Q.2. (a) What is Database Management System? What are the advantages of using database approach?

Ans: DATABASE MANAGEMENT SYSTEM

Data may be used to represent thing like name, telephone number, address, people name. The data can be stored using Microsoft ACCESS, EXCEL, etc. The data base is collections of data are group together to make large information. The database system is collection of programs that enables the users to create and manipulate a dbase. For example, to creates a new account in a bank and creates a new account in yahoo. The database management system is a general purpose software system, in which specifies defining, constructing and manipulating of the database.

1. It is collection of programs that enables users to create and manipulate a dbase
   - a. Ex: creates a new account in yahoo.com
   - b. Ex: creates a new account in a bank
2. It is a general-purpose software system
3. It specifies defining, constructing and manipulating.

Advantages of using DBMS

1. Controlling redundancy
   - a. Duplication effort
   - b. Wastage of storage space
   - c. Inconsistency in data
2. Restricting unauthorized access
   - a. Some users will not be authorized to access all information in Dbase
   - b. Ex: Financial data is often considered confidential
3. Good UI
   - a. DBMS should provide a menu driven S/W so that user can access the data without remembering(commands)
4. Providing multiple user interfaces
   - a. Provides a variety of users interfaces
5. System should support various types of users with varying knowledge
   - a. Query language interface for casual users
   - b. Programming language interface for application programmers
   - c. Formal and command interfaces for particular users
   - d. Menu-driven interface and natural language interface for standalone users
6. Representing complex relationship among data
   - a. Ex: Student name is Prasad in student table
7. Enforcing Integrity Constraints
   - a. Each record in a table should have a proper semantic relationship with a record of another table.

b. If employee works in dept no.5, then there must be dept table in which has dept no.5.

8. Providing backup and recovery
    a. It should provide facilities for recovering data from s/w and h/w failures.

Q.2 (b) Describe the three-schema architecture. What is data independence? Explain**.**

Ans:-The goal of the three-schema architecture is to separate the user applications and the physical database.

In this architecture, schemas can be defined at the following three levels:

1. The internal level has an internal schema, which describes the physical storage structure of the database. The internal schema uses a physical data model and describes the complete details of data storage and access paths for the database.
2. The conceptual level has a conceptual schema, which describes the structure of the whole database for a community of users. The conceptual schema hides the details of physical storage structures and concentrates on describing entities, data types, relationships, user operations, and constraints. A high-level data model or an implementation data model can be used at this level.
3. The external or view level includes a number of external schemas or user views. Each external schema describes the part of the database that a particular user group is interested in and hides the rest of the database from that user group. A high-level data model or an implementation data model can be used at this level.

The three-schema architecture is a convenient tool for the user to visualize the schema levels in a database system. Most DBMS do not separate the three levels completely, but support the three- schema architecture to some extent. Some DBMSs may include physical-level details in the conceptual schema. In most DBMSs that support user views, external schemas are specified in the same data model that describes the conceptual-level information. Some DBMSs allow different data models to be used at the conceptual and external levels.

Data Independence

The three-schema architecture can be used to explain the concept of data independence, which can be defined as the capacity to change the schema at one level of a database system without having to change the schema at the next higher level. We can define two types of data independence:

1. Logical data independence is the capacity to change the conceptual schema without having to change external schemas or application programs. We may change the conceptual schema to expand the database (by adding a record type or data item), or to reduce the database (by removing a record type or data item). In the latter case, external schemas that refer only to the remaining data should not be affected.

Only the view definition and the mappings need be changed in a DBMS that supports logical data independence. Application programs that reference the external schema constructs must work as before, after the conceptual schema undergoes a logical reorganization. Changes to constraints can be applied also to the conceptual schema without affecting the external schemas or application programs.

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

The three-schema architecture can make it easier to achieve true data independence, both physical and logical.

Q.3) (a) Illustrate and explain the main phases of database design.

Ans: Figure shows a simplified description 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 ixith retrievals and updates. In software design, it is common to use *data flow diagrams*, *sequence diagrams*, *scenarios*, and other techniques for specifying functional requirements.

Once all 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 the database designers to concentrate on specifying the properties of the data, without being concerned with storage details. Consequently, it is easier for them to come up with a good conceptual database design.

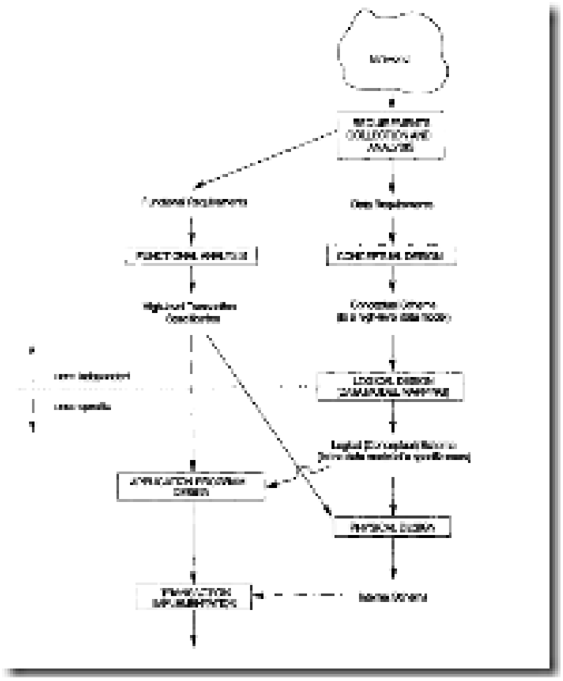

Figure: A simplified diagram to illustrate the main phases of database design

During or after the conceptual schema design, the basic data model operations can be used r.o specify the high-level user 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 mode! – 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, and its result is a database schema in the implementation data model of the DBMS.

The last step is the physical design phase, during which the internal storage structures, indexes, access paths, and file organizations for the database tiles are specified, iii parallel with these activities, application programs are designed and implemented as database transactions corresponding to the high-level transaction specifications.

Q.3 (b) what are the Weak Entity Types? Explain with the help of examples.

Ans: Entity types that do not have key attributes of their own are called weak entity types. In contrast, regular entity types that do have a key attribute – which include all the examples we discussed so far – are called strong entity types. Entities belonging to a weak entity type are identified by being related to specific entities from another entity type in combination with one of their attribute values. We call this other entity type identifying or owner entity type, and we call the relationship type that relates a weak entity type to its Tier the identifying relationship of the weak entity type A weak entity type always has a total participation astraint (existence dependency) with respect to its identifying relationship, because a weak entity cannot identified without an owner entity. However, not every existence dependency results in a weak entity type, For example, a driver_j_icense entity cannot exist unless it is related to a PERSON entity, even though it has own key (LicenseNumber) and hence is not a weak entity.

Consider the entity type DEPENDENT, related to EMPLOYEE, which is used to keep track of the dependents each employee via a 1:N relationship. The attributes of DEPENDENT are Name (the first name the dependent), DOB, Sex, and Relationship (to the employee). Two dependents of two distinct employees say, by chance, have the same values for Name, DOB, Sex, and Relationship, but they are still distinct entities. They are identified as distinct entities only after determining the particular employee entry to which each dependent is related. Each employee entity is said to own the dependent entities that are related to it.

A weak entity type normally has a partial key, which is the set of attributes that can uniquely identify weak entities that are related to the same owner entity.10 In our example, if we assume that no two dependents of the same employee ever have the same first name, the attribute Name of DEPENDENT is the partial key. In the worst case, a composite attribute of all the weak entity's attributes will be the partial key.

In ER diagrams, both a weak entity type and its identifying relationship are distinguished by surrounding their boxes and diamonds with double lines. The partial key attribute is underlined with a hashed or dotted line.

Q.4.a) Explain Relational database design using ER - to - Relational Mapping.

Ans: - ER-to-Relational Mapping

The steps of an algorithm for ER-to-relational mapping: -

1: 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 primary key for R. If the chosen key of E is composite, the set of simple attributes that form it will together form the primary key of R.

2: 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 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.

3: 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. Choose one of the relations—S, say—and include as foreign key in S the primary key of T. It is better to choose an entity type with *total participation* in R in the role of S. Include all the simple attributes (or simple components of composite attributes) of the 1:1 relationship type R as attributes of S.

4: 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; this is 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.

5: 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—as we did for 1:1 or 1:N relationship types—because of the M:N cardinality ratio.

6: 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 an attribute. The primary key of R is the combination of A and K. If the multivalued attribute is composite, we include its simple components.

7: 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.4.b) Explain Join and Division operations.

Ans: JOIN

The JOIN operation, denoted by , is used to combin related tuples from two relations into single tuples. This operation is very important for any relational database with more than a single relation, because it allows us to process relationships among relations. To illustrate join, suppose that we want to retrieve the name of the manager of each department. To get the manager’s name, we need to combine each department tuple with the employee tuple whose SSN value matches the MGRSSN value in the department tuple. We do this by using the JOIN operation, and then projecting the result over the necessary attributes, as follows:

DEPT_MGRãDEPARTMENTMGRSSN=SSN EMPLOYEE

RESULTãpDNAME, LNAME, FNAME(DEPT_MGR)

Combine each department tuple with the employee tuple whose SSN value matches the MGRSSN value in the department tuple. We do this by using the JOIN operation, and then projecting the result over the necessary attributes, as follows:

DEPT_MGRãDEPARTMENTMGRSSN=SSN EMPLOYEE

RESULTãpDNAME, LNAME, FNAME(DEPT_MGR)

EMP_DEPENDENTSãEMPNAMES x DEPENDENT
ACTUAL_DEPENDENTSãsSSN=ESSN (EMP_DEPENDENTS)
with a single JOIN operation:

ACTUAL_DEPENDENTSãEMPNAMESSSN=ESSN DEPENDENT

The DIVISION Operation
The DIVISION operation is useful for a special kind of query that sometimes occurs in database applications. An example is "Retrieve the names of employees who work on all the projects that ‘John Smith’ works on." To express this query using the DIVISION operation, proceed as follows. First, retrieve the list of project numbers that ‘John Smith’ works on in the intermediate relation SMITH_PNOS:

SMITHãsFNAME=’John’ AND LNAME=’Smith’(EMPLOYEE)
SMITH_PNOSãpPNO(WORKS_ONESSN=SSN SMITH)
Next, create a relation that includes a tuple <PNO, ESSN> whenever the employee whose social security
number is ESSN works on the project whose number is PNO in the intermediate relation SSN_PNOS:
SSN_PNOSãpESSN,PNO (WORKS_ON)
Finally, apply the DIVISION operation to the two relations, which gives the desired employees’ social
security numbers:
SSNS(SSN) ãSSN_PNOS ÷ SMITH_PNOS

RESULTãpFNAME, LNAME(SSNS * EMPLOYEE)

Q.5 (a) Discuss the following SQL commands with example:-

(iv) Alter Table

Ans: - ALTER TABLE Command

The definition of a base table can be changed by using the ALTER TABLE command, which is a schema evolution command. The possible *alter table actions* include adding or dropping a column (attribute), changing a column definition, and adding or dropping table constraints. For example, to add an attribute for keeping track of jobs of employees to the EMPLOYEE base relations in the COMPANY schema, we can use the command:

ALTER TABLE COMPANY.EMPLOYEE ADD JOB VARCHAR(12);

We must still enter a value for the new attribute JOB for each individual EMPLOYEE tuple. This can be done either by specifying a default clause or by using the UPDATE command. If no default clause is specified, the new attribute will have NULL in all the tuples of the relation immediately after the command is executed; hence, the NOT NULL constraint is *not allowed* in this case.

To drop a column, we must choose either CASCADE or RESTRICT for drop behavior. If CASCADE is chosen, all constraints and views that reference the column are dropped automatically from the schema, along with the column. If RESTRICT is chosen, the command is successful only if no views or constraints reference the column. For example, the following command removes the attribute ADDRESS from the EMPLOYEE base table:

ALTER TABLE COMPANY.EMPLOYEE DROP ADDRESS CASCADE;

Q.6 (a) Explain Second and third Normal Form with the help of suitable examples.

Ans: 1. Second normal form (2NF) is based on the concept of full functional dependency. A functional dependency *X*â *Y* is a full functional dependency if removal of any attribute *A* from *X* means that the dependency does not hold any more; that is, for any attribute *A X*, (*X* - {*A*}) *does not* functionally determine *Y*. A functional dependency *X* â *Y* is a partial dependency if some attribute *A X* can be removed from *X* and the dependency still holds; that is, for some *A X*, (*X* - {*A*}) â *Y*. In Figure, {SSN, PNUMBER} âHOURS is a full dependency (neither SSNâHOURS nor PNUMBERâ HOURS holds). However, the dependency {SSN, PNUMBER} âENAME is partial because SSNâENAME holds.

The test for 2NF involves testing for functional dependencies whose left-hand side attributes are part of the primary key. If the primary key contains a single attribute, the test need not be applied at all. A relation schema *R* is in 2NF if every nonprime attribute *A* in *R* is *fully functionally dependent* on the primary key of *R*.

2. Third Normal Form

Third normal form (3NF) is based on the concept of *transitive dependency*. A functional dependency *X*â *Y* in a relation schema *R* is a transitive dependency if there is a set of attributes *Z* that is neither a candidate key nor a subset of any key of *R*, and both *X*â *Z* and *Z*â*Y* hold.

According to Codd's original definition, a relation schema *R* is in 3NF if it satisfies 2NF *and* no nonprime attribute of *R* is transitively dependent on the primary key.

Q.6 (b) What do you mean by Functional dependencies? Explain.

Ans: A functional dependency is a constraint between two sets of attributes from the database. Suppose that our relational database schema has *n* attributes , , . . . ; let us think of the whole database as being described by a single **universal** relation schema. We do not imply that we will actually store the database as a single universal table; we use this concept only in developing the formal theory of data dependencies .
A functional dependency, denoted by *X* â*Y*, between two sets of attributes *X* and *Y* that are subsets of *R* specifies a *constraint* on the possible tuples that can form a relation state *r* of *R*. The constraint is that, for any two tuples and in r that have [*X*] = [*X*], we must also have [*Y*] = [*Y*]. This means that the values of the *Y* component of a tuple in *r* depend on, or are *determined by,* the values of the *X* component; or alternatively, the values of the *X* component of a tuple uniquely (or functionally) determine the values of the *Y* component. We also say that there is a functional dependency from *X* to*Y* or that *Y* is functionally dependent on *X*. The abbreviation for functional dependency is FD or f.d.
The set of attributes *X* is called the left-hand side of the FD, and *Y* is called the right-hand side.
Thus *X* functionally determines *Y* in a relation schema *R* if and only if, whenever two tuples of *r*(*R*) agree on their *X*-value, they must necessarily agree on their *Y*-value.

If a constraint on *R* states that there cannot be more than one tuple with a given *X*-value in any relation instance *r*(*R*)—that is, *X* is a candidate key of *R*—this implies that *X*â*Y* for any subset of attributes *Y* of *R* (because the key constraint implies that no two tuples in any legal state *r*(*R*) will have the same value of *X*).

• If *X* â*Y* in *R*, this does not say whether or not *Y* â*X* in *R.*
A functional dependency is a property of the semantics or meaning of the attributes. The database designers will use their understanding of the semantics of the attributes of *R*—that is, how they relate to *r* of *R.* Whenever the semantics of two sets of attributes in *R* indicate that a functional dependency should hold, we specify the dependency as a constraint. Relation extensions *r*(*R*) that satisfy the functional dependency constraints are called legal extensions (or legal relation states) of *R*, because they obey the functional dependency constraints. Hence, the main use of functional dependencies is to describe further a relation schema *R* by specifying constraints on its attributes that must hold *at all times.* Certain FDs can be specified without referring to a specific relation, but as a property of those attributes.

Q.7 (a) Explain fourth normal form.

Ans: Fourth Normal Form is violated when a relation has undesirable multivalued dependencies, and hence can be used to identify and decompose such relations. A relation schema *R* is in 4NF with respect to a set of dependencies *F* (that includes functional dependencies and multivalued dependencies) if, for every *nontrivial* multivalued dependency *X Y* in, *X* is a superkey for R.

Multivalued dependencies are a consequence of first normal form (1NF), which disallowed an attribute in a tuple to have a *set of values.* If we have two or more multivalued *independent* attributes in the same relation schema, we get into a problem of having to repeat every value of one of the attributes with every value of the other attribute to keep the relation state consistent and to maintain the independence among the attributes involved. This constraint is specified by a multivalued dependency.

For example, consider the relation EMP. A tuple in EMP relation represents the fact that an employee whose name is ENAME works on the project whose name is PNAME and has a dependent whose name is DNAME. An employee may work on several projects and may have several dependents, and the employee's projects and dependents are independent of one another. To keep the relation state consistent, we must have a separate tuple to represent every combination of an employee's dependent and an employee's project. This constraint is specified as a multivalued dependency on the EMP relation. Informally, whenever two *independent* 1:N relationships *A:B* and *A:C* are mixed in the same relation, an MVD may arise.

The EMP relation is not in 4NF because in the nontrivial MVDs ENAME PNAME and ENAME DNAME, ENAME is not a superkey of EMP. We decompose EMP into EMP_PROJECTS and EMP_DEPENDENTS, .Both EMP_PROJECTS and EMP_ DEPENDENTS are in 4NF, because the MVDs ENAME PNAME in EMP_PROJECTS and ENAME DNAME in EMP_DEPENDENTS are trivial MVDs. No other nontrivial MVDs hold in either EMP_PROJECTS or EMP_DEPENDENTS. No FDs hold in these relation schemas either.

Q.8 (a) Discuss the basic operations on files.

Ans: Operations on files are usually grouped into retrieval operations and update operations. The former does not change any data in the file, but only locate certain records so that their field values can be examined and processed. The latter change the file by insertion or deletion of records or by modification of field values. In either case, we may have to select one or more records for retrieval, deletion, or modification based on a selection condition (or filtering condition), which specifies criteria that the desired record or records must satisfy.

Consider an EMPLOYEE file with fields NAME, ENO, SALARY, JOBCODE, and DEPARTMENT. A simple selection condition may involve an equality comparison on some field value—for example, (ENO = ‘123456789′) or (department = ‘Research’). More complex conditions can involve other types of comparison operators, such as > or >; an example is (SALARY > 30000). The general case is to have an arbitrary Boolean expression on the fields of the file as the selection condition.

Search operations on files are generally based on simple selection conditions. A complex condition must be decomposed by the DBMS (or the programmer) to extract a simple condition that can be used to locate the records on disk. Each located record is then checked to determine whether it satisfies the full selection condition. For example, we may extract the simple condition (DEPARTMENT = ‘Research’) from the complex condition ((SALARY > 30000) AND (DEPARTMENT = ‘Research’)); each record satisfying (DEPARTMENT = ‘Research’) is located and then tested to see if it also satisfies (salary > 30000).

When several file records satisfy a search condition, the *first* record – with respect to the physical sequence of file records – is initially located and designated the current record. Subsequent search operations commence from this record and locate the next record in the file that satisfies the condition.

Actual operations for locating and accessing file records vary from system to system. Below, we present a set of representative operations. Typically, high-level programs, such as DBMS software programs, access the records by using these commands, so we sometimes refer to program variables in the following descriptions:

· Open: Prepares the file for reading or writing. Allocates appropriate buffers (typically at least two) to hold file blocks from disk, and retrieves the file header. Sets the file pointer to the beginning of the file.

· Reset: Sets the file pointer of an open file to the beginning of the file.

· Find (or Locate): Searches for the first record that satisfies a search condition. Transfers the block containing that record into a main memory buffer (if it is not already there). The file pointer points to the record in the buffer and it becomes the current record. Sometimes, different verbs are used to indicate whether the located record is to be retrieved or updated.

· Read (or Get): Copies the current record from the buffer to a program variable in the user program. This command may also advance the current record pointer to the next record in the file, which may necessitate reading the next file block from disk.

· FindNext: Searches for the next record in the file that satisfies the search condition. Transfers the block containing that record into a main memory buffer (if it is not already there). The record is located in the buffer and becomes the current record.

· Delete: Deletes the current record and (eventually) updates the file on disk to reflect the deletion.

· Modify: Modifies some field values for the current record and (eventually) updates the file on disk to reflect the modification.

· Insert: Inserts a new record in the file by locating the block where the record is to be inserted, transferring that block into a main memory buffer (if it is not already there), writing the record into the buffer, and (eventually) writing the buffer to disk to reflect the insertion.

· Close: Completes the file access by releasing the buffers and performing any other needed cleanup operations.

Q.8 (b) Explain the Types of single-level ordered indexes.

Ans: For a file with a given record structure consisting of several fields (or attributes), an index access structure is usually defined on a single field of a file, called an indexing field (or indexing attribute). The index typically stores each value of the index field along with a list of pointers to all disk blocks that contain records with that field value. The values in the index are ordered so that we can do a binary search on the index. The index file is much smaller than the data file, so searching the index using a binary search is reasonably efficient. Multilevel indexing does away with the need for a binary search at the expense of creating indexes to the index itself.

There are several types of ordered indexes. A primary index is specified on the ordering key field of an ordered file of records.

Three types of single-level indexes.

Primary indexes A primary index is an ordered file whose records are of fixed length with two fields. The first field is of the same data type as the ordering key field called the primary key of the data file, and the second field is a pointer to a disk block (a block address). There is one index entry (or index record) in the index file for each block in the data file. Each index entry has the value of the primary key field for the first record in a block and a pointer to that block as its two field values.

We will refer to the two field values of index entry i as <K(i), P(i) >.

To create a primary index on the ordered file, we use the NAME field as primary key, because that is the ordering key field of the file (assuming that each value of NAME is unique). Each entry in the index has a NAME value and a pointer. The first three index entries are as follows:

<K(1) = (Abbas), P(l) = address of block 1>

<K(2) = (Agarkar), P(2) = address of block 2>

<K(3) = (Akthar), P(3) = address of block 3 >

The total number of entries in the index is the same as the number of disk blocks in the ordered data file. The first record in each block of the data file is called the anchor record of the block, or simply the block anchor.

Indexes can also be characterized as dense or sparse. A dense index has an index entry for every search key value (and hence every record) in the data file. A sparse (or nondense) index, on the other hand, has index entries for only some of the search values. A primary index is hence a nondense (sparse) index, since it includes an entry for each disk block of the data file and the keys of its anchor record rather than for every search value (or every record).

Secondary indexes A secondary index provides a secondary means of accessing a file for which some primary access already exists. The secondary index may be on a field which is a candidate key and has a unique value in every record, or a nonkey with duplicate values. The index is an ordered file with two fields. The first field is of the same data type as some nonordering *field* of the data file that is an indexing field. The second field is either a *block* pointer or a *record* pointer. There can be *many* secondary indexes (and hence, indexing fields) for the same file.

We first consider a secondary index access structure on a key field that has a *distinct value* for every record. Such a field is sometimes called a secondary key. In this case there is one index entry for *each record* in the data file, which contains the value of the secondary key for the record and a pointer either to the block in which the record is stored or to the record itself. Hence, such an index is dense.

We again refer to the two field values of index entry *i* as *<K(i), P(i)>*. The entries are ordered by value of *K(i)*, so we can perform a binary search. Because the records of the data file are *not* physically ordered by values of the secondary key field, we *cannot* use block anchors. That is why an index entry is created for each record in the data file, rather than for each block, as in the case of a primary index.

A secondary index usually needs more storage space and longer search time than does a primary index, because of its larger number of entries. However, the *improvement* in search time for an arbitrary record is much greater for a secondary index than for a primary index, since we would have to do a *linear search* on the data file if the secondary index did not exist.

Q.9 (a) How to translate SQL queries into relational algebra?

Ans: SQL is the query language that is used in most commercial RDBMSs. An SQL query is first translated into an equivalent extended relational algebra expression—represented as a query tree data structure—that is then optimized. Typically, SQL queries are decomposed into query blocks, which form the basic units that can be translated into the algebraic operators and optimized. A query block contains a single SELECT-FROM-WHERE expression, as well as GROUP BY and HAVING clauses if these are part of the block. Hence, nested queries within a query are identified as separate query blocks. Because SQL includes aggregate operators—such as MAX, MIN, SUM, AND COUNT—these operators must also be included in the extended algebra.

Consider the following SQL query on the EMPLOYEE relation

```
SELECT LNAME, FNAME
FROM  EMPLOYEE
WHERE  SALARY >  (SELECT   MAX (SALARY)
      FROM   EMPLOYEE
      WHERE   DNO=5);
```

This query includes a nested subquery and hence would be decomposed into two blocks.
The inner block is

```
  (SELECT MAX (SALARY)
  FROM  EMPLOYEE
  WHERE  DNO=5)
```

and the outer block is

```
  SELECT LNAME, FNAME
FROM  EMPLOYEE
  WHERE  SALARY > c
```

where c represents the result returned from the inner block. The inner block could be translated into the extended relational algebra expression
MAX SALARY(sDNO=5(EMPLOYEE))
and the outer block into the expression
pLNAME, FNAME(sSALARY>C(EMPLOYEE))
The query optimizer would then choose an execution plan for each block.
the inner block needs to be evaluated only once to produce the maximum salary, which is then used—as the constant c—by the outer block.

Q.9 (b) What are the aggregate operations and how they can be implemented?

Ans: Implementing Aggregate Operations:-

The aggregate operators (MIN, MAX, COUNT, AVERAGE, SUM), when applied to an entire table, can be computed by a table scan or by using an appropriate index, if available.

For example, consider the following SQL query:

SELECT MAX (SALARY) FROM EMPLOYEE;

If an (ascending) index on SALARY exists for the EMPLOYEE relation, then the optimizer can decide on using the index to search for the largest value by following the *rightmost* pointer in each index node from the root to the rightmost leaf. That node would include the largest SALARY value as its *last* entry. In most cases, this would be more efficient than a full table scan of EMPLOYEE, since no actual records need to be retrieved. The MIN aggregate can be handled in a similar manner, except that the *leftmost* pointer is followed from the root to leftmost leaf. That node would include the smallest SALARY value as its *first* entry.

The index could also be used for the COUNT, AVERAGE, and SUM aggregates, but only if it is a dense index—that is, if there is an index entry for every record in the main file. In this case, the associated computation would be applied to the values in the index. For a nondense index, the actual number of records associated with each index entry must be used for a correct computation (except for COUNT DISTINCT, where the number of distinct values can be counted from the index itself).

When a GROUP BY clause is used in a query, the aggregate operator must be applied separately to each group of tuples. Hence, the table must first be partitioned into subsets of tuples, where each partition (group) has the same value for the grouping attributes. In this case, the computation is more complex.

TEXTBOOK: Fundamentals of database systems, ELMASRI, Navathe, Somayajulu, Gupta, Pearson Education, 2006.