Q.2 a. List the main characteristics of database approach versus file-processing approach.

Answer:
The main characteristics of the database approach versus the file-processing approach are the following:

- Self-describing nature of the database system.
- Insulation between programs and data, and data abstraction.
- Support of multiple views of the data.

Sharing of data and multiuser transaction processing.

b. Discuss the different types of user-friendly interfaces and the types of users who typically use each.

Answer:

User-friendly interfaces provided by a DBMS may include the following.

- **Menu-Based Interfaces for Browsing:** These interfaces present the user with lists of options, called **menus,** that lead the user through the formulation of a request. Menus do away with the need to memorize the specific commands and syntax of a query language; rather, the query is composed step by step by picking options from a menu that is displayed by the system. Pull-down menus are becoming a very popular technique in window-based user hethe contents of a database in an exploratory and unstructured manner.

- **Forms-Based Interfaces:** A forms-based interface displays a **form** to each user. Users can fill out all of the form entries to insert new data, or they fill out only certain entries, in which case the DBMS will retrieve matching data for the remaining entries. Forms are usually designed and programmed for naive users as interfaces to canned transactions. Many DBMSs have **forms specification languages,** special languages that help programmers specify such forms. Some systems have utilities that define a form by letting the end user interactively construct a sample form on the screen.

- **Graphical User Interfaces:** A graphical interface (GUI) typically displays a schema to the user in diagrammatic form. The user can then specify a query by manipulating the diagram. In many cases, GUIs utilize both menus and forms. Most GUIs use a **pointing device,** such as a mouse, to pick certain parts of the displayed schema diagram.

- **Natural Language Interfaces:** These interfaces accept requests written in English or some other language and attempt to "understand" them. A natural language interface usually has its own "schema," which is similar to the database conceptual schema. The natural language interface refers to the words in its schema, as well as to a set of standard words, to interpret the request. If the interpretation is successful, the interface generates a high-level query

corresponding to the natural language request and submits it to the DBMS for processing; otherwise, a dialogue is started with the user to clarify the request.

- **Interfaces for Parametric Users:** Parametric users, such as bank tellers, often have a small set of operations that they must perform repeatedly. Systems analysts and programmers design and implement a special interface for a known class of naive users. Usually, a small set of abbreviated commands is included, with the goal of minimizing the number of keystrokes required for each request. For example, function keys in a terminal can be programmed to initiate the various commands. This allows the parametric user to proceed with a minimal number of keystrokes.

- **Interfaces for the DBA:** Most database systems contain privileged commands that can be used only by the DBA's staff. These include commands for creating accounts, setting system parameters, granting account authorization, changing a schema, and reorganizing the storage structures of a database.

c. Describe the following attributes of ER – model:
(i) Simple versus composite
(ii) Single-valued versus multivalued
(iii)Stored versus derived

Answer:

Simple versus composite

(i) Composite attributes can be divided into smaller subparts, which represent more basic attributes with independent meanings. For example, the Address attribute of the EMPLOYEE entity shown in Figure (A) below can be subdivided into Street_address, City, State, and Zip, with the values '2311 Kirby', 'Houston', 'Texas', and '77001.' Attributes that are not divisible are called simple or atomic attributes. Composite attributes can form a hierarchy; for example, Street_address can be further subdivided into three simple component attributes: Number, Street, and Apartment_number, as shown in Figure (B) below. The value of a composite attribute is the concatenation of the values of its component simple attributes.

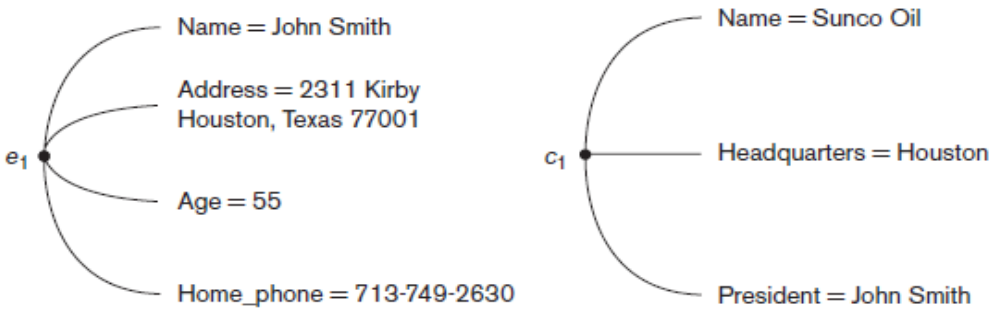


Figure (A): Two entities EMPLOYEE e_1, and COMPANY c_1, and their attributes

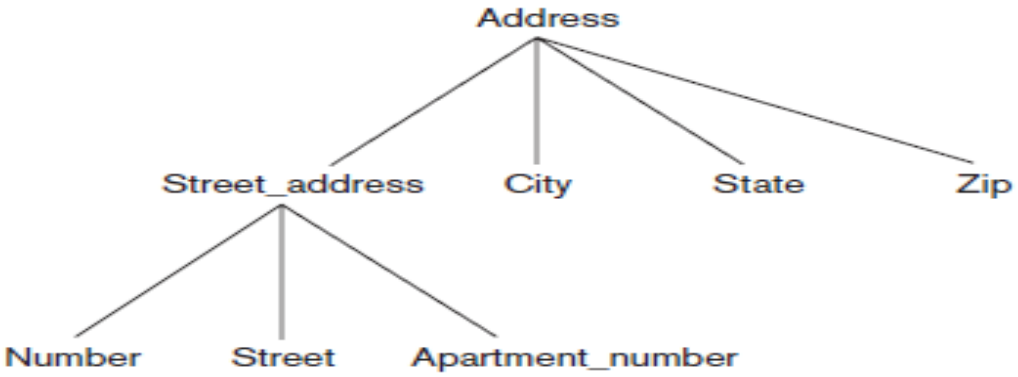


Figure (B): A hierarchy of composite attributes

(i) Single- valued versus multivalued

Most attributes have a single value for a particular entity; such attributes are called **single-valued**. For example, Age is a single-valued attribute of a person. In some cases an attribute can have a set of values for the same entity—for instance, a Colors attribute for a car, or a College

Composite attributes are useful to model situations in which a user sometimes refers to the composite attribute as a unit but at other times refers specifically to its components. If the degrees attribute for a person. Cars with one color have a single value, whereas two-tone cars have two color values. Similarly, one person may not have a college degree, another person may have one, and a third person may have two or more degrees; therefore, different people can have different *numbers* of *values* for the College_degrees attribute. Such attributes are called **multivalued**. A multivalued attribute may have lower and upper bounds to constrain the *number of values* allowed for each individual entity. For example, the Colors attribute of a car may be restricted to have between one and three values, if we assume that a car can have three colors at most.

(iv) Stored versus derived

In some cases, two (or more) attribute values are related—for example, the Age and Birth_date attributes of a person. For a particular person entity, the value of Age can be determined from the current (today's) date and the value of that person's Birth_date. The Age attribute is hence called a **derived attribute** and is said to be derivable from the Birth_date attribute, which is called a **stored attribute**. Some attribute values can be derived from *related entities*; for example, an attribute Number_of_employees of a DEPARTMENT entity can be derived by counting the number of employees related to (working for) that department.

Q.3 a. Consider the following schema:
Suppliers(*sid:* integer, *sname:* string, *address:* string)
Parts(*pid:* integer, *pname:* string, *color:* string)
Catalog(*sid:* integer, *pid:* integer, *cost:* real)

The key fields are underlined, and the domain of each field is listed after the field name. Therefore *sid* is the key for Suppliers, *pid* is the key for Parts, and *sid* and *pid* together form the key for Catalog. The Catalog relation lists the prices charged for parts by Suppliers. Write the following queries in relational algebra.

(i) Find the *name*s of suppliers who supply some red part.
(ii) Find the *sid*s of suppliers who supply some red part and some green part.
(iii) Find the *pid*s of the most expensive parts supplied by suppliers named SHANKER.

Answer:

$\pi_{sname}(\pi_{sid}((\pi_{pid}\sigma_{color='red'}\ Parts) \bowtie Catalog) \bowtie Suppliers)$

(i) $\rho(R1, \pi_{sid}((\pi_{pid}\sigma_{color='red'}\ Parts) \bowtie Catalog))$
$\rho(R2, \pi_{sid}((\pi_{pid}\sigma_{color='green'}\ Parts) \bowtie Catalog))$
$R1 \cap R2$

(ii) $\rho(R1, \pi_{sid}\sigma_{sname='SHANKER'}\ Suppliers)$
$\rho(R2,R1 \bowtie Catalog)$
$\rho(R3,R2)$
$\rho(R4(1 \rightarrow sid, 2 \rightarrow pid, 3 \rightarrow cost), \sigma_{R3.cost<R2.cost}(R3 \times R2))$
$\pi_{pid}(R2 - \pi_{sid,pid,cost}\ R4)$

b. Discuss the characteristics of a relation that make them different from ordinary tables and files?

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.*

(c) What is meant by a safe expression in relational calculus?

Answer:

Whenever we use universal quantifiers, existential quantifiers, or negation of predicates in a calculus expression, we must make sure that the resulting expression makes sense. A **safe expression** in relational calculus is one that is guaranteed to yield a *finite number of tuples* as its result; otherwise, the expression is called **unsafe.** For example, the expression

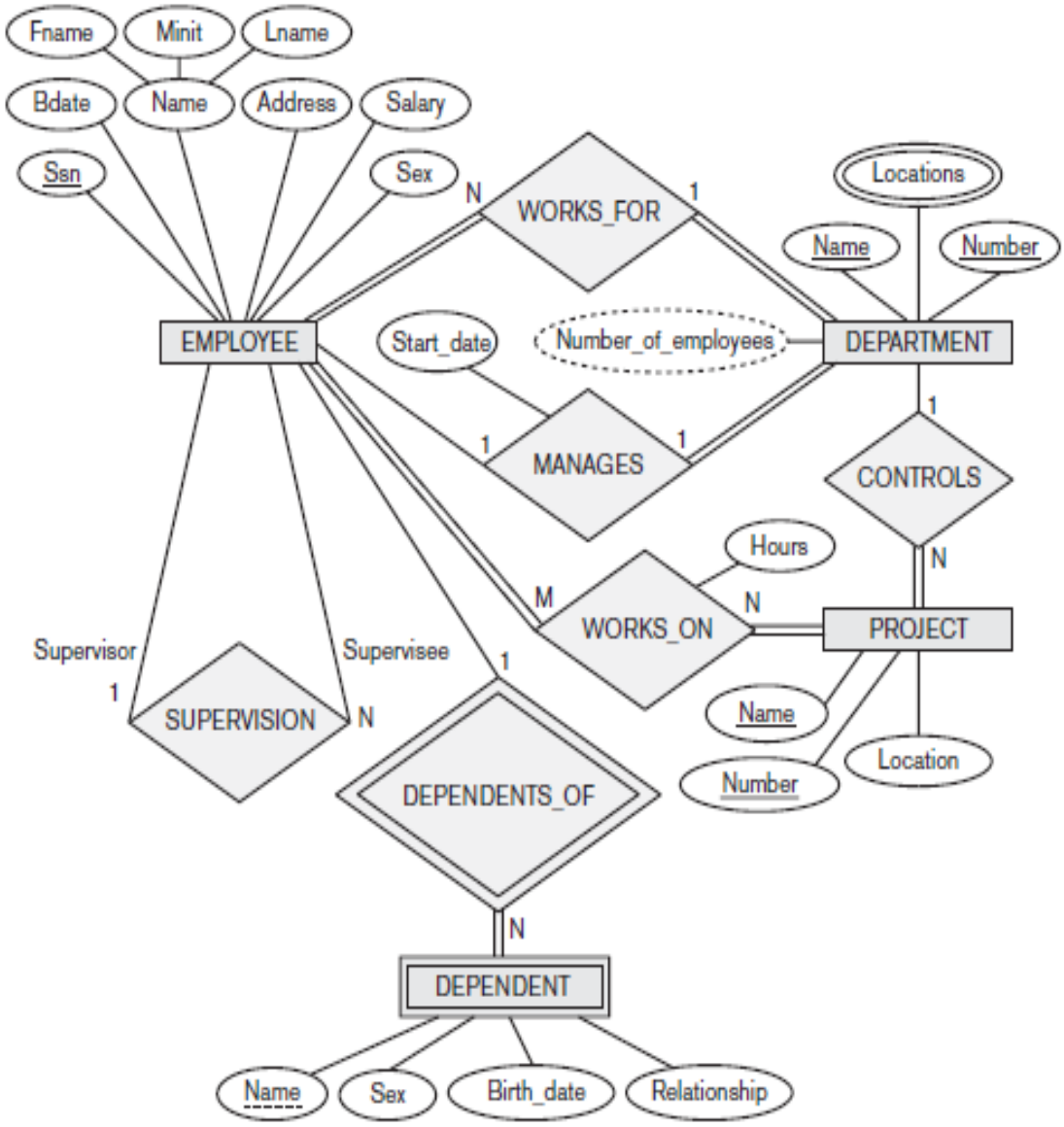
{t | **not** (EMPLOYEE(t))}

is *unsafe* because it yields all tuples in the universe that are *not* EMPLOYEE tuples, which are infinitely numerous. If we follow the rules, we will get a safe expression when using universal quantifiers. We can define safe expressions more precisely by introducing the concept of the *domain of a tuple relational calculus expression:* This is the set of all values that either appear as constant values in the expression or exist in any tuple of the relations referenced in the expression. The domain of {t | **not**(EMPLOYEE(t))} is the set of all attribute values appearing in some tuple of the EMPLOYEE relation (for any attribute).

An expression is said to be **safe** if all values in its result are from the domain of the expression. Notice that the result of {t | **not**(EMPLOYEE(t))} is unsafe, since it will, in

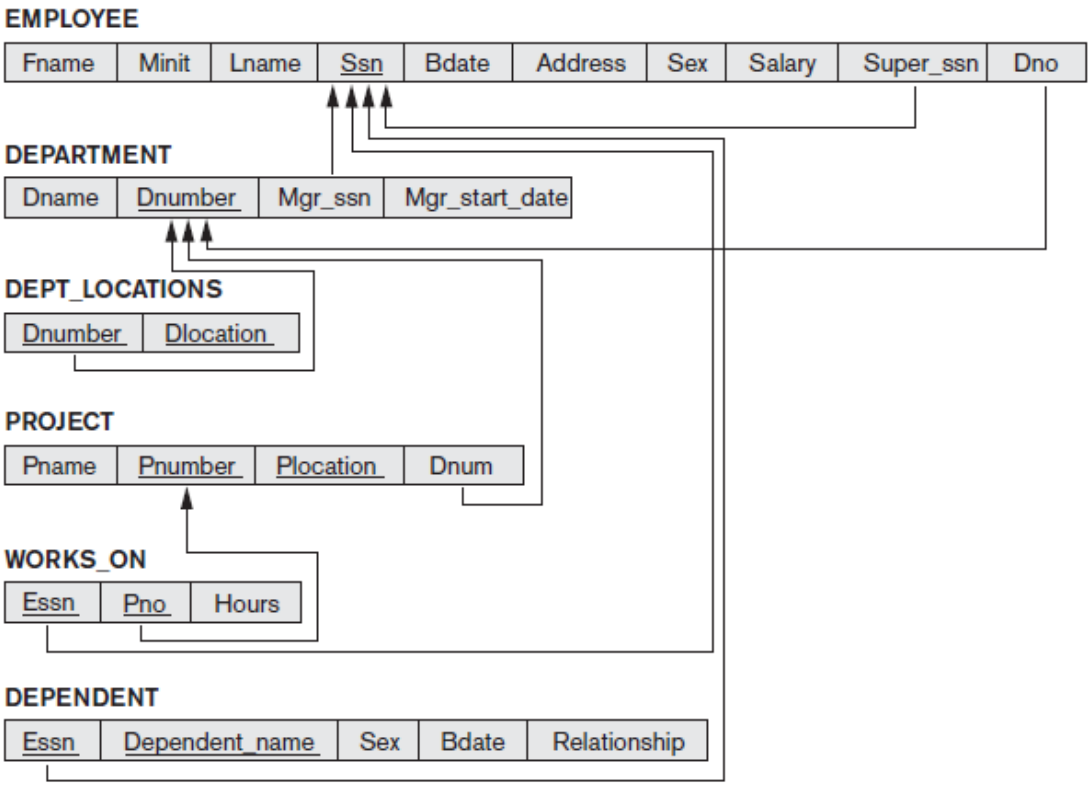
general, include tuples (and hence values) from outside the EMPLOYEE relation; such values are not in the domain of the expression.

Q.4 a. Consider the following ER conceptual schema diagram for the COMPANY database. Map the given ER diagram into relational database schema.



Answer:

Result of mapping the given COMPANY ER schema into a relational database schema is as follows:



b. Describe the circumstances in which you would choose to use embedded SQL rather than SQL alone or only a general-purpose programming language?

Answer:
SQL provides a powerful declarative query language. Writing queries in SQL is usually much easier than coding the same queries in a general-purpose programming language. However, a programmer must have access to a database from a general purpose programming language for at least two reasons:

1. Not all queries can be expressed in SQL, since SQL does not provide the full expressive power of a general-purpose language. That is, there exist queries that can be expressed in a language such as C, Java, or Cobol that cannot be expressed in SQL. To write such queries, we can embed SQL within a more powerful language. SQL is designed so that queries written in it can be optimized automatically and executed efficiently—and providing the full power of a programming language makes automatic optimization exceedingly difficult.

2. Nondeclarative actions—such as printing a report, interacting with a user, or sending the results of a query to a graphical user interface—cannot be done from within SQL. Applications usually have several components, and querying or updating data is only one component; other components are written in general-purpose programming languages. For an integrated application, the programs written in the programming language must be able to access the database.

The SQL standard defines embeddings of SQL in a variety of programming languages, such as C, Cobol, Pascal, Java, PL/I, and Fortran. A language in which SQL queries are embedded is referred to as a *host* language, and the SQL structures permitted in the host language constitute *embedded* SQL.

Programs written in the host language can use the embedded SQL syntax to access and update data stored in a database. This embedded form of SQL extends the programmer's ability to manipulate the database even further. In embedded SQL, all query processing is performed by the database system, which then makes the result of the query available to the program one tuple (record) at a time.

An embedded SQL program must be processed by a special preprocessor prior to compilation. The preprocessor replaces embedded SQL requests with host-language declarations and procedure calls that allow run-time execution of the database accesses. Then, the resulting program is compiled by the host-language compiler. To identify embedded SQL requests to the preprocessor, we use the EXEC SQL statement; it has the form

EXEC SQL <embedded SQL statement > END-EXEC

The exact syntax for embedded SQL requests depends on the language in which SQL is embedded. For instance, a semicolon is used instead of END-EXEC when SQL is embedded in C. The Java embedding of SQL (called SQLJ) uses the syntax

SQL *{* <embedded SQL statement > *}*;

We place the statement SQL INCLUDE in the program to identify the place where the preprocessor should insert the special variables used for communication between the program and the database system. Variables of the host language can be used within embedded SQL statements, but they must be preceded by a colon (:) to distinguish them from SQL variables.

Q.5 a. (i) When are two sets of functional dependencies equivalent? What conditions are to be satisfied to define a set of functional dependencies F to be minimal?

(ii) Let the given set of functional dependencies be E: {B → A, D → A, AB → D}. Find the minimal cover of E.

Answer:

Two sets of functional dependencies E and F are equivalent if $E^+ = F^+$. Therefore, equivalence means that every FD in E can be inferred from F, and every FD in F can be inferred from E; that is, E is equivalent to f if both the conditions E cover F and F covers E hold.

Conditions to be satisfied to define a set of functional dependencies F to be minimal are as follows:

1. Every dependency in F has a single attribute for its right-hand side.
2. We cannot replace any dependency X → A in F with a dependency Y → A, where Y is a proper subset of X, and still have a set of dependencies that is equivalent to F.
3. We cannot remove any dependency from F and still have a set of dependencies that is equivalent to F.

(ii) Let the given set of functional dependencies be E: {B → A, D → A, AB → D}. Find the minimal cover of E.

Answer:

- All above dependencies are already in canonical form. In next step we determine if AB → D has any redundant attribute on the left-hand side; i.e., can it be replaced by B → D or A → D?
- Since B → A, hence by augmenting with B on both sides, we have BB → AB, or B → AB. But it is given AB → D.
- Hence by transitive rule, we get B → D. Hence, AB → D may be replaced by B → D.
- Now we get a set equivalent to original E, say E' : {B → A, D → A, B → D}. No further reduction is possible since all FDS have a single attribute on the left-hand side.
- Now we look for a redundant FD in E'. By using transitive rule on B → A and D → A, we drive B → A. Hence B → A is redundant in E' and can be eliminated.
- Hence the minimum cover of E is **{B → A, D → A}**.

b. Define decomposition. State the properties that must be satisfied by a relation R to be decomposed into a set of relations.

Answer:

Decompositiion means breaking the relation schema into smaller schema. We may also call this schema refinement. Suppose a relation schema R(A, B, C, D, E, F, G) is split into R1(A, B, C, D) and R2(E, F, G) then we can say that R1 and R2 are decomposition of R.

We can say a relation R is decomposed into a set of relations R1 and R2 if and only if it satisfies the following properties of decomposition.

Attribute Prevention

- If R1 and R2 are projections of some relation R and R1 and R2 between them include all attributes of R, then we say that R is decomposed into R1 and R2. For e.g., consider the relation R(A, B, C, D, E, F, G) and the decomposition of this relation are R1(A, B, C, D) and R2(E, F, G) are the attribute preserving decompositions.

Loss-Less Join Decomposition

- We can say that the decomposition of R into R1, with attribute set X1 and R2 with attribute set X2 is loss-less decomposition if by joining R1 and R2 we get back the original decomposition.
- The word loss-less refers to loss of information and not loss of tuples.
- Let R be a relation with attribute set X and F is the set of FDs that hold over R. The decomposition of R into R1 and R2 with attributes X1 and X2 respectively is loss-less if and only if F^+ contains either

  $X1 \cap X2 \rightarrow X1$ or,

  $X1 \cap X2 \rightarrow X2$

In other words, the attributes common to R1 and R2 must contain a key for either R1 or R2.

- If an FD, $X \rightarrow Y$ holds over a relation schema R and $X \cap Y$ is empty, the decomposition of R into R – Y and XY is loss-less.
  For e.g., consider the relation schema R(A, B, C) with FDs $AB \rightarrow C$ and $C \rightarrow B$. The relation is not in BCNF since $C \rightarrow$ holds in R. We decompose this relation into R1(A1, C) (i.e. R – Y) and R2(B, C) (i.e. XY) is a loss-less decomposition because $B \cap C$ is empty.

Dependency Preservation

- If R is decomposed into X, Y and Z and we enforce the FDs that hold on X, on Y and on Z, and then all FDs that were given to hold on R must also hold.
- The decomposition of relation R with FDs F into R1 and R2 is said to be dependency preserving if $F_{R1} \cup F_{R2})^+ = F^+$
  In other words, if we take the dependencies of F_{R1} and F_{R2} and compute the closure of their union, we get back the original FDs in F.
  For e.g., consider the relation R(A, B, C, D) with FDs $A \rightarrow B$ and $C \rightarrow$ is decomposed into R1(A, B) and R2(C, D) is a dependency preserving decomposition.
- It is always possible to find a dependency preserving decomposition with respect to an F such that the resulting relations are in 3NF.
- In general, there may not be a dependency preserving decomposition that also decomposes relations in BCNF. For e.g., consider a relation R(A, B, C) with FDs $AB \rightarrow C$ and $C \rightarrow B$ is a relation that cannot be decomposed to satisfy both dependency preserving and BCNF.

Q.6 a. What are the reasons for having variable length records?

Answer:

A **file** is a *sequence* of records. In many cases, all records in a file are of the same record type. If every record in the file has exactly the same size (in bytes), the file is said to be made up of **fixed-length records.** If different records in the file have different sizes, the file is said to be made up of **variable-length records.**

A file may have variable-length records for several reasons:

- The file records are of the same record type, but one or more of the fields are of varying size (**variable-length fields**). For example, the NAME field of EMPLOYEE can be a variable-length field.

- The file records are of the same record type, but one or more of the fields may have multiple values for individual records; such a field is called a **repeating field** and a group of values for the field is often called a **repeating group.**

- The file records are of the same record type, but one or more of the fields are **optional;** that is, they may have values for some but not all of the file records (**optional fields**).

- The file contains records of *different record types* and hence of varying size (**mixed file**). This would occur if related records of different types were *clustered* (placed together) on disk blocks.

b. Briefly explain any two hashing techniques that allow dynamic file expansion.

Answer: Page Number 495 of Text Book.

c. Why does the index file for a primary index need substantially fewer blocks than the data file?

Answer:
The index file for a primary index needs substantially fewer blocks than does the data file, for two reasons:

- First, there are fewer index entries than there are records in the data file.

- Second, each index entry is typically smaller in size than a data record because it has only two fields; consequently more index entries than data records can fit in one block. Therefore, a binary search on the index file requires fewer block accesses than a binary search on the data file.

Q.7 a. 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?

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.

b. Discuss the different phases of external sorting. Also give an outline of the algorithm used.

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 $n_R = [(b/n_B)]$ 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;
        }
```

Q.8 a. Define Deadlock. What are the necessary four conditions for a deadlock to occur? Discuss the different methods that can be used for deadlock prevention.

Answer:

Deadlock: A system is in a deadlock state if there exists a set of transactions such that every transaction in the set is waiting for another transaction in the set. More precisely, there exists a set of waiting transactions *{T*0, *T*1, . . ., *Tn}* such that *T*0 is waiting for a data item that *T*1 holds, and *T*1 is waiting for a data item that *T*2 holds, and . . ., and *Tn*−1 is waiting for a data item that *Tn* holds, and *Tn* is waiting for a data item that *T*0 holds. None of the transactions can make progress in such a situation.

There are two principal methods for dealing with the deadlock problem. We can use a **deadlock prevention** protocol to ensure that the system will *never* enter a deadlock state. Alternatively, we can allow the system to enter a deadlock state, and then try to recover by using a **deadlock detection** and **deadlock recovery** scheme.

Necessary conditions for a deadlock to occur

Mutual Exclusion: Some data items must be locked by some transactions in Exclusive Mode. These data items are not accessible to other transactions than the one currently holding it.

Hold and Wait: Some transactions must be holding Exclusive Locks on some data items and at the same waiting for grant of Exclusive Lock on some other data items, which may be currently locked by other transactions.

No Pre-emption: The data items locked exclusively by a Transaction cannot be forcibly pre-empted. The Transaction will release the locks on such items only voluntarily, when it has finished with the data items.

Cyclic Wait: There must exist a situation, wherein a set of n transactions say (T1, T2, …….Tn) are waiting in a cyclic manner for the data items locked by each other i.e *T*0 is waiting for a data item that *T*1 holds, and *T*1 is waiting for a data item that *T*2 holds, and . . ., and *Tn*−1 is waiting for a data item that *Tn* holds, and *Tn* is waiting for a data item that *T*0 holds.

Two different deadlock prevention schemes using timestamps are as below:

(i) The **wait–die** scheme is a non-preemptive technique. When transaction *Ti* requests a data item currently held by *Tj* , *Ti* is allowed to wait only if it has a timestamp smaller than that of *Tj* (that is, *Ti* is older than *Tj*). Otherwise, *Ti* is rolled back (dies).

For example, suppose that transactions *T1*, *T*2, and *T3* have timestamps 5, 10, and 15, respectively. If *T1* requests a data item held by *T*2, then *T1* will wait. If *T3* requests a data item held by *T*2, then *T3* will be rolled back.

(ii) The **wound–wait** scheme is a preemptive technique. It is a counterpart to the wait–die scheme. When transaction *Ti* requests a data item currently held by *Tj* , *Ti* is allowed to wait only if it has a timestamp larger than that of *Tj* (that is, *Ti* is younger than *Tj*). Otherwise, *Tj* is rolled back (*Tj* is *wounded* by *Ti*).

Consider the same example, with transactions *T1*, *T*2, and *T3*, if *T1* requests a data item held by *T2*, then the data item will be preempted from *T*2, and *T*2 will be rolled back. If *T3* requests a data item held by *T*2, then *T3*will wait.

b. What are the conditions that lead to the two schedules being view equivalent. When a schedule S is said to be view serializable?

Answer:
Two schedules S and S' are said to be view equivalent if the following three conditions hold:

1. The same set of transactions participants in S and S', and S and S' include the same operations of those transactions.
2. For any operation $r_i(X)$ of T_i in S, if the value of X read by the operation has been written by an operation $w_j(X)$ of T_j (or if it is the original value of X before the schedule started), the same condition must hold for the value of X read by operation $r_i(X)$ of T_i in S'.
3. If the operation $w_k(Y)$ of T_k is the last operation to write item in S, then $w_k(Y)$ of T_k must also be the last operation to write item Y in S'.

The idea behind view equivalence is that, as long as each read operation of a transaction reads the result of the same write operation in both schedules, the write operations of each transaction must produce the same results. The read operations are hence said to *see the same view* in both schedules. Condition 3 ensures that the final write operation on each data item is the same in both schedules, so the database state should be the same at the end of both schedules.

"A schedule *S* is said to be **view serializable** if it is view equivalent to a serial schedule."

c. What are the rules followed when shared / exclusive locking scheme is used?

Answer:
When we use the shared/exclusive locking scheme, the system must enforce the following rules:

1. A transaction *T* must issue the operation read_lock(*X*) or write_lock(*X*) before any read_item(*X*) operation is performed in *T*.
2. A transaction *T* must issue the operation write_lock(*X*) before any write_item(*X*) operation is performed in *T*.
3. A transaction *T* must issue the operation unlock(*X*) after all read_item(*X*) and write_item(*X*) operations are completed in *T*.
4. A transaction *T* will not issue a read_lock(*X*) operation if it already holds a read (shared) lock or a write (exclusive) lock on item *X*. This rule may be relaxed.
5. A transaction *T* will not issue a write_lock(*X*) operation if it already holds a read (shared) lock or write (exclusive) lock on item *X*. This rule may also be relaxed.

6. A transaction *T* will not issue an unlock(*X*) operation unless it already holds a read (shared) lock or a write (exclusive) lock on item *X*.

Q.9 a. What are checkpoints and why are they important? List the actions taken by the recovery manager during checkpoints.

Answer:
Checkpoint is a type of entry in the log . A checkpoint record is written into the log periodically at that point when the system writes out to the database on disk all DBMS buffers that have been modified. As a consequence of this, all transactions that have their [commit, T] entries in the log before a checkpoint entry do not need to have their WRITE operations *redone* in case of a system crash, since all their updates will be recorded in the database on disk during checkpointing.

Actions taken by the recovery manager during Checkpoint

The recovery manager of a DBMS must decide at what intervals to take a checkpoint. The interval may be measured in time—say, every *m* minutes—or in the number *t* of committed transactions since the last checkpoint, where the values of *m* or *t* are system parameters. Taking a checkpoint consists of the following actions:
1. Suspend execution of transactions temporarily.
2. Force-write all main memory buffers that have been modified to disk.
3. Write a [checkpoint] record to the log, and force-write the log to disk.
4. Resume executing transactions.
As a consequence of Step 2, a checkpoint record in the log may also include additional information, such as a list of active transaction ids, and the locations (addresses) of the first and most recent (last) records in the log for each active transaction. This can facilitate undoing transaction operations in the event that a transaction must be rolled back.

b. Explain the term *"steal and no-steal"* approach in standard DBMS recovery schemes.

Answer:
Standard DBMS recovery terminology includes the terms **steal/no-steal,** which specify when a page from the database can be written to disk from the cache:
If a cache page updated by a transaction *cannot* be written to disk before the transaction commits, this is called a **no-steal approach.** The pin-unpin bit indicates if a page cannot be written back to disk. Otherwise, if the protocol allows writing an updated buffer *before* the transaction commits, it is called **steal.** Steal is used when the DBMS cache (buffer) manager needs a buffer frame for another transaction and the buffer manager replaces an existing page that had been updated but whose transaction has not committed.
The deferred update recovery scheme follows a *no-steal* approach. However, typical database systems employ a *steal* strategy. The advantage of steal is that it avoids the need for a very large buffer space to store all updated pages in memory.

c. Discuss the two main techniques for recovery from non-catastrophic transaction.

Answer:

The two main techniques for recovery from noncatastrophic transaction failures are as follows:

- **Deferred Update:** The deferred update techniques do not physically update the database on the disk until after a transaction reaches its commit point; then the updates are recorded in the database. Before reaching commit, all transaction updates are recorded in the local transaction workspace (or buffers). During commit, the updates are first recorded persistently in the log and then written to the database. If a transaction fails before reaching its commit point, it will not have changed the database in any way, so UNDO is not needed. It may be necessary to REDO the effect of the operations of a committed transaction from the log, because their effect may not yet have been recorded in the database. Hence, deferred update is also known as the **NO-UNDO/REDO algorithm.**

Immediate Update: In the immediate techniques, the database may be update by some operations of a transaction before the transaction reaches its commit point. However, these operations are typically recorded in the log on disk by force writing before they are applied to the database, making recovery still possible. If a transaction fails after recording some changes in the database but before reaching its commit point, the effect of its operations on the database must be done; i.e., the transaction must be rolled back. In the general case of immediate update, both undo and redo may be required during recovery. This technique, known as the **UNDO/REDO algorithm,** requires both operations, and is used most often in practice. A variation of the algorithm where all updates are recorded in the database before a transaction commits requires undo also, so it is known as the **UNDO/NO-REDO algorithm.**

TEXT BOOK

Fundamentals of Database systems, Elmasri, Navathe, Somayajalu, Gupta, Pearson Education, 2006.