Q.2 (a) What are the different types of database end users? Discuss the main activities of each.

Answer:

End users are the people whose jobs require access to the database for querying, updating, and generating reports; the database primarily

exists for their use. There are several categories of end users:

• **Casual end users** occasionally access the database, but they may need different information each time. They use a sophisticated database query language to specify their requests and are typically middle- or high-level managers or other occasional browsers.

• **Naive** or **parametric end users** make up a sizable portion of database end users. Their main job function revolves around constantly querying and updating the database, using standard types of queries and updates—called **canned transactions—**that have been carefully programmed and tested.

- **Sophisticated end users** include engineers, scientists, business analysts, and others who thoroughly familiarize themselves with the facilities of the DBMS so as to implement their applications to meet their complex requirements.

• **Stand-alone users** maintain personal databases by using ready-made program packages that provide easy-to-use menu- or graphics-based interfaces. An example is the user of a tax package that stores a variety of personal financial data for tax purposes.

Q.2 (b) Describe the three-schema architecture. What is the difference between logical data independence and physical data independence?

Answer:

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 schem levels in a database system. Most DBMSs 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) Discuss the database design process using high level data model.

Answer:

The database design process consists of a number of steps listed below.

Step 1: Requirements Collection and Analysis

- Prospective users are interviewed to understand and document data requirements
- This step results in a concise set of user requirements, which should be detailed and complete.
- The functional requirements should be specified, as well as the data requirements. Functional requirements consist of user operations that will be applied to the database, including retrievals and updates.

- Functional requirements can be documented using diagrams such as sequence diagrams, data flow diagrams, scenarios, etc.

Step 2: Conceptual Design
- Once the requirements are collected and analyzed, the designers go about creating the conceptual schema.
- Conceptual schema: concise description of data requirements of the users, and includes a detailed description of the entity types, relationships and constraints.
- The concepts do not include implementation details; therefore the end users easily understand them, and they can be used as a communication tool.
- The conceptual schema is used to ensure all user requirements are met, and they do not conflict.

Step 3: Database Implementation
- Many DBMS systems use an implementation data 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, which results in the implementation data model of the DBMS.

Step 4: Physical Design
- Internal storage structures, indexes, access paths and file organizations are specified.

Application programs are designed and implemented.

Q.3 (b.1) Define the entity type and entity set.
Answer:
An **entity type** defines a collection of entities that have the same attributes. Each entity type in the database is described by its name and attributes. The entity shares the same attributes, but each entity has its own value for each attribute.

Entity Type Example:
- *Entity Type*:
  Student
- *Entity Attributes***:**
  StudentID,
  Name,
  Surname,
  Date of Birth,
  Department
- The collection of all entities of a particular entity type in the database at any point in time is called an entity set. The entity type (Student) and the entity set (Student) can be referred to using the same name.

Entity Set Example:
- Entity Type: Student

- Entity Set:
  [123, John, Smith, 12/01/1981, Computer Technology]
  [456, Jane, Doe, 05/02/1979, Mathematics]
  [789, Semra, Aykan, 02/08/1980, Linguistics]

The entity type describes the **intension**, or schema for a set of entities that share the same structure. The collection of entities of a particular entity type is grouped into the entity set, called the **extension**.

Q.3. (b.2) Define foreign key .What is this concept used for?

Answer:

The conditions for a foreign key, given below, specify a referential integrity constraint between the two relation schemas R1 and R2. A set of attributes FK in relation schema R1 is a foreign key of R1 that references relation R2 if it satisfies the following two rules:

1. The attributes in FK have the same domain(s) as the primary key attributes PK of R2; the attributes FK are said to reference or refer to the relation R2.
2. A value of FK in a tuple t1 of the current state r1 (R1) either occurs as a value of PK for some tuple t2 in the current state r2 (R2) or is null. In the former case, we have t1[FK] = t2 [PK], and we say that the tuple t1 references or refers to the tuple t2. R1 is called the referencing relation and R2 is the referenced relation.

Q.4 (a) What are the unary relational operations? Explain.

Answer:

SELECT Operation

The **SELECT** operation is used to select a *subset* of the tuples from a relation that satisfy a **selection condition.** One can consider the SELECT operation to be a *filter* that keeps only those tuples that satisfy a qualifying condition. For example, to select the EMPLOYEE tuples whose department is 4, or those whose salary is greater than $30,000, we can individually specify each of these two conditions with a SELECT operation as follows:

sDNO=4(EMPLOYEE)

sSALARY>30000(EMPLOYEE)

In general, the SELECT operation is denoted by

s<selection condition>(R)

Where the symbol s (sigma) is used to denote the SELECT operator, and the selection condition is a Boolean expression specified on the attributes of relation R. Notice that R is generally a *relational algebra expression* whose result is a relation; the simplest expression is just the name of a database relation. The relation resulting from the SELECT operation has the *same attributes* as R. The Boolean expression specified in <selection condition> is made up of a number of **clauses** of the form

<attribute name> <comparison op> <constant value>, or

<attribute name> <comparison op> <attribute name>

where <attribute name> is the name of an attribute of R, <comparison op> is normally one of the operators {=, <, 1, >, , }, and <constant value> is a constant value from the attribute domain. Clauses can be arbitrarily connected by the Boolean operators AND, OR, and NOT to form a general selection condition.

2. The PROJECT Operation

If we think of a relation as a table, the SELECT operation selects some of the *rows* from the table while discarding other rows. The **PROJECT** operation, on the other hand, selects certain *columns* from the table and discards the other columns. If we are interested in only certain attributes of a relation, we use the PROJECT operation to *project* the relation over these attributes only. For example, to list each employee's first and last name and salary, we can use the PROJECT operation as follows:

pLNAME, FNAME, SALARY(EMPLOYEE)

p<attribute list>(R)

The general form of the PROJECT operation is where p (pi) is the symbol used to represent the PROJECT operation and <attribute list> is a list of attributes from the attributes of relation R. Again, notice that R is, in general, a *relational algebra expression* whose result is a relation, which in the simplest case is just the name of a database relation. The result of the PROJECT operation has only the attributes specified in <attribute list> and *in the same order as they appear in the list.* Hence, its **degree** is equal to the number of attributes in <attribute list>.

Q.4 (b). Discuss the steps in relational database design using ER-to-Relational mapping.

Answer:

ER-to-Relational Mapping Algorithm

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

STEP 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.

STEP 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.

STEP 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.

STEP 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.

STEP 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.

STEP 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.

STEP 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.5 (a) Discuss the following SQL commands with example:-
(i) Create table
(ii) Drop Table
(iii) Drop Schema
(iv) Alter Table

Answer:

(i)The CREATE TABLE command is used to specify a new relation by giving it a name and specifying its attributes and constraints. The attributes are specified first, and each attribute is given a name, a data type to specify its domain of values, and any attribute constraints such as NOT NULL. The key, entity integrity, and referential integrity constraints can be specified—within the CREATE TABLE statement.

CREATE TABLE statements are executed. Alternatively, we can explicitly attach the schema name to the relation name, separated by a period. For example, by writing:

CREATE TABLE COMPANY.EMPLOYEE ...

rather than

CREATE TABLE EMPLOYEE ...

(ii)The DROP SCHEMA Commands

If a whole schema is not needed any more, the DROP SCHEMA command can be used. There are two *drop behavior* options: CASCADE and RESTRICT. For example, to remove the COMPANY database schema and all its tables, domains, and other elements, the CASCADE option is used as follows:

DROP SCHEMA COMPANY **CASCADE;**

If the RESTRICT option is chosen in place of CASCADE, the schema is dropped only if it has *no elements* in it; otherwise, the DROP command will not be executed.

If a base relation within a schema is not needed any longer, the relation and its definition can be deleted by using the DROP TABLE command.

(iii)**DROP TABLE** DEPENDENT **CASCADE;**

If the RESTRICT option is chosen instead of CASCADE, a table is dropped only if it is *not referenced* in any constraints (for example, by foreign key definitions in another relation) or views. With the CASCADE option, all such constraints and views that reference the table are dropped automatically from the schema, along with the table itself.

(iv) 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 (see Section 8.4). If no default clause is specified, the new attribute will have NULLs 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) Discuss the insertion, deletion and modification anomalies.

Answer:

Insertion Anomalies

These 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 the attribute values of department 5 correctly so that they are *consistent* with 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 causes a problem because SSN is the primary key of EMP_DEPT, and each tuple is supposed to represent an employee entity—not a department entity. Moreover, when the first employee is assigned to that department, we do not need the tuple with null values any more.

Deletion Anomalies

This problem is related to the second insertion anomaly situation discussed above. 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 should not be the case.

Q.6 (b) Explain first, second and third normal forms with suitable examples.

Answer:

First normal form (1NF) is now considered to be part of the formal definition of a relation in the basic (flat) relational model; it was defined to disallow multivalued attributes, composite attributes, and their combinations. It states that the domain of an attribute must include only *atomic* (simple, indivisible) *values* and that the value of any attribute in a tuple must be a *single value* from the domain of that attribute. Hence, 1NF disallows having a set of values, a tuple of values, or a combination of both as an attribute value for a *single tuple.* In other words, 1NF disallows "relations within relations" or "relations as attributes of tuples." The only attribute values permitted by 1NF are single **atomic** (or **indivisible**) **values.**

Consider the DEPARTMENT relation schema shown in Figure1, whose primary key is DNUMBER, and suppose that we extend it by including the DLOCATIONS attribute as

shown in Figure 2(a). We assume that each department can have *a number of* locations. The DEPARTMENT schema and an example extension are shown in Figure 2. As we can see, this is not in 1NF because DLOCATIONS is not an atomic attribute, as illustrated by the first tuple in Figure 2(b). There are two ways we can look at the DLOCATIONS attribute:

- The domain of DLOCATIONS contains atomic values, but some tuples can have a set of these values. In this case, DLOCATIONS *is not* functionally dependent on DNUMBER.

• The domain of DLOCATIONS contains sets of values and hence is nonatomic. In this case, DNUMBERâDLOCATIONS, because each set is considered a single member of the attribute domain.

2**. 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*.

3 .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.

Figure 1 Simplified version of the COMPANY relational database schema.

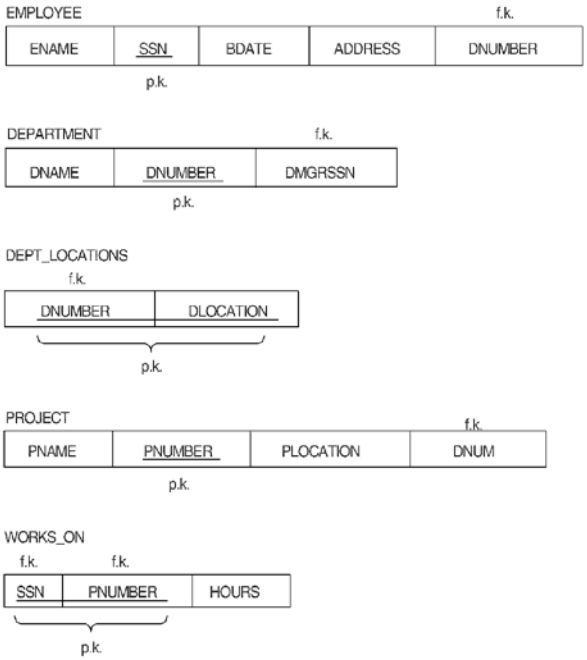


Figure Normalization into 1NF. (a) Relation schema that is not in 1NF. (b) Example relation instance. (c) 1NF relation with redundancy.

(a)

DEPARTMENT

DNAME	DNUMBER	DMGRSSN	DLOCATIONS

(b)

DEPARTMENT

DNAME	DNUMBER	DMGRSSN	DLOCATIONS
Research	5	333445555	{Bellaire, Sugarland, Houston}
Administration	4	987654321	{Stafford}
Headquarters	1	888665555	{Houston}

(c)

DEPARTMENT

DNAME	DNUMBER	DMGRSSN	DLOCATION
Research	5	333445555	Bellaire
Research	5	333445555	Sugarland
Research	5	333445555	Houston
Administration	4	987654321	Stafford
Headquarters	1	888665555	Houston

Q.7 (a). Discuss multivalued dependencies and fourth normal form.

Answer:

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.

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.

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.

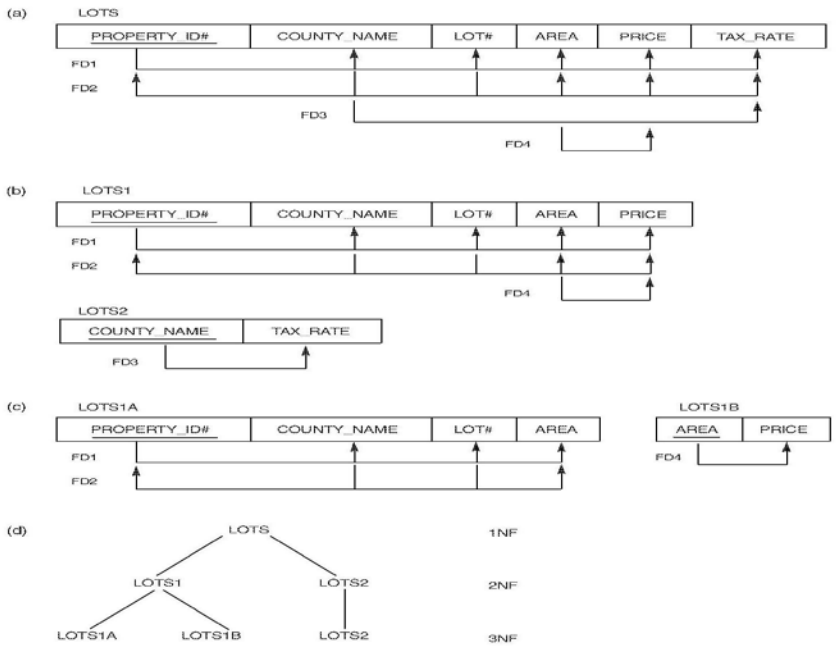
Q.7 (b) Write note on Boyce-Codd normal form.

Answer:

Boyce-Codd normal form (BCNF) was proposed as a simpler form of 3NF, but it was found to be stricter than 3NF, because every relation in BCNF is also in 3NF; however, a relation in 3NF is *not necessarily* in BCNF. Intuitively, we can see the need for a stronger normal form than 3NF by going back to the LOTS relation schema of Figure with its four functional dependencies, FD1 through FD4. Suppose that we have thousands of lots in the relation but the lots are from only two counties: Dekalb and Fulton. Suppose also that lot sizes in Dekalb County are only 0.5, 0.6, 0.7, 0.8, 0.9, and 1.0 acres, whereas lot sizes in Fulton County are restricted to 1.1, 1.2, ..., 1.9, and 2.0 acres. In such a situation we would have the additional functional dependency FD5: AREAâCOUNTY_NAME. If we add this to the other dependencies, the relation schema LOTS1A still is in 3NF because COUNTY_NAME is a prime attribute.

The area of a lot that determines the county, as specified by FD5, can be represented by 16 tuples in a separate relation *R*(AREA, COUNTY_NAME), since there are only 16 possible AREA values. This representation reduces the redundancy of repeating the same information in the thousands of LOTS1A tuples. BCNF is a *stronger normal form* that would disallow LOTS1A and suggest the need for decomposing it.

The formal definition of BCNF differs slightly from the definition of 3NF. A relation schema *R* is in **BCNF** if whenever a *nontrivial* functional dependency *X*â *A* holds in *R*, then *X* is a superkey of *R*. The only difference between the definitions of BCNF and 3NF is that condition (b) of 3NF, which allows *A* to be prime, is absent from BCNF



Q.8 (a). Discuss the following:-

(i) Files of unordered records

(ii) Files of ordered (sorted) records

Answer:

Files of Unordered Records (Heap Files)

In this simplest and most basic type of organization, records are placed in the file in the order in which they are inserted, so new records are inserted at the end of the file. Such an organization is called a **heap** or **pile file.** It is also used to collect and store data records for future use.

Inserting a new record is *very efficient:* the last disk block of the file is copied into a buffer; the new record is added; and the block is then **rewritten** back to disk. The address of the last file block is kept in the file header. However, searching for a record using any search condition involves a **linear search** through the file block by block—an expensive procedure. If only one record satisfies the search condition, then, on the average, a program will read into memory and search half the file blocks before it finds the record. For a file of *b* blocks, this requires searching (*b*/2) blocks, on average. If no records or several records satisfy the search condition, the program must read and search all *b* blocks in the file.

To delete a record, a program must first find its block, copy the block into a buffer, then delete the record from the buffer, and finally **rewrite the block** back to the disk. This leaves unused space in the disk block. Deleting a large number of records in this way results in wasted storage space. Another technique used for record deletion is to have an extra byte or bit, called a **deletion marker,** stored with each record. A record is deleted by setting the deletion marker to a certain value. A different value of the marker indicates a valid (not deleted) record. Search programs consider only valid records in a block when conducting their search. Both of these deletion techniques require periodic **reorganization** of the file to reclaim the unused space of deleted records. During reorganization, the file blocks are accessed consecutively, and records are packed by removing deleted records. After such reorganization, the blocks are filled to capacity once more. Another possibility is to use the space of deleted records when inserting new records, although this requires extra bookkeeping to keep track of empty locations.

We can use either spanned or unspanned organization for an unordered file, and it may be used with either fixed-length or variable-length records. Modifying a variable-length record may require deleting the old record and inserting a modified record, because the modified record may not fit in its old space on disk.

To read all records in order of the values of some field, we create a sorted copy of the file. Sorting is an expensive operation for a large disk file, and special techniques for **external sorting** are used .

(ii) Files of Ordered Records (Sorted Files)

We can physically order the records of a file on disk based on the values of one of their fields—called the **ordering field.** This leads to an **ordered** or **sequential** file . If the ordering field is also a **key field** of the file—a field guaranteed to have a unique value in each record—then the field is called the **ordering key** for the file.

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.

Inserting and deleting records are expensive operations for an ordered file because the records must remain physically ordered. To insert a record, we must find its correct position in the file, based on its ordering field value, and then make space in the file to insert the record in that position. For a large file this can be very time-consuming because, on the average, half the records of the file must be moved to make space for the new record. This means that half the file blocks must be read and rewritten after records are moved among them. For record deletion, the problem is less severe if deletion markers and periodic reorganization are used.

Modifying the ordering field means that the record can change its position in the file, which requires deletion of the old record followed by insertion of the modified record.

Reading the file records in order of the ordering field is quite efficient if we ignore the records in overflow, since the blocks can be read consecutively using double buffering. To include the records in overflow, we must merge them in their correct positions; in this case, we can first reorganize the file, and then read its blocks sequentially. To reorganize the file, first sort the records in the overflow file, and then merge them with the master file. The records marked for deletion are removed during the reorganization.

Ordered files are rarely used in database applications unless an additional access path, called a **primary index,** is used; this results in an **indexed-sequential file.**

Q.8 (b). Discuss the single-level ordered indexes and their types.

Answer:

The idea behind an ordered index access structure is similar to that behind the index used in a textbook, which lists important terms at the end of the book in alphabetical order along with a list of page numbers where the term appears in the book. We can search an index to find a list of *addresses*—page numbers in this case—and use these addresses to locate a term in the textbook by *searching* the specified pages. The alternative, if no other guidance is given, would be to sift slowly through the whole textbook word by word to find the term we are interested in; this corresponds to doing a linear search on a file. Of course, most books do have additional information, such as chapter and section titles, that can help us find a term without having to search through the whole book. However, the index is the only exact indication of where each term occurs in the book.

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 (see Section 6.2) 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.

1. Primary Index

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) = (Aaron,Ed), P(1) = address of block 1>

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

<K(3) = (Alexander,Ed), 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 rather than for every search value (or every record). .

2 Clustering Indexes

If records of a file are physically ordered on a nonkey field—which *does not* have a distinct value for each record—that field is called the **clustering field.** We can create a different type of index, called a **clustering index,** to speed up retrieval of records that have the same value for the clustering field. This differs from a primary index, which requires that the ordering field of the data file have a *distinct value* for each record.

A clustering index is also an ordered file with two fields; the first field is of the same type as the clustering field of the data file, and the second field is a block pointer. There is one entry in the clustering index for each *distinct value* of the clustering field, containing the value and a pointer to the *first block* in the data file that has a record with that value for its clustering field.

A clustering index is another example of a *nondense* index, because it has an entry for every *distinct value* of the indexing field rather than for every record in the file.

3 Secondary Indexes

A **secondary index** is also 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). Write the methods for implementing JOIN operations.
Answer:
Methods for Implementing Joins •

1. ***Nested-loop join (brute force):*** For each record *t* in *R* (outer loop), retrieve every record *s* from *S* (inner loop) and test whether the two records satisfy the join condition *t*[*A*] = *s*[*B*] .
2. ***Single-loop join*** *(using an access structure to retrieve the matching records):* If an index (or hash key) exists for one of the two join attributes—say, *B* of *S*—retrieve each record *t* in *R*, one at a time (single loop), and then use the access structure to retrieve directly all matching records *s* from *S* that satisfy *s*[*B*] = *t*[*A*].
3. ***Sort–merge join****:* If the records of *R* and *S* are *physically sorted* (ordered) by value of the join attributes *A* and *B*, respectively, we can implement the join in the most efficient way possible. Both files are scanned concurrently in order of the join attributes, matching the records that have the same values for *A* and *B.* If the files are not sorted, they may be sorted first by using external sorting (see Section 18.2.1). In this method, pairs of file blocks are copied into memory buffers in order and the records of each file are scanned only once each for matching with the other file—unless both *A* and *B* are nonkey attributes, in which case the method needs to be modified slightly.
4. ***Hash-join****:* The records of files *R* and *S* are both hashed to the same hash file, using the same

Q.9 (b). Write short note on Aggregate Operations.
Answer:
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.

Consider the following query:

SELECT DNO, **AVG** (SALARY) FROM EMPLOYEE **GROUP BY** DNO;

Text Books

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