Q2. (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 interfaces. They are often used in browsing interfaces, which allow a user to look through the 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 DBMS 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.

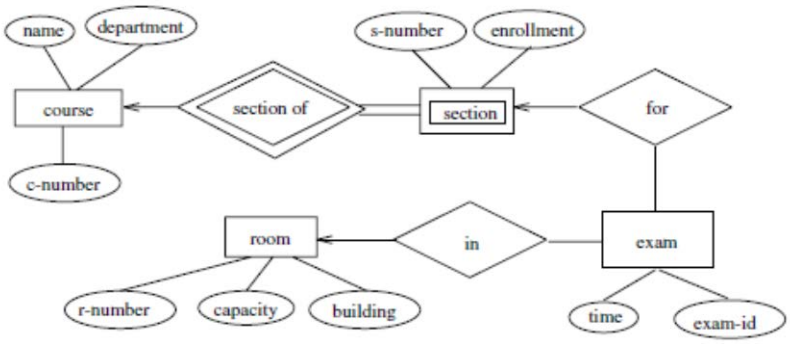
- Interfaces for the DBA: Most database systems contain privileged commands that can be used only by the DBA's staff. These include commands for creating accounts, setting system parameters, granting account authorization, changing a schema, and reorganizing the storage structures of a database.

(c) Consider a university database for the scheduling of classrooms for final exams. This database could be modeled as the single entity set *exam*, with attributes *course-name*, *section-number*, *room-number*, and *time*. Alternatively, one or more additional entity sets could be defined, along with relationship sets to replace some of the attributes of the *exam* entity set, as

• *course* with attributes *name*, *department*, and *c-number*
• *section* with attributes *s-number* and *enrollment*, and dependent as a weak entity set on *course*
• *room* with attributes *r-number*, *capacity*, and *building*

Show an E-R diagram illustrating the use of the listed entity sets listed.

Answer:



ER- diagram for Exam Scheduling

Q3 (a) List the categories in which constraints on database can be divided?

Answer: Constraints on databases can be generally divided into three main categories:

- Constraints that are inherent in the data model. We call these inherent model-based or implicit constraints.
- Constraints that can be directly expressed in schemas of the data model, typically by specifying them in the data definition language. We call these scheme-based or explicit constraints.
- Constraints that cannot be directly expressed in schemas of the data model, and hence must be expressed and enforced by the application programs. We call these application-based or semantic constraints or business rules.

(b) 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.

(c) Let the following relation schemas be given:

R = (A, B, C)
S = (D, E, F)

Let relations *r(R)* and *s(S)* be given. Give equivalent SQL statements for the following queries.

(i) $\Pi_A(r)$
(ii) $\sigma_{B=33}(r)$
(iii) rXs
(iv) $\Pi_{A,F}(\sigma_{C=D}(r \times s))$

Answer:

(i) $\Pi_A(r)$

```
select distinct A
from r
```

(ii) $\sigma_{B=33}(r)$

```
select *
from r
where B = 33
```

(iii) r X s

```
select distinct *
from r, s
```

(iv) $\Pi_{A,F}(\sigma_{C=D}(r \times s))$

```
select distinct A, F
from r, s
where C = D
```

Q4 (a) Discuss the following SQL commands with examples:

(i) DROP

The DROP command can be used to drop schema elements, such as tables, domains, or constraints. One can also drop a schema. For example, if a whole schema is no needed, the DROP SCHEMA command can be used. There are two drop behaviour options: CASCADE and RESTRICT. For example, to remove the database schema and all its tables, domains, and other elements, the CASCADE options is used as follows:

DROP SCHEMA *database_name* 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.
Also, if the RESTRICT option is chosen instead of CASCADE, a table is drooped 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.
The DROP command can also be used to drop other types of named schema elements, such as constraints or domains.

(ii) ALTER

The definition of a base table or of other named schema elements can be changed by using the ALTER TABLE command. For base tables, 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 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 TABLECOMPANY.EMPLOYEE DROP ADDRESS CASCADE;

It is also possible to alter a column definition by dropping an existing default clause or by defining a new default clause. The following examples illustrate this clause:

ALTER TABLE COMPANY.DEPARTMENT **ALTER** MGRSSN
DROP DEFAULT;
ALTER TABLE COMPANY.DEPARTMENT **ALTER** MGRSSN
SET DEFAULT "333445555";

One can change the constraints specified on a table by adding or dropping a constraint. To be dropped, a constraint must have been given a name when it was specified.

(iii) INSERT

In its simplest form, INSERT is used to add a single tuple to a relation. We must specify the relation name and a list of values for the tuple. The values should be listed in the same order in which the corresponding attributes were specified in the CREATE TABLE command. For example, to add a new tuple to the EMPLOYEE relation shown in Figure is as follows:

INSERT INTO
EMPLOYEE
VALUES (‘Richard’, ’k’, ’Marini’, ‘653298653, ‘1962-12-30’,’98 Oak Forest, Katy, TX’,’M’,37000,’987654321’, 4);

A second form of the INSERT statement allows the user to specify explicit attribute names that correspond to the values provided in the INSERT command. This is useful if a relation has many attributes, but only a few of those attributes are assigned values in the new tuple. These attribute must include all attributes with NOT NULL specification and no default value; attributes with NULL allowed or Default values are the ones that can be left out. For example, to enter a tuple for a new EMPLOYEE for whom we know only the FNAME, LNAME, DNO, and SSN attribute, we can use:

INSERT INTO

EMPLOYEE(FNAME, LNAME, DNO, SSN)
VALUES ('Richard', 'Marini', '4', '653298653'4);

Attributes not specified are set to their DEFAULT or to NULL, and the values are listed in the same order as the attributes are listed in the INSERT command itself,. It is also possible to insert into a relation multiple tuples separated by commas in a single INSERT command. The attribute values forming each tuple are enclosed in parentheses.

(iv) UPDATE

The UPDATE command is used to modify attribute values of one or more selected tuples. As in the DELETE command, a WHERE-clause in the YPDATE command selects the tuples to be modified from a single relation. However, updating a primary key value may propagate to the foreign key values of tuples in other relations if such a referential triggered action is specified in the referential integrity constraints of the DDL. An additional SET-clause specifies the attributes to be modified and their new values. For example, to change the location and controlling department number of project number 10 to 'Bellaire' and 5, respectively, we use:

UPDATE PROJECT
SET Plocation = 'Bellaire', Dnum=5
WHERE Pnumber =10;

Several tuples can be modified with a single UPDATE command. An example is to give all employees in the 'Research' department a 10 percent raise in salary, the command is shown below. In this request, the modified SALARY value depends on the original SALARY value in each tuple, so two references to the SALARY attribute are needed. In the SET-clause, the reference to the SALARY attribute on the right refers to the old SALARY value before modification, and the one on the left refers to the new SALARY value after modification:

UPDATE EMPLOYEE
SET Salary = Salary * 1.1
WHERE Dno IN(**SELECT** Dnumber
FROM DEPARTMENT
WHERE Dname ="Research");

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

Ans. 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 powerful language. SQL is designed so that queries written in it can be optimised automatically and executed efficiently- and providing the full power of a programming language makes automatic optimisation exceedingly difficult.
2. Non-declarative 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 pre-processor 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 pre-processor, 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 pre-processor 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.

Q5(a) What is a minimal set of functional dependencies? Does every set of dependencies have a minimal equivalent set? Give and algorithm for finding a minimal cover G for F.

Ans. A set of functional dependencies F is minimal if it satisfies the following conditions:

1. Every dependencies 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.

We can think of a minimal set of dependencies as being a set of dependencies in a standard or canonical form and with no redundancies. Condition 1 ensures that every dependency is in canonical form with a single attribute on the right-hand side. Conditions 2 and 3 ensures that there is no redundancies in the dependencies either by having redundant attributes on the left-hand side of a dependency, or by having dependency that can be inferred from the remaining FDs in F. A minimal cover of a set of functional dependencies that is equivalent to F. unfortunately, there can be several minimal covers for a set of functional dependencies.

Algorithm: Finding a minimal cover G for F

1. Set G: = F.
2. Replace each functional dependency X →{A_1, A_2 …… A_n} in G by the n functional dependencies
   X→ A_1, X→ A_2…… X→ A_n.
3. For each functional dependency X→A in G for each attribute B that is an element of X
   if ((G-{X→A} U {X –{B}) →A}) is equivalent to G.
4. For each remaining functional dependency X→A in G
   if (G-{X→A}) is equivalent to G,
   then remove X→A from G.

(b) With the help of examples, differentiate between candidate key, primary key and secondary key.

Ans. Page No. 360 of Textbook 'Fundamentals of DataBase Systems, 5th Edition' by Elmasri Navathe.

(c) Briefly describe Boyce-Codd Normal Form?

Ans. Boyce-Codd Normal Form: A relation schema R is in BCNF with respect to a set F of functional dependencies if, for all functional dependencies in *F+* of the form $\alpha \rightarrow \beta$, where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds:

- $\alpha \rightarrow \beta$ is a trivial functional dependency (that is, $\beta \subseteq \alpha$).
- α is a superkey for schema R.

A database design is in BCNF if each member of the set of relation schemas that constitutes the design is in BCNF.

Consider the following relation schemas and their respective functional dependencies:

- Customer-schema =(customer-name, customer-street, customer-city)
  Customer-name→ customer-street customer-city
- Branch-schema = (branch-name, assets, branch-city)
  Branch-name→ assets branch-city
- Loan-info-schema = (branch-name, customer-name, loan-number, amount)
  Loan-number→ amount branch-name

Customer-schema is in BCNF, and customer-name is a candidate key for the schema. The only nontrivial functional dependencies that hold on customer-schema have customer-name on the left side of the arrow. Since customer-name on the left side do not violate the definition of BCNF. Similarly, it can be shown easily that the relation schema Branch-schema is in BCNF. The schema Loan-info-schema, however, is not in BCNF. First, note that loan-number is not a superkey for Loan-info-schema, we could have a pair of tuples representing a single loan made to two people---for example

(Downtown, John Bell, L-44,1000)
(Downtown, Jane Bell, L-44,1000)

Q6(a) What are the reasons for having variable_length records?

Ans. 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 individuals records; such a field is called 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 of different types were clustered (placed together) on disk blocks.

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

Ans. Ordered records have some advantages over unordered files.

- First, reading the records in order of the ordering key values becomes extremely efficient, because no sorting is required.
- Second, finding the next record from the current one in order of the ordering key usually requires no additional block accesses, because the next record is in the same block as the current one (unless the current record is the last one in the block).
- Third, using a search condition based on the value of an ordering key field results in faster access when the binary search technique is used, which constitutes an improvement over linear searches, although it is not often used for disk files.

(c) What is Partitioned Hashing? What are its advantage and disadvantage?

Ans. Partitioned hashing is an extension of static external hashing (Section 5.9.2) that allows access on multiple keys. It is suitable only for equality comparisons; range queries are not supported. In partitioned hashing, for a key consisting of n components, the hash function is designed to produce a result with n separate hash addresses. The bucket address is a concatenation of these n addresses. It is then possible to search for the required composite search key by looking up the appropriate buckets that match the parts of the address.

An advantage of partitioned hashing is that it can be easily extended to any number of attributes. The bucket addresses can be designed so that high order bits in the addresses correspond to more frequently accessed attributes. Additionally, no separate access structure needs to be maintained for the individual attributes.

The main drawback of partitioned hashing is that it cannot handle range queries on any of the component attributes.

Q7(a) Discuss the cost components for a cost function that are used to estimate query execution cost. Where is this information kept?

Ans. 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) Briefly explain the different methods for implementing joins?

Ans. Different methods for implementing joins are as follows:

- 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 $t[A] = s[B]$ (Note 11).

- 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 $s[B] = t[A]$.

- 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 . 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. In external sorting 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. The indexes provide the ability to access (scan) the records in order of the join attributes, but the records themselves are physically scattered all over the file blocks, so this method may be quite inefficient, as every record access may involve accessing a different disk block.

- Hash-join: The records of files R and S are both hashed to the same hash file, using the same hashing function on the join attributes A of R and B of S as hash keys. First, a single pass through the file with fewer records (say, R) hashes its records to the hash file buckets; this is called the partitioning phase, since the records of R are partitioned into the hash buckets. In the second phase, called the

probing phase, a single pass through the other file (*S*) then hashes each of its records to *probe* the appropriate bucket, and that record is combined with all matching records from *R* in that bucket. This simplified description of hash-join assumes that the smaller of the two files *fits entirely into memory buckets* after the first phase.

Q8 (a) Briefly explain the following problems that arise because of concurrent execution of transaction is

1. Lost Update Problem
2. Dirty Read Problem
3. Incorrect Summary Problem

Ans. Page No. 616 of Textbook 'Fundamentals of Database Systems' 5th Edition' by Elmasri Navathe.

(b) What is timestamp? What are the rules followed to ensure serializability in multiversion techniques based on timestamp ordering?

Ans. A timestamp is a unique identifier created by the DBMS to identify a transaction. Typically, timestamp values are assigned in the order in which the transactions are submitted to the system, so a timestamp can be thought of as the *transaction start time.* We will refer to the timestamp of transaction T as TS(T)**.** Concurrency control techniques based on timestamp ordering do not use locks; hence, *deadlocks cannot occur*.
Timestamps can be generated in several ways. One possibility is to use a counter that is incremented each time its value is assigned to a transaction. The transaction timestamps are numbered 1, 2, 3, . . . in this scheme. A computer counter has a finite maximum value, so the system must periodically reset the counter to zero when no transactions are executing for some short period of time. Another way to implement timestamps is to use the current date/time value of the system clock and ensure that no two timestamp values are generated during the same tick of the clock.

Multiversion Technique Based on Timestamp Ordering

In the Multiversion Technique Based on Timestamp Ordering, several versions X_1, X_2, ...,X_k of each data item *X* are maintained. For *each version,* the value of version X_i and the following two timestamps are kept:

1. read_TS(X_i)**:** The read timestamp of X_i is the largest of all the timestamps of transactions that have successfully read version X_i .

2. write_TS(X_i): The write timestamp of X_i is the timestamp of the transaction that wrote the value of version X_i .

Whenever a transaction T is allowed to execute a write_item(*X*) operation, a new version X_{k+1} of item *X* is created, with both the write_TS(X_{k+1}) and the read_TS(X_{k+1}) set to

TS(T). Correspondingly, when a transaction T is allowed to read the value of version *Xi*, the value of read_TS(X_i) is set to the larger of the current read_TS(X_i) and TS(T).

To ensure serializability, the following two rules are used:

1. If transaction T issues a write_item(*X*) operation, and version *i* of *X* has the highest write_TS(X_i) of all versions of *X* that is also *less than or equal to* TS(T), and read_TS(X_i) > TS(T), then abort and roll back transaction T; otherwise, create a new version of X_j with read_TS(X_j) = write_TS(X_j) = TS(T).

2. If transaction T issues a read_item(*X*) operation, find the version *i* of *X* that has the highest write_TS(X_i) of all versions of *X* that is also *less than or equal to* TS(T); then return the value of X_i to transaction T, and set the value of read_TS(X_i) to the larger of TS(T) and the current read_TS(X_i).

As in case 2, a read_item(*X*) is always successful, since it finds the appropriate version X_i to read based on the write_TS of the various existing versions of *X*. In case 1, however, transaction T may be aborted and rolled back. This happens if T is attempting to write a version of *X* that should have been read by another transaction T′ whose timestamp is read_TS(X_i); however, T′ has already read version Xi, which was written by the transaction with timestamp equal to write_TS(X_i). If this conflict occurs, T is rolled back; otherwise, a new version of *X*, written by transaction T, is created. Notice that, if T is rolled back, cascading rollback may occur. Hence, to ensure recoverability, a transaction T should not be allowed to commit until after all the transactions that have written some version that T has read have committed.

Q9 (a) What are checkpoints and why are they important? List the actions taken by the recovery manager during checkpoints?

Ans. 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) Briefly explain the shadow paging recovery scheme.

Ans. Page No. 684 of Textbook 'Fundamentals of Database Systems' 5th Edition' by Elmasri Navathe.

(c) Describe the three phase of the ARIES recovery method?

Ans. The ARIES recovery procedure consists of three main steps: (1) analysis, (2) REDO and (3) UNDO. The analysis step identifies the dirty (updated) pages in the buffer, and the set of transactions active at the time of the crash. The appropriate point in the log where the REDO operation should start is also determined. The REDO phase actually reapplies updates from the log to the database. Generally, the REDO operation is applied to only committed transactions. However, in ARIES, this is not the case. Certain information in the ARIES log will provide the start point for REDO, from which REDO operations are applied until the end of the log is reached. In addition, information stored by ARIES and in the data pages will allow ARIES to determine whether the operation to be redone has actually been applied to the database and hence need not be reapplied. Thus *only the necessary REDO operations* are applied during recovery. Finally, during the UNDO phase, the log is scanned backwards and the operations of transactions that were active at the time of the crash are undone in reverse order. The information needed for ARIES to accomplish its recovery procedure includes the log, the Transaction Table, and the Dirty Page Table. In addition, checkpointing is used. These two tables are maintained by the transaction manager and written to the log during checkpointing.
In ARIES, every log record has an associated log sequence number (LSN) that is monotonically increasing and indicates the address of the log record on disk. Each LSN corresponds to a *specific change* (action) of some transaction. In addition, each data page will store the LSN of the *latest log record corresponding to a change for that page.* A log record is written for any of the following actions: updating a page (write), committing a transaction (commit), aborting a transaction (abort), undoing an update (undo), and ending a transaction (end). The need for including the first three actions in the log has been discussed, but the last two need some explanation. When an update is undone, a *compensation log record* is written in the log. When a transaction ends, whether by committing or aborting, an *end log record* is written.
Common fields in all log records include: (1) the previous LSN for that transaction, (2) the transaction ID, and (3) the type of log record. The previous LSN is important because it links the log records (in reverse order) for each transaction. For an update (write)

action, additional fields in the log record include: (4) the page ID for the page that includes the item, (5) the length of the updated item, (6) its offset from the beginning of the page, (7) the before image of the item, and (8) its after image.

Besides the log, two tables are needed for efficient recovery: the Transaction Table and the Dirty Page Table, which are maintained by the transaction manager. When a crash occurs, these tables are rebuilt in the analysis phase of recovery. The Transaction Table contains an entry for *each active transaction,* with information such as the transaction ID, transaction status, and the LSN of the most recent log record for the transaction. The Dirty Page Table contains an entry for each dirty page in the buffer, which includes the page ID and the LSN corresponding to the earliest update to that page.

Checkpointing in ARIES consists of the following: (1) writing a begin_checkpoint record to the log, (2) writing an end_checkpoint record to the log, and (3) writing *the LSN of* the begin_checkpoint record to a special file. This special file is accessed during recovery to locate the last checkpoint information. With the end_checkpoint record, the contents of both the Transaction Table and Dirty Page Table are appended to the end of the log. To reduce the cost, fuzzy checkpointing is used so that the DBMS can continue to execute transactions during checkpointing. In addition, the contents of the DBMS cache do not have to be flushed to disk during checkpoint, since the Transaction Table and Dirty Page Table—which are appended to the log on disk—contain the information needed for recovery.

After a crash, the ARIES recovery manager takes over. Information from the last checkpoint is first accessed through the special file. The analysis phase starts at the begin_checkpoint record and proceeds to the end of the log. When the end_checkpoint record is encountered, the Transaction Table and Dirty Page Table are accessed (recall that these tables were written in the log during checkpointing). During analysis, the log records being analyzed may cause modifications to these two tables. For instance, if an end log record was encountered for a transaction *T* in the Transaction Table, then the entry for *T* is deleted from that table. If some other type of log record is encountered for a transaction T′, then an entry for T′ is inserted into the Transaction Table, if not already present, and the last LSN field is modified. If the log record corresponds to a change for page *P*, then an entry would be made for page *P* (if not present in the table) and the associated LSN field would be modified. When the analysis phase is complete, the necessary information for REDO and UNDO has been compiled in the tables.

The REDO phase follows next. To reduce the amount of unnecessary work, ARIES starts redoing at a point in the log where it knows (for sure) that previous changes to dirty pages *have already been applied to the database on disk.* It can determine this by finding the smallest LSN, *M,* of all the dirty pages in the Dirty Page Table, which indicates the log position where ARIES needs to start the REDO phase. Any changes corresponding to a LSN < *M,* for redoable transactions, must have already been propagated to disk or already been overwritten in the buffer; otherwise, those dirty pages with that LSN would be in the buffer (and the Dirty Page Table). So, REDO starts at the log record with LSN = *M* and scans forward to the end of the log. For each change recorded in the log, the REDO algorithm would verify whether or not the change has to be reapplied. For example, if a change recorded in the log pertains to page *P* that is not in the Dirty Page Table, then this change is already on disk and need not be reapplied. Or, if a change

recorded in the log (with LSN = *N*, say) pertains to page *P* and the Dirty Page Table contains an entry for P with LSN greater than *N*, then the change is already present. If neither of these two conditions hold, page *P* is read from disk and the LSN stored on that page, LSN(*P*), is compared with *N*. If *N* < LSN(*P*), then the change has been applied and the page need not be rewritten to disk.

Once the REDO phase is finished, the database is in the exact state that it was in when the crash occurred. The set of active transactions—called the undo_set—has been identified in the Transaction Table during the analysis phase. Now, the UNDO phase proceeds by scanning backward from the end of the log and undoing the appropriate actions. A compensating log record is written for each action that is undone. The UNDO reads backward in the log until every action of the set of transactions in the undo_set has been undone. When this is completed, the recovery process is finished and normal processing can begin again.

Lsn	Last-lsn	Tran_id	Type	Page_id	Other_information
1	0	T_1	update	C	…
2	0	T_2	update	B	…
3	1	T_1	commit		…
4	begin checkpoint				
5	end checkpoint				
6	0	T_3	update	A	…
7	2	T_2	update	C	…
8	7	T_2	commit		…

Figure (a)

TRANSACTION TABLE

Transaction_id	Last_lsn	Status
T_1	3	commit
T_2	2	in progress

DIRTY PAGE TABLE

Page_id	Lsn
C	1
B	2

Figure (b)

TRANSACTION TABLE

Transaction_id	Last_lsn	Status
T_1	3	commit
T_2	8	commit
T_3	6	in progress

DIRTY PAGE TABLE

Page_id	Lsn
C	1
B	2
A	6

Figure (c)

Consider the recovery example shown in figure. There are three transactions: T_1, T_2, and T_3 . T_1 updates page *C*, T_2 updates pages *B* and C, T3 and updates page *A*. Figure (a) shows the partial contents of the log and Figure (b) shows the contents of the Transaction

Table and Dirty Page Table. Now, suppose that a crash occurs at this point. Since a checkpoint has occurred, the address of the associated begin_checkpoint record is retrieved, which is location 4. The analysis phase starts from location 4 until it reaches the end. The end_checkpoint record would contain the Transaction Table and Dirty Page table in Figure (b), and the analysis phase will further reconstruct these tables. When the analysis phase encounters log record 6, a new entry for transaction T_3 is made in the Transaction Table and a new entry for page *A* is made in the Dirty Page table. After log record 8 is analyzed, the status of transaction T_2 is changed to committed in the Transaction Table. Figure (c) shows the two tables after the analysis phase.

For the REDO phase, the smallest LSN in the Dirty Page table is 1. Hence the REDO will start at log record 1 and proceed with the REDO of updates. The LSNs {1, 2, 6, 7} corresponding to the updates for pages *C, B, A,* and *C,* respectively, are not less than the LSNs of those pages (as shown in the Dirty Page table). So those data pages will be read again and the updates reapplied from the log (assuming the actual LSNs stored on those data pages are less then the corresponding log entry). At this point, the REDO phase is finished and the UNDO phase starts. From the Transaction Table (Figure c), UNDO is applied only to the active transaction T_3 . The UNDO phase starts at log entry 6 (the last update for T_3) and proceeds backward in the log. The backward chain of updates for transaction T_3 (only log record 6 in this example) is followed and undone.

TEXTBOOK

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