X → A} U {(X – {B}) → A}} is equivalent to F,
                then replace X → A with (X – {B}) → A in F.
```

4. For each remaining functional dependency $X \rightarrow A$ in F

```
        if {F – {X → A}} is equivalent to F,
                then remove X → A from F.
```

Q.7 a. Define the following: (2×4)
(i) Multivalued Dependency
(ii) Join Dependency
(iii) Fourth normal form
(iv) Fifth normal form

Answer:

(i) Multivalued Dependency

A **multivalued dependency** $X \rightarrow\rightarrow Y$ 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 t_1 and t_2 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 $(R - (X \cup Y))$:

- $t_3[X] = t_4[X] = t_1[X] = t_2[X]$.
- $t_3[Y] = t_1[Y]$ and $t_4[Y] = t_2[Y]$.
- $t_3[Z] = t_2[Z]$ and $t_4[Z] = t_1[Z]$.

(ii) Join Dependency

A **join dependency** (**JD**), denoted by JD($R_1, R_2, \ldots, R_n$), 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 nonadditive join decomposition into $R_1, R_2, \ldots, R_n$. Hence, for every such *r* we have

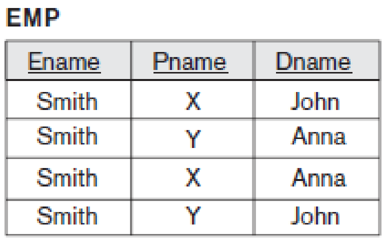
$(\pi_{R1}(r), \pi_{R2}(r), \ldots, \pi_{Rn}(r)) = r$

(iii) Fourth normal form

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

We can state the following points:

- An all-key relation is always in BCNF since it has no FDs.
- An all-key relation such as the EMP relation in Figure below, which has no FDs but has the MVD Ename→→ Pname | Dname, is not in 4NF.



EMP

Ename	Pname	Dname
Smith	X	John
Smith	Y	Anna
Smith	X	Anna
Smith	Y	John

- A relation that is not in 4NF due to a nontrivial MVD must be decomposed to convert it into a set of relations in 4NF.
- The decomposition removes the redundancy caused by the MVD.

(iv) Fifth normal form

A relation schema R is in **fifth normal form** (5NF) (or project-join normal form (PJNF)) with respect to a set F of functional, multivalued, and join dependencies if, for every nontrivial join dependency JD(R_1, R_2, ..., R_n) in F^+ (that is, implied by F), every R_i is a superkey of R.

b. Describe the two desirable properties of decomposition? (8)

Answer:

The two desirable properties of decomposition are the dependency preservation property and the nonadditive (or lossless) join property.

Dependency Preservation Property of a Decomposition

It would be useful if each functional dependency $X \rightarrow Y$ specified in F either appeared directly in one of the relation schemas R_i in the decomposition D or could be inferred from the dependencies that appear in some R_i. Informally, this is the *dependency preservation condition.* We want to preserve the dependencies because each dependency in F represents a constraint on the database. If one of the dependencies is not represented in some individual relation R_i of the decomposition, we cannot enforce this constraint by dealing with an individual relation. We may have to join multiple relations so as to include all attributes involved in that dependency.
It is not necessary that the exact dependencies specified in F appear themselves in individual relations of the decomposition D. It is sufficient that the union of the dependencies that hold on the individual relations in D be equivalent to F.

Given a set of dependencies F on R, the projection of F on R_i, denoted by $\pi_{R_i}(F)$ where R_i is a subset of R, is the set of dependencies $X \rightarrow Y$ in F^+ such that the attributes in $X \cup Y$ are all contained in R_i. Hence, the projection of F on each relation schema R_i in the decomposition D is the set of functional dependencies in F^+, the closure of F, such that all their left- and right-hand-side attributes are in R_i. We say that a decomposition $D = \{R_1, R_2, ..., R_m\}$ of R is dependency-preserving with respect to F if the union of the projections of F on each R_i in D is equivalent to F; that is, $((\pi_{R_1}(F)) \cup ... \cup (\pi_{R_m}(F)))^+ = F^+$.
If a decomposition is not dependency-preserving, some dependency is lost in the decomposition. To check that a lost dependency holds, we must take the JOIN of two or more relations in the decomposition to get a relation that includes all left and right-hand-side attributes of the lost dependency, and then check that the dependency holds on the result of the JOIN—an option that is not practical.

Nonadditive (Lossless) Join Property of a Decomposition

Another property that a decomposition D should possess is the nonadditive join property, which ensures that no spurious tuples are generated when a NATURAL JOIN operation is applied to the relations resulting from the decomposition. Because this is a property of a decomposition of relation *schemas,* the condition of no spurious tuples should hold on *every legal relation state*—that is, every relation state that satisfies the functional dependencies in F. Hence, the lossless join property is always defined with respect to a specific set F of dependencies.

Formally, a decomposition $D = \{R_1, R_2, ..., R_m\}$ of R has the lossless (nonadditive) join property with respect to 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: $*(\pi_{R1}(r), ..., \pi_{Rm}(r)) = r$.

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 (π) and NATURAL JOIN (*) operations are applied; these additional tuples represent erroneous or invalid information. We prefer the term *nonadditive join* because it describes the situation more accurately. Although the term *lossless join* has been popular in the literature, *we will henceforth use the term nonadditive join*, which is self-explanatory and unambiguous. The nonadditive join property ensures that no spurious tuples result after the application of PROJECT and JOIN operations. We may, however, sometimes use the term lossy design to refer to a design that represents a loss of information.

Q.8 a. Why most databases are stored permanently on magnetic disk? Why are disks used to store online database files and not tapes? (5)

Answer:

Most databases are stored permanently (or *persistently*) on magnetic disk secondary storage, for the following reasons:

- Generally, databases are too large to fit entirely in main memory.
- The circumstances that cause permanent loss of stored data arise less frequently for disk secondary storage than for primary storage. Hence, we refer to disk—and other secondary storage devices—as **nonvolatile storage**, whereas main memory is often called **volatile storage**.
- The cost of storage per unit of data is an order of magnitude less for disk secondary storage than for primary storage.

Magnetic tapes are frequently used as storage medium for backing up databases because storage on tape costs even less than storage on disk. However, access to data on tape is quite slow. Data stored on tapes is **offline**; that is, some intervention by an operator—or an automatic loading device—to load a tape is needed before the data becomes available. In contrast, disks are **online** devices that can be accessed directly at any time.

b. What are the advantages of ordered files over unordered files? (5)

Answer:

Ordered records have some advantages over unordered files.

- First, reading the records in order of the ordering key values becomes extremely efficient, because no sorting is required.
- Second, finding the next record from the current one in order of the ordering key usually requires no additional block accesses, because the next record is in the same block as the current one (unless the current record is the last one in the block).

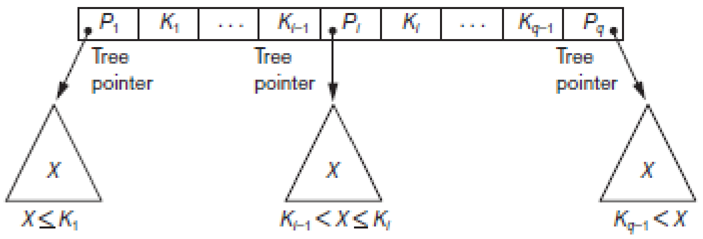
- Third, using a search condition based on the value of an ordering key field results in faster access when the binary search technique is used, which constitutes an improvement over linear searches, although it is not often used for disk files.

c. Describe the structure of internal nodes of a B^+ tree of order p. (6)

Answer:

The structure of the *internal nodes* of a B^+- tree of order p as shown in figure is as follows:

- Each internal node is of the form $<P_1, K_1, P_2, K_2, ..., P_{q-1}, K_{q-1}, P_q>$ where $q \le p$ and each P_i is a **tree pointer**.
- Within each internal node, $K_1 < K_2 < ... < K_{q-1}$.
- For all search field values X in the subtree pointed at by P_i, we have
  $K_{i-1} < X \le K_i$ for $1 < i < q$;
  $X \le K_i$ for $i = 1$;
  and $K_{i-1} < X$ for $i = q$.
- Each internal node has at most p tree pointers.
- Each internal node, except the root, has at least $\lceil(p/2)\rceil$ tree pointers. The root node has at least two tree pointers if it is an internal node.
- An internal node with q pointers, $q \le p$, has $q - 1$ search field values.



Internal nodes of a B^+-tree with q – 1 search values

Q.9 a. Discuss the different phases of external sorting? Also give an outline of the algorithm used? (8)

Answer:

External sorting refers to sorting algorithms that are suitable for large files of records stored on disk that do not fit entirely in main memory, such as most database files. The typical external sorting algorithm uses a **sort-merge strategy**, which starts by sorting small subfiles called **runs,** of the main file and then merges the sorted runs, creating larger sorted subfiles that are merged in turn. The sort-merge algorithm, like other database algorithms, requires *buffer space* in main memory, where the actual sorting and merging of the runs is performed. The buffer space in main memory is part of the **DBMS cache**—an area in the computer's main memory that is controlled by the DBMS. The buffer space is divided into individual buffers, where each **buffer** is the same size in bytes as the size of one disk block. Thus, one buffer can hold the contents of exactly *one disk block*. The basic algorithm consists of two phases:

- **The sorting phase:** In the **sorting phase**, runs (portions or pieces) of the file that can fit in the available buffer space are read into main memory, sorted using an *internal* sorting algorithm, and written back to disk as temporary sorted subfiles (or runs). The size of each run and the **number of initial runs (n_R)** are dictated by the **number of file blocks (b)** and the **available buffer space (n_B)**. For example, if the number of available main memory buffers n_B = 5 disk blocks and the size of the file b = 1024 disk blocks, then n_R= $\lceil(b/n_B)\rceil$ or 205 initial runs each of size 5 blocks (except the last run which will have only

4 blocks). Hence, after the sorting phase, 205 sorted runs (or 205 sorted subfiles of the original file) are stored as temporary subfiles on disk.

- **The merging phase:** In the **merging phase**, the sorted runs are merged during one or more **merge passes**. Each merge pass can have one or more merge steps. The **degree of merging (d_M)** is the number of sorted subfiles that can be merged in each merge step. During each merge step, one buffer block is needed to hold one disk block from each of the sorted subfiles being merged, and one additional buffer is needed for containing one disk block of the merge result, which will produce a larger sorted file that is the result of merging several smaller sorted subfiles. Hence, d_M is the smaller of $(n_B - 1)$ and n_R, and the number of merge passes is $[(\log_{dM}(n_R))]$. In our example where $n_B = 5$, $d_M = 4$ (four-way merging), so the 205 initial sorted runs would be merged 4 at a time in each step into 52 larger sorted subfiles at the end of the first merge pass. These 52 sorted files are then merged 4 at a time into 13 sorted files, which are then merged into 4 sorted files, and then finally into 1 fully sorted file, which means that *four passes* are needed. The minimum d_M of 2 gives the worst-case performance of the algorithm, which is
  $(2 * b) + (2 * (b * (\log_2 n_R)))$

Outline of the Sort-Merge algorithm for external sorting

```
set     i ← 1;
        j ← b; {size of the file in blocks}
        k ← nB; {size of buffer in blocks}
        m ← ⌈( j/k)⌉;

{Sorting Phase}
while (i ≤ m)
        do {
                read next k blocks of the file into the buffer or if there are less than k blocks
                remaining, then read in the remaining blocks;
                sort the records in the buffer and write as a temporary subfile;
                i ← i + 1;
        }

{Merging Phase: merge subfiles until only 1 remains}
set     i ← 1;
        p ← [logk–1m] {p is the number of passes for the merging phase}
        j ← m;

while (i ≤ p)
        do {
                n ← 1;
                q ← ( j/(k–1)⌉ ; {number of subfiles to write in this pass}

                while (n ≤ q)
                        do {
                                read next k–1 subfiles or remaining subfiles (from previous pass)
                                one block at a time; merge and write as new subfile one block at a
                        time;
                n ← n + 1;
            }
        j ← q;
        i ← i + 1;
}
```

b. Discuss the cost components for a cost function that is used to estimate query execution cost. What are the different parameters that are used in cost functions? Where is this information kept? (3+3+2)

Answer:

The cost of executing a query includes the following components:

1. ***Access cost to secondary storage:*** This is the cost of searching for, reading, and writing data blocks that reside on secondary storage, mainly on disk. The cost of searching for records in a file depends on the type of access structures on that file, such as ordering, hashing, and primary or secondary indexes. In addition, factors such as whether the file blocks are allocated contiguously on the same disk cylinder or scattered on the disk affect the access cost.

2. ***Storage cost:*** This is the cost of storing any intermediate files that are generated by an execution strategy for the query.

3. ***Computation cost:*** This is the cost of performing in-memory operations on the data buffers during query execution. Such operations include searching for and sorting records, merging records for a join, and performing computations on field values.

4. ***Memory usage cost:*** This is the cost pertaining to the number of memory buffers needed during query execution.

5. ***Communication cost:*** This is the cost of shipping the query and its results from the database site to the site or terminal where the query originated.

For large databases, the main emphasis is on minimizing the access cost to secondary storage. Simple cost functions ignore other factors and compare different query execution strategies in terms of the number of block transfers between disk and main memory. For smaller databases, where most of the data in the files involved in the query can be completely stored in memory, the emphasis is on minimizing computation cost. In distributed databases, where many sites are involved, communication cost must be minimized also. It is difficult to include all the cost components in a (weighted) cost function because of the difficulty of assigning suitable weights to the cost components. That is why some cost functions consider a single factor only—disk access

To estimate the costs of various execution strategies, we must keep track of any information that is needed for the cost functions. This information may be stored in the DBMS catalog, where it is accessed by the query optimizer. First, we must know the size of each file. For a file whose records are all of the same type, the **number of records (tuples) (*r*),** the (average) **record size (*R*),** and the **number of blocks (*b*)** (or close estimates of them) are needed. The **blocking factor (*bfr*)** for the file may also be needed. We must also keep track of the *primary access method* and the *primary access attributes* for each file. The file records may be unordered, ordered by an attribute with or without a primary or clustering index, or hashed on a key attribute. Information is kept on all

secondary indexes and indexing attributes. The **number of levels (*x*)** of each multilevel index (primary, secondary, or clustering) is needed for cost functions that estimate the number of block accesses that occur during query execution. In some cost functions the **number of first-level index blocks** is needed.
Another important parameter is the **number of distinct values (*d*)** of an attribute and its **selectivity (*sl*),** which is the fraction of records satisfying an equality condition on the attribute. This allows estimation of the **selection cardinality (*s = sl * r*)** of an attribute, which is the *average* number of records that will satisfy an equality selection condition on that attribute. For a *key attribute,* $d = r$, $sl = 1/r$ and $s = 1$. For a *nonkey attribute,* by making an assumption that the *d* distinct values are uniformly distributed among the records, we estimate $sl = (1/d)$ and so $s = (r/d)$ (Note 21).
Information such as the number of index levels is easy to maintain because it does not change very often. However, other information may change frequently; for example, the number of records *r* in a file changes every time a record is inserted or deleted. The query optimizer will need reasonably close but not necessarily completely up-to-the-minute values of these parameters for use in estimating the cost of various execution strategies.

TEXT BOOK

I. Fundamentals of Database Systems, Elmasri, Navathe, Somayajulu, Gupta, Pearson Education, 2006