SPPU New Syllabus

A Book Of

DATABASE MANAGEMENT SYSTEMS - II

For B.C.A.(Science) : Semester - III

[Course Code 232 : Credit - 4]

CBCS Pattern

As Per New Syllabus, Effective from June 2020

Dr. Ms. Manisha Bharambe
M.Sc(Comp. Sci),M.Phil, Ph.D (Comp. Sci)
Vice Principal and Associate Professor,
Computer Science Department,
MES Abasaheb Garware College,
Pune 4

Mrs. Veena Gandhi
M.C.S., M.Phil (Comp. Sci.), UGC-NET
Assistant Professor,
Computer Science Department,
Abeda Inamdar Senior College,
Pune 1

Price ₹ 250.00

NIRALI PRAKASHAN
ADVANCEMENT OF KNOWLEDGE

N5412

Syllabus ...

Unit I - Relational Database Design **16 Hrs**

1.1 PL/PG SQL: Language structure

1.2 Controlling the program flow, conditional statements, loops

1.3 Views

1.4 Functions

1.5 Handling errors and exceptions

1.6 Cursors

1.7 Triggers

Unit II - Transaction Concepts **4 Hrs**

2.1 Transaction, properties of transaction, states of transactions

2.2 Concurrent execution of transactions and conflicting operations

2.3 Schedules, types of schedules, concept of serializability, precedence graph for serializability

Unit III - Concurrency Control **10Hrs**

3.1 Ensuring serializability by locks, different lock modes

3.2 2PL and its variations

3.3 Multiple Granularity locking protocol

3.4 Basic timestamp method for concurrency, Thomas Write Rule

3.5 Locks with multiple granularity, dynamic database concurrency (Phantom Problem)

3.6 Timestamps versus locking

3.7 Optimistic concurrency control algorithm, multi version concurrency control

3.8 Deadlock handling methods -

3.8.1 Detection and Recovery (Wait for graph).

3.8.2 Prevention algorithms (Wound-wait, Wait-die)

Unit IV - Crash Recovery **8 Hrs**

4.1 Transaction Failure classification

4.2 Recovery concepts

4.3 Checkpoints

4.4 Recovery with concurrent transactions (Rollback, checkpoints, commit)

4.5 Log base recovery techniques (Deferred and Immediate update)

4.6 Buffer Management

4.7 Database backup and recovery from catastrophic failures

4.8 Shadow paging

Unit V - Database Security **06 Hrs**

5.1 Introduction to database security concepts

5.2 Methods for database security

5.3 Discretionary access control method

5.4 Mandatory access control and role based access control for multilevel security

5.5 Use of views in security enforcement

5.6 Overview of encryption technique for security

5.7 Statistical database security

Unit VI - Database System Architectures **04 Hrs**

6.1 Centralized and Client – Server Architectures

6.2 Server System Architectures

6.3 Introduction to Parallel Systems

6.4 Introduction to Distributed Systems

6.5 Introduction to Object Based Databases

❖❖❖

Contents ...

1.	Relational Database Design	1.1 – 1.48
2.	Transaction Concepts	2.1 – 2.34
3.	Concurrency Control	3.1 – 3.30
4.	Crash Recovery	4.1 – 4.30
5.	Database Security	5.1 – 5.20
6.	Database System Architectures	6.1 – 6.18
*	Assignments	A.1 – A.58

❖❖❖

1... Relational Database Design

Objectives...

- To know the Language Structure of PL/pgSQL
- To learn Programming Control Flow Statements like Conditional Statements, Loops etc.
- To define Views, User Defined Functions, Cursors and Triggers
- To handle Errors and Exceptions

1.1 INTRODUCTION

- PL/pgSQL (Procedural Language/PostgreSQL) is a loadable procedural language for the PostgreSQL database system.
- A procedural language is a programming language which specifies order or sequence of steps that are followed to produce a desired result or output.
- We can use PL/pgSQL to group sequences of SQL and programming statements together within a database server. It allows us to develop complex functions in PostgreSQL that may not be possible using plain SQL statements.
- Currently procedural languages available in the standard PostgreSQL distribution are PL/pgSQL, PL/Tcl, PL/Perl and PL/Python.

1.1.1 Features of PL/pgSQL (S-18)

- The features of PL/pgSQL are listed below:
  1. PL/pgSQL is easy to use.
  2. PL/pgSQL adds control structures to the SQL language.
  3. PL/pgSQL can be used to create functions and trigger procedures.
  4. It can perform complex computations.
  5. PL/pgSQL inherits all user-defined types, functions, and operators.
  6. It can be defined to be trusted by the server.

1.1.2 Advantages of Using PL/pgSQL

- Advantages of using PL/pgSQL are listed below:
  1. **Portability:** One of important aspect of using PL/pgSQL is its portability as its functions are compatible with all platforms that can operate the PostgreSQL database system.
  2. **SQL Support:** The use of SQL within PL/pgSQL code can increase the power, flexibility, and performance of the programs. One can access to all data types, operators, and functions within PL/pgSQL code. If multiple SQL statements are executed from a PL/pgSQL code block, the statements are processed at one time, instead of the normal behavior of processing a single statement at a time.
  3. **Better Performance:** PostgreSQL databases use SQL as query language. But every SQL statement must be executed individually by the database server. So client application must send each query to the database server, wait for it to be processed, receive and process the results, and do some computation, then send further queries to the server. All this incurs lot of inter-process communication and will also incur network overhead if the client is on a different machine than the database server. But with PL/pgSQL one can group a block of computation and a series of queries inside the database server, thus having the power of a procedural language, the ease of use of SQL and considerable savings of client/server communication overhead. It eliminates extra round trips between client and server, also avoids multiple times query parsing. This can result in a considerable performance increase as compared to an application that does not use PL/pgSQL.

1.1.3 Developing in PL/pgSQL

- Developing in PL/pgSQL is very easy and simple. There are two ways to develop in PL/pgSQL:
  1. We can use the text editor to create our functions and use psql to load and test those functions.
  2. Another good way to develop in PL/pgSQL is with a GUI database access tool that facilitates development in a procedural language such as PgAccess.

1.1.4 To Add Language PL/pgSQL to Database

- A procedural language must be "installed" into each database where it is to be used. But procedural languages installed in the default database are automatically available in all subsequently created databases.
- To add PL/pgSQL to the PostgreSQL database, we can either use the *createlang* application from the command line, or the 'CREATE LANGUAGE' SQL command from within a database client such as psql.

1. **Using psql to add PL/pgSQL:**

- The use of the CREATE LANGUAGE command first requires the creation of the PL/pgSQL call handler, which is the function that actually processes and interprets the PL/pgSQL code.
- CREATE LANGUAGE is the SQL command which adds procedural languages to the currently connected database. Before it can be used, however, the CREATE FUNCTION command must first be used to create the procedural call handler.
- Here, is the **syntax** to create a PL/pgSQL call handler with CREATE FUNCTION:

```
CREATE FUNCTION plpgsql_call_handler()
RETURNS OPAQUE AS '/postgres_library_path/plpgsql.so' LANGUAGE 'C'
```

- In above syntax, `postgres_library_path` is the absolute system path to the installed PostgreSQL library files. This path, by default, is `/usr/local/pgsql/lib`.
- The special return type of OPAQUE tells the database that this function does not return one of the defined SQL data types and is not directly usable in SQL statements.
- Example 1 uses the CREATE FUNCTION command to create the PL/pgSQL call handler, assuming the `plpgsql.so` file is in the default location.

Example 1: Creating the PL/pgSQL Call Handler.

```
postgres=#     CREATE FUNCTION plpgsql_call_handler ()
postgres-#     RETURNS OPAQUE
postgres-#     AS '/usr/local/pgsql/lib/plpgsql.so'
postgres-#     LANGUAGE 'C';
CREATE
```

- Example 1 only creates the function handler; the language itself must also be added with the CREATE LANGUAGE command. Here, is the **syntax** to add PL/pgSQL to a database with following command:

```
CREATE LANGUAGE 'plpgsql' HANDLER plpgsql_call_handler
LANCOMPILER 'PL/pgSQL'
```

- In above syntax, `plpgsql` is the name of the language to be created, the `plpgsql_call_handler` is the name of the call handler function (e.g., the one created in Example 1), and the PL/pgSQL string constant following the LANCOMPILER keyword is an arbitrary descriptive note.
- Example 2 adds PL/pgSQL to the postgres database with the CREATE LANGUAGE command.

Example 2: Adding PL/pgSQL with CREATE LANGUAGE.

```
postgres=# CREATE LANGUAGE 'plpgsql' HANDLER plpgsql_call_handler
postgres-# LANCOMPILER 'PL/pgSQL';
CREATE
```

2. Using *createlang* to add PL/pgSQL:

- The *createlang* utility is simpler to use, as it abstracts the creation of the call handler and the language away from the user.
- To execute *createlang*, we will first need to be at the command prompt. If the operating system username (we are currently logged into) is the same as that of a database superuser account on the target database, we can call *createlang* with the command shown in following Example 3.

Example 3: Using *createlang* as a Database Superuser.

```
$ cd /usr/local/pgsql/bin
postgres=# createlang plpgsql postgres
```

- The *createlang* program will return you to a shell prompt upon successful execution.

1.2 PL/PgSQL: LANGUAGE STRUCTURE

- In this section, we will discuss the block organization of PL/pgSQL code, how to use comments, how PL/pgSQL expressions are organized, and the usage of statements.
- PL/pgSQL is a block-structured and case-insensitive language. A block comprises statements inside the same set of the DECLARE/BEGIN and END statements.
- A block in PL/pgSQL can be defined as follows:

```
 <<label>> ]
[ DECLARE
    declarations ]
BEGIN
    statements
END [ label ];
```

- Each declaration and each statement within a block is terminated by a semicolon (;). A block that appears within another block must have a semicolon after END, as shown above; however the final END that concludes a function body does not require a semicolon.

1.2.1 Structure of a PL/pgSQL Code Block

- PL/pgSQL code is organized as block structured code where each portion of code is designed to exist as a function.
- Code blocks are entered within a SQL CREATE FUNCTION call that creates the PL/pgSQL function in the PostgreSQL database.
- All variables must be declared before being used, and code segments accept arguments when called and return arguments at their end.

- The following example shows the structure of PL/pgSQL:

Example 4: Structure of a PL/pgSQL code block.

```
CREATE FUNCTION identifier (arguments) RETURNS type AS'
  DECLARE
    declaration;
    [...]
  BEGIN
    statement;
    [...]
  END;
' LANGUAGE 'plpgsql';
```

- The declaration section is denoted by the DECLARE keyword. In the declaration section of a code block, all variables are declared, and optionally initialized to a default value. A variable declaration specifies the variable's name and type. Each variable declaration is ended with a semicolon (;).
- After declaring variables, the main body of the code block is started with the BEGIN keyword. The code block's statements should appear after the BEGIN keyword. The END keyword designates the end of the code block. The main block of a PL/pgSQL function should return a value of its specified return type.
- A block of PL/pgSQL code can contain sub-blocks, which are code blocks nested within other code blocks. Sub-blocks can be useful for the organization of code within a large PL/pgSQL function. All sub-blocks must follows normal block structure, meaning they must start with the DECLARE keyword, followed by the BEGIN keyword and a body of statements, then end with the END keyword.

1.2.2 Comments

- There are two methods of commenting in PL/pgSQL: Single line comment and Block comments (multiple line comments).

1. **Single Line Comment:**

- Single line comment begin with two dashes (--) and have no end-character, because the parser interprets any characters after the two dashes as a comment, until the end of the line.
- Example 5 demonstrates the use of single line comments.

Example 5: Using Single line Comments.

```
--This will be interpreted as a single line comment.
```

2. **Multi-line or Block Comment:**

- The second type of comment is same as comment in C programming languages. Block comments begin with the forward slash and asterisk characters (/*) and end with the asterisk and forward slash characters (*/).
- Block comments can span multiple lines. While single line comments can be nested within block comments, block comments cannot be nested within other block comments.
- Example 6 shows the usage of a block comment.

Example 6: Using block comments.

```
/*
  This is a
  block
  comment.
 */
```

1.2.3 Statements and Expressions

- Like most programming languages, PL/pgSQL code is composed of statements and expressions. Most of code will be made of statements and expressions which are essential to certain types of data manipulation.

1.2.3.1 Statements

- A statement performs an action within PL/pgSQL code, such as assignment of a value to a variable or the execution of a query.
- The organization of statements within a PL/pgSQL code block controls the order in which operations are executed within that code block.
- Declarative statements should appear in the declaration section after the DECLARE keyword, but these should only declare and/or initialize the variables that will be referenced within the code block.
- The most of statements will be placed between BEGIN keyword and before the END keyword. Every statement should end with a semicolon character (;).
- Basic types of statements in PL/pgSQL are as follows:

1. **Assignment Statement:** It assigns a value to a PL/pgSQL variable and written as:

Syntax: `variable := expression;`

Example 7: Example for Assignment statement.

```
sum := sum + 100;
```

2. **Perform Statement:** This statement executes query and discards the result. Write the query in the same way as SQL SELECT command, but replace the initial keyword SELECT with PERFORM. A PERFORM statement sets FOUND true if it produces (and discards) one or more rows, false if no row is produced. This statement also used to call function and ignores its return data.

   Syntax: `PERFORM query ;`

   OR

   ```
   PERFORM function_name(arguments);
   ```

   Example 8: Example for PERFORM statement.

   ```
   PERFORM * from Emp where eno := 1;
   ```

   Example 9:

   ```
   PERFORM add_product(pno,pname);
   ```

   Here, `add_product` is function name with two arguments `pno` and `pname`.

3. **SELECT INTO Statement:** The result of a SQL command giving a single row can be assigned to a record variable, row-type variable, or list of scalar variables. This is done by writing the base SQL command and adding an INTO clause.

   Syntax: `SELECT select_expressions INTO [STRICT] target FROM table_name`

   ```
   WHERE condition;
   ```

   where, `target` can be a record variable, a row variable, or a comma-separated list of simple variables and record/row fields. If a row or a variable list is used as target, the query's result columns must exactly match the structure of the target as to number and data types, or else a run-time error occurs. When a record variable is the target, it automatically configures itself to the row type of the query result columns.

   The following example shows usage of SELECT INTO statement.

   Example 10: Example for SELECT INTO statement.

   ```
   SELECT * INTO myrec FROM product WHERE pname = prod_name;
   IF NOT FOUND THEN
       RAISE EXCEPTION ' PRODUCT % not found ', prod_name;
   END IF;
   ```

4. **EXECUTE Statement:** Sometimes, we want to generate dynamic commands inside the PL/pgSQL function which involve different tables or different data types each time they are executed. There is no plan caching for commands executed via EXECUTE.

   Instead, the command is prepared each time the statement is run. Thus the query string can be dynamically created within the function to perform actions on different tables and columns.

Syntax:

```
EXECUTE query-string [INTO [STRICT] target][USING expression [,...]];
```

where, `query-string` is normally SELECT SQL statement. The optional `target` is a record variable, a row variable, or a comma-separated list of simple variables and record/row fields, into which the results of the command will be stored. The optional `USING` expressions supply values to be inserted into the command.

Example 11: Example for EXECUTE statement.

```
EXECUTE 'SELECT * FROM product WHERE pno = ' || a INTO r;
```

5. **GET DIAGNOSTICS Statement:** There are several ways to determine the effect of a command. The first method is to use the GET DIAGNOSTICS command.

Syntax:

```
GET DIAGNOSTICS variable = item [ , ... ];
```

This command allows retrieval of system status indicators. Each item is a key word identifying a state value to be assigned to the specified variable .The currently available status items are ROW_COUNT, the number of rows processed by the last SQL command sent to the SQL engine, and RESULT_OID, the OID of the last row inserted by the most recent SQL command.

Example 12: Example for GET DIAGNOSTICS statement.

```
GET DIAGNOSTICS integer_var = ROW_COUNT;
```

1.2.3.2 Expressions

- Expressions are calculations or operations that return their results as one of PostgreSQL's base data types.
- For example expression is x := a + b, which adds the variables a and b, then assigns the result to the variable x.
- Example 13 shows a simple PL/pgSQL function that assigns result of a multiplication expression to the variable x.

Example 13: Using Expressions.

```
CREATE FUNCTION mult() RETURNS int4 AS '
DECLARE
   x int 4;
BEGIN
   x := 10 * 10;
   return x;
END;
' LANGUAGE 'plpgsql';
```

Execution of Code:

```
postgres=# SELECT mult() AS output;
```

Output:

```
-----------------
    100
(1 row)
```

1.2.4 Declaration (S-18, 19; W-18)

- Variables are used within PL/pgSQL code to store data of specified type.
- Variables in PL/pgSQL can be represented by any of SQL's standard data types such as an INTEGER or CHAR.
- In addition to SQL data types, PL/pgSQL also provides the additional RECORD data type, which is designed to allow us to store row information without specifying the columns that will be supplied when data is inserted into the variable.
- All variables that we will be using within a code block must be declared under the DECLARE keyword. If a variable is not initialized to a default value when it is declared, its value will default to the SQL NULL type.
- A list of supported data types is as below:

1. boolean	1. char
3. integer	4. double precision
5. date	6. time.

Syntax for Variable Declaration:

```
Variable_name [CONSTANT] type [NOT NULL] [{DEFAULT|:=} expression];
```

- If `DEFAULT` clause given, it initializes the variable when the block is entered. If the `DEFAULT` clause is absent, then the variable is initialized to the SQL null value.
- The `CONSTANT` option prevents the variable from being modified to after initialization, so that its value will remain constant for the duration of the block. All variables declared as `NOT NULL` must have a non null default value specified otherwise error exists.

Example 14: Variable declaration.

```
uid  integer;
qty  numeric(5);
str  varchar(20);
emp_row   tablename%ROWTYPE;   (emp_row  emp%ROWTYPE ;)
emp_var   tablename.columnname%TYPE; ( emp_var  emp.salary%TYPE;)
emp_rec  RECORD;
```

1.2.4.1 Declaring Function Parameters

- Functions can accept and return values called function parameters or arguments.
- Parameters passed to functions are named with the identifiers $1, $2, etc. Optionally, aliases can be declared for $n parameter names for increased readability. Either the alias or the numeric identifier can then be used to refer to the parameter value.
- There are two ways to create an alias. The preferred way is to give a name to the parameter in the CREATE FUNCTION command.

Example 15: Example for CREATE FUNCTION command.

```
CREATE FUNCTION IT_tax(subtotal  real) RETURNS real AS'
BEGIN
    RETURN subtotal * 0.10;
END;
'LANGUAGE 'plpgsql';
```

- The another way is available before PostgreSQL 8.0, to explicitly declare an alias, using the declaration **syntax:**

```
name ALIAS FOR $n;
```

Example 16:

```
CREATE FUNCTION IT_tax(real) RETURNS real AS'
DECLARE
    subtotal ALIAS FOR $1;
BEGIN
    RETURN subtotal * 0.10;
END;
'LANGUAGE 'plpgsql';
```

- We can declare an alias for any variable, not just function parameters. The different name can be assigned for variables with predetermined names, such as NEW or OLD within a trigger procedure.

Example 17:

```
DECLARE
  prior ALIAS FOR old;
  updated ALIAS FOR new;
```

1.2.4.2 Attributes (W-18; S-19)

- PL/pgSQL provides variable attributes to assist in working with database objects.
- Use attributes to assign a variable either the type of a database object, with the %TYPE attribute, or the row structure of a row with the %ROWTYPE attribute.

1. **The %TYPE Attribute:**

- The %TYPE is used to declare a variable with the type of a referenced database object.

Syntax:

```
variable_name table_name.column_name%TYPE
```

Example 18: Using the %TYPE attribute.

```
CREATE FUNCTION get_emp(text) RETURNS text AS '
DECLARE
   f_name ALIAS FOR $1;
   l_name emp.last_name%TYPE;
BEGIN
   SELECT INTO l_name last_name FROM emp WHERE first_name = f_name;
      RETURN f_name || '' '' || l_name;
END;
' LANGUAGE 'plpgsql';
```

Execution of Code:

```
postgres=# SELECT get_emp('Kiran');
```

Output:

```
 get_emp
---------------------
Kiran Gandhi
(1 row)
```

2. **The %ROWTYPE Attribute**

- The %ROWTYPE is used to declare a PL/pgSQL row with the same structure as the row specified during declaration.
- It is similar to the RECORD data type, but a variable declared with %ROWTYPE will have the exact structure of that table row, whereas a RECORD variable is not structured and will accept a row from any table.

Example 19: Using the %ROWTYPE attribute.

```
CREATE FUNCTION get_emp (text) RETURNS text AS '
  DECLARE
     f_name ALIAS FOR $1;
     found_emp   emp%ROWTYPE;
  BEGIN
    SELECT INTO found_emp  * FROM emp  WHERE first_name = f_name;
      RETURN found_emp.first_name || '' '' || found_emp.last_name;
  END;
' LANGUAGE 'plpgsql';
```

Execution of Code:

```
postgres=#SELECT get_emp('Kiran');
```

Output:

```
 get_emp
---------------------
Kiran Gandhi
(1 row)
```

1.2.4.3 Record (S-19)

- Record variables are similar to ROW variables, but they have no predefined structure. Their structure depends on the structure of the actual row assigned to them during SELECT or FOR command.
- RECORD is only placeholder, not true datatype.

Syntax:

```
Variable_name  RECORD;
```

Example 20: Example for RECORD in PL/pgSQL.

```
DECLARE
emp_rec RECORD;
```

1.2.4.4 Rename

- Variable identifiers can be renamed using the RENAME keyword.
- Renaming a variable's identifier only alters the way the variable is referenced; we will not modify the value of a variable by renaming it.
- Variables declared with %TYPE and %ROWTYPE can also renamed with the RENAME keyword.

Syntax:

```
RENAME old_identifier TO new_identifier;
```

Example 21: Example for RENAME command

```
RENAME uid to USER_ID;
```

1.3 CONTROLLING THE PROGRAM FLOW (W-18)

- Like the most programming languages, PL/pgSQL also provide ways controlling flow of program execution by using conditional statements and loops.

1.3.1 Conditional Statements

- A conditional statement specifies an action or set of actions that should be executed based on the result of logical condition specified within the statement. If a given condition is true, the specified action should be taken.

- PL/pgSQL has three forms of IF i.e., IF ... THEN, IF ... THEN ... ELSE, and IF ... THEN ... ELSIF ... THEN ... ELSE. Let us see in detail each control statement.

1. IF....THEN Statement:

- In IF....THEN statement a statement or block of statements is executed if a given condition evaluates true.

Syntax:

```
IF condition THEN
  statement;
END IF;
```

- In the following example, *book-id* and *edition number* is passed as input parameter and *no. of books in stock* is fetched for the same.

Example 22: Using the IF-THEN statement.

```
CREATE FUNCTION stk_amount (integer, integer) RETURNS integer AS '
  DECLARE
    id ALIAS FOR $1;
    edtn ALIAS FOR $2;
    bn TEXT;
    amount INTEGER;
  BEGIN
      SELECT INTO bn isbn FROM editions WHERE
      book_id = id AND edition = edtn;
    IF bn IS NULL THEN
      RETURN -1;
    END IF;
    SELECT INTO amount stock FROM stock WHERE isbn = bn;
    RETURN amount;
  END;
'LANGUAGE 'plpgsql';
```

Execution of Code:

```
postgres=# SELECT stk_amount(7808,1);
```

Output:

```
 stk_amount
---------------------
      22
(1 row)
```

2. IF...THEN....ELSE Statement

- The IF...THEN....ELSE statement allows you to execute a block of statements *if* a condition evaluates to true, otherwise a block of statements in *else* part is executed.

Syntax:

```
IF condition THEN
  statements
ELSE
   statements
END IF;
```

- In Example 23, first the ISBN number is retrieved, store it and then use it to retrieve the stock number of the book in question.

Example 23: Using the `IF..THEN..ELSE` statement.

```
CREATE FUNCTION Chk_stock (integer,integer) RETURNS boolean AS '
  DECLARE
    b_id ALIAS FOR $1;
    b_edition ALIAS FOR $2;
    _isbn TEXT;
    stock_amount INTEGER;
  BEGIN
    SELECT INTO b_isbn isbn FROM editions WHERE
      book_id = b_id AND edition = b_edition;
    IF b_isbn IS NULL THEN
      RETURN FALSE;
    END IF;
     SELECT INTO stock_amount stock FROM stock WHERE isbn = b_isbn;
    IF stock_amount <= 0 THEN
      RETURN FALSE;
    ELSE
      RETURN TRUE;
    END IF;
  END;
' LANGUAGE 'plpgsql';
```

Execution of Code:

```
postgres=# SELECT Chk_stock(4513,2);
```

Output:

```
 chk_stock
-----------
     t
(1 row)
```

3. IF...THEN...ELSEIF...THEN...ELSE Statement:

- The IF...THEN...ELSEIF...THEN...ELSE statement provides a convenient method of checking several alternatives in turn.
- The IF conditions are tested successively until the first one that is true is found ,then the associated statement(s) are executed, after which control passes to the next statement after END IF. If none of the IF conditions is true, then the ELSE block (if any) is executed.

Syntax:

```
IF condition THEN
       Statements
 ELSE IF condition THEN
        statements
 ELSE IF condition THEN
    statements
 ELSE
      statements
END IF;
```

- The following example accepts number and check whether it is positive, negative, zero or null.

Example 24: Using the IF...THEN...ELSEIF...THEN...ELSE statement.

```
CREATE FUNCTION chk_no(interger) RETURNS text AS'
DECLARE
number ALIAS FOR $1;
result text;
BEGIN
  IF number = 0 THEN
   result := 'zero';
ELSIF number>0 THEN
     result := 'positive';
ELSIF number<0 THEN
     result := 'negative';
ELSE
   result := 'NULL';
END IF;
END;
' LANGUAGE 'plpgsql';
```

Execution of Code:

```
postgres=# SELECT chk_no(0);
```

Output:

```
chk_no
----------
 zero
```

Simple CASE Statement:

- The simple form of CASE provides conditional execution based on equality of operands. The search-expression is evaluated (once) and successively compared to each expression in the WHEN clauses.
- If a match is found, then the corresponding statements are executed, and then control passes to the next statement after END CASE. If no match is found, the ELSE statements are executed; but if ELSE is not present, then a CASE_NOT_FOUND exception is raised.

Syntax:

```
CASE search-expression
WHEN expression [,expression [ ... ]] THEN
 statements
[ WHEN expression [,expression [ ... ]] THEN
statements
... ]
[ ELSE
statements ]
END CASE;
```

- The following example shows different prices depending on rate information.

Example 25: Using CASE statement.

```
CREATE FUNCTION get_price(p_film_id integer)RETURNS VARCHAR(50)AS‘
DECLARE
 rate NUMERIC;
 price_segment VARCHAR(50);
BEGIN
    SELECT INTO rate rental_rate
    FROM film
    WHERE film_id = p_film_id;
CASE rate
  WHEN 0.99 THEN
            price_segment = 'average';
  WHEN 2.99 THEN
            price_segment = 'normal';
  WHEN 4.99 THEN
            price_segment = 'High';
```

```
ELSE
     price_segment = 'Unspecified';
END CASE;
    RETURN price_segment;
END;
' LANGUAGE 'plpgsql';
```

1.3.2 Loops

- Loops conditions are used to perform some task repeatedly for fixed number of times or until some are valid. PL/pgSQL provides three iterative loops i.e., the basic loop, WHILE loop and the FOR loop.

1.3.2.1 The Basic Loop

- This is the basic unconditional loop which starts with keyword LOOP and executes the statements within its body until terminated by an EXIT or RETURN statement.

Syntax:

```
[<<label>>]
LOOP
  statements
END LOOP;
```

- The optional `label` can be used by EXIT statements in nested loops to specify which level of nesting should be terminated.

```
EXIT [ label ] [ WHEN expression ];
```

- If no label is given, the innermost loop is terminated and the statement following END LOOP is executed next.
- If label is given, it must be the label of the current or some outer level of nested loop or block. Then the named loop or block is terminated and control continues with the statement after the loop's/block's corresponding END.
- If WHEN present, loop EXIT is occurs only if the specified condition is true, otherwise control passes to the statement after EXIT. This can be used to cause early exit from all types of loops; it is not limited to use with unconditional loops.
- The following example shows usage of loop statement.

Example 26: Using LOOP statement.

```
CREATE FUNCTION square_loop (integer) RETURNS integer AS '
  DECLARE
    num1 ALIAS FOR $1;
    result integer;
  BEGIN
     result := num1;
```

```
        LOOP
        result := result * result;
        EXIT WHEN result >= 50;
        END LOOP;
       RETURN result;
     END;
   ' LANGUAGE 'plpgsql';
```

Execution of Code:

```
postgres=# SELECT square_loop(3);
```

Output:

```
 square_loop
-----------------------
        81
(1 row)
```

1.3.2.2 WHILE Loop (S-19)

- The WHILE statement repeats a sequence of statements till the condition evaluates to true.
- It is top tested or entry controlled loop where the condition is checked just before each entry to the loop body.

Syntax:

```
[<<label>>]
WHILE condition LOOP
Statements
END LOOP [label];
```

Example 27: Using WHILE loop.

```
CREATE FUNCTION Example_while_loop (integer, integer) RETURNS integer AS'
  DECLARE
    low ALIAS FOR $1;
    high ALIAS FOR $2;
   result INTEGER = 0;
  BEGIN
    WHILE result != high LOOP
      result := result + 1;
    END LOOP;
    RETURN result;
  END;
' LANGUAGE 'plpgsql';
```

1.3.2.3 FOR Loop

- This form of FOR creates a loop that iterates over a range of integer values. The variable name is automatically defined as type integer and exists only inside the loop.
- The two expressions giving the lower and upper bound of the range are evaluated once when entering the loop. If the BY clause is not specified the iteration step is 1, otherwise it is the value specified in the BY clause, which again is evaluated once on loop entry. If REVERSE is specified then the step value is subtracted, rather than added after each iteration.

  Syntax:

```
[<<label>>]
FOR name IN [ REVERSE ]expression ..expression [ BYexpression ] LOOP
statements
END LOOP [label ];
```

- Some examples of integer FOR loops:

```
FOR i IN 1..10 LOOP
-- i will take on the values 1,2,3,4,5,6,7,8,9,10 within the loop
END LOOP;
FOR i IN REVERSE 10..1 LOOP
-- i will take on the values 10,9,8,7,6,5,4,3,2,1 within the loop
END LOOP;
FOR i IN REVERSE 10..1 BY 2 LOOP
-- i will take on the values 10,8,6,4,2 within the loop
END LOOP;
```

- If the lower limit is greater than the upper limit (or less than, in the REVERSE case), then loop body will not execute at all. No error is raised. If a label is attached to the FOR loop, then the integer loop variable can be referenced with a qualified name, using that label.
- The following example shows odd number generation.

Example 28: Using FOR loop.

```
CREATE FUNCTION Example_for_loop() RETURNS integer AS '
DECLARE
   counter interger;
BEGIN
  FOR counter IN 1..6 BY 2 LOOP
    RAISE NOTICE 'Counter:%', counter;
  END LOOP;
END;
return counter;
' LANGUAGE 'plpgsql';
```

Output:

```
NOTICE:  Counter:1
NOTICE:  Counter:3
NOTICE:  Counter:5
```

1.3.2.4 Looping through Query Results

- Using a different type of FOR loop, we can iterate through the results of a query and manipulate that data accordingly.

  Syntax:

```
[<<label>>]
FOR rec_1 IN query LOOP
   statements
END LOOP [label ];
```

- The rec_1 is a record variable, row variable, or comma-separated list of scalar variables. The rec_1 is successively assigned each row resulting from the query and the loop body is executed for each row. Here, is an example which fetches book title for given subject id.

Example 29: Using FOR loop through query result set.

```
CREATE FUNCTION extract_title (integer) RETURNS text AS '
  DECLARE
     sub_id ALIAS FOR $1;
     text_output TEXT :=''\n'';
     row_data books%ROWTYPE;
   BEGIN
     FOR row_data IN SELECT * FROM books
     WHERE subject_id = sub_id ORDER BY title  LOOP
       text_output := text_output || row_data.title || ''\n'';
     END LOOP;
     RETURN text_output;
   END;
 ' LANGUAGE 'plpgsql';
```

Execution of Code:

```
postgres=# SELECT extract_title(2);
```

Output:

```
      extract_title
-----------------------------------
    Wings of Fire
    Two states
```

(1 row)

- If the loop is terminated by an EXIT statement, the last assigned row value is still accessible after the loop.
- The query used in this type of FOR statement can be any SQL command that returns rows to the caller. SELECT is the most common case, but we can also use INSERT, UPDATE, or DELETE with a RETURNING clause.
- The RETURNING clause allows you to retrieve values of columns (and expressions based on columns) that were modified by an insert, delete or update.

1.3.2.5 FOR-IN-EXECUTE Statement

- The FOR-IN-EXECUTE statement is another way to iterate over rows of record set.

  Syntax:

```
[<<label>>]
FOR rec_1  IN EXECUTE text_expression [ USING expression [, ... ] ] LOOP
     statements
END LOOP [label];
```

- Here, the source query is specified as a string expression, which is evaluated and re-planned on each entry to the FOR loop. This allows the programmer to choose the speed of a preplanned query or the flexibility of a dynamic query.
- As with `EXECUTE`, parameter values can be inserted into the dynamic command via `USING`. The following example demonstrates how to use the FOR loop statement to loop through a dynamic query.
- It accepts two parameters: first parameter `sort_type`: 1 means sort the query result by title, 2 means sort the result by release year. The other parameter `n` is the number of rows to query from the `film` table. Notice that it will be used in the `USING` clause.

Example 30: Using `FOR-IN-EXECUTE` loop.

```
CREATE FUNCTION for_loop_dyn_query (sort_type INTEGER, n INTEGER)
                                                        RETURNS VOID AS'
DECLARE
    rec RECORD;
    query text;
BEGIN
  query := 'SELECT title, release_year FROM film ';
 IF sort_type = 1 THEN
     query := query || 'ORDER BY title';
 ELSIF sort_type = 2 THEN
   query := query || 'ORDER BY release_year';
 ELSE
     RAISE EXCEPTION 'Invalid sort type %s', sort_type;
```

```
 END IF;
  query := query || ' LIMIT $1';
  FOR rec IN EXECUTE query USING n
    LOOP
    RAISE NOTICE '% - %', rec.release_year, rec.title;
    END LOOP;
END;
'  LANGUAGE ' plpgsql';
```

1.4 VIEWS (W-18)

- We can use views to restrict table access so that the users see only specific rows or columns of a table.
- A view is a subset of a real table, selecting certain columns or certain rows from an ordinary table.
- Views are pseudo-tables means they are not real tables.
- When we create a view, we basically create a query and assign it name, therefore a view is useful for wrapping a commonly used complex query.
- A view can be created from one or many tables, which depends on the written PostgreSQL query.
- A normal view does not store any data except the materialized view. Since views are not ordinary tables, we may not be able to execute a DELETE, INSERT, or UPDATE statement on a view.
- Note that in a PostgreSQL, we can create a special view called a Materialized View that stores data physically and refreshes the data periodically from the base tables.

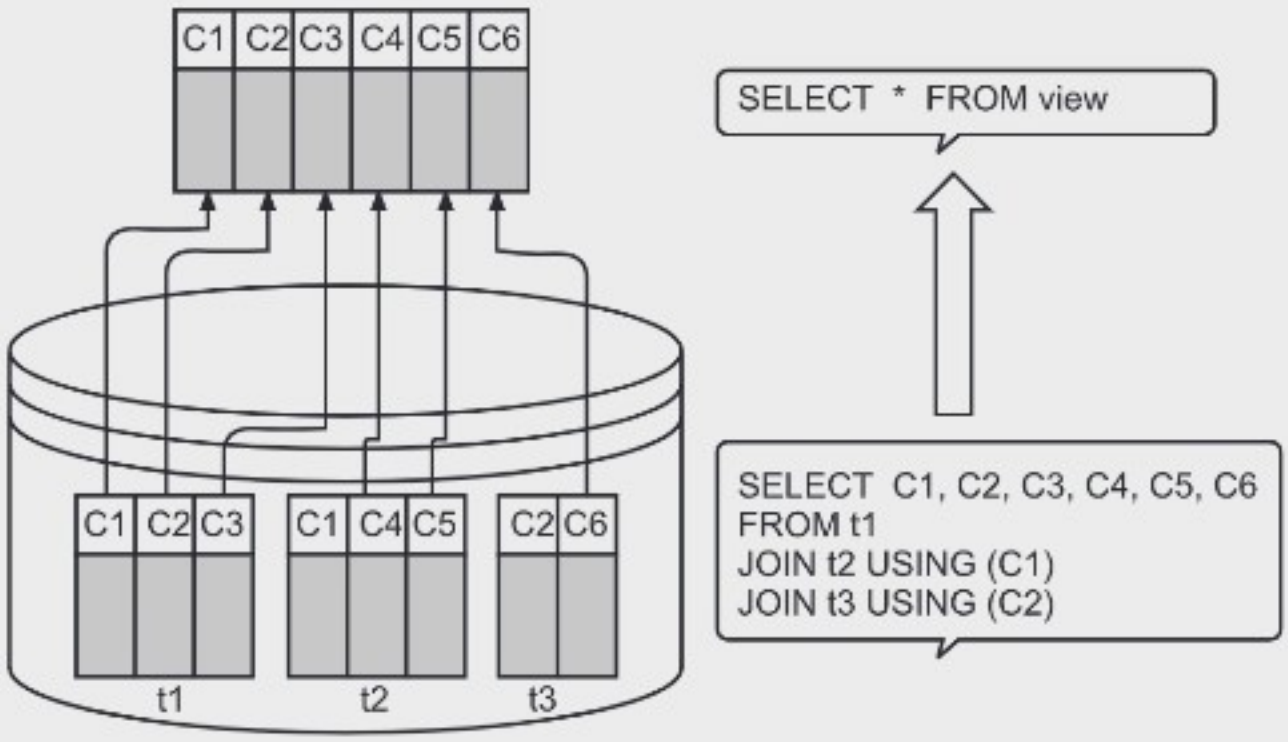


Fig. 1.1: Concept of View

- Views which are kind of virtual tables. These allow users to do the following:
  1. Structure data in a way that users or classes of users find natural or intuitive.
  2. Restrict access to the data such that a user can only see limited data instead of complete table.
  3. Summarize data from various tables, which can be used to generate reports.

1.4.1 Creating Views (S-19)

- The PostgreSQL views are created using the CREATE VIEW statement.
- The PostgreSQL views can be created from a single table, multiple tables, or another view.

Syntax:

```
CREATE [TEMP | TEMPORARY] VIEW view_name AS
SELECT column1, column2.....
FROM table_name
WHERE [condition];
```

- We can include multiple tables in the SELECT statement in very similar way as we use them in normal PostgreSQL SELECT query.
- If the optional TEMP or TEMPORARY keyword is present, the view will be created in the temporary space.
- Temporary views are automatically dropped at the end of the current session.
- Consider following emp table:

ID	NAME	AGE	ADDRESS	SALARY
1	Paul	32	Pune	20000
2	Allen	25	Mumbai	15000
3	Teddy	23	Pune	20000
4	Mark	25	Bangalore	65000
5	David	27	Mumbai	85000

Example 31: Example for VIEW command.

```
postgres=# CREATE VIEW COMPANY_VIEW AS
SELECT ID, NAME, AGE
FROM COMPANY;
```

- Now, we can query COMPANY_VIEW in a similar way as we query an actual table.

```
postgres =# SELECT * FROM COMPANY_VIEW;
```

ID	NAME	AGE
1	Paul	32
2	Allen	25
3	Teddy	23
4	Mark	25
5	David	27

- To change the defining query of a view, we use the `CREATE VIEW` statement with `OR REPLACE` addition as follows:

```
CREATE OR REPLACE view_name
AS Query
```

- To change the definition of a view, use the ALTER VIEW statement.
- PosgreSQL allows us to set a default value for a column name, change the view's schema, set or reset options of a view.
- For example, we can change the name of the view from `customer_master` to `customer_info` by using the following statement:

```
ALTER VIEW customer_master RENAME TO customer_info
```

1.4.2 Dropping Views

- To drop a view, simply use the `DROP VIEW` statement with the `view_name`.

Syntax:

```
DROP VIEW [IF EXISTS] view_name;
```

- We specify the name of the view that you want to remove after `DROP VIEW` clause. Removing a view that does not exist in the database will result an error. To avoid this, we normally add `IF EXISTS` option to the statement to instruct PostgreSQL to remove the view if it exists, otherwise do nothing.
- Following command will delete `COMPANY_VIEW` view, which previously created.

```
postgres=# DROP VIEW IF EXISTS COMPANY_VIEW;
```

1.5 FUNCTIONS (S-18,19)

- PostgreSQL functions are also known as Stored Procedures. These functions allow us to carry out operations that would normally take several queries and round trips in a single function within the database.
- Functions allow database reuse as other applications can interact directly with the stored procedures instead of a middle-tier or duplicating code.

- Functions can be created in a language PL/pgSQL using following **syntax:**

```
CREATE [OR REPLACE] FUNCTION function_name (arguments)
RETURNS return_datatype AS '
DECLARE
declaration;
[...]
BEGIN
< function_body >
[...]
RETURN { variable_name | value }
END;
'LANGUAGE 'plpgsql';
```

where,

`Function_name` specifies the name of the function.

`[OR REPLACE]` option allows modifying an existing function.

- Argument can be of types like IN, OUT, INOUT and VARIADIC as explained below:
  - **(i) IN:** By default, the parameter's type of any parameter in PostgreSQL is IN parameter. We can pass the IN parameters to the function but you cannot get them back as a part of result.
  - **(ii) OUT:** The OUT parameter is a part of the function arguments list and you can get the result back as the part of the result. To define OUT parameters, we use OUT keyword.
  - **(iii) INOUT:** The INOUT parameter is the combination IN and OUT parameters. It means that the caller can pass the value to the function. The function then changes the argument and passes the value back as a part of the result.
  - **(iv) VARIADIC:** A PostgreSQL function can accept a variable numbers of arguments with one condition that all arguments have the same data type. The arguments are passed to the function as an array.
- The function must contain a RETURN statement.
- RETURN clause specifies that data type we are going to return from the function. The `return_datatype` can be a base, composite, or domain type, or can reference the type of a table column. The return value of function cannot be left undefined.

  Syntax:

```
RETURN expression;
```

- `function_body` contains the executable part.
- The `AS` keyword is used for creating a standalone function.
- `plpgsql` is the name of the language that the function is implemented in. For backward compatibility, the name can be enclosed by single quotes.

Example 32: Example for CREATE FUNCTION

```
CREATE OR REPLACE FUNCTION hi_lo(IN a NUMERIC, IN b NUMERIC, IN  c
                              NUMERIC,  OUT hi NUMERIC, OUT lo NUMERIC) AS'
BEGIN
 hi := GREATEST(a,b,c);
 lo := LEAST(a,b,c);
END;
'LANGUAGE 'plpgsql';
```

1.5.1 Calling a Function

- The normal syntax to call another PL/pgSQL function from within PL/pgSQL is to either reference the function in a SQL SELECT statement, or during the assignment of a variable.
- For example:
  1. `SELECT function_name(arguments);`
  2. `variable_identifier := function_name(arguments);`
- The use of assignments and SELECT statements to execute functions is standard in PL/pgSQL because all functions in a PostgreSQL database must return a value of some type.
- We can also use the PERFORM keyword to call a function and ignore its return data.

  `PERFORM function_identifier(arguments);`

Example 33: Calling a function.

```
postgres=#SELECT hi_lo(10,20,30);
```

Output:

```
hi_lo
------------
30,10
```

1.5.2 Dropping a Function

- We can remove a function by using following syntax:

  `DROP FUNCTION name ( [ type [, ...] ] ) [ CASCADE | RESTRICT ]`

  where,

  Type: The data type(s) of the function's arguments.

  CASCADE: Automatically drop objects that depend on the function (such as operators or triggers).

  RESTRICT: Refuse to drop the function if any objects depend on it.

Example 34: Dropping function.

```
postgres=#DROP function hi_lo();
```

1.6 HANDLING ERRORS AND EXCEPTIONS (W-18)

RAISE statement:

- RAISE statement is used to raise errors and exceptions during a PL/pgSQL function's operation.
- A RAISE statement sends specified information to the PostgreSQL *elog* mechanism which is the standard PostgreSQL error logging utility, which typically logs data.

Syntax:

```
RAISE level ' 'format string' ' [, identifier [...] ];
```

- Followed the `RAISE` statement is the `level` option that specifies the error severity.
- There are following levels in PostgreSQL:
  - `DEBUG`
  - `LOG`
  - `NOTICE`
  - `INFO`
  - `WARNING`
  - `EXCEPTION`

1. **DEBUG:** DEBUG level statements send the specified text as a DEBUG: message to the PostgreSQL Log and the client program if the client is connected to a database cluster running in debug mode. DEBUG level RAISE statements will be ignored by a database running in production mode.
2. **NOTICE:** This level statement sends the specified text as a NOTICE: message to the PostgreSQL log and the client program in any PostgreSQL operation mode.
3. **EXCEPTION:** This statement sends the specified text as an ERROR: message to the client program and the PostgreSQL database log. The EXCEPTION level also causes the current transaction to be aborted. If we don't specify the level, by default, the RAISE statement will use EXCEPTION level that raises an error and stops the current transaction.

- The `format_string` is a string that specifies the message. The format uses percentage (%) placeholders that will be substituted by the next arguments. The number of placeholders must match the number of arguments; otherwise PostgreSQL will report the error message.
- In Example 35, the first RAISE statement raises a debug level message. The second and third RAISE statements send a notice to the user. Notice the use of the percent sign (%) in the third RAISE statement to mark the location in the string at which the value of an integer is to be inserted. Finally, the fourth RAISE statement displays an error and throws an exception, causing the function to end and the transaction to be aborted.

Example 35: Using the RAISE statement.

```
CREATE FUNCTION raise_test_example () RETURNS integer AS'
  DECLARE
       cnt INTEGER = 1;
  BEGIN
     RAISE DEBUG ''The raise_test() function began.''; cnt = cnt + 1;
     RAISE NOTICE ''Variable cnt was changed.'';
     RAISE NOTICE ''Variable cnt's value is now %.'', cnt;
     RAISE EXCEPTION ''Variable % changed.  Transaction aborted.'', cnt;
    RETURN 1;
  END;
' LANGUAGE 'plpgsql';
```

Execution of Code:

```
postgres=# SELECT raise_test_example();
```

Output:

```
NOTICE: Variable cnt was changed.
NOTICE: Variable cnt's value is now 2.
ERROR:  Variable 2 changed. Transaction aborted.
```

- We can attach additional information to the error report by writing USING followed by `option = expression` items. Each expression can be any string-valued expression. The allowed option keywords are:
  - **(i) MESSAGE:** Sets the error message text. This option can't be used in the form of RAISE that includes a format string before USING.
  - **(ii) DETAIL:** Supplies an error detail message.
  - **(iii) HINT:** Supplies a hint message.
  - **(iv) ERRCODE:** Specifies the error code (SQLSTATE) to report, either by condition name or directly as a five-character SQLSTATE code. All messages emitted by the PostgreSQL server are assigned five-character error codes that follow the SQL standard's conventions for "SQLSTATE" codes. Applications that need to know which error condition has occurred should usually test the error code, rather than looking at the textual error message.
- The following example will abort the transaction with the given error message and hint:

```
RAISE EXCEPTION 'Nonexistent ID --> %', user_id USING HINT =
'Please check your user ID';
```

- These two examples show equivalent ways of setting the SQLSTATE:

```
RAISE 'Duplicate user ID: %', user_id USING ERRCODE = 'unique_violation';
RAISE 'Duplicate user ID: %', user_id USING ERRCODE = '23505';
```

- The following RAISE syntax in which the main argument is the condition name or SQLSTATE to be reported:

```
RAISE division_by_zero;
RAISE SQLSTATE '22012';
```

Trapping Errors:

- By default, any error occurring in a PL/pgSQL function aborts execution of the function, and indeed of the surrounding transaction as well.
- We can trap errors and recover from them by using a BEGIN block with an EXCEPTION clause.
- The **syntax** is an extension of the normal syntax for a BEGIN block:

```
[ <<label>> ]
[ DECLARE
    declarations ]
BEGIN
    statements
EXCEPTION
    WHEN condition [ OR condition ... ] THEN
        handler_statements
    [ WHEN condition [ OR condition ... ] THEN
        handler_statements
      ... ]
END;
```

- If no error occurs, this form of block simply executes all the statements, and then control passes to the next statement after END. But if an error occurs within the statements, further processing of the statements is abandoned, and control passes to the EXCEPTION list.
- The list is searched for the first condition matching the error that occurred. If a match is found, the corresponding `handler_statements` are executed, and then control passes to the next statement after END. If no match is found, it aborts processing of the function. The condition names can be like for example:

```
WHEN division_by_zero THEN ...
WHEN no_data_found THEN..
WHEN SQLSTATE '22012' THEN ...
```

- When an error is caught by an EXCEPTION clause, the local variables of the PL/pgSQL function remain as they were when the error occurred, but all changes to persistent database state within the block are rolled back.

1.7 CURSORS (S-18)

- A PL/pgSQL cursor allows us to encapsulate a query and process each individual row at a time. We use cursors when we want to divide a large result set into parts and process each part individually. If we process it at once, we may have a memory overflow error. In addition, we can develop a function that returns a reference to a cursor. This is an efficient way to return a large result set from a function. The caller of the function can process the result set based on the cursor reference.
- Cursors can be of two types: implicit cursors and explicit cursors.
  1. **Implicit cursors** are declared and managed by PL/pgSQL for all DML and PL/pgSQL SELECT statements.
  2. **Explicit cursors** are declared and managed by the programmer.

  We will see how to manage explicit cursor operations.
- Explicit cursor operations are as follows:

  Step 1: Declare a cursor.

  Step 2: Open the cursor.

  Step 3: Fetch rows from the result set into a target.

  Step 4: Check if there are more rows left to fetch. If yes, go to step 3, otherwise go to step 5.

  Step 5 : Close the cursor.

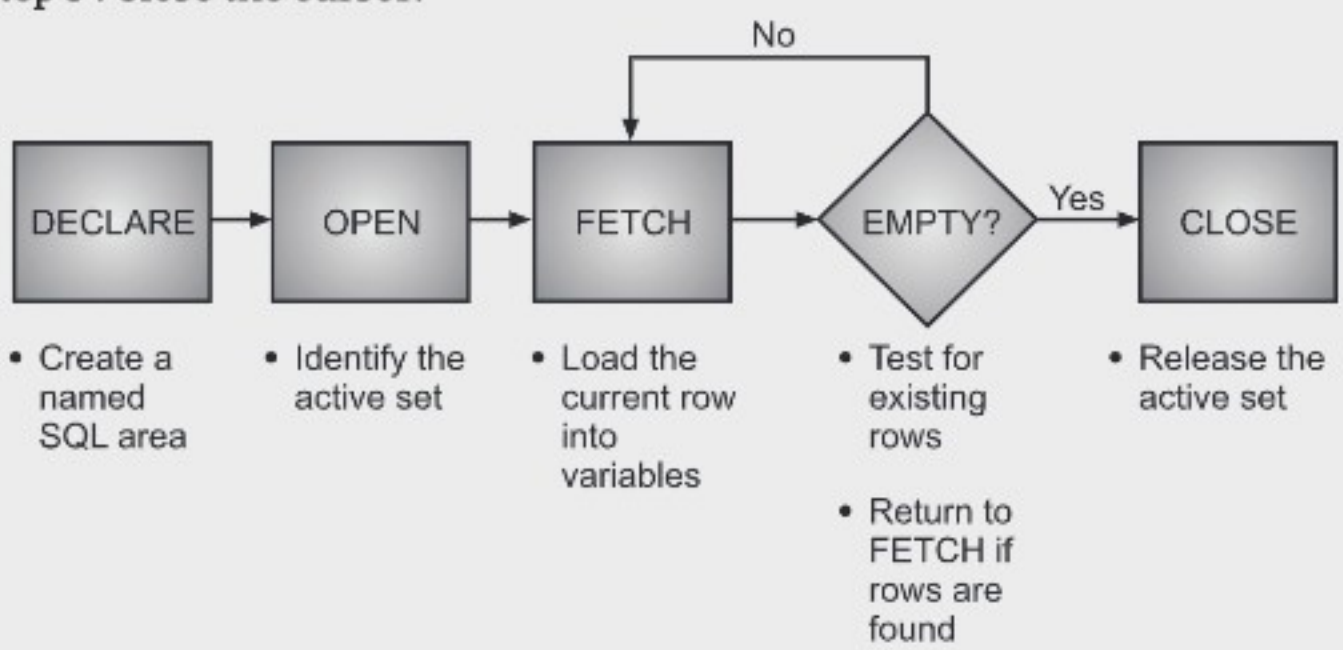


Fig. 1.2: Explicit Cursor Operations

- We will examine each step in more detail in the following sections.

1.7.1 Declaring Cursor Variables

- To access cursors in PL/pgSQL, we have to go through cursor variables. Cursor variables are always of the special data type REFCURSOR.

- There are two ways to create a cursor variable:
  1. First, we need to declare a cursor variable at the declaration section of a block. PostgreSQL provides us with a special type called REFCURSOR to declare a cursor variable.

     Syntax:

     ```
     DECLARE
     my_cursor REFCURSOR;
     ```

  2. Another way to declare a cursor that bounds to a query is using the following syntax:

     ```
     cursor_name [ [NO] SCROLL ] CURSOR [( name datatype, name data
                                                    type, ...)] FOR query;
     ```

     where, cursor_name specify a variable name for the cursor.
- Next, we specify whether the cursor can be scrolled backward using the SCROLL. If we use NO SCROLL, the cursor cannot be scrolled backward.
- Next the CURSOR keyword followed by a list of comma-separated arguments (name datatype) that defines parameters for the query. These arguments will be substituted by values when the cursor is opened.
- After that, we specify a query following the FOR keyword. We can use any valid SELECT statement here.

Example 36: Using declarations for CURSORs.

```
DECLARE
   curs1 refcursor;
   curs2 CURSOR FOR SELECT * FROM tenk1;
curs3 CURSOR (key integer) IS SELECT * FROM tenk1 WHERE unique1 = key;
```

- In above example, all three of these variables have the data type REFCURSOR.
  - The first declaration may be used with any query. The variable curs1 is said to be unbound since it is not bound to any particular query.
  - The second cursor is said to be bound cursor as specified query already bound to it.
  - The last has a parameterized query bound to it (key will be replaced by an integer parameter value when the cursor is opened.) So it is called Parameterized Cursor.

1.7.2 Opening Cursor (W-18)

- Cursors must be opened before they can be used to fetch query rows. PostgreSQL provides syntax for opening unbound and bound cursors.

1.7.2.1 Opening Unbound Cursor (W-18)

- To open an unbound cursor there are two forms.

1. The first form syntax is as follows:

```
OPEN unbound_cursor_variable [[ NO ]SCROLL ] FOR query;
```

- Because unbound cursor variable is not bounded to any query when we declared it, we have to specify the query when we open it.
- For example,

```
OPEN my_cursor FOR SELECT * FROM city WHERE counter = p_country
```

2. PostgreSQL allows us to open a cursor and bound it to a dynamic query.

   Here, is the **syntax:**

```
OPEN unbound_cursor_variable[ [ NO ] SCROLL ]
FOR EXECUTE query_string [USING expression [, ... ] ];
```

- In the following example, a dynamic query can be built that sorts rows based on a year parameter, and open the cursor that executes the dynamic query.

```
query := 'SELECT * FROM city ORDER BY $1';
OPEN cur_city FOR EXECUTE query USING year ;
```

1.7.2.2 Opening Bound Cursor

- A bound cursor already bounds to a query when we declared it, so when we open it, we just need to pass the arguments to the query if necessary.

```
OPEN cursor_variable[ (name:=value,name:=value,...)];
```

- In the following example, we open bound cursors `curs2` and `curs3` that we declared above:

```
OPEN curs2;
OPEN curs3(42);
```

1.7.3 Fetching Rows

- After opening a cursor, we can manipulate it using FETCH, MOVE, UPDATE, or DELETE statement. These manipulations need not occur in the same function that opened the cursor to begin with. We can return a REFCURSOR value out of a function and let the caller operate on the cursor.
- Internally, a REFCURSOR value is simply the string name of a so-called portal containing the active query for the cursor. This name can be passed around, assigned to other REFCURSOR variables, and so on, without disturbing the portal.
- All portals are implicitly closed at transaction end. Therefore, a REFCURSOR value is usable to reference an open cursor only until the end of the transaction.

1. **Fetching the Next Row:**

   Syntax:

```
FETCH [direction {FROM|IN}] cursor_variable INTO target_variable;
```

- The FETCH statement retrieves the next row from the cursor and assign it a `target_variable`, which could be a record, a row variable, or a comma-separated list of variables. If no more rows found, the `target_variable` is set to NULL.
- By default, a cursor gets the next row if you don't specify the direction explicitly. The following is the valid directions for the cursor:

```
NEXT
LAST
PRIOR
FIRST
ABSOLUTE count
RELATIVE count
FORWARD
BACKWARD
```

- Note that FORWARD and BACKWARD directions are only for cursors declared with SCROLL option.

  For example,

```
FETCH cur2 INTO row1;
FETCH LAST FROM cur3 INTO title, release_year;
```

2. **Moving the Cursor:**

```
MOVE [ direction { FROM | IN } ] cursor_variable;
```

- If we want to move the cursor only without retrieving any row, we can use the MOVE statement. The direction accepts the same value as the FETCH statement.

  For example:

```
MOVE cur2;
MOVE LAST FROM cur3;
MOVE RELATIVE -1 FROM cur2;
MOVE FORWARD 3 FROM cur3;
```

3. **Deleting or Updating Row:**

- Once a cursor is positioned, we can delete or update row identifying by the cursor using DELETE WHERE CURRENT OF or UPDATE WHERE CURRENT OF statement as follows.

  Syntax:

```
UPDATE table_name
SET column = value, ...
```

```
WHERE CURRENT OF cursor_variable;
DELETE FROM table_name
WHERE CURRENT OF cursor_variable;
```

For example,

```
UPDATE book SET release_year = p_year
WHERE CURRENT OF cur_book;
```

1.7.4 Closing Cursors

- To close an opening cursor, we use CLOSE statement. The **syntax** is as follows:

```
CLOSE cursor_variable;
```

- For example, `CLOSE cur2;`
- The CLOSE statement releases resources or frees up cursor variable.

Example 37: The following example shows steps of cursor.

```
CREATE OR REPLACE FUNCTION print_data() RETURNS integer AS '
DECLARE
-- declaring cursor
prod_cursor CURSOR FOR SELECT * FROM product;
rec product%ROWTYPE;
 BEGIN
-- opening cursor
 OPEN prod_cursor;
loop
--fetch the table row inside the loop
FETCH prod_cursor INTO rec;
--check if there is no record
--exit from loop when record not found
   if not found then
        exit ;
   end if;
raise notice ''%'',rec.pname;
end loop;
CLOSE prod_cursor;
RETURN 1;
END;
'LANGUAGE 'plpgsql';
postgres=# \i c1.sql
CREATE FUNCTION
```

Execution of Code:

```
postgres=# select print_data();
```

Output:

```
NOTICE:  pen
NOTICE:  box
 print_data
-----------
     1
(1 row)
```

1.7.5 Returning Cursors

- PL/pgSQL functions can return cursors to the caller. This is useful to return multiple rows or columns, especially with very large result sets.
- To do this, the function opens the cursor and returns the cursor name to the caller. The caller can then fetch rows from the cursor.
- The cursor can be closed by the caller, or it will be closed automatically when the transaction closes.

Example 38: Example shows one way a cursor name can be supplied by the caller.

```
CREATE TABLE testdata (col text);
INSERT INTO testdata VALUES ('123');
CREATE FUNCTION reffunc1(refcursor) RETURNS refcursor AS'
BEGIN
OPEN $1 FOR SELECT col FROM testdata;
RETURN $1;
END;
' LANGUAGE plpgsql;
BEGIN;
SELECT reffunc1('Cur1');
FETCH ALL IN Cur1;
COMMIT;
```

1.7.6 Looping through a Cursor's Result

- There is a variant of the FOR statement that allows iterating through the rows returned by a cursor.
- The **syntax** is:

```
[<<label>>]
FOR recordvar  IN bound_cursorvar [ (argument_values ) ] LOOP
statements
END LOOP [label ];
```

- The cursor variable must have been bound to some query when it was declared, and it cannot be open. The FOR statement automatically opens the cursor.

- When loop exits, it closes the cursor again. A list of actual argument value expressions must appear if and only if the cursor was declared to take arguments.
- These values will be substituted in the query, in just the same way as during an OPEN. The variable `recordvar` is automatically defined as type record and exists only inside the loop.
- Each row returned by the cursor is successively assigned to this record variable and the loop body is executed.

Example 39: Example for `CREATE OR REPLACE FUNCTION`

```
CREATE OR REPLACE FUNCTION print_data1() RETURNS integer AS '
DECLARE
prod_cursor CURSOR FOR SELECT * FROM product;
rec product%ROWTYPE;
 BEGIN
for rec in prod_cursor loop
     raise notice ''%'',rec.pname;
end loop;
RETURN 1;
END;
'LANGUAGE 'plpgsql';
postgres=# \i c2.sql
CREATE FUNCTION
```

Execution of Code:

```
postgres=# select print_data1();
```

Output:

```
NOTICE:  pen
NOTICE:  box
print_data1
-------------------
     1
(1 row)
```

1.8 TRIGGERS (S-18, 19; W-18)

- A trigger is a set of SQL statements stored in the database catalog.
- A trigger is a set of actions that are run automatically when a specified change operation (SQL INSERT, UPDATE, DELETE or TRUNCATE statement) is performed on a specified table or view.
- **Definition:** A trigger is, "a statement that is executed automatically by the system as a side effect of a modification to the database".

- To design a trigger mechanism, we must:
  1. Specify the conditions under which the trigger is to be executed.
  2. Specify the actions to be taken when the trigger executes.
- A trigger is a special user-defined function that binds to a table. To create a new trigger, we must define a trigger function first, and then bind this trigger function to a table.
- The difference between a trigger and a user-defined function is that a trigger is automatically invoked when an event occurs whereas a stored procedure must be called explicitly.
- Triggering events can be insert, delete or update.
- Trigger can be set to fire BEFORE an event occur or AFTER an event occur or even we can bypass the event by using the INSTEAD OF command.
- If we configured a trigger to fire BEFORE an INSERT or UPDATE, we will have the additional benefit of modifying the new data to be inserted or updated and even skipping the operation itself.
- There are two types of trigger i.e., Row Level Trigger and Statement Level trigger.
- A **row level trigger** is fired for each affected row.
- A **statement level trigger** is fired only once for a statement.
- For example, consider the statement:

```
UPDATE account_current SET balance = balance + 100 WHERE balance >
100000;
```

- Suppose executing this statement may affect 20 rows. If a row level trigger is defined for the table, the trigger will be fired for each 20 updated rows. But if it was a statement level trigger, it would have fired only once.

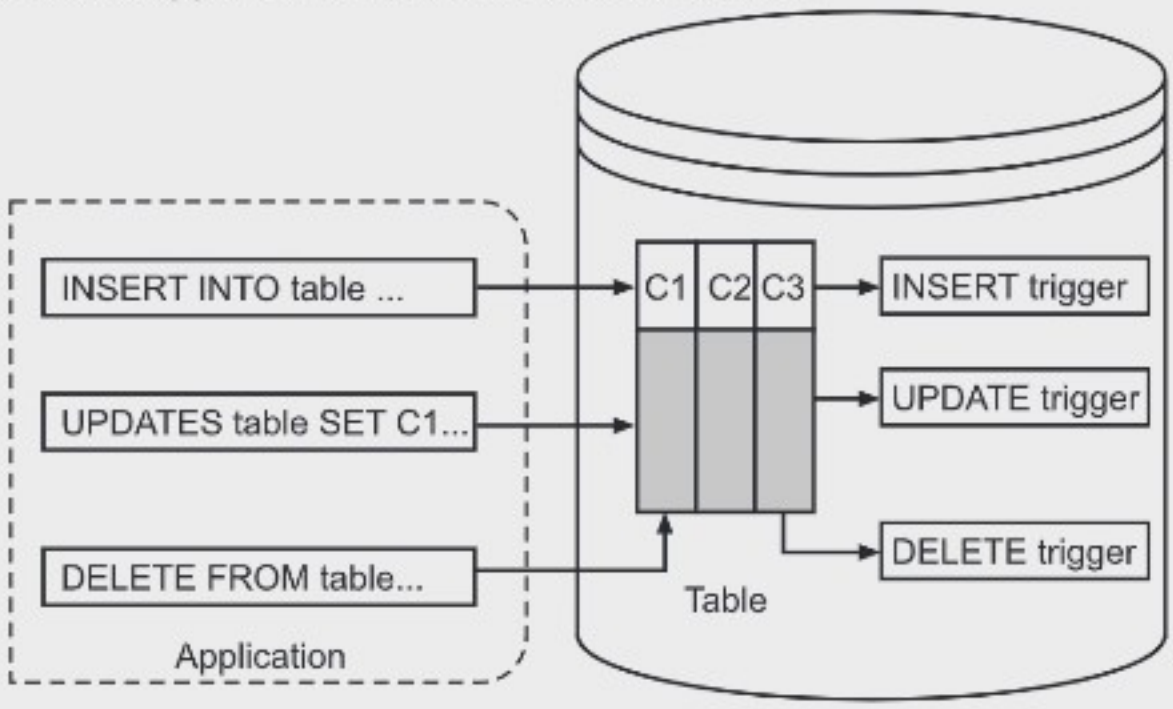


Fig. 1.3: Concept of Trigger

- It is important to understand trigger's advantages and disadvantages so that you can use it appropriately.

Advantages of Using Triggers:

1. Triggers provide an alternative way to check the integrity of data.
2. Triggers can catch errors in business logic in the database layer.
3. SQL triggers provide an alternative way to run scheduled tasks. By using SQL triggers, you don't have to wait to run the scheduled tasks because the triggers are invoked automatically before or after a change is made to the data in the tables.
4. Triggers are very useful to audit the changes of data in tables.

Disadvantages of Using SQL Triggers:

1. SQL triggers only can provide an extended validation and they cannot replace all the validations. Some simple validations have to be done in the application layer. For example, we can validate user's inputs in the client side by using JavaScript or in the server side using server-side scripting languages such as JSP, PHP, ASP.NET, Perl, etc.
2. SQL triggers are invoked and executed invisible from the client applications; therefore, it is difficult to figure out what happen in the database layer.
3. Triggers may increase the overhead of the database server.

1.8.1 Creating Trigger

- To create a new trigger in PostgreSQL, we need to:
  1. Create a trigger function using CREATE FUNCTION statement.
  2. Bind this trigger function to a table using CREATE TRIGGER statement.
- A trigger procedure is created with the CREATE FUNCTION command, declaring it as a function with no arguments and a return type of trigger.
- The function must be declared with no arguments even if it expects to receive arguments specified in CREATE TRIGGER, trigger arguments are passed via TG_ARGV, as described below.
- When a PL/pgSQL function is called as a trigger, several special variables are created automatically in the top-level block. They are:
  1. **NEW:** Data type RECORD; variable holding the new database row for INSERT/UPDATE operations in row-level triggers. This variable is NULL in statement-level triggers and for DELETE operations.
  2. **OLD:** Data type RECORD; variable holding the old database row for UPDATE/DELETE operations in row-level triggers. This variable is NULL in statement-level triggers and for INSERT operations.

3. **TG_NAME:** Data type name; variable that contains the name of the trigger actually fired.
4. **TG_WHEN:** Data type text; a string of BEFORE, AFTER, or INSTEAD OF, depending on the trigger's definition.
5. **TG_LEVEL:** Data type text; a string of either ROW or STATEMENT depending on the trigger's definition.
6. **TG_OP:** Data type text; a string of INSERT,UPDATE,DELETE, or TRUNCATE telling for which operation the trigger was fired.
7. **TG_RELID:** Data type oid; the object ID of the table that caused the trigger invocation.
8. **TG_TABLE_NAME:** Data type name; the name of the table that caused the trigger invocation.
9. **TG_TABLE_SCHEMA:** Data type name; the name of the schema of the table that caused the trigger invocation.
10. **TG_NARGS:** Data type integer; the number of arguments given to the trigger procedure in the CREATE TRIGGER statement.
11. **TG_ARGV[]:** Data type array of text; the arguments from the CREATE TRIGGER statement. The index counts from 0. Invalid indexes (less than 0 or greater than or equal to tg_nargs) result in a null value.

Creating the Trigger Function:

- A trigger function is similar to an ordinary function, except that it does not take any arguments and has return value type trigger as follows:

```
CREATE FUNCTION trigger_function() RETURN trigger AS'
```

Creating the Trigger:

- To create a new trigger, we use the CREATE TRIGGER statement.

```
CREATE TRIGGER trigger_name {BEFORE | AFTER | INSTEAD OF} {event [OR ...]}
   ON table_name
   [FOR [EACH] {ROW | STATEMENT}]
   EXECUTE PROCEDURE trigger_function
```

CREATE TRIGGER trigger_name: This clause creates a trigger with the given name.

- **{BEFORE | AFTER | INSTEAD OF }:** This clause indicates at what time should the trigger get fired. i.e. for example, before or after updating a table. INSTEAD OF is used to create a trigger on a view. before and after cannot be used to create a trigger on a view.
- **Event:** This clause determines the triggering event. Event can be {INSERT [OR] | UPDATE [OR] | DELETE}. More than one triggering events can be used together separated by OR keyword. The trigger gets fired at all the specified triggering event.

- **[ON table_name]:** This clause specifies table name on which event occurs.
- **[FOR EACH ROW]:** This clause is used to determine whether a trigger must fire when each row gets affected (i.e. a Row Level Trigger) or just once when the entire sql statement is executed (i.e. statement level Trigger).
- **EXECUTE PROCEDURE:** specifies trigger function.

Example 40: A Trigger to ensure that the quantity entered for a product is never < 0.

```
postgres=# select * from product;
```

pid	pname	qty	rate
1	pen	100	10
2	box	20	100

(2 rows)

```
create or replace function chk_product_data()  returns trigger as '
declare
begin
if  NEW.qty < 0  then
   raise exception ‘‘ QTY is less than zero ’’;
end if;
end;
'LANGUAGE  'plpgsql';
```

Execution of Code:

```
postgres=# \i f1.sql
CREATE FUNCTION
CREATE TRIGGER check_qty
 BEFORE INSERT ON product
 FOR EACH ROW
 EXECUTE PROCEDURE chk_product_data();
```

Execution of Code:

```
postgres=# \i t1.sql
CREATE TRIGGER
```

Output:

```
postgres=# insert into product values(3,'paper',-20,100);
ERROR:   QTY is less than zero
postgres=#
```

1.8.2 Listing Trigger

- We can list down all the triggers in the current database from pg_trigger table as follows:

```
testtown=# SELECT * FROM pg_trigger;
```

- The above given PostgreSQL statement will list down all triggers.

1.8.3 Dropping a Trigger

- To delete or destroy or drop a trigger, use a DROP TRIGGER statement.
- To execute this command, the current user must be the owner of the table for which the trigger is defined.

Syntax:

```
DROP TRIGGER [ IF EXISTS ] name ON table_name [ CASCADE | RESTRICT ]
```

Table 1.1: Trigger Parameters

Name	Description
IF EXISTS	Do not throw an error if the trigger does not exist. A notice is issued in this case.
Name	The name of the trigger to remove.
table_name	The name of the table for which the trigger is defined.
CASCADE	Automatically drop objects that depend on the trigger.
RESTRICT	Refuse to drop the trigger if any objects depend on it. This is the default.

- For example,

```
postgres=# DROP trigger chk_qty on product;
```

Summary

- PL/pgSQL (Procedural Language/PostgreSQL) is a loadable procedural language for the PostgreSQL database system.
- Currently procedural languages available in the standard PostgreSQL distribution are PL/pgSQL, PL/Tcl, PL/Perl and PL/Python.
- CREATE LANGUAGE is the SQL command which adds procedural languages to the currently connected database. Before it can be used, however, the CREATE FUNCTION command must first be used to create the procedural call handler.
- The *createlang* utility is simpler to use, as it abstracts the creation of the call handler and the language away from the user.

- PL/pgSQL is a block-structured and case-insensitive language.
- PL/pgSQL code is organized as block structured code where each portion of code is designed to exist as a function.
- A block of PL/pgSQL code can contain sub-blocks, which are code blocks nested within other code blocks.
- There are two methods of commenting in PL/pgSQL. First single line (- -) comments, which represent a single line comment, and block comments (/* .... */), which represent multiple line comments.
- A statement performs an action within PL/pgSQL code, such as assignment of a value to a variable or the execution of a query.
- Expressions are calculations or operations that return their results as one of PostgreSQL's base data types.
- Variables are used within PL/pgSQL code to store data of specified type.
- Functions can accept and return values called function parameters or arguments.
- The %TYPE attribute is used to declare a variable with the type of a referenced database object.
- The %ROWTYPE attribute is used to declare a PL/pgSQL row with the same structure as the row specified during declaration.
- Record variables are similar to ROW variables, but they have no predefined structure.
- Variables declared with %TYPE and %ROWTYPE can also renamed with the RENAME keyword.
- A conditional statement specifies an action or set of actions that should be executed based on the result of logical condition specified within the statement.
- In IF....THEN statement a statement or block of statements is executed if a given condition evaluates true.
- The IF...THEN....ELSE statement allows us to execute a block of statements if a condition evaluates to true, otherwise a block of statements in else part is executed.
- The simple form of CASE provides conditional execution based on equality of operands.
- Loops conditions are used to perform some task repeatedly for fixed number of times or until some is valid. PL/pgSQL provides three iterative loops i.e., the basic LOOP, WHILE loop and the FOR loop.
- The WHILE statement repeats a sequence of statements till the condition evaluates to true.
- Using a different type of FOR loop, we can iterate through the results of a query and manipulate that data accordingly.
- The FOR-IN-EXECUTE statement is another way to iterate over rows of record set.

- A view is a subset of a real table, selecting certain columns or certain rows from an ordinary table.
- The PostgreSQL views are created using the CREATE VIEW statement.
- To change the definition of a view, use the ALTER VIEW statement.
- PostgreSQL functions, also known as Stored Procedures, allow us to carry out operations that would normally take several queries and round trips in a single function within the database.
- RAISE statement is used to raise errors and exceptions during a PL/pgSQL function's operation.
- Cursors can be of two types: implicit cursors and explicit cursors.
- PostgreSQL provides us with a special type called REFCURSOR to declare a cursor variable.
- Cursors must be opened before they can be used to fetch query rows. PostgreSQL provides syntax for opening unbound and bound cursors.
- PL/pgSQL functions can return cursors to the caller. The FOR statement automatically opens the cursor.
- A trigger is a set of SQL statements stored in the database catalog.
- A trigger is a special user-defined function that binds to a table. To create a new trigger, we must define a trigger function first, and then bind this trigger function to a table.
- A trigger procedure is created with the CREATE FUNCTION command, declaring it as a function with no arguments and a return type of trigger.
- To create a new trigger, we use the CREATE TRIGGER statement.
- To delete or destroy or drop a trigger, use a DROP TRIGGER statement.

Check Your Understanding

1. A nested cursor is implicitly opened when the containing row is fetched from the parent cursor. The nested cursor is closed in which of the following case(s)?
   (a) The nested cursor is explicitly closed by the user
   (b) The parent cursor is re-executed
   (c) The parent cursor is closed
   (d) The parent cursor is cancelled
   (e) All mentioned above
2. Which statement associates a cursor variable with a multi-row query executes the query and identifies the result set?
   (a) OPEN-FOR (b) FETCH
   (c) CLOSE (d) All mentioned above

3. Which of the following executes the query and identifies the result set, consisting of all rows that meet the query search criteria?
   (a) Fetching with a Cursor
   (b) Opening a cursor
   (c) Fetching bulk data with a cursor
   (d) None of the above
4. Which of the following retains duplicate rows in the result of a query or in an aggregate expression?
   (a) ALL (b) DISTINCT
   (c) PRIOR (d) None of the above
5. From the following PL/SQL Delimiters which symbol is exponentiation operator?
   (a) <> (b) ~=
   (c) ** (d) –
6. PL/SQL is completely portable, high-performance transaction processing language that offers which of the following advantages ?
   (a) Tight Integration with SQL
   (b) Better Performance
   (c) Higher Productivity, Full Portability, Tight Security
   (d) All mentioned above
7. Which of the following is used to declare a record?
   (a) %ROWTYPE (b) %TYPE
   (c) Both A & B (d) None of the above

ANSWER KEY

1. (e)	2. (a)	3. (b)	4. (a)	5. (c)	6. (d)	7. (a)

Practice Questions

Q.I. Answer the following questions in short.

1. Explain features of PL/pgSQL.
2. Explain Advantages of PL/pgSQL.
3. Explain language structure of PL/pgSQL.
4. How comments are specified in PL/pgSQL?
5. What are different types of basic statements available in PL/pgSQL?
6. What are different types of loops available in PL/pgSQL?
7. What are different types of attributes in PL/pgSQL?
8. What are different levels of RAISE statement?

Q.II. Answer the following questions.

1. Explain conditional statements in PL/pgSQL with example.
2. Explain concept of view in detail with example.
3. How errors and exception handled in PL/pgSQL?
4. How to create and call functions in PL/pgSQL?
5. Explain concept of cursor in detail with example.
6. Explain explicit cursor operations.
7. Explain concept of trigger in detail with example.
8. State the events on which trigger can be fired.
9. What is row-level and statement-level trigger?
10. What are advantages and disadvantages of using triggers?
11. Explain special variables created when trigger is called automatically.
12. Consider the following Student –Marks database.

- Student (rollno integer, name varchar(30), address varchar(50), class varchar(10))
- Subject (scode varchar(10), subject_name varchar(20))
- Student-Subject are related with M-M relationship with attributes marks_scored.
- Create the above database in PostgreSQL.

Query:

Execute the following queries in PostgreSQL:

(i) Display the names of students scoring the maximum total marks.

(ii) List the distinct names of all the subjects.

(iii) Display classwise and subjectwise student list.

Cursors and Triggers:

(i) Write a stored function using cursors, to accept a address from the user and display the name, subject and the marks of the students staying at that address.

(ii) Write a stored function using cursors which will calculate total and percentage of each student.

(iii) Write a trigger before deleting a student record from the student table. Raise a notice and display the message "student record is being deleted".

(iv) Write a trigger to ensure that the marks entered for a student, with respect to a subject is never < 10 and greater than 100.

Stored Function:

Write a stored function using cursors to accept an address from the user and display the name, subject and the marks of the student staying at that address.

Create a View:

(i) To list student name, class & total marks scored by each student, sorted by student name.

(ii) To list student names along with subject name and marks who scored more than 40 marks.

(iii) Containing all the details of student named 'Omkar'.

Q.III. Define following terms:

1. implicit cursor,
2. explicit cursor,
3. bound cursor,
4. unbound cursor.
5. Record,
6. rename,
7. %type attribute,
8. assignment statement,
9. For loop,
10. function.

Previous Exams Questions

Summer 2019

1. The.....is used to declare a PL/PG-SQL row with the same structure as the row specified during declaration. **(1M)**

(a) % type (b) % Rowtype

(c) Declare (d) @ Type

Ans.: Refer to Section 1.2.4.2

2. Record is a..... **(1M)**

(a) Placeholder (b) Variable

(c) Data type (d) Keyword

Ans.: Refer to Section 1.2.4.3

3. Give the syntax of creating view. **(1M)**

Ans.: Refer to Section 1.4.1

4. What is stored function? Explain with an example. **[5M]**

Ans.: Refer to Section 1.5.

5. Consider the following tables : [4] Student (rno, name, address, class) Subject (scode, subj-name). Student_Subject related with M_M Relationship with descriptive attribute marks scored.

Ans.: Refer to 'Assignments' Chapter.

6. Define a trigger to ensure that the marks entered for a student, with respect to a subject is never less than 10 and greater than 100. **(5M)**

Ans.: Refer to Section 1.8

7. Consider the following Project-Employee database maintained by a company which stores the details of the projects assigned to the employees.

Project (pno, pname, ptype, duration)

Employee (eno, ename, qualification, joindate)

Relationship : M_M with descriptive attributes as start_date, no_of_hours_worked.

Write a stored function using cursor to accept project name as input and print the names of employees working on the project. **[4M]**

Ans.: Refer to 'Assignments' chapter

8. Define view. Explain how to create view with example. **[3M]**

Ans.: Refer to Section 1.4.1

9. Explain different data types in PL/PG-SQL. **[3M]**

Ans.: Refer to Section 1.2.4

10. Consider the following database and relationships : Bus (bno, capacity, depot_name) Route (rno, source, destination, no._of_stations) Driver (dno, dname, license no., address, dage, salary) Relationship : Bus_Route : M_1 Bus_Driver : M_M.

Write a stored function using cursor to display the details of buses that run on Route no. = 1 and Route no. = 2. (Use two different cursors for route_no. = 1 and route_no. = 2).

Ans.: Refer to 'Assignments' Chapter.

11. Explain the syntax of while loop. **[3M]**

Ans.: Refer to Section 1.3.2.2

Winter 2018

1. The.....is an unconditional loop that will execute until an exit statement is met. **(1M)**

(a) While loop (b) For loop

(c) Basic loop (d) If statement

Ans.: Refer to Section 1.3

2. The.....attribute is used to declare a variable to have the same structure as a row in the table. **(1M)**

(a) % record type (b) % tuple type

(c) % type (d) % row type

Ans.: Refer to Section 1.2.4.2

3. Give the syntax for creating a bounded and unbounded cursor **(1M)**

Ans.: Refer to Section 1.7.2

4. Give any four datatypes used in PL/pgsql. **(1M)**

Ans.: Refer to Section 1.2.4

5. Consider the relation : Employee (eno, ename, designation, salary, dept_no)

Write a stored function with cursor, which accepts dept_no as input and prints the names of all employees working in that department. **[5M]**

Ans.: Refer to 'Assignments' chapter.

6. Explain with examples how triggers can be used to enforce referential integrity. **[4M]**

Ans.: Refer to Section 1.8

7. What is a view ? Explain with example how views can be used. **[3M]**

Ans.: Refer to Section 1.4

8. What are the different levels used with the RAISE statement ? Give syntax of each.

Ans.: Refer to Section 1.6 **[3M]**

Summer 2018

1. ..... attribute is used to declare a variable with datatype of referenced data base object. **(1M)**

(a) $ type (b) * type
(c) @ type (d) % type

Ans.: (d)

2. List any two features of PL/PgSQL. **(1M)**

Ans.: Refer to Sections 1.1.1.

3. What is stored procedure? Explain with example. **[5M]**

Ans.: Refer to Section 1.5.

4. What are various datatypes available in PL/PGSQL. **[5M]**

Ans.: Refer to Section 1.2.4.

5. Consider the following entities and the relationship. **[4M]**

Employee (eno, ename, sex, joining date, designation, salary, dno),
Dept (dno, dname).

Write a PL/SQL block to list the names of all employees. Who are Male and are earning the maximum salary in their department?

Ans.: Refer to 'Assignments' chapter.

6. What are types of triggers? **[3M]**

Ans.: Refer to Section 1.8.

7. What is a cursor? How to declare it? Explain with example. **[4M]**

Ans.: Refer to Section 1.7.

8. How triggers are declared and implemented. Explain with example. **[3M]**

Ans.: Refer to Section 1.8.

❖❖❖

2... Transaction Concepts

Objectives...

- To understand Transaction Concepts
- To learn Schedule and their Types
- To get information about serializability and precedence graph

2.1 INTRODUCTION (S-19)

- A transaction can be considered as a unit of program execution that accesses and updates various data items of a database.
- Collection of operations that forms a single logical unit of work is called transaction.
- A transaction accesses and possibly updates various data items. After every transaction, the database should be in a consistent state.
- Usually a transaction is the result of execution of a user program, written in a high-level data manipulation language or programming language.
- Every transaction is delimited by statements or function calls of the form begin transaction and end transaction.
- A computer system is an electronic device and it is subject to failures of various types. During the execution of a transaction if some failure occurs, the transaction may result in some inconsistent state of database. Hence, the reliability of DBMS is linked to the reliability of computer system, and some solution must be there to deal with such computer system failures.
- Recovery system is the main component of transaction management/processing deals with different failures. It makes the database fault tolerant.
- Number of transactions can be executed at the same time and they may be accessing the same database. Such concurrent access to the database may result in some inconsistent state of database.

- This chapter deals with the topics like basic concepts of transaction and serializability.

2.1.1 Transaction Concepts [S-18]

- A transaction is a program unit whose execution accesses and possibly updates the contents of a database.
- A transaction can be defined as, "a logical unit of database processing that includes one or more database access operations".
- If the database was in consistent state before a transaction then on execution of transaction, the database will be in a consistent state.
- In day-to-day life, a transaction can be defined as, "an act of giving something to a person and receiving something from that person in return".
- For example, if a customer wants to buy a commodity from a shopkeeper, the customer first pays an amount for that commodity and then receives the commodity from the shopkeeper in return. Here a customer cannot receive a commodity without paying amount. Similarly, after receiving an amount, the shopkeeper has to handover the commodity to customer. It means that omission of any step will result into invalid transaction.
- Clearly, a transaction is an atomic (not splittable) unit. Either all instructions from transaction should be completed or none of the instruction should be completed.

2.1.2 Properties of Transaction [S-18, 19; W-18]

- To ensure accuracy, completeness and the integrity of data, database system maintains following properties of transaction.
  1. **Atomicity:** Atomicity property ensures that at the end of the transaction, either no changes have occurred to the database or the database has been changed in a consistent manner. At the end of a transaction, the updates made by the transaction will be accessible to other transactions and processes outside the transaction.
  2. **Consistency:** Consistency property of transaction implies that if the database was in consistent state before the start of a transaction, then on termination of a transaction, it will also be in a consistent state. In other words, Data is in a consistent state when a transaction starts and when it ends.
  3. **Isolation:** Isolation property of transaction indicates that action performed by a transaction will be hidden from outside the transaction until the transaction terminates. Thus each transaction is unaware of other transactions executing concurrently in the system.
  4. **Durability:** Durability property of a transaction ensures that once a transaction completes successfully (commits), the changes it has made to the database persist, even if there are system failures.

- These four properties are often called ACID (Atomicity, Consistency, Isolation, and Durability) properties of transaction.

Significance of ACID Properties:

- Consider a banking system consisting of several accounts and a set of transactions that accesses and updates those accounts. Here consider that the database resides on disk, but some portion of database is temporarily stored in main memory.
- Following are the functions to access the database:
  1. **read (X):** Which transfers the data item X from the database to a local buffer, belonging to the transaction that executed the read operation.
  2. **write (X):** Which transfers the data item X from the local buffer of the transaction that executed the write back to the database.
- Let T_i be a transaction that transfers ₹ 50 from account A to account B. This transaction can be defined as:

  T_i : Read (A);
  A := A – 50;
  write (A);
  Read (B);
  B:= B + 50;
  write (B).

  Let us now consider the significance of ACID properties.
- **Atomicity:**

  This property ensures that either all the operations of a transaction reflect in database or none.

  Example: Suppose in a banking system Account A has a balance of ₹ 400 and B has ₹ 700. Account A is transferring ₹ 100 to Account B. This is a transaction that has two operations i.e., Withdrawal of ₹ 100 from A's balance and Depositing ₹ 100 to B's balance. Let's say first operation passed successfully while second failed, in this case A's balance would be ₹ 300 while B would be having ₹ 700 instead of ₹ 800. This is unacceptable in a banking system. Either the transaction should fail without executing any of the operation or it should process both the operations. Thus the Atomicity property ensures that a transaction is said to be atomic if a transaction always executes all its actions in one step or not executes any actions at all.
- **Consistency:**

  To preserve the consistency of database, the execution of transaction should take place in isolation that means no other transaction should run concurrently when there is a transaction already running.

  For example, account A is having a balance of ₹ 400 and it is transferring ₹ 100 to account B and C both. So we have two transactions here. Let's say these transactions

run concurrently and both the transactions read ₹ 400 balance, in that case the final balance of A would be ₹ 300 instead of ₹ 200. This is wrong. If the transaction were to run in isolation then the second transaction would have read the correct balance ₹ 300 (before debiting ₹ 100) once the first transaction went successful.

- **Isolation:**

  Even if the consistency and atomicity properties are ensured for each transaction and if several transactions are executed concurrently, their operations may interleave in some undesirable way. The isolation property ensures that for every pair of transactions, one transaction should start execution only when the other finished execution.

- **Durability:**

  The durability assures that once transaction completes successfully, all updates that it carried out on the database persist even if there is a system failure. We can guarantee durability by ensuring that either:

  1. The updates carried out by a transaction have been written to disk before the transaction completes.
  2. Information about updates carried out by the transaction and written to disk is sufficient to enable the database to reconstruct the updates when the database system is restarted after failure.

2.1.3 States of Transaction [W-18; S-19]

- A transaction goes through many different states throughout is life cycle. These states are called **Transaction States**.
- Fig. 2.1 shows a state transition diagram of a transaction. It illustrates how a transaction moves through its execution states.

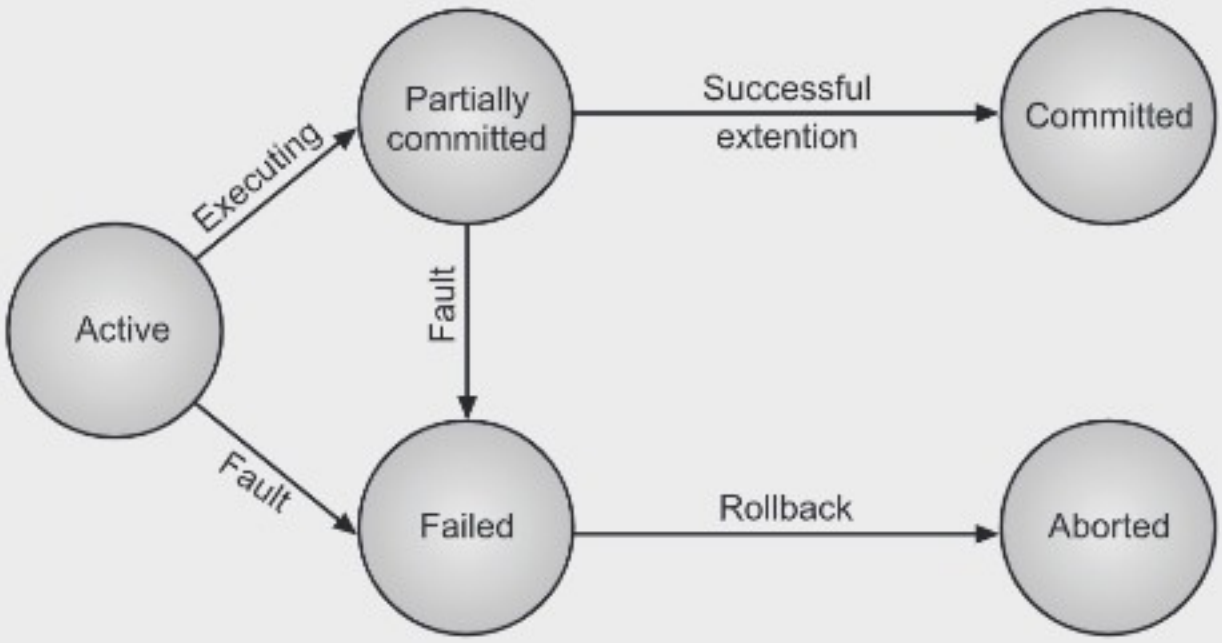


Fig. 2.1: State Transition diagram of a Transcation

- Fig. 2.1 shows following states of a transaction:
  1. **Active State (Initial State):** This is the initial state of transaction. A transaction is active when it is executing. A transaction always starts with active state. It remains in active state till all commands of that transaction are executed.
  2. **Partially Committed:** When a transaction completes its last statement (command), it enters in partially committed state.
  3. **Failed:** If the system decides that the normal execution of the transaction can no longer proceed, then transaction is termed as failed. If some failure occurs in active state or partially committed state, transaction enters in failed state.
  4. **Committed:** When the transaction completes its execution successfully it enters committed state from partially committed state.
  5. **Aborted:** To ensure the atomicity property, changes made by failed transaction are undone i.e. the transaction is rolled back. After rollback, that transaction enters in aborted state. When the transaction is in failed state, it rollbacks that transaction and enters in aborted state.
- When the transaction is in aborted state, the system has two options:
  1. If the transaction was aborted as a result of some hardware or software error (software error which is not created because of some internal logic of transaction), then such transaction can be restarted and that transaction is considered to be a new transaction.
  2. If the transaction was aborted because of some internal logical error (which can be corrected only by rewriting of the application program) or because of the input was bad or the desired data were not found in the database, then system can kill such transactions.

2.2 CONCURRENT EXECUTION OF TRANSACTIONS AND CONFLICTING OPERATIONS

- In concurrent execution, the DBMS controls the execution of two or more transactions in parallel. However, it allows only one operation of any transaction to occur at any given time within the system.
- The DBMS interleaves the actions of different transactions to improve performance, in terms of increased throughput or improved response times for short transactions, but not all interleaving should be allowed.
- Ensuring transaction isolation while permitting such concurrent execution is difficult but is necessary for performance reasons.
  1. While one transaction is waiting for a page to be read in from disk, the CPU can process another transaction. This is because I/O activity can be done in parallel with CPU activity in a computer. Overlapping I/O and CPU activity reduces the

amount of time disks and processors are idle and increases system throughput (the average number of transactions completed in a given time).

2. Interleaved execution of a short transaction with a long transaction usually allows the short transaction to complete quickly. In serial execution, a short transaction could get stuck behind along transaction leading to unpredictable delays in response time, or average time taken to complete a transaction.

T_1	T_2
R (A)	
W (A)	
	R (A)
	W (B)
R (C)	
W (C)	

Conflicting Operations:

- When two or more transactions in a non-serial schedule execute concurrently, then there may be some conflicting operations.
- Two operations are said to be conflicting, if they satisfy all of the following conditions:
  1. The operations belong to different transactions.
  2. At least one of the operations is a write operation.
  3. The operations access the same object or item.

Examples:

1. The following set of operations is conflicting:

T_1	T_2	T_3
R(X)		
	W(X)	
		W(X)

- In this schedule, operations on same object X and there are two write operations.

2. While the following sets of operations are not conflicting:

T_1	T_2	T_3
R(X)		
	R(X)	
		R(X)

(a) No Write operation on same Object

T_1	T_2	T_3
R(X)		
	W(Y)	
		R(X)

(b) Write operation on different Object

Fig. 2.2: Conflicting and Non-conflicting operations

2.3 SCHEDULE (W-18, S-19)

- Schedule represents the chronological order in which instructions are executed in the system.
- A schedule can have many transactions in it, each comprising of a number of instructions/tasks.
- A schedule can be defined as, "a sequence of operations by a set of concurrent transactions that preserves the order of the operations in each of the individual transactions".
- For example: Consider two transactions T_1 and T_2 are executing concurrently as:

T_1	T_2
Read(X)	Read (Y)
X=X+5	Y=Y+100
Write(X)	Read(X)
Commit;	X=X+Y
	Write(X)
	Commit;

- Then a schedule S of { T_1,T_2} is given as:

T_1	T_2
Read(X)	Read(X)
X=X+5	X=X+Y
	Read (Y)
	Y=Y+100
Write(X)	
Commit;	
	Read(X)
	X=X+Y
	Write(X)
	Commit;

Complete schedule:

- A schedule that contains either a commit or an abort action for each transaction.

Example:

Complete Schedule	
T_1	T_2
R(A)	
	R(B)
W(A)	
	W(B)
Commit	
	Abort

Complete Schedule	
T_1	T_2
R(A)	
W(A)	
	R(B)
Commit	
	W(B)
	Abort

Complete Schedule	
T_1	T_2
R(A)	
W(A)	
Commit	
	R(B)
	W(B)
	Abort

Fig. 2.3: Example of Complete Schedule

2.3.1 Types of Schedules [S-18; W-18]

- Following chart shows various types of schedules in DBMS.

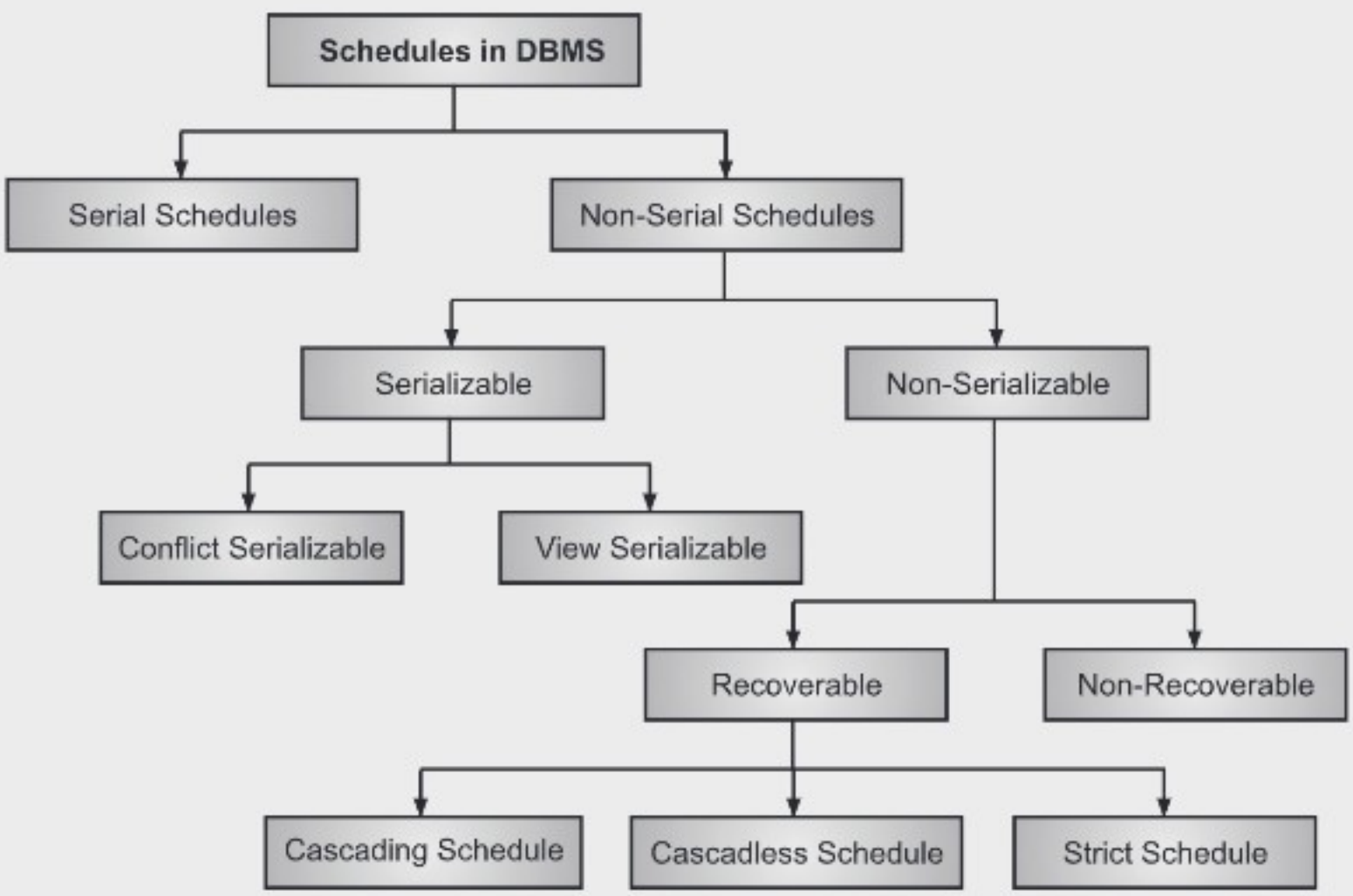


Fig. 2.4: Types of schedules in RDBMS

- Let us see details of each type of schedule.

2.3.1.1 Serial Schedule

- Serial Schedule consists of a sequence of instructions from various transactions where the instructions belonging to one single transaction appear together in that schedule.
- **Definition:** "If the actions (operations) of different transactions are not interleaved i.e., transactions are executed one-by-one from start to finish, the schedule is called as a Serial Schedule".
- **For example:** Consider the banking system of several accounts and a set of transactions that accesses and updates those accounts. Let T_1 and T_2 be two transactions. Assume initial balances of A and B are 1000 and 2000 respectively.

T_1: Transfers ₹ 50 from account A to account B.

T_1 : read (A);

A = A – 50;

write (A);

read (B);

B = B + 50;

write (B).

T_2: Transfers 10% of balance from account A to B.

T_2: read (A);

temp: = A . 0.1;

A:= A – temp;

write (A);

read (B);

B:= B + temp;

write (B).

For T_1 and T_2, two serial schedules are possible:

(i) **<T_1, T_2> Serial Schedule:** It will execute transaction T_1 first and then T_2.

After executing both transactions, account balance of A is ₹ 855 and account balance of B is ₹ 2145 i.e., <T_1 T_2> preserves A + B i.e. the database are consistent.

Table 2.1: Schedule 1

T_1	T_2
read (A) A:= A – 50 write (A) read (B) B:= B + 50 Write (B)	
	read (A) temp:= A * 0.1 A:= A – temp write (A) read (B) B:= B + temp write (B)

(ii) <T_2, T_1> Serial Schedule: It executes T_2 first and then T_1.

Table 2.2: Schedule 2

T_1	T_2
	read (A) temp:= A * 0.1 A:= A – temp write (A) read (B) B:= B + temp write (B)
read (A) A:= A – 50 write (A) read (B) B:= B + 50 write (B)	

- Here, after executing both the transactions, values of A and B are ₹ 850 and ₹ 2150 respectively. In this case also A + B is constant and it preserves the consistency. Thus, for a set of n transactions, there exist n! different valid serial schedules.

2.3.1.2 Concurrent Schedule (Non-Serial Schedule)

- When several transactions are executed concurrently, the corresponding schedule is called Concurrent Schedule.
- Several execution sequences are possible since the various instructions from both transactions may now be interleaved.

- In general, it is not possible to predict exactly how many instructions of a transaction will be executed before CPU switches to other transaction. Thus, the number of possible concurrent schedules for a set of n transactions is much larger than n! .
- **For example:** A concurrent schedule for T_1 and T_2 is given in Table 2.3.

  (i) Consistent Concurrent Schedule for T_1 and T_2: This concurrent schedule preserves the consistency of database and A + B is constant.

Table 2.3: Schedule 3

T_1	T_2
read (A)	
A:= A – 50	
write (A)	
	read (A)
	temp:= A * 0.1
	A:= A – temp
	write (A)
read (B)	
B:= B + 50	
write (B)	
	read (B)
	B:= B + temp
	write (B)

(ii) Inconsistent concurrent schedule for T_1 and T_2:

Table 2.4: Schedule 4 (concurrent schedule)

T_1	T_2
read (A)	
A:= A – 50	
	read (A)
	temp:= A * 0.1
	A:= A – temp
	write (A)
	write (B)
write (A)	
read (B)	
B:= B + 50	
write (B)	
	B:= B + temp
	write (B)

2.3.1.2.1 Serializable Schedule (W-18)

- For a transaction, a serial schedule always results in a consistent database and not every concurrent schedule can result in consistent database.
- But a concurrent schedule results in a consistent state if its result is equivalent to a serial schedule of that transaction. Such concurrent schedule is known as **Serializable**.
- A Serializable Schedule is defined as,

  "given (an interleaved execution) a concurrent schedule for n transactions; the following conditions hold for each transaction in the set:

  1. All transactions are correct i.e. if any one of the transactions is executed on a consistent database, the resulting database is also consistent.
  2. Any serial execution of the transactions is also correct and preserves the consistency of the database."
- Serialization refers to the transactions running serially, one after the other rather than concurrently.

Types of Serializability:

- There are two types of serializable schedule:
  1. Conflict serializable schedule
  2. View serializable schedule

1. Conflict Serializability:

- Consider that T_1 and T_2 are two transactions and S is a schedule for T_1 and T_2. I_i and I_j are two instructions. If I_i and I_j refer to different data items, then I_i and I_j can be executed in any sequence.
- But, if I_i and I_j refer to same data items then the order of two instructions may matter. Here, I_i and I_j can be a read or write operation only. Hence, following 3 conditions are possible.

  (i) I_i = read (A)

  I_j = read (A)

  The order of I_i and I_j does not matter because both are reading the data.

  (ii) I_i = read (A) I_j = write (A)

  I_i = write (A) I_j = read (A)

  Here, if read (A) is executed before write (A) then it will read the original value of A otherwise it will read that value of A which is written by write (A). Hence, the order of I_i and I_j matters.

  (iii) I_i = write (A) I_j = write (A)

  Here, order of I_i and I_j does not affect either T_i or T_j. But the database is changed, and it makes difference for next read.

- We say that I_i and I_j conflict if they are operated by different transactions on the same data item and at least one of them is write operation. i.e. only in case 1, I_i and I_j do not conflict.
- Consider an example of concurrent schedule 5.

Table 2.5: Concurrent Schedule 5

T_1	T_2
read (A)	
write (A)	
	read (A)
	write (A)
read (B)	
write (B)	
	read (B)
	write (B)

- Here, write (A) of T_1 conflicts with read (A) of T_2, similarly write (B) of T_1 conflicts with read (B) of T_2. But write (A) of T_2 does not conflict with read (B) of T_1 because both are accessing different data items.
- If I_i and I_j are two consecutive instructions of schedule S and if they do not conflict, then we can swap the order of I_i and I_j, to produce new schedule S'. We say that S and S' are equivalent since all instructions appear in the same order except for I_i and I_j whose order does not matter.
- Equivalent schedule for a schedule given in Table 2.5 can be obtained by following swap.
- Swap write (A) of T_2 with read (B) of T_1.

Table 2.6: Schedule 6 (Schedule after swapping instructions)

T_1	T_2
Read (A)	
write (A)	
	read (A)
read (B)	
	write (A)
write (B)	
	read (B)
	write (B)

- Similarly swap the following in schedule given in Table 2.6.
  (i) read (B) instruction and read (A) instruction of T_1 and T_2 respectively.
  (ii) write (B) instruction and write (A) instruction of T_1 and T_2 respectively.
  (iii) write (B) instruction and read (A) instruction of T_1 and T_2 respectively.
- The final schedule S' after these swapping is given below:

Table 2.7: Schedule 7

T_1	T_2
read (A)	
write (A)	
read (B)	
write (B)	
	read (A)
	write (A)
	read (B)
	write (B)

- This is a serial schedule of T_1 and T_2. Thus concurrent schedule S is transferred to serial schedule S' by a series of swaps of non-conflicting instructions and schedules S and S' are conflict equivalent.
- We say that a schedule S is conflict serializable, if it is conflict equivalent to a serial schedule. Consider the schedule shown in Table 2.8.

Table 2.8: Schedule 8

T_1	T_2
read (Q)	
	write (Q)
read (Q)	

- This schedule is not conflict serializable, since it is not conflict equivalent to any serial schedule $<T_1, T_2>$ or $<T_2, T_1>$.
- There may be any two schedules which are not conflict equivalent but produce same outcome. Schedule given in Table 2.9 is not conflict serializable.

Table 2.9: Schedule 9 (Concurrent schedule)

T_1	T_5
read (A)	
A:= A – 50	
write (A)	
	read (B)
	B:= B – 10
	write (B)
read (B)	
B:= B + 50	
write (B)	
	read (A)
	A:= A + 10
	write (A)

- Result of above schedule is same as serial schedule <T_1, T_5>, but this is not conflict serializable, since, in above schedule write (B) of T_5 conflicts with read (B) of T_1. Thus, we cannot move all instructions of T_1 before those of T_5 by swapping consecutive non-conflicting instructions.

2. View Serializability:

- Consider two schedules S and S', where same set of transactions participate in both schedules.
- The schedules S and S' are said to be view equivalent if the following three conditions are satisfied:
  (i) For each data item Q if transaction T_i reads the initial value of Q in schedule S, then transaction T_i in schedule S' must also read the initial value of Q.
  (ii) For each data item Q if transaction T_i executes read (Q) in schedule S, and that value was produced by transaction T_j (if any), then transaction T_i in schedule S', must also read the value of Q that was produced by T_j transaction.
  (iii) For each data item Q, the transaction that performs the final write (Q) operation in schedule S must perform the final write (Q) operation in schedule S'.
- A schedule S is view serializable if it is view equivalent to a serial schedule.
- Example of view equivalence: Schedule 1 is not view equivalent to schedule 2 since, in schedule 1 the value of account A read by transaction T_2 was produced by T_i, whereas this is not the case in schedule 2. Schedule 1 is view equivalent to schedule 3, because values of account A and B read by transaction T_2 were produced by T_1 in both schedules.

- Example of view serializable schedule, A schedule given in Table 2.10 is view serializable schedule.

Table 2.10: Schedule 10

T_3	T_4	T_6
read (A)		
	write (A)	
write (A)		
		write (A)

This schedule is view equivalent to serial schedule $<T_3, T_4, T_6>$.

Note: Transactions T_4 and T_6 perform write (A) operations without having performed a read (A) operation. Writes of this form are called blind writes.

- Every conflict serializable schedule is view serializable, but there are view serializable schedules that are not conflict serializable.
- A view serializable schedule in which blind write appear is not a conflict serializable. Schedule 10 in Table 2.10 is view serializable but it is not conflict serializable.

Testing for Serializability:

- A serializability schedule gives same result as some serial schedule.
- A serial schedule always gives correct result. i.e. a schedule that is serializable schedule is always correct.
- Hence, we must show that schedules generated by concurrency control scheme are serializable. This section deals with methods for determining conflict and view serializability.

1. **Testing of Conflict Serializability:**

- There is an algorithm to establish the serializability of a given schedule for a set of transactions.
- This algorithm uses a directed graph called precedence graph, constructed from given schedule.

Precedence Graph:

- A precedence graph, also named conflict graph and serializability graph, is used in the context of concurrency control in databases.
- It is a directed Graph consists of a pair G = (V, E) where V : set of vertices and E : the set of edges.
- The set of vertices consists of all transactions participating in the schedule.

- The set of edges consists of all edges $T_i \rightarrow T_j$ for which one of the following three conditions hold:
  (i) T_i executes write (Q) before: T_j executes read (Q)
  (ii) T_i executes read (Q) before: T_j executes write (Q)
  (iii) T_i executes write (Q) before: T_j executes write (Q)
- A precedence graph is said to be acyclic if there are no cycles in the graph otherwise it is a cyclic graph.

Algorithm: Conflict serializability:

Step 1: Construct a precedence graph G for given schedule S.

Step 2: If the graph G has a cycle, schedule S is not conflict serializable.

Topological Sorting:

- A topological sort or topological ordering of a directed graph is a linear ordering of its vertices such that for every directed edge *uv* from vertex *u* to vertex *v*, *u* comes before *v* in the ordering.
- If the graph is acyclic, then find a serial schedule using the topological sort as follows:
  (i) Initialize the serial schedule as empty.
  (ii) Find a transaction T_i, such that there are no arcs entering T_i, T_i is the next transaction in the serial schedule.
  (iii) Remove T_i and all edges emitting from T_i. If the remaining set is non-empty, return to (ii), otherwise the serial schedule is complete.

Example:

(i) Given schedule is:

Table 2.11: Schedule 11

T_{11}	T_{12}	T_{13}
Read (A)		
	Read (B)	
A:= f_1 (A)		
		Read (C)
	B:= f_2 (B)	
	write (B)	
		C:= f_3 (C)
		write (C)
write (A)		
	Read (A)	
	A:= f_4 (A)	
Read (C)		
	Write (A)	
C:= f_5 (C)		
Write (A)		
		B:= f_6 (B)
		write (B)

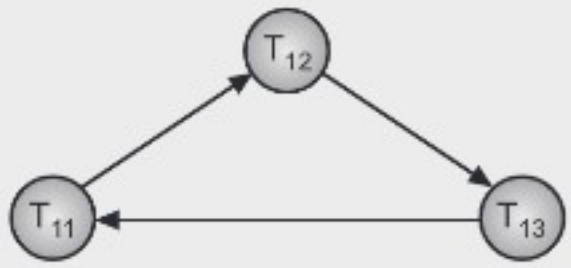


Fig. 2.5 (a): Precedence graph for the schedule

- This graph contains cycle. Hence, the schedule is not conflict serializable.

(ii) The given schedule is:

Table 2.12: Schedule 12

T_{14}	T_{15}	T_{16}
Read (A)		
A:= f_1 (A)		
Read (C)		
write (A)		
A:= f_2 (C)		
	Read (B)	
write (C)		
	Read (A)	
		Read (C)
	B:= f_3 (B)	
	write (B)	
		C:= f_4 (C)
		Read (B)
		write (C)
	A:= f_5 (A)	
	write (A)	
		B:= f_6 (B)
		write (B)

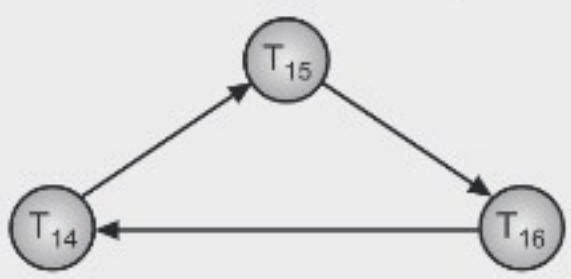


Fig. 2.5 (b): The precedence graph for the given schedule

- The graph is acyclic. The conflict equivalent serial schedule for given schedule can be obtained using step 2 of algorithm.
- T_{14} is the transaction with no arcs entering in T_{14}. Hence, T_{14} is the first transaction in serial schedule. Remove T_{14} and all edges emitting from T_{14}.
- T_{15} is the next schedule, since it has no incoming edges. Remove T_{15} and edges emitting from T_{15}.
- T_{16} is the last schedule. Hence the serial schedule which is conflict equivalent to given schedule is:

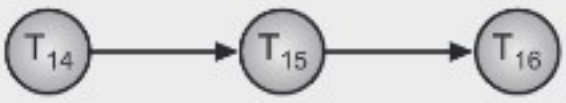


Fig. 2.6: Precedence graph

- Hence, the schedule is conflict serializable.

(iii) Consider schedule 3, the precedence graph for it is given as:

Fig. 2.7: Precedence graph for schedules

- Since, T_1 executes write (A) and write (B) before T_2 executes read (A) and read (B).
- This graph is acyclic and the conflict equivalent serial schedule is $T_1 \rightarrow T_2$. Hence, schedule 3 is conflict serializable.

(iv) Consider schedule 4, the precedence graph for it is:

Fig. 2.8: Precedence graph for schedules

This is a cyclic graph and hence it is not conflict serializable.

(v) Consider the precedence graph given in Fig. 2.9.

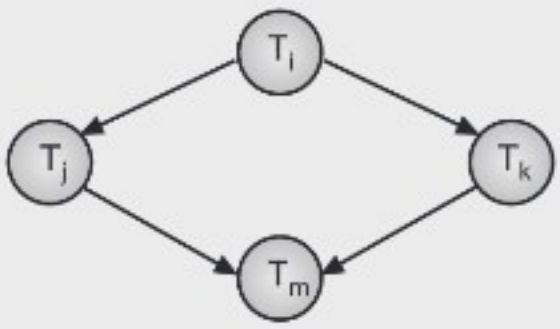


Fig. 2.9: Precedence graph

- The graph is acyclic. The conflict equivalent serial schedule is equivalent to this are:

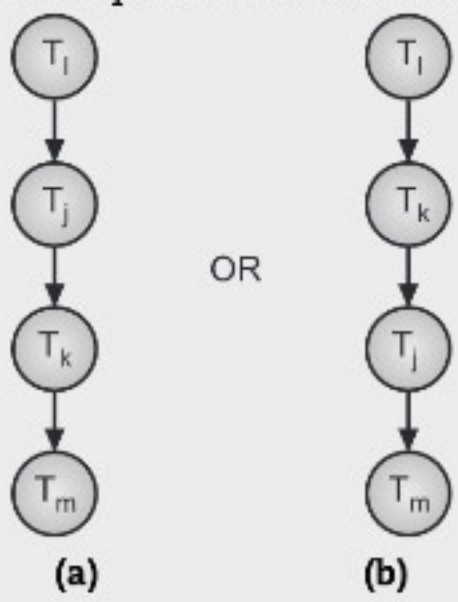


Fig. 2.10: Precedence graph

- Hence, the schedule corresponding to precedence graph in Fig. 2.10 is conflict serializable.

(vi) Consider the precedence graph in Fig. 2.11.

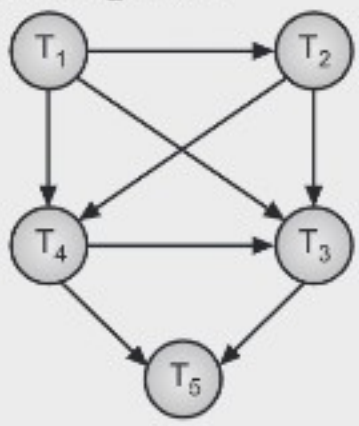


Fig. 2.11: Precedence graph

- The graph is acyclic. The serial schedules that are conflict equivalent to given schedule are:

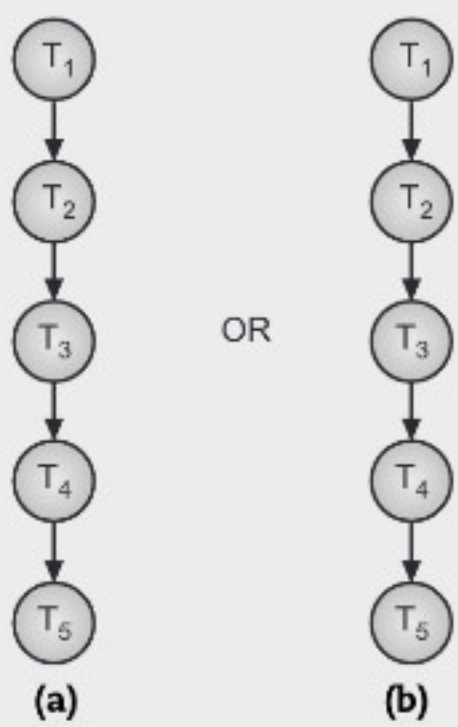


Fig. 2.12: Precedence graph

- Hence, the schedule corresponding to given precedence graph is conflict serializable.

2. Testing of View Serializability:

- Testing for view serializability is complicated. It has been shown that testing for view serializability is itself NP-complete.
- Thus there exists no algorithm to test for view serializability. However concurrency control schemes can still use sufficient conditions for view serializability.
- That is if sufficient conditions are satisfied, the schedule is view serializable schedule. But there may be view serializable schedules that do not satisfy the sufficient conditions.

EXAMPLES

Example 1: Consider the following Transaction. Give two non-serial schedules that the serializable:

T_1	T_2
Read (X)	Read (Z)
X = X + 100	Read (X)
Write (X)	X = X − Z
Read (Y)	Write (X)
Read (Z)	Read (Y)
Y = Y + Z	Y = Y − 100
Write (Y)	Write (Y)

Solution:

(i)

T_1	T_2
Read(X)	
X=X+100	
Write(X)	
	Read(Z)
	Read(X)
	X=X-Z
	Write(X)
Read(Y)	
Read(Z)	
Y=Y+Z	
Write(Y)	
	Read(Y)
	Y=Y-100
	Write(Y)

(ii)

T_1	T_2
Read(X)	
X=X+100	
Write(X)	
	Read(Z)
Read(Y)	
Read(Z)	
	Read(X)
	X=X-Z
	Write(X)
Y=Y+Z	
Write(Y)	
	Read(Y)
	Y=Y-100
	Write(Y)

Example 2: Consider the following transactions. Give two non-serial schedules that are serializable.

T_1	T_2
Read (Y)	Read (X)
Read (A)	Read (A)
Y = Y + A	X = X + A
Write (Y)	Write (X)
	Read (Y)
	Y = Y + A
	Write (Y)

Solution:

(i)

T_1	T_2
Read(Y)	
Read(A)	
	Read(X)
	Read(A)
Y=Y+A	
Write(Y)	
	X=X+A
	Write(X)
	Read(Y)
	Y=Y+A
	Write(Y)

(ii)

T_1	T_2
	Read(X)
	Read(A)
	X=X+A
	Write(X)
Read(Y)	
Read(A)	
Y=Y+A	
Write(Y)	
	Read(Y)
	Y=Y+A
	Write(Y)

Example 3: Consider the following Transaction. Give two non-serial schedules that the serializable:

T_1	T_2
Read (A)	Read (B)
A = A – 1000	B = B + 100
Write (A)	Write (B)
Read (B)	Read (C)
B = B – 100	C = C + 100
Write (B)	Write (C)

Solution:

(i)

T_1	T_2
Read(A)	
A=A-1000	
Write(A)	
	Read(B)
	B=B+100
	Write(B)
Read(B)	
B=B-100	
Write(B)	
	Read(C)
	C=C+100
	Write(C)

(ii)

T_1	T_2
Read(A)	
A=A-1000	
	Read(B)
Write(A)	
	B=B+100
	Write(B)
Read(B)	
B=B-100	
	Read(C)
Write(B)	
	C=C+100
	Write(C)

Example 4: Consider the following transactions. Give two non-serial schedules that are serializable.

T_1	T_2
Read(x)	Read(x)
x=x+10	x=x-10
Write(x)	Write(x)
Read(y)	Read(y)
y=y+20	y=y-20
Write(y)	Write(y)
Read(z)	
z=z+30	
Write(z)	

Give two non-serial schedules that are serializable:

Solution:

(i)

T_1	T_2
Read(x)	
x=x+10	
Write(x)	
	Read(x)
	x=x-10
	Write(x)
Read(y)	
y=y+20	
Write(y)	
	Read(y)
	y=y-20
	Write(y)
Read(z)	
z=z+30	
Write(z)	

(ii)

T_1	T_2
Read(x)	
x=x+10	
Write(x)	
	Read(x)
Read(y)	
	x=x-10
	Write(x)
y=y+20	
Write(y)	
	Read(y)
Read(z)	
	y=y-20
	Write(y)
z=z+30	
Write(z)	

Example 5: Consider the following Transactions. Give two non-serial schedules that are serializable to serial schedule <T_1, T_2, T_3>.

T_1	T_2	T_3
Read (a)	Read (c)	Read (b)
a = a + 100	Read (b)	b = b + 100
Write (a)	b = b + c	Write (b)
Read (b)	Write (b)	Read (a)
b = b – 100	Read (a)	c = c – 100
Write (b)	a = a – c	Write (c)
	Write (a)	

Solution:

(i)

T_1	T_2	T_3
Read(a)		
a=a+100		
Write(a)		
Read(b)		
	Read(c)	
b=b-100		
Write(b)		
	Read(b)	
	b=b+c	
	Write(b)	
		Read(b)
		b=b+100
		Write(b)
	Read(a)	
	a=a-c	
	Write(a)	
		Read(c)
		c=c-100
		Write(c)

(ii)

T_1	T_2	T_3
Read(a)		
a=a+100		
Write(a)		
	Read(c)	
Read(b)		
b=b-100		
Write(b)		
	Read(b)	
	b=b+c	
	Write(b)	
		Read(b)
		b=b+100
		Write(b)
	Read(a)	
		Read(c)
	a=a-c	
	Write(a)	
		c=c-100
		Write(c)

Example 6: Consider the following transactions. Give two non-serial schedules that are serializable to serial schedule <T_1, T_2>.

T_1	T_2
Read (c)	Read (c)
Read (a)	Read (a)
a = a – c	a = a + c
Write (a)	Write (a)
Read (b)	
b = b – c	
Write (b)	

Solution:

(i)

T_1	T_2
Read(c)	
Read(a)	
a=a-c	
Write(a)	
	Read(c)
	Read(a)
Read(b)	
	a=a+c
	Write(a)
b=b-c	
Write(b)	

(ii)

T_1	T_2
Read(c)	
Read(a)	
	Read(c)
a=a-c	
Write(a)	
	Read(a)
Read(b)	
b=b-c	
Write(b)	
	a=a+c
	Write(a)

2.3.1.2.2 Non-Serializable Schedule

- This is classification of schedules based on Recoverability.
- This Schedule has following types:
  1. Recoverable Schedule
  2. Non-recoverable Schedule

1. Recoverable Schedule **(S-19)**

- In recoverability, there is need to address the effect of transaction failures on concurrently running transactions.
- Once a transaction T is committed, it should never be necessary to rollbacks T.
- The schedules those meet this condition are called recoverable schedules and those do not, are called non-recoverable.
- In other words, A recoverable schedule is one where for each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i, the commit operation of T_i appears before the commit operation of T_j. Otherwise, the schedule is non-recoverable.
- Consider the schedule given in Table 2.13. Transaction T_9 reads the data written by T_8. Commit of transaction T_8 occurs after commit of transaction T_9. Hence, it is a non-recoverable schedule.

Table 2.13: Schedule 13

T_8	T_9
Read(A)	
Write(A)	
	Read(A)
Read(B)	

- Recoverable Schedules in DBMS are of three types: Cascadeless Schedule, Strict Schedule and Cascading Rollback Schedule or Cascading Abort.

(a) Cascadeless Schedule **(S-19)**

- Even if a schedule is recoverable, to recover correctly from the failure of a transaction T_i, we may have to rollback the transaction.
- Consider the partial schedule shown below:

Table 2.14: Schedule 14

T_{10}	T_{11}	T_{12}
Read(A)		
Read(B)		
Write(A)		
	Read(A)	Read(A)
	Write(A)	

- Transaction T_{10} writes a value of A that is read by transaction T_{11}. Transaction T_{11} writes a value of A that is read by transaction T_{12}. Suppose that at a point transaction T_{10} fails. T_{10} must be rolled back. Since, T_{11} is dependent on T_{10}, T_{11} must be rolled back. Since, T_{12} is dependent on T_{11}, T_{12} must be rolled back.
- This concept, in which a single transaction failure results in a series of transaction rollbacks, is called cascading rollback.
- A cascadeless schedule is one where, for each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i, the commit operation of T_i appears before commit operation of T_j.

(b) Strict Schedule

- If in a schedule, a transaction is neither allowed to read nor write a data item until the last transaction that has written it is committed or aborted, then such a schedule is called as a **Strict Schedule**.
- In other words,
  - Strict schedule allows only committed read and write operations.
  - Clearly, strict schedule implements more restrictions than cascadeless schedule.

(c) Cascading Rollback or Cascading abort schedule

- If in a schedule, failure of one transaction causes several other dependent transactions to rollback or abort then such a schedule is called as a Cascading Rollback or Cascading Abort or Cascading Schedule. It simply leads to the wastage of CPU time.
- These Cascading Rollbacks occur because of Dirty Read problems.

2. Non-recoverable Schedule

- Consider a situation where the transaction T_i fails before it commits. Since T_j has read the value written by T_i, we must abort T_j also to ensure atomicity. But this is not possible as transaction T_j has already been committed. Thus we have a situation where it is impossible to recover correctly from the failure of T_i. The above situation is an example of a non-recoverable schedule.
- Non-recoverable schedules are not allowed. All DBMSs require that all schedules are recoverable.

Summary

- Collection of operations that forms a single logical unit of work is called transaction.
- A transaction is the result of execution of a user program, written in a high-level data manipulation language or programming language.
- Concurrency control in transaction management preserves the consistency of database in case of concurrent accesses.
- These four properties are often called ACID (Atomicity, Consistency, Isolation, and Durability) properties of transaction.
- In concurrent execution, the DBMS controls the execution of two or more transactions in parallel.
- Schedule represents the chronological order in which instructions are executed in the system.

- If the actions (operations) of different transactions are not interleaved i.e., transactions are executed one-by-one from start to finish, the schedule is called as a serial schedule.
- When several transactions are executed concurrently, the corresponding schedule is called concurrent schedule.
- A concurrent schedule results in a consistent state if its result is equivalent to a serial schedule of that transaction. Such concurrent schedule is known as serializable.
- Serialization refers to the transactions running serially, one after the other, rather than concurrently.
- Every conflict serializable schedule is view serializable, but there are view serializable schedules that are not conflict serializable.
- A view serializable schedule in which blind write appear is not a conflict serializable.
- The non-serializable schedule is divided into two types, Recoverable and Non-recoverable Schedule.
- A schedule is said to be recoverable if it is recoverable as name suggest i.e only reads are allowed before write operation on same data.
- When no read or write-write occurs before execution of transaction then corresponding schedule is called cascadeless schedule.
- If schedule contains no read or write before commit then it is known as strict schedule.
- Cascading Abort can also be rollback.

Check Your Understanding

1. The "all-or-none" property is commonly referred to as _____.
   (a) Isolation (b) Durability
   (c) Atomicity (d) None of the mentioned
2. Which of the following is a property of transactions?
   (a) Atomicity (b) Durability
   (c) Isolation (d) All of the mentioned
3. Execution of translation in isolation preserves the _____ of a database.
   (a) Atomicity (b) Consistency
   (c) Durability (d) All of the mentioned
4. Which of the following is not a transaction state?
   (a) Active (b) Partially committed
   (c) Failed (d) Compensated

5. The execution sequences in concurrency control are termed as _____.
   (a) Serials (b) Schedules
   (c) Organizations (d) Time tables
6. The scheme that controls the interaction between executing transactions is called as _____.
   (a) Concurrency control scheme (b) Multiprogramming scheme
   (c) Serialization scheme (d) Schedule scheme
7. A set of changes that must be all made together is called as a _____.
   (a) Atom (b) Transaction
   (c) Concurrency (d) None of the above.
8. A transaction must be _____.
   (a) Atomic (b) Small
   (c) Large (d) None of the above.
9. ACID stands for _____.
   (a) Again, Common, Internal, Data
   (b) Atomic, Common, Internal, Data
   (c) Atomic, Consistent, Isolation, Durability
   (d) None of the above.
10. The initial state of a transaction is known as _____.
    (a) Active state (b) Partially committed state
    (c) Failed state (d) Aborted state.
11. A list of actions from a set of transactions is known as _____.
    (a) Statement (b) Schedule
    (c) Transaction Set (d) None of the above.
12. A schedule in which the operations from a set of concurrent transactions are interleaved is called as a _____.
    (a) Non-serial schedule (b) Serial schedule
    (c) Complete schedule (d) None of the above.
13. A schedule that will always produce identical results _____.
    (a) Equivalent schedule (b) Complete schedule
    (c) Serial schedule (d) Non-serial schedule.

14. If the precedence graph has no cycles, then the schedule S is _____.
    (a) Conflict Serializable (b) Conflict Equivalent
    (c) Serial (d) None of the above.

ANSWER KEY

1. (c)	2. (d)	3. (b)	4. (d)	5. (b)	6. (b)	7. (b)	8. (a)
9. (c)	10. (a)	11. (b)	12. (a)	13. (a)	14. (a)		

Practice Questions

Q.I. Answer the following questions in short.

1. What is a transaction?
2. Which are states of transaction?
3. What is serializability?
4. Explain the term Schedule.
5. Write any two types of schedules.
6. Which are four properties of ACID?

Q.II. Answer the following questions.

1. With suitable diagram explain different states of transaction.
2. Explain ACID properties of transaction in detail.
3. Which is the main difference between view serializability and conflict serializability?
4. What is the need for concurrent execution of transactions?
5. Explain when schedule is conflict serializable when it is view serializable? Test if the given schedule is view serializable.

T_3	T_4	T_5
read(Q)		
	write(Q)	
write(Q)		
		write(Q)

6. Differentiate between serial and serializable schedule.
7. What is the test view serializability?

8. What is recoverable schedule? Why it is desirable?
9. What is serializability? Explain conflict serializability.

Q.III. Define following terms:

1. Cascadeless schedule
2. Recoverable schedule
3. Non-Recoverable schedule
4. View serializability
5. Conflict serializability
6. Precedence graph

Previous Exams Questions

Summer 2019

1. If there are n transactions, then how many different schedules are possible? **(1M)**
   (a) n! (b) (n – 1)!
   (c) n(n – 1)! (d) None of these

Ans. Refer to Section 2.1

2. Durability of the transaction is ensured by..... **(1M)**
   (a) Concurrency control (b) Application programs
   (c) System server (d) Recovery manager

Ans. Refer to Section 2.1.2

3. A schedule where operations of each transaction are executed consecutively without interruption is..... **(1M)**
   (a) Serial schedule (b) Aborted schedule
   (c) Concurrent schedule (d) Committed schedule

Ans. Refer to Section 2.3

4. Define transaction and list its properties. **(1M)**

Ans. Refer to Section 2.1

5. Explain life cycle of transaction with a diagram. **(5M)**

Ans. Refer to Section 2.1.3

6. Explain cascadeless schedule and recoverable schedule. **(3M)**

Ans. Refer to Sections 2.3.1.2.1

7. Check whether the below schedule is conflict serializable or not, using precedence graph : **(4M)**

S : R1 (B), R3 (C), R1 (A), W2 (A), W1 (A), W2 (B), W3 (B), W1 (B), W3 (B), W3 (C).

Ans. Refer to Section 'Examples'

Winter 2018

1. A schedule in which transactions are executed one after another is called ____ **(1M)**

(a) Linear schedule (b) Serial Schedule

(c) Unrecoverable schedule (d) None of these

Ans. Refer to Section 2.3.1.

2. The.....lists are constructed when the recovery procedure starts. **(1M)**

(a) Redo (b) Undo

(c) Both (a) and (b) (d) None of these

3. What is a complete schedule ? **(1M)**

Ans. Refer to Section 2.3

4. State the atomicity properties of a transaction. **(1M)**

Ans. Refer to Section 2.1.2

5. With the help of a state transition diagram, explain the states in which a transaction can be during its execution. **(5M)**

Ans. Refer to Section 2.1.3

6. Explain conflict serializability with suitable example. **(4M)**

Ans. Refer to Section 2.3.1.2

Summer 2018

1. Isolation of transaction is ensured by ..... **(1M)**

(a) System Server (b) Recovery Manager

(c) Application Programmer (d) Concurrency control

Ans. (d)

2. A schedule where operation of each transaction are executed consecutively without interruption is ..... **(1M)**

(a) Serial Schedule (b) Aborted schedule

(c) Concurrent Schedule (d) Committed schedule

Ans. (a)

3. Define: (i) Transaction (ii) Atomicity. **(1M)**

Ans. Refer to Sections 2.1.1 and 2.1.2.

4. What is schedule? Give types of schedule. **(1M)**

Ans. Refer to Section 2.3.1.

5. List and explain properties of transaction. **(5M)**

Ans. Refer to Sections 2.1.2.

6. Consider the transaction:

T_1	T_2
Read (x)	Read (y)
x=x–70	y=y+10
Write (x)	Write (y)
Read (y)	Read (z)
b:=b+70;	z=z-5;
Write(y)	Read (x)
	Write (z)
	x=x-15
	Write (x)

Give at least 2 non-serial schedules that are serializable. **(4M)**

Ans. Refer to Section 2.3.1.

❖❖❖

3... Concurrency Control

Objectives...

- To understand locking concepts and different lock modes
- To know concept of Timestamping
- To learn about Deadlocks and Concurrency Control

3.1 INTRODUCTION

- Concurrency control in transaction management preserves the consistency of database in case of concurrent accesses.
- In the concurrency control, the multiple transactions can be executed simultaneously. It may affect the transaction result. It is highly important to maintain the order of execution of those transactions.

Problems of Concurrency Control:

- Simultaneous execution of transactions over a shared database can create several data integrity and consistency problems. These problems are:
  - **Lost Update:** This problem occurs when two transactions that access the same database items, have their operations interleaved in a way that makes the value of some database items incorrect.
  - **Dirty Read Problems:** This problem occurs when one transaction reads changes the value while the other reads the value before committing or rolling back by the first transaction.
  - **Inconsistent Retrievals:** This problem occurs when a transaction accesses data before and after another transaction(s) finish working with such data.
- In previous chapter, we have studied about serializibility, now we will see how to ensure serializability by locks.

3.2 ENSURING SERIALIZABILITY BY LOCKS

- If all schedules in a concurrent environment are restricted to serializable schedule, the result will be consistent with some serial execution of transactions and will be considered correct.
- This section deals with some concurrency control schemes/algorithms. These schemes ensure the serializability property of the schedules.
- Following are the different concurrency control schemes:
  1. Lock-based protocol
  2. Time-stamp based protocol
  3. Validation based protocol
  4. Multiple granularity
  5. Multiversion schemes

3.2.1 Lock-based Protocol [S-19]

- Serializability can easily be ensured if access to database is done in mutually exclusive manner i.e. if one transaction is accessing a data item, no other transaction can modify that data item.
- The most common method to implement mutual exclusion is to use locks.

1. Different Lock Modes:

- Consider that database is made up of data-items. A lock is a variable associated with each data item. Manipulating the value of lock is called locking.
- The value of lock is used in locking schemes to control the concurrent access and to manipulate the associated data items.
- Following are the two types or modes of locks.
  - **(i) Shared:** If a transaction T_i has obtained a shared mode lock (denoted by S) on item A, then T_i can read but it cannot write A. It is also called a read-locked item, since multiple transactions are allowed to read a database item concurrently.
  - **(ii) Exclusive:** If a transaction T_i, has obtained an exclusive mode lock (denoted by X) on item A, then T_i can both read and write A. It is also called a write-locked item since a transaction exclusively holds the lock on an item, until it finishes updating the item.
- Depending on the type of the operation, the transaction requests a lock in an appropriate mode on data item.
- The request is made to the concurrency-control manager, and the transaction can proceed with operation only after the concurrency-control manager grants the lock to that transaction.

2. Compatibility Function:

- Given a set of lock modes, the compatibility function can be defined as:
  - Let A and B represent arbitrary lock modes. Suppose that a transaction T_i requests a lock of mode A on item Q on which transaction T_j currently holds a lock of mode B.
  - If transaction T_i can be granted a lock on Q immediately, in spite of presence of the lock of mode B, then we say that mode A is compatible with B. Such a function can be represented conveniently by a matrix. Compatibility of two modes of lock is given in compatibility matrix.

	S	X
S	True	false
X	False	false

'comp' Lock-compatibility Matrix:

- If comp (A, B) is true it means that A is compatible with B. Notice that shared mode (S) is compatible with shared mode (S).
- At any time, several shared-mode locks can be held simultaneously on a particular data item.
- A transaction requests a shared lock on data item Q by executing the lock-S(Q) instruction and an exclusive lock is requested through lock-X(Q) instruction. A data item Q can be unlocked via the unlock (Q) instruction.
- Transaction T_i can unlock the data item Q by executing unlock (Q) instruction. But transaction must hold a lock on a data item, as long as it accesses the data item.
- Locking protocols indicate when a transaction may lock and unlock each of the data items. Each transaction must follows the set of rules specified by locking protocols.
- Locking protocols restrict the number of possible serializable schedules. This section deals with only those locking protocols which allows the conflict serializable schedules.

3. Starvation of Locks:

- Transaction starvation usually happens to large or long transactions that need to lock many data items.
- Suppose that transaction:

  T_2 - has a shared mode lock on data item, and

  T_1 - requests an exclusive mode lock on same data item.
- Clearly T_1 has to wait for T_2 to release the shared mode lock.
- Meanwhile, suppose that,

  T_3 - request a shared mode lock on same data item.

- This lock request is compatible with the lock granted to T_2, so T_3 may be granted the shared mode lock.
- At this point, T_2 may release the lock but still T_1 has to wait for T_3. But again there may be a new transaction T_4 requesting a shared mode lock on the same data-item.
- It is possible that there is a sequence of transactions that each requests a shared mode lock on same data item and T_1 never gets the exclusive mode lock on the data item. The transaction T_1 may never make progress and is said to be starved.
- Starvation of transactions can be avoided by granting locks as follows. When a transaction T_i requests a lock on a data item Q in a particular mode M, the lock is granted provided that:
  (i) There is no other transaction holding a lock on Q in a mode that conflicts with M.
  (ii) There is no other transaction that is waiting for a lock on Q and that made its lock request before T_i.

3.2.2 Two-Phase Locking (2PL) Protocol [S-18, 19; W-18]

- A locking protocol is a set of rules followed by all transactions while requesting and releasing locks.
- 2PL protocol requires that each transaction issue a lock and unlock requests in two phases:
  1. **Growing Phase:** A transaction may obtain locks, but may not release any lock.
  2. **Shrinking Phase:** A transaction may release locks, but may not obtain any new locks. Initially the transaction is in growing phase. In this it acquires locks as needed. Once, the transaction releases a lock, it enters the shrinking phase and it can issue no more lock requests.
- The point in the schedule where the transaction has obtained its final lock (the end of its growing phase) is called the lock point of the transaction. The transactions can be ordered according to lock points.
- This ordering gives the serializability ordering for transaction. This serial schedule is conflict equivalent i.e. the two-phase locking protocol ensures conflict serializability.
- **Example:** Following two transactions are two phase transactions.

```
T3 :   lock-X (B);
       read (B);
       B : = B – 50;
       write (B);
       lock-X(A);
       read (A);
```

A : = A + 50;
write (A);
Unlock (B);
Unlock (A).

T_4 : lock-S (A);
read (A);
lock-S (B);
read (B);
display (A + B);
Unlock (A);
Unlock (B).

- The unlock instructions do not need to appear at the end of the transaction.
- Two phase locking does not ensure freedom from deadlock. Transactions T_3 and T_4 are two phase, but they are deadlocked in the schedule.

Table 3.1: Schedule1

T_3	T_4
Lock-X (B)	
read (B)	
B := B – 50	
write (B)	
	lock- S (A)
	read (A)
	lock-S (B)
Lock-X (A)	
read (A)	
A := A + 50	
write (A)	
unlock (B)	
unlock (A)	
	read (B);
	display (A + B);
	unlock (A);
	unlock (B)

- Cascading rollback may occur under two phase locking.

3.2.2.1 Variations of Two-Phase Locking

Variations of 2PL Locking are explained below:

1. **Strict Two-Phase Locking Protocol: (Strict 2PL):**

- Cascading rollbacks can be avoided by a modification of Two-Phase locking called Strict Two-phase locking protocol.
- It requires that in addition to locking being two-phase, all exclusive-mode locks taken by a transaction must be held until that transaction commits.
- This requirement ensures that any data written by an uncommitted transaction are locked in exclusive mode until the transaction commits, preventing any other transaction from reading the data.

2. **Rigorous Two-Phase Locking Protocol: (Rigorous 2PL):**

- It requires that all locks to be held until the transaction commits.
- With rigorous two-phase locking, transactions can be serialized in the order in which they commit. Most database system implements strict or rigorous two-phase locking.

3. **Conservative 2PL:**

- Conservative 2PL also called a static 2PL, which requires a transaction to lock all the items it accesses before the transaction begins execution by predeclaring its read set and write set.
- Read set of a transaction is the set of all items that the transaction reads and write set of a transaction is the set of all items that the transaction writes. If any of the predeclared items needed can't be locked, the transaction doesn't lock any item. Instead, it waits until all the items are available for locking.
- Conservative 2PL is a deadlock free protocol.
- For example: Consider the two transactions in two-phase locking protocol:

Table 3.2: Schedule 2

T_1	T_2
Lock X (B)	Lock S(A)
Read (B)	Read (A)
B = B – 50	Lock S(B)
Write (B)	Read (B)
Lock X (A)	Display (A + B)
Read (A)	Unlock (A)
A = A + 50	Unlock (B)
Write (A)	
Unlock (B)	
Unlock (A)	

- The solution is, two phase locking does not ensure freedom from deadlock T_1 and T_2 are two phase transmission but they are deadlocked.

Table 3.3: Schedule 3

T_1	T_2
Lock X(B)	
Read (B)	
B = B – 50	
Write (B)	
	Lock S (A)
	Read (A)
	lock (B)
lock X(A)	
Read (A)	
A = A + 50	
Write (A)	
unlock (B)	
unlock (A)	
	Read (B)
	display (A + B)
	unlock (A)
	unlock (B)

- Here, T_1 has a X lock and T_2 wants a S on B and T_2 has a S lock and T_1 wants a X on A. Cascading rollback may occur under 2PL.

3.3 BASIC TIMESTAMP METHOD FOR CONCURRENCY (W-18)

Timestamp ordering protocol decides the ordering of transactions in advance to determine the serializability order. For this it uses timestamp ordering scheme.

3.3.1 Timestamp

- With each transaction T_i in the system, a fixed value called timestamp is associated, denoted by TS (T_i). This timestamp is assigned by database system before T_i starts execution.
- If transaction T_i is assigned a timestamp TS (T_i) and a new transaction T_j enters the system, then TS (T_i) < T_s (T_j). There are two simple methods for implementing timestamp schemes.
  1. Use the value of system clock as the timestamp i.e. a transaction's timestamp is equal to the value of system clock, when the transaction enters the system.

2. Use a logical counter that is incremented after a new timestamp has been assigned. i.e. a transaction's timestamp is equal to the value of the counter when the transaction enters the system.

- Timestamp of transaction determines the serializability order.
- If TS (T_i) < TS (T_j) then the system must ensure that the resultant schedule is equivalent to serial schedule in which T_i appears before T_j.
- To implement this scheme, two timestamp values are associated with each Data item Q.
  1. **W-Timestamp (Q):** It denotes the largest timestamp of any transaction that executed write (Q) successfully.
  2. **R-Timestamp (Q):** It denotes the largest timestamp of any transaction that executed read (Q) successfully.
- Whenever, a new read(Q) or write(Q) instruction is executed, these timestamps are updated.

3.3.2 Timestamp Ordering Protocol (S-19)

- Timestamp ordering protocol ensures that any conflicting read and write operations are executed in timestamp order.
- This protocol works as follows:
  1. **Suppose that transaction T_i issues read (Q).**
     (i) If TS (T_i) < W-Timestamp (Q), then T_i needs to read a value of Q that was already overwritten. Hence, the read operation is cancelled, and T_i is rolled back.
     (ii) If TS (T_i) ≥ W-Timestamp (Q), then the read operation is executed, and R-Timestamp (Q) is set to maximum of R-Timestamp (Q) and TS (T_i).
  2. **Suppose that T_i issues write (Q).**
     (i) If TS (T_i) < R-Timestamp (Q) then the value of Q that T_i is producing was needed previously and the system assumed that value would never be produced. Hence, the write operation is cancelled, and T_i is rolled back.
     (ii) If TS (T_i) < W-timestamp (Q), then T_i is attempting to write an outdated value of Q. Hence, this write operation is cancelled, T_i is rolled back.
     (iii) Otherwise the write operation is executed and W-timestamp is set to TS (T_i).
- A transaction T_i, that is rolled back by the concurrency control scheme as a result of either a read or write operation being issued, is assigned a new timestamp and is restarted.

- **Example:** Consider the given transactions T_{14} and T_{15}.

T_{14}: read (B);
read (A);
display (A + B)

T_{15}: read (B);
B := B – 50;
write (B);
read (A);
A := A + 50;
write (A);
display (A + B).

- Concurrent schedule for the two transactions is given below.
- Here, we consider that TS (T_{14}) < TS (T_{15}). The schedule given in Table 3.4 is possible under the timestamp protocol.

Table 3.4: Schedule 4

T_{14}	T_{15}
read (B)	
	read (B)
	B : = B – 50
	write (B)
read (A)	
	read (A)
display (A + B)	
	A : = A + 50
	write (A)
	display (A + B)

1. read (B) of T_{14} is executed because TS (T_{14}) ≥ W-timestamp (B) and it sets R-Timestamp (B) = TS (T_{14}).
2. read (B) of T_{15} is executed because TS (T_{15}) ≥ W-timestamp (B) and it sets R-Timestamp (B) = TS (T_{15}).
3. write (B) of T_{15} is executed because TS (T_{15}) = R-Timestamp (B).
   Similarly, read (A) of T_{14} and write (A) of T_{15} are executed.

3.3.3 Thomas Write Rule (S-18; W-18)

- Thomas Write Rule is the modification to the basic timestamp ordering, in which the rules for write operations are slightly different from those of basic timestamp ordering.
- It can be summarized as ignore outdated writes.
- It states that, if a more recent transaction has already written the value of an object, then a less recent transaction does not need perform its own write since it will eventually be overwritten by the more recent one.
- Consider the schedule given in Table 3.5.

Table 3.5: Schedule 5

T_{16}	T_{17}
read (Q)	
	write (Q)
write (Q)	

- Apply Timestamp-ordering protocol to given schedule. Since, T_{16} starts before T_{17}, TS (T_{16}) < TS (T_{17}), read (Q) of T_{16} and write (Q) operations succeed.
- When T_{16} attempts its write(Q) operation, we observe that, TS (T_{16}) < W-Timestamp (Q), since W_timestamp (Q) = TS ($_{17}$). According to Timestamp protocol, write (Q) must be rejected, T_{16} will be rolled back.
- Timestamp ordering protocol rolls back the transaction T_{16}, but the value of write (Q) Operation of T_{16} is already written by write (Q) of T_{17}, and the value that write (Q) of T_{16} is attempting to write will never be read. i.e. we can ignore the write (Q) of T_{16}.
- This leads to a modification of Timestamp-ordering protocol. This modified protocol operates as follows:

1. **Suppose that transaction T_i issues read (Q):**
   (i) If TS (T_i) <W-Timestamp (Q), then T_i needs to read a value of Q that was already over written. Hence, the read operation is rejected, and T_i is rolled back.
   (ii) If TS (T_i) ≥ W-timestamp (Q), then read operation is executed, and R-timestamp (Q) is set to the maximum of R-timestamp (Q) and TS (T_i).
2. **Suppose that transaction T_i issues write (Q):**
   (i) If TS (T_i) < R-Timestamp (Q) then the value of Q that T_i is producing was previously needed, and it was assumed that the value would never be produced. Hence, the write operation is cancelled, and T_i is rolled back.
   (ii) If TS (T_i) <W-timestamp (Q), then T_i is attempting write an outdated value of Q. Hence, the write operation can be ignored.

(iii) Otherwise, the write operation is executed, and W-timestamp (Q) is set to TS (T_i).

i.e. here if TS (T_i) <W-Timestamp (Q), then we ignore the obsolete write operation. This modification to timestamp ordering protocol is called **Thomas Write Rule**.

- Under Thomas Write Rule, the write(Q) operation of T_{16} is ignored. Therefore, the schedule in table 3.5 is a view equivalent to the serial schedule of <T_{16}, T_{17}>.

3.4 LOCKS WITH MULTIPLE GRANULARITY (S-18)

- In the concurrency control schemes described so far, we have used each individual data item as the unit on which synchronization is performed.
- But in some cases it would be advantageous to group several data items and to treat them as one individual synchronization unit.
- **For example:** If a transaction T_i needs to access the entire database, and locking protocol is used, then it must lock each data item in the database.
- It would be better if T_i could issue a single lock request to lock the entire database. Similarly, if transaction T_i needs to access only one record in the database, it should not be required to lock the entire database.
- That is we need a mechanism to allows the system to define multiple levels of granularity.
- We can make one by allowing data items to be of various sizes and defining a hierarchy of data granularities where the small granularities are nested within larger ones. Such a hierarchy can be represented by tree.
- As an illustration consider the tree of Fig. 3.1.

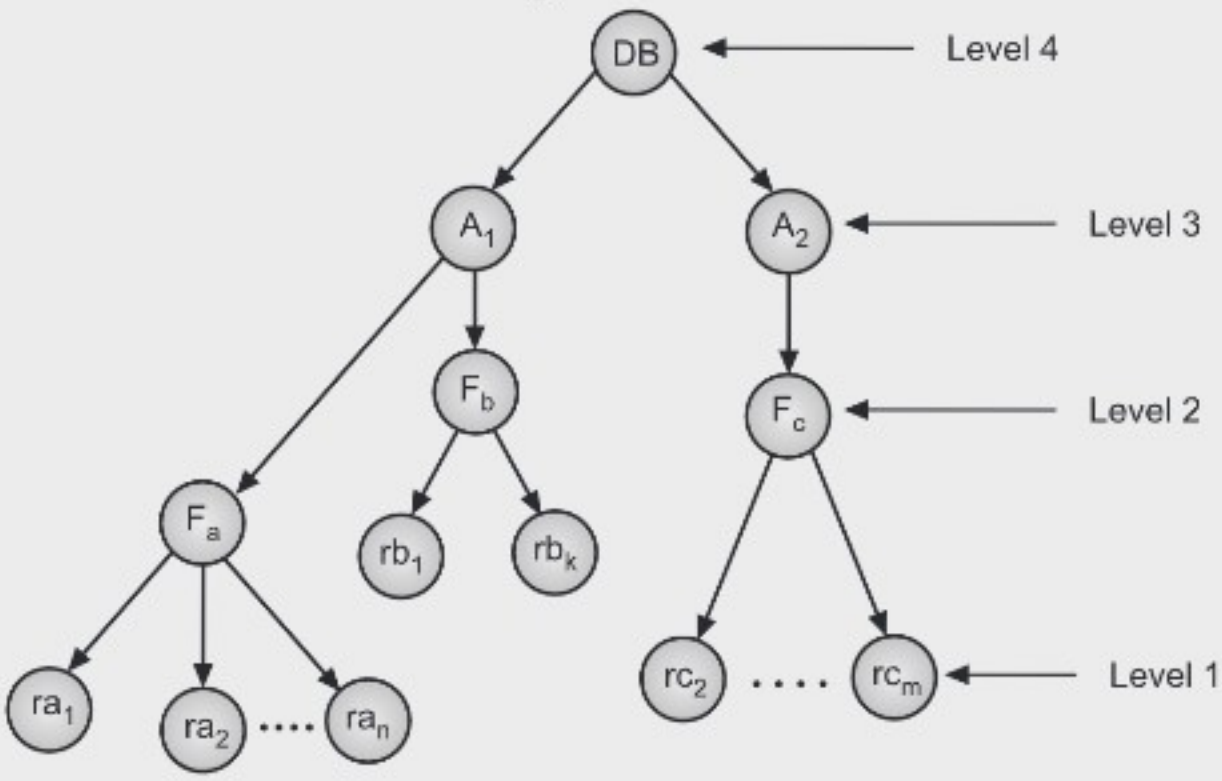


Fig. 3.1: Tree Hierarchy of Multiple Granularities

- The highest level represents the entire database (DB). The second level represents a node of type area (A).
- Each area in turn has nodes of type file (F) as its children. Each area contains exactly those files that are its child nodes. No file is in more than one area.
- Each file has nodes of type record (ra). The file consists of those exactly records that are its child nodes and no record can be present in more than one file.

Modes of Lock:

- Each node in the tree can be locked individually.
- Following are modes of lock used in **Multiple Granularity** scheme.
  1. Shared mode lock (S).
  2. Exclusive mode lock (X).
  3. Intension-shared mode lock (IS).
  4. Intension-exclusive mode lock (IX).
  5. Shared and Intension-exclusive mode lock (SIX).
- When transaction locks a node in shared or exclusive mode, the transaction also has locked all the descendants of that node in the same lock mode.
- If a node is locked in intension-shared mode, then explicit locking is being done at lower level of the tree, but with only shared mode locks.
- If a node is locked in shared and intension-exclusive mode, the sub-tree rooted by that node is locked explicitly in shared mode and that explicit locking is being done at a lower level with exclusive-mode locks.
- The compatibility matrix is given below:

Table 3.6: Compatibility Matrix

Lock Modes	IS	IX	S	SIX	X
IS	true	true	True	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false

- Multiple granularity locking protocol that follows ensures serializability.
- Each transaction T_i can lock a node Q, using the following rules:
  1. The lock compatibility function given in Table 3.6 must be observed.
  2. The root of the tree must be locked first, and can be locked in any mode.

3. A node Q can be locked by T_i in S or IS mode only if the parent Q is currently locked by T_i in either IX or IS mode.
4. A node Q can be locked by T_i in X, SIX or IX mode only if the parent of Q is currently locked by T_i in either IX or SIX mode.
5. T_i can lock a node only if it has not previously unlocked any node.
6. T_i can unlock a node Q only if none of the children of Q are currently locked by T_i.

Example:

- Consider the tree given in Fig. 3.1 and following transactions.
  1. Suppose that transaction T_{18} reads record r_{a2} in file F_a. Then T_{18} needs to lock, the database area A_1 and file F_a in 'IS' mode and to lock r_{a2} in S mode.
  2. Suppose that transaction T_{19} modifies record r_{a9} in file F_a. Then T_{19} needs to lock the database area A_1 and file F_a in 'IX' mode and finally to lock r_{a9} in 'X' mode.
  3. Suppose that transaction T_{20} reads all records in file F_a. Then T_{20} needs to lock the database area A_1 in 'IS' mode and finally to lock file F_a in 'S' mode.
  4. Suppose that transaction T_{21} reads the entire database. It can do so after locking the database in 'S' mode.

Advantages and Disadvantages:

1. This protocol ensures concurrency and reduces lock overhead.
2. It is particularly useful in applications that include a mix of:
   - Short transactions that access only a few data items.
   - Long transactions that produce reports from an entire file or set of files.
3. Deadlock is possible in this protocol. But there are techniques to reduce deadlock frequency to eliminate deadlock entirely.

3.4.1 Dynamic Database Concurrency (Phantom Problem)

- If the concurrency control is performed at the tuple granularity, the conflict may be undetected. This is also called as Phantom Problem.
- **For example:** We search for a dept-name = 'Physics'. Say T_8 requires to access all the records (tuples) pertaining Physics.

```
SELECT count(*) from Department where dept_name = 'Physics';
```

- And if T_9 is a transaction which is executed.

```
INSERT into department values (14,'XYZ','Physics',12300);
```

- Let us consider S is the schedule having T_8 and T_9 transactions.
- If T_8 uses the newly added tuple by T_9 while computing count(*) then T_8 reads a value written by T_9. So in serial schedule T_9 must come first before T_8.

3.5 TIMESTAMPING VS LOCKING

- The difference between timestamping and locking is shown in Table 3.8.

Table 3.8: Difference between Timestamping and Locking

Sr. No.	Timestamping	Locking
1.	It is used to decide whether transaction should wait or rollback.	It is used for concurrency control.
2.	It is used for deadlock prevention.	It is used to improve performance.
3.	Timestamp is a unique identifier to identify a transaction.	A lock is a variable associated with a data item which describes the status of item with respect to possible operations applied on data items.
4.	Timestamping methods assign timestamps to each transaction and enforced serializability by ensuring that the transaction timestamps match the schedule for the transactions.	Locking methods prevents unsearializable schedules by keeping more than one transaction from accessing the same data elements.
5.	Timestamping methods may have causing more transaction abort than a locking protocol.	Locking methods does not have to abort transaction because they prevent potentially conflicting transactions from interacting with other transactions.

3.6 OPTIMISTIC CONCURRENCY CONTROL

- Optimistic techniques are based on the assumption that conflict is rare and that is more efficient to allow transactions to proceed without imposing delays to ensure serializability.
- When a transaction wishes to commit a check is performed to determine whether conflict has occurred. If there has been a conflict the transaction must be rolled back and restarted.
- There are two or three phases to an optimistic concurrency control protocol, depending on whether it is a read only or an update transaction:
  1. **Read Phase:** This extends from the start of the transaction until immediately before the commit. The transaction reads the values of all data items it needs

from the database and stores them in local variables. Updates are applied to a local copy of data, not to the database itself.

2. **Validation Phase:** This phase follows the Read phase. Checks are performed to ensure that the serializability is not violated if the transaction updates are applied to the database. For a read only transaction, this consists of checking that the data values read are still current values for the corresponding data items. If no interference occurred then the transaction is committed otherwise it is aborted. For a transaction that updates, validation consists of determining whether the current transaction leaves the database in a consistent state, with serializability maintained. If not then the transaction is aborted and restarted.
3. **Write Phase:** This phase follows the successful Validation phase for update transaction. During this phase, the updates made to the local copy are applied to the database.

- The validation phase examines the reads and writes of transactions that may cause interference.
- Each transaction T is assigned a timestamp at the start of its execution, start(T). One at the start of the validation phase: validation(T) and other at its finish time: finish(T).
- To pass the validation test, one of the following condition must be true:
  1. All transactions S with earlier timestamps must have finished before transaction T started; that is finish(S) < start (T).
  2. If transaction T starts before an earlier one S finishes:
     (i) The set of data items written by the earlier transaction are not the ones read by the current transaction.
     (ii) The earlier transaction completes its write phase before the current transaction enters its validation phase, that is start(T)< finish(TS) < validation(T).

3.6.1 Multiversion Concurrency Control

- Multiversion two-phase locking scheme combines the advantages of multiversion concurrency control with two phase locking. This protocol differentiates between read-only transactions and update transactions.
- Update transactions perform rigorous two-phase locking i.e. they hold all locks up to the end of the transaction.
- Thus, they can be serialized according to their commit order. Each data item has a single timestamp.
- The timestamp in this case is not a real clock based timestamp. But rather it is counter, which is called TS-counter. It is incremented during commit processing.

- Read-only transactions follow the multiversion timestamp ordering scheme. Timestamp is assigned to read-only transactions by reading the current value of TS-counter before they start execution.
- Thus, when a read-only transaction issues a read (Q) request, the value read is the content of the version whose timestamp is the largest timestamp less than TS (T_i).

1. **Update Transactions:**
   (i) When an update transaction reads an item, it gets a shared lock on the item and reads the largest version of that item.
   (ii) When an update transaction wants to write a item, it first gets exclusive lock on the items and then creates a new version of the data item. The timestamp of the new version is initially set to a value ∞, which value is greater than that of any possible time stamp.
   (iii) When update transaction T_i completes its actions, it carries out commit processing.
   - First T_i sets the timestamp on every version it has created to 1 more the value of TS-counter; then T_i increments TS-counter by 1.
   - Only one update transaction is allowed to perform commit processing at a time.
2. **Read-only Transactions:**
   (i) Read-only transactions that start after T_i increments TS-counter, can see the values updated by T_i.
   (ii) Read-only transactions that start before T_i increments TS-counter, can see the values before updated by T_j.
   - Versions are deleted in a manner similar to that of multiversion timestamp ordering.

3.7 DEADLOCK HANDLING METHODS

- A system is in deadlock state if there exists a set of transactions such that every transaction in the set is waiting for another transaction in the set. If $\{T_0, T_1, ..., T_n\}$ is the set of transactions such that T_0 is waiting for a data item that is held by T_1 and T_1 is waiting for a data item that is held by T_2 ... and T_{n-1} is waiting for a data item that is held by T_n. T_n is waiting for a data item that is held by T_0. Hence, none of the transactions can make progress in such situation. That is the system is in deadlock state.
- There are two different types of resource deadlocks, and they need to be programmed differently.

1. **Single resource:**

- There is only one resource of each type. This means that multiple desktops share one scanner, one printer, and so on. We use resource graphs to detect a deadlock in this scenario.
- We will take a look at a resource deadlock with just one resource. To detect a deadlock, if all resource types have only one instance, then we can use a graph called Wait-for-graph.

2. **Multiple resources**:

- There is more than one resource of each type. This means there are multiple printers for multiple desktops. We need two matrices, the current allocation matrix, and the request matrix to detect a deadlock in this scenario.
- Deadlock detection algorithm that is valid for such a system, works several time-varying data structures that are alike to those used in the **banker's algorithm**.
- Data structures used in this algorithm :
  - **Available:** A vector with length m specifies the number of available resources of each type.
  - **Allocation:** An n x m matrix that defines the number of resources of each type which is currently allocated to each process.
  - **Request:** An n x m matrix shows the current request of each process. If Request [i][j] = k, then the process Pi is requesting k more instances of resource type Rj.

Methods for handling deadlocks:

- There are two methods for dealing with deadlock problems:
  1 Deadlock detection and deadlock recovery (Wait-for Graph), and
  2. Deadlock prevention (Wound-Wait ,Wait-die).

3.7.1 Deadlock Detection and Recovery (Wait-for Graph)

1. **Deadlock Detection:**

- This is one method for dealing with deadlock. It allows the system to enter in a deadlock state, and then try to recover using deadlock detection and dead-lock recovery scheme.
- If the probability that system enters deadlock state is relatively low, this method is efficient.
- An algorithm that examines the state of the system is executed periodically to determine whether a deadlock has occurred.
- If it has occurred then the system must attempt to recover from the deadlock.
- Following are the requirements for deadlock detection and recovery:

(i) Keep information about the current allocation of data items to transactions, as well as any outstanding data item requests.

(ii) Provide an algorithm that uses this information to find out whether the system has entered a deadlock state.

(iii) Recover from the deadlock when detection algorithm finds out that a deadlock exists.

Wait-for graph:

- A Wait-for graph in computer science is a directed graph used for deadlock detection in operating systems and relational database systems.
- The Wait-for graph consists of a pair G = (V, E) where, V is a set of vertices and E is a set of edges. The set of vertices consists of all transactions in the system. Each element in the set E of edges is an ordered pair $T_i \rightarrow T_j$.
- A directed edge $T_i \rightarrow T_j$ in graph imply that T_i is waiting for transaction T_j to release the data item.
- The edge $T_i \rightarrow T_j$ is removed when T_j is no longer holding a data item needed by transaction T_i.
- A deadlock exist in the system if the Wait-for graph for that system contain a cycle (or loop).
- Consider the Wait-for graph given in Fig. 3.2.

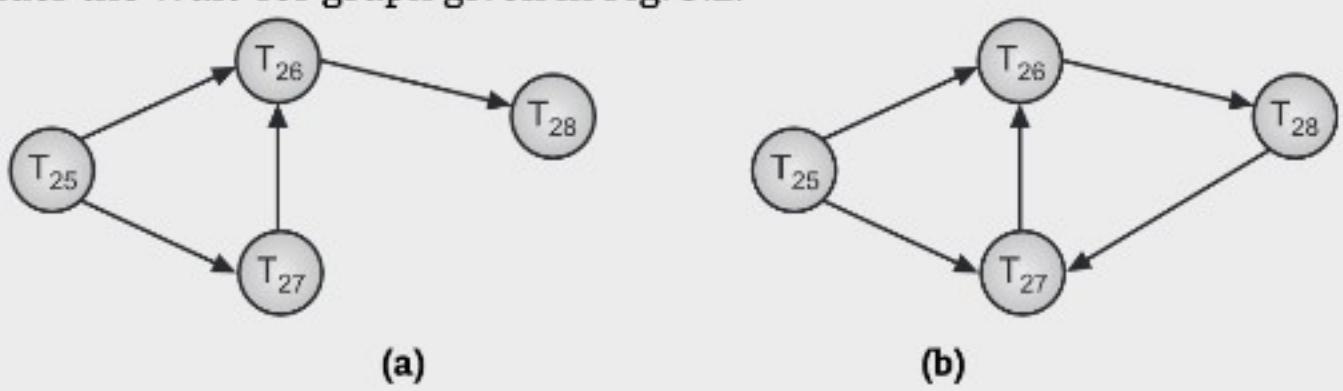


Fig. 3.2: Wait-for Graph

Transaction T_{25} is waiting for T_{26} and T_{27}.
Transaction T_{26} is waiting for T_{28}.
Transaction T_{27} is waiting for T_{26}.

- The graph contains no cycle. Hence, the system is not in deadlock state.

  But now consider the Fig. 3.2 (b). It contains a loop,

  $$T_{26} \rightarrow T_{28} \rightarrow T_{27} \rightarrow T_{26}$$

- Hence, the system is in deadlock state. In this way using this algorithm deadlocks can be detected. Depending on the frequency of deadlock occurrence, the deadlock detection algorithm should be executed.

2. Deadlock Recovery:

- The most common solution to recover from deadlock is to rollback one or more transactions to break the deadlock.
- In the next section 3.7.3, we will see Deadlock Recovery Techniques.

3.7.2 Deadlock Prevention Algorithms (Wound-wait, Wait-die)

(W-18, S-19)

- Deadlock prevention means that we design such a system where there is no chance of having a deadlock.
- There are two approaches to deadlock prevention as given below:
  1. It ensures that cyclic waits can be avoided by ordering the requests for locks or requiring all locks to be acquired together.
  2. It performs transaction rollbacks instead of waiting for a lock, whenever the wait could potentially result in a deadlock.
- Following are the schemes under first approach. Lock all the data items before a transaction begins its execution. But there are two main disadvantages of using this protocol.
  1. It is often hard to predict, before the transaction, what data items need to be locked.
  2. Data item utilization may be very low, since many of the data items may be locked but unused for a long time.
- Another scheme is to impose a partial ordering of all data items, and transaction can lock the data items only in that order. Using tree protocol, this scheme can be implemented.
- Following are the schemes for second approach. The second approach uses preemption and transaction rollbacks.
- When a transaction T_2 requests a lock that is held by a transaction T_1, the lock granted to T_1, may be preempted by rolling back of T_1 and granting of lock to T_2.
- To control the preemption, we assign a unique timestamp to each transaction. These timestamps will be used to decide whether the transaction should wait or rollback. Locking is used for concurrency control. If a transaction is rolled back, it retains the old timestamp when restarted.
- Two different deadlock prevention schemes using timestamps under the second approach are:

1. **Wait-die:** Scheme is based on non-preemptive technique. When a transaction T_i requests a lock on a data item currently held by T_j, T_i is allowed to wait only if it has a timestamp smaller than that of T_j. Otherwise T_i is rolled back (die).

Example: $TS(T_i) = 5$
$TS(T_j) = 10$
$TS(T_k) = 15$

If T_i requests a lock on a data item held by T_j then T_i will wait since $TS(T_i) < TS(T_j)$. If T_k requests a lock on a data item held by T_j, then T_k will be rolled back since $TS(T_k) > TS(T_j)$.

2. **Wound-wait:** This scheme is based on preemptive technique. When a transaction T_i requests a lock on a data item, currently held by T_j, T_i is allowed to wait only if it has a timestamp larger than that of T_j. Otherwise T_j is rolled back (T_j is wounded by T_i).

   Example: Consider the timestamps given in previous example for T_i, T_j and T_k transactions.
   If T_i requests a lock on data item held by T_j, then T_j will be rolled back.

   $$TS(T_i) < TS(T_j)$$

   If T_k requests a lock on a data item held by T_j, then T_k will wait since,

   $$TS(T_k) > TS(T_j)$$

- **Comparison of Wait-die and Wound-wait Schemes:**
  1. Both the Wait-die and Wound-wait schemes avoid starvation.
  2. In Wait-die scheme, an older transaction must Wait-for younger one to release the data item. By contrast, in Wound-wait scheme the oldest transaction never waits for a latest transaction.
  3. Number of rollbacks in Wait-wound scheme is fewer as compared to Wait-die scheme.
  4. Major problem in both the schemes is unnecessary rollbacks.
  5. Wait-die is a non-preemptive technique for deadlock prevention whereas wound-wait It is a preemptive technique.

3.7.3 Deadlock Recovery Techniques

- After the detection of a deadlock, we need to recover from it.
- Following are some ways to recover from a deadlock:

1. **Process Termination:** To eliminate deadlock, we can simply kill one or more processes. For this, we use two methods:
   - Abort all the processes involved in a deadlock
   - Abort one process at a time until the deadlock is eliminated.
2. **Resource Preemption:** To eliminate deadlocks using resource preemption, we preempt some resources from processes and give those resources to other processes. Preempt means that we are taking away the resources from a process.

- If preemption is mandatory to deal with deadlocks, then three issues required to be addressed:

(a) Selection of Victim:

- Given a set of deadlocked transactions, we should determine which transaction to rollback to break the deadlock.
- Following should be kept in mind for determining the roll back of transaction:
  (i) How many data items the transaction has used?
  (ii) How many more data items the transaction needs for it to complete?
  (iii) How many transactions will be involved in rollback?

(b) Rollback:

- Once we have decided that a particular transaction must be rolled back, we must determine how far this transaction should be rolled back.
- The simplest solution is total roll back. Abort the transaction and restart it.
- However it is efficient to roll back the transaction to break the deadlock.
- The deadlock detection mechanism should decide which locks the selected transaction needs to release in order to break the deadlock.

(c) Starvation:

- In a system where the selection of victims is based primarily on cost factors. It may happen that the same transaction is always picked as a victim.
- As a result, the transaction never completes its designated task, thus there is a transaction.
- We must be ensure that a particular transaction is rolled back for finite times.

EXAMPLES

Example 1: Following is the list of events in an interleaved execution of set T_1, T_2, and T_3 assuming 2PL (Two Phase Lock). Is there a Deadlock? If yes, which transactions are involved in Deadlock?

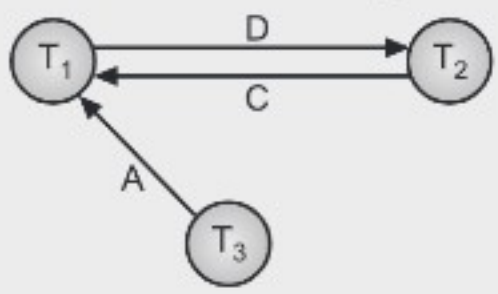
Time	Transaction	Code
t_1	T_1	Lock (A, X)
t_2	T_2	Lock (B, S)
t_3	T_3	Lock (A, S)
t_4	T_1	Lock (C, X)
t_5	T_2	Lock (D, X)
t_6	T_1	Lock (D, S)
t_7	T_2	Lock (C, S)

Solution: As per the given list of events (from time t_1 to t_7) in the problem, the schedule can be written as follows:

T_1	T_2	T_3
X(A)		
	S(B)	
		S(A)
X(C)		
	X(D)	
S(D)		
	S(C)	

Where X(A) indicates exclusive lock on object A.

The waits-for graph for the above schedule is as given below:



From the above waits-for graph, there is cycle between T_1, T_2. Hence, there is deadlock.

Example 2: Following is the list of events in an interleaved execution of set T_1, T_2, T_3 and T_4 assuming 2PL. Is there a Deadlock? If yes, which transactions are involved in Deadlock?

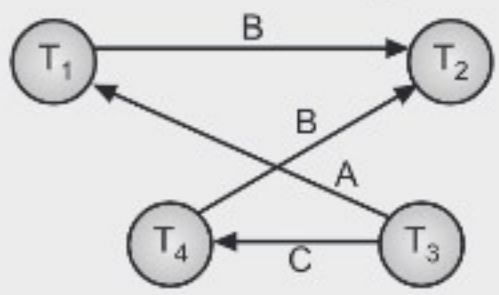
Time	Transaction	Code
t_1	T_1	Lock (A, X)
t_2	T_2	Lock (B, X)
t_3	T_3	Lock (A, S)
t_4	T_4	Lock (B, S)
t_5	T_1	Lock (B, S)
t_6	T_2	Lock (D, S)
t_7	T_3	Lock (C, S)
t_8	T_4	Lock (C, X)

Solution: As per the given list of events (from time t_1 to t_8) in the problem, the schedule can be written as follows:

T_1	T_2	T_3	T_4
X(A)			
	X(B)		
		S(A)	
			S(B)
S(B)			
	S(D)		
		S(C)	
			X(C)

Where, X(A) indicates exclusive lock on object A.

The waits-for graph for the above schedule is as given below:



From the above waits-for graph, there is no cycle. Hence, there is no deadlock.

Example 3: Following is the list of events in an interleaved execution of set T_1, T_2, T_3 and T_4 assuming 2PL (Two Phase Lock). Is there a Deadlock? If yes, which transactions are involved in Deadlock?

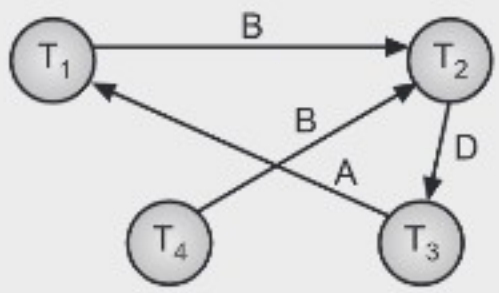
Time	Transaction	Code
t_1	T_1	Lock (A, X)
t_2	T_2	Lock (B, X)
t_3	T_3	Lock (A, S)
t_4	T_4	Lock (B, S)
t_5	T_1	Lock (B, S)
t_6	T_3	Lock (D, X)
t_7	T_2	Lock (D, S)
t_8	T_4	Lock (C, X)

Solution: As per the given list of events (from time t_1 to t_8) in the problem, the schedule can be written as follows:

T_1	T_2	T_3	T_4
X(A)			
	X(B)		
		S(A)	
			S(B)
S(B)			
		X(D)	
	S(D)		
			X(C)

Where, X(A) indicates exclusive lock on object A.

The waits-for graph for the above schedule is as given below:



From the above waits-for graph, there is cycle among T_1, T_2 and T_3. Hence there is deadlock.

Example 4: Following is the list of events in an interleaved execution of set T_1, T_2, T_3 and T_4 assuming 2PL. Is there a Deadlock? If yes, which transactions are involved in Deadlock?

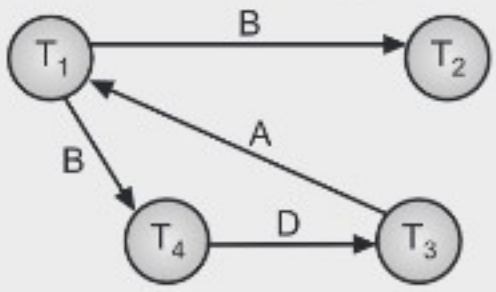
Time	Transaction	Code
t_1	T_1	Lock (A, X)
t_2	T_2	Lock (B, S)
t_3	T_3	Lock (A, S)
t_4	T_4	Lock (B, S)
t_5	T_1	Lock (B, X)
t_6	T_2	Lock (C, S)
t_7	T_3	Lock (D, S)
t_8	T_4	Lock (D, X)

Solution: As per the given list of events (from time t_1 to t_8) in the problem, the schedule can be written as follows:

T_1	T_2	T_3	T_4
X(A)			
	S(B)		
		S(A)	
			S(B)
X(B)			
	S(C)		
		S(D)	
			X(D)

Where X(A) indicates exclusive lock on object A.

The waits-for graph for the above schedule is as given below:



From the above waits-for graph, there is cycle among T_1, T_4 and T_3. Hence, there is deadlock.

Example 5: Following is the list of events in an interleaved execution of set of transaction T_1, T_2, T_3 and T_4 assuming 2PL. Is there a deadlock? If yes, which transactions are involved in deadlock?

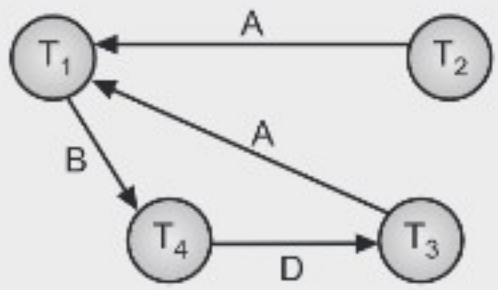
Time	Transaction	Code
t_1	T_1	Lock(A,X)
t_2	T_2	Lock(A,S)
t_3	T_3	Lock(A,S)
t_4	T_4	Lock(B,S)
t_5	T_1	Lock(B,X)
t_6	T_2	Lock(C,X)
t_7	T_3	Lock(D,S)
t_8	T_4	Lock(D,X)

Solution: As per the given list of events (from time t_1 to t_7) in the problem, the schedule can be written as follows:

T_1	T_2	T_3	T_4
X(A)			
	S(A)		
		S(A)	
			S(B)
X(B)			
	X(C)		
		S(D)	
			X(D)

Where, X(A) indicates exclusive lock on object A.

The Wait-for graph for the above schedule is as given below:



From the above Wait-for graph, there is cycle among T_1,T_4 and T_3. Hence there is deadlock.

Summary

- A lock is a variable associated with each data item. Manipulating the value of lock is called locking.
- A locking protocol is a set of rules followed by all transactions while requesting and releasing locks.
- 2PL protocol requires that each transaction issue a lock and unlock requests in two phases i.e., growing phase (a transaction may obtain locks, but may not release any lock) and shrinking phase (a transaction may release locks, but may not obtain any new locks. Initially the transaction is in growing phase).
- The point in the schedule where the transaction has obtained its final lock (the end of its growing phase) is called the lock point of the transaction.
- Cascading rollbacks can be avoided by a modification of two phase locking called strict two-phase locking protocol.
- Conservative 2PL also called a static 2PL, which requires a transaction to lock all the items it accesses before the transaction begins execution by predeclaring its read set and write set.

- Conservative 2PL is a deadlock free protocol.
- With each transaction T_i in the system, a fixed value called timestamp is associated, denoted by TS (T_i).
- Timestamp ordering protocol ensures that any conflicting read and write operations are executed in timestamp order.
- If the concurrency control is performed at the tuple granularity, the conflict may be undetected. This is also called as Phantom Problem.
- Optimistic techniques are based on the assumption that conflict is rare and that is more efficient to allow transactions to proceed without imposing delays to ensure serializability.
- Multiversion two-phase locking scheme combines the advantages of multiversion concurrency control with two phase locking.
- It allows the system to enter a deadlock state, and then try to recover using deadlock detection and dead-lock recovery scheme.
- Deadlocks can be detected using a directed graph called Wait-for graph.
- The Wait-for graph consists of a pair G = (V, E) where V is a set of vertices and E is a set of edges.

Check Your Understanding

1. Which of the following timestamp based protocols generates serializable schedules?
   (a) Thomas write rule (b) Timestamp ordering protocol
   (c) Validation protocol (d) None of the mentioned
2. The default timestamp ordering protocol generates schedules that are _____.
   (a) Recoverable (b) Non-recoverable
   (c) Starving (d) None of the mentioned
3. R-timestamp(Q) denotes?
   (a) The largest timestamp of any transaction that can execute write(Q) successfully
   (b) The largest timestamp of any transaction that can execute read(Q) successfully
   (c) The smallest timestamp of any transaction that can execute write(Q) successfully
   (d) The smallest timestamp of any transaction that can execute read(Q) successfully

4. A logical counter is after a new timestamp has been assigned _____.
   (a) Incremented (b) Decremented
   (c) Doubled (d) Remains the same
5. The rigorous two-phase locking protocol permits releasing all locks at the _____.
   (a) Beginning of transaction (b) During execution of transaction
   (c) End of transaction (d) Never in the life-time of transaction
6. The system must deal with the deadlocks that are not prevented by using schemes of _____.
   (a) Validation (b) Deadlock detection
   (c) Deadlock recovery (d) Both A and B

ANSWER KEY

1. (a)	2. (b)	3. (b)	4. (a)	5. (c)	6. (d)

Practice Questions

Q.I. Answer the following questions in short.

1. What is meant by concurrency control?
2. What is meant by deadlock handling?
3. What is a lock?
4. What is timestamp?
5. What is deadlock prevention?

Q.II. Answer the following questions.

1. Explain two-phase lock protocol with example.
2. Explain timestamp-based and lock-based protocols.
3. What is deadlock detection? Explain one method for deadlock detection.
4. State the rules for a timestamp based concurrency control protocol and demonstrate its working.
5. Explain any one locking protocol.
6. Describe procedure for deadlock recovery.
7. State and Explain Thomas write rule with example.
8. Write a short note on Phantom phenomenon.
9. Compare between Timestamp and locking.

Q.III. Define following terms.

1. Shared mode of lock
2. Exclusive mode of lock

3. Wait-for graph method
4. Wound-wait method,
5. Wait-die method
6. W-Timestamp
7. R-Timestamp
8. Two-phase protocol
9. Multiversion two-phase locking

Previous Exams Questions

Summer 2019

1. Explain two-phase locking protocol. **(4M)**

Ans. Refer to Section 3.2.2

2. Explain deadlock prevention algorithm. **(4M)**

Ans. Refer to Section 3.7.2

3. Explain Timestamp ordering protocol. **(4M)**

Ans. Refer to Section 3.3.2

4. Define : (1) Lock (2) Shared Lock (3) Exclusive lock. **(3M)**

Ans. Refer to Section 3.2.1

Winter 2018

1. The.....lock can be held by more than one transaction on an object at the same time. State the Thomas write rule. **(1M)**

(a) Shared (b) Binary

(c) Exclusive (d) All of these

Ans. Refer to Section 3.3.3

2. In the.....protocol, the locks held by a transaction are released only when the transaction commits. **(1M)**

(a) 2PL (b) Conservative 2PL

(c) Rigorous 2PL (d) Strict 2PL

Ans. Refer to Section 3.2.2

3. State the Thomas write rule. **(1M)**

Ans. Refer to Section 3.3.3

4. Explain the Basic Timestamp ordering method for concurrency control. **(3M)**

Ans. Refer to Section 3.3

5. Explain the two-phase locking protocol and list its variants. **(4M)**

Ans. Refer to Section 3.2.2

6. What are the different deadlock prevention algorithms? Explain any one in detail.

Ans. Refer to Section 3.7.2 **(3M)**

7. Consider the following schedule of interleaved execution of transactions. Check if there is a deadlock in the system : **(4M)**

Ans.

T_1	T_2	T_3
		R(X)
		W(X)
	R(Y)	
R(Z)		
W(Z)		
	W(Y)	
R(X)		
W(X)		
		R(Y)
		W(Y)
	R(Z)	
	W(Z)	

Summer 2018

1. Explain how 2PL ensures serializibility. **(4M)**

Ans. Refer to Section 3.2.2.

2. What is multiple granularity? **(3M)**

Ans. Refer to Section 3.4.

3. State and Explain Thomas write Rule. **(3M)**

Ans. Refer to Section 3.3.3.

❖❖❖

4... Crash Recovery

Objectives...

- To understand concept of Crash Recovery
- To learn Recovery Concepts and checkpoints in Database
- To study about Database Backup and log-based Recovery
- To learn Buffer Management and Shadow Paging

4.1 INTRODUCTION

- A crash is the sudden failure of a software application or operating system or of a hardware device such as a hard disk.
- An important feature of conventional database systems is the ability to recover from system crashes. Recovery means restoring data into its previous state before the incident.
- Database recovery is the process of restoring the database to a correct (consistent) state in the event of a failure. In other words, it is the process of restoring the database to the most recent consistent state that existed shortly before the time of system failure.
- Recovery is required to protect the database from data inconsistencies and data loss.
- Transactions (or units of work) against a database can be interrupted unexpectedly. If a failure occurs before all of the changes that are part of the unit of work are completed, committed, and written to disk then the database is left in an inconsistent and unstable state.
- Crash recovery is the process by which the database is moved back to a consistent and usable state. This is done by rolling back incomplete transactions and completing committed transactions that were still in memory when the crash occurred.

4.2 CLASSIFICATION OF TRANSACTION FAILURE (W-18)

- There are different types of failure that may occur in a system, each of failure needs to be dealt with in a different manner.
- The simplest form of failure is one that does not result in the loss of information or data in the system.
- The failures that are more difficult or critical to deal with are those that result in loss of information or data (content).

1. Transaction Failure:

- There are two types of errors that may lead to transaction failure. They are:
  - (i) **System Error:** The system has entered an undesirable state as a result of which a transaction cannot continue with its normal execution. The transaction, however, can be re-executed at a later time.
  - (ii) **Logical Error:** The transaction can no longer continue with its normal execution, owing the some internal condition, such as data not found, bad input or resource limit overflow exceeded.

2. System Crash:

- There is a hardware bug or error in the database software or the operating system that causes the loss of the content of volatile storage and leads transaction processing to a halt.
- The content of non-volatile storage remains undamaged and is not corrupted.

3. Disk Failure:

- In disk failures disk block losses its content as a result of either a head failure during a data transfer operation. Copies of the data on other disks on tertiary media such as tapes, disks etc., are used to recover from the failure.
- To find out how the system should recover from failure or crash, we need to recognize the failure modes of those devices used for storing data.
- Next, we must consider how these failure modes or crash modes affect the contents or information of the database. We can then use algorithms to ensure database consistency and transaction atomicity despite failures known as Recovery Algorithms.
- Although they have two parts, one is actions taken following a failure to recover the database contents or data to a state that ensures database transaction atomicity, durability and consistency and other actions taken during normal transaction processing to ensure that enough information exists to allow recovery from failures.

4.3 RECOVERY CONCEPTS

- Consider the Banking System and transaction T_i that transfers ₹ 1000 from account X to account Y, with initial values of X and Y being ₹ 5000 and ₹ 10000 respectively.
- Suppose that a system crash has occurred during the execution of transaction T_i, after output (Y_A) has taken place, but before output (Y_B) was executed, where Y_A and Y_B denote the buffer blocks on which X and Y are stored. Since the memory contents were lost, users do not know the fate of the transaction; thus, we could use one of two possible recovery procedures.
  1. **Re-execute T_i:** This procedure will result in the value of X becoming ₹ 3000, rather than ₹ 4000. Thus, the system enters an inconsistent state.
  2. **Do Not Re-execute T_i:** The current system state is value of ₹ 4000 and ₹ 10000 for X and Y respectively. Thus, the system enters an inconsistent state.
- To achieve our goal of atomicity, we must first output information describing the updations of stable storage, without modifying the database itself. As we shall see, this procedure will allows us to output all the updations made by a committed transaction, despite failures.

4.4 CONCEPT OF LOG (W-18)

- The most widely used structure for recording database modifications is the log.
- It is a sequence of log records and maintains a record of all the update activities in the database.
- There are several types of log records to record significant events during transaction processing.
- An update log record describes a single database write. It has the following fields:
  - **Transaction identifier** is the unique identifier of the transaction that performed the write operation.
  - **Data-item identifier** is the unique identifier of the data item written. Typically, it is the location on disk of the data item.
  - **Old value** is the value of the data item prior to the write.
  - **New value** is the value that the data item will have after the write. Other special log records exist to record significant events during transaction processing, such as the start of a transaction and the commit or abort of a transaction.
- We denote the various types of log records as:
  1. **Start of transaction :**
     denoted as; <T_i start>

2. **Update log:**
   It describes a single database write and it is denoted as:
   $<T_i, X_j, V_1, V_2>$
   where, $T_i \rightarrow$ Transaction identifier
   $X_j \rightarrow$ Data item identifier
   $V_1 \rightarrow$ Old value of X_j
   $V_2 \rightarrow$ New value of X_j after the write operation.
3. **Transaction commits:**
   denoted as: $<T_i$ commit>
4. **Transaction abort:**
   denoted as: $<T_i$ abort>

Log is stored in stable storage.

- Whenever a transaction performs a write, it is essential that the log record for that write be created before the database is modified.
- Once a log record exists, we can output the modification to the database if that is desirable.
- Also, we have the ability to undo a modification that has already been output to the database. We undo it by using the old-value field in log records.

4.5 RECOVERY WITH CONCURRENT TRANSACTIONS (ROLLBACK, CHECKPOINT, COMMIT)

- Now we will discuss how user can modify and extend the log-based recovery scheme to deals with multiple concurrent transactions.
- The number of concurrent transactions, computer system has a single disk buffer and a single log. The buffer blocks are shared by all transactions.
- User allows immediate updates and permits a buffer block to have data items updated by one or more transactions.

4.5.1 Rollback

- User rollback a failed transaction, T_i, using the log.
- The log is scanned backward; for every log record of the form $<T_i, X_j, V_1, V_2>$ found in the log, the data item X_j is restored to its old value V_1.
- The scanning of the log terminates when the log record $<T_i$, start> is found.
- Scanning the log backward is more important, since a transaction may have updated a data item more than once. Consider the pair of log records as given below:

  $<T_i, A, 10, 20>$
  $<T_i, A, 20, 30>$

- They represent a modification of data item A by T_i, followed by another modification of A by T_i. Scanning the log backward set A correctly to 10.
- If the log were scanned in the forward direction, A would be set to 20, which value is incorrect.
- If strict two-phase locking is used for concurrency control, locks held by a transaction T may be released only after the transaction has been rolled back as described.
- Once, transaction T has updated a data item, no other transaction could have updated the same data item, due to the concurrency-control requirements.

Interaction with Concurrency Control :

- The recovery scheme depends on the concurrency-control technique that is used. To roll back a failed transaction, user must undo the updates or modification performed by the transaction.
- Suppose that a transaction T_0 has to be rolled back and a data item Q that was updated by T_0 has to be restored to its old value. Using the log-based schemes for recovery, we restore the value using the undo information in a log record.
- Suppose a second transaction T_1 has performed yet another update on Q before T_0 is rolled back. Then, the updation performed by T_1 will be lost if T_0 is rolled back.
- If a transaction T has updated a data item Q, no other transaction may update the same data item until T has committed. We can ensure this requirement easily by using strict two-phase locking.

4.5.2 Checkpoint (S-18, 19; W-18)

- When a system failure occurs, some transactions need to be redone and some need to be undone. Log record can find out this. But for that we need to search the entire log.
- There are two major difficulties with this approach.
  1. The search process is time consuming.
  2. Most of the transactions that will be redone have already written their updates into the database. Hence, it is better to avoid such redo operations.
- To reduce these types of overheads, checkpoints are introduced. During the execution the system maintains the log, using immediate database or deferred database modification technique.
- In addition, the system periodically performs checkpoints, which require following sequence of operations:
  1. Output onto stable storage, all log records currently stored in main memory.
  2. Output to the disk, all modified buffer blocks.
  3. Output onto stable storage, a log record <checkpoint>.

- Transactions are not allowed to perform any update actions, such as writing to a buffer block or writing a log record, while a checkpoint is in progress.
- The presence of <checkpoint> record in log allows the system to streamline its recovery procedure. After the failure has occurred, the recovery system examines the log to determine the most recent transaction T_i that started execution before the most recent checkpoint took place.
- It can find such a transaction by searching the log backward, from the end of the log until it finds the first <checkpoint> record, then it continues the search backward until it finds the next <T_i start> record. This record identifies a transaction T_i.
- Once the transaction is identified, redo or undo operation can be applied to transaction T_i and all the transactions T executing after T_i. The earlier part of transaction can be ignored. The recovery can be done by using immediate database modification or deferred database modification technique.

4.5.3 Commit

- If a transaction executes all its operations successfully, it is said to be committed. All its effects are now permanently established on the database system.
- Commit command is used to permanently save any transaction into database.

  Syntax:

  ```
  commit;
  ```

  For example:

  ```
  INSERT into class values (5,'Rahul');
  commit;
  ```

- After commit point the transaction writes a commit record [COMMIT, T_i] into the log.
- If a system failure occurs, the log is searched back for all transactions T that have written a [START_TRANSACTION, T_i] records in the log but do not make [COMMIT, T_i]. Then these transactions may have to be rolled back to undo their effect on the database.
- Transactions that are committed must also have recorded all their write entries in the log. So, their effect on the database can be redone from the log entries.

4.6 LOG-BASED RECOVERY (DEFERRED AND IMMEDIATE UPDATE) (S-18, 19)

- The log-based recovery techniques maintain transaction logs to keep track of all update operations of the transaction.
- It assumes that transactions are executed serially, i.e. only one transaction is active at a time. It uses a structure called log to store the database modifications.

Recovery Procedures:

- Two operations applied to recover from a failure are: *undo* and *redo*. These are applied with the help of the log on the last consistent state of the database.

1. **Undo**: This operation reverses (rolls back) the changes made to the database by an uncommitted transaction and restores the database to the consistent state that existed before the start of the transaction.
2. **Redo:** This operation reapplies the changes of a committed transaction and restores the database to the consistent state it would be at the end of the transaction. This operation is required when the changes of a committed transaction are not or partially reflected to the database on disk. The redo modifies the database on disk to the new values for the committed transaction.

Recovery Algorithms :

- Conceptually, there are two main techniques for recovery from non-catastrophic transaction failures:
  1. Deferred update (or NO UNDO/REDO) algorithm.
  2. Immediate update (or UNDO/REDO) algorithm

4.6.1 Deferred update

- The idea behind deferred update is to defer or postpone any actual updates to the database itself until the transaction completes its execution successfully and reaches its commit point. During transaction execution, the updates are recorded only in the log and in the transaction workspace. After the transaction reaches its commit point and the log is force-written to disk, the updates are recorded in the database itself. If a transaction fails before reaching its commit point, there is no need to undo any operations, because the transaction has not affected the database in any way.
- The steps involved in the Deferred update protocol are as follows:
  1. When a transaction starts, write an entry start_transaction(T) to the log.
  2. When any operation is performed that will change values in the database, write a log entry write_item(T, X, old_value, new_value).
  3. When a transaction is about to commit, write a log record of the form commit(T); write all log records to disk.
  4. Commit the transaction, using the log to write the updates to the database; the writing of data to disk need not occur immediately.
  5. If the transaction aborts, ignore the log records and do not write the changes to disk.
- The database is never updated until after the transaction commits, and there is never a need to UNDO any operations. Hence this technique is known as the NO-UNDO/REDO algorithm. The REDO is needed in case the system fails after the transaction commits but before all its changes are recorded in the database. In this case, the transaction operations are redone from the log entries.

- The protocol and how different entries are affected can be best summarized as shown:

Log entry	Log written to disk	Changes written to database buffer	Changes written on disk
start_transaction(T)	No	N/A	N/A
read_item(T, X)	No	N/A	N/A
write_item(T, X)	No	No	No
commit(T)	Yes	Yes	*Yes
checkpoint	Yes	Undefined	Yes (of committed Ts)

*Yes: writing back to disk may occur not immediately.

4.6.2 Immediate Update

- In the immediate update techniques, the database may be updated by the operations of a transaction immediately, 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, so that recovery is possible.
- When immediate update is allowed, provisions must be made for undoing the effect of update operations on the database, because a transaction can fail after it has applied some updates to the database itself. Hence recovery schemes based on immediate update must include the capability to roll back a transaction by undoing the effect of its write operations.
  1. When a transaction starts, write an entry start_transaction(T) to the log;
  2. When any operation is performed that will change values in the database, write a log entry write_item(T, X, old_value, new_value);
  3. Write the log to disk;
  4. Once the log record is written, write the update to the database buffers;
  5. When convenient, write the database buffers to the disk;
  6. When a transaction is about to commit, write a log record of the form commit(T);
  7. Write the log to disk.
- The protocol and how different entries are affected can be best summarised below:

Log entry	Log written to disk	Changes written to database buffer	Changes written on disk
start_transaction(T)	No	N/A	N/A
read_item(T, X)	No	N/A	N/A
write_item(T, X)	Yes	Yes	*Yes
commit(T)	Yes	Undefined	Undefined
checkpoint	Yes	Undefined	Yes (of committed Ts)

*Yes: writing back to disk may occur not immediately.

Difference between Deferred update and Immediate update:

- **Deferred update: (NO-UNDO, REDO)**
  - The database is not physically updated until after a transaction reaches its commit point.
  - UNDO is not needed
  - REDO may be necessary
- **Immediate update: (UNDO, NO-REDO)**
  - The database is physically updated before the transaction reaches its commit point.
  - UNDO may be necessary when a transaction fails
  - REDO is not needed.

4.6.3 Deferred Database Modification

- Deferred database modification technique stores the database modifications in the log.
- Execution of *write* operation is done when the transaction is in a partially committed state.
- According to the transaction state diagram, a transaction is in a partially committed state when the transaction completes executing the last instruction. Execution of transaction T_i proceed as follows:
  - Before T_i starts its execution record, a record <T_i start> is written to log. Write(X) operation of T_i results in writing a new record <T_i, X, V_2> to the log.
  - When T_i partially commits a record <T_i commit> is written to log. When T_i partially commits, the records associated with it in the log are used in executing the deferred writes. If a system crashes before the transaction completes its execution or if the transaction aborts, then the information on the log is ignored.
  - If some failure occurs while updating the database using log records, the log record is written in some stable storage, hence, the transaction can resume its database modification.
  - Using log the system can handle any failure that results in loss of information on volatile storage.

    For example,
    - **redo (T_i)** sets the value of all data items updated by transaction T_i to the new values. The set of data items and their respective new values can be found in the log. The redo operation must be idempotent i.e. executing it several times must be equivalent to executing it once.
    - A transaction can execute redo (T_i) if the log contains both the record <T_i start> and the record <T_i commit>.

- **Example:** Consider the transaction T_i, it transfers ₹ 50 from account A to account B. Original values of account A and B are ₹ 1000 and ₹ 2000 respectively. This transaction is defined as:

  T_0 : read (A);
  A:= A – 50;
  write (A);
  read (B);
  B:= B + 50;
  write (B).

- Consider the second transaction T_1 that withdraws ₹ 100 from account C. This transaction is defined as: (Assume that an original value of account C is 700).

  T_1 : read (C);
  C:= C – 100;
  write (C).

- These two transactions are executed serially <T_0, T_1>.
- The log containing relevant information on these two transactions is given below:

  <T_0 start>
  <T_0, A, 950>
  <T_0, B, 2050>
  <T_0 commit>
  <T_0 start>
  <T_1, C, 600>
  <T_1 commit>

- It shows the log that results from the complete execution of T_0 and T_1.
- The actual output can take place in the database system in various orders. One such order is given below:

Log	Database
<T_0 start>	
<T_0, A, 950>	
<T_0, B, 2050>	
<T_0 commit>	
	A = 900
	B = 2050
<T_1 start>	
<T_1, C, 600>	
<T_1 commit>	C = 600

- If system crashes before the completion of transactions:

  Case 1: Crash occurs just after the log record for write (B) operation.

  Log contents after the crash are:

  <T_0 start>

  <T_0, A, 950>

  <T_0, B, 2050>

  <T_0 commit> is not written; hence no redo operation is possible.

  The values of account remain unchanged i.e. account balance of A is ₹ 1000 and B is ₹ 2000.

  Case 2: Crash occurs just after log record for write (C) operation.

  The log contents after the crash are:

  <T_0 start>

  <T_0, A, 950>

  <T_0, B, 2050>

  <T_0 commit>

  <T_1, start>

  <T_1, C, 600>

  When the system comes back it finds <T_0 start> and <T_0 commit>. But there is no <T_1 commit> for <T_0 start>. Hence, the system can execute redo (T_0) but not redo <T_1>. Hence, the value of account C remains unchanged.

  Case 3: Crash occurs just after the log record <T_1 commit>.

  The log contents after the crash are:

  <T_0 start>

  <T_0, A, 950>

  <T_0, B, 2050>

  <T_0, commit>

  <T_1 start>

  <T, C, 600>

  <T_1 commit>

- When the system comes back it can execute both redo (T_0) and redo (T_1) operations.
- For each commit record, the redo (T_i) operation is performed. redo (T_i) writes the values to the database independent of the values currently in the database. Hence, the redo (T_i) is idempotent.

4.6.4 Immediate Database Modifications

- It allows the database modifications to be output to the database while the transaction is still in active state.
- Database modifications written by active transactions are called uncommitted modifications.
- Execution of transaction proceeds as follows:
  1. Before T_i starts its execution, the record <T_i start> is written to the log.
  2. Before executing any write (X) i.e. before modifying the database for write operation, it writes an update record <T_i, X, V_1, V_2> to the log.
  3. When T_i partially commits, the record <T_i commit> is written to log.
- As an illustration consider the same example of bank accounts and transactions T_0 and T_1. Transactions T_0 and T_1 are executed serially.
- The log corresponding to this execution is given below:

  <T_0 start>
  <T_0, A, 1000, 950>
  <T_0, B, 2000, 2050>
  <T_0 commit>
  <T_1 start>
  <T_1, C, 700, 600>
  <T_1 commit>
- The order in which output took place to both database system and log as a result of execution of T_0 and T_1 is:

Data	Database
<T_0 start>	
<T_0, A, 1000, 950>	
<T_0, B, 2000, 2050>	
<T_0 commit> →	A = 950 B = 2050
<T_1 start>	
<T_1, C, 700, 600> →	C = 600
<T_1 commit >	

- Using the log, the system can handle any failure. Two recovery procedures are there to recover:
  1. **Undo (T_i):** It restores the value of all data items updated by transaction T_i to the old values.
  2. **Redo (T_i):** It sets the value of all data items updated by transaction T_i to the new values.

- The undo (T_i) and redo (T_i) operations must be idempotent.
- Depending on the log, the recovery scheme determines which transaction need to be redone and which need to be undone.
- If the log record contains the record <T_i start> but does not contain the record <T_i commit> then T_i needs to be undone.
- If the log record contains both the records <T_i start> and <T_i commit> then T_i needs to be redone i.e. If a failure occurs before <T_i commit> the database modifications are rolled back by undo (T_i) operation. If a failure occurs after <T_i commit> then the transaction is re-executed by redo (T_i) operation.
- Consider the following conditions of failure for transactions T_0 and T_1.

  Case 1: Failure occurs just after the log record for write (B) operation.

  Log contents and database are:

Log	**Database**
<T_0 start>	
<T_0 ,A, 1000, 950>	
<T_0, B, 2000, 2050>	
	A = 950
	B = 2050

- Log contains <T_0 start> but does not contain <T_0, commit>. Hence T_0 must be undone hence undo (T_0) is executed; the values of account A and B are restored to 1000 and 2000 respectively.

  Case 2: If some failure occurs just after the log record for write (C) T_i has written to log.

  The log and database contents are:

Log		**Database**
<T_0,start>		
<T_0 A, 1000, 950>		
<T_0 B, 2000, 2050>		
	→	A = 950
		B = 2050
<T_0 commit>		
<T_1 start>		
<T_1 C, 700, 600>	→	C = 600
		A = 950

- Log contains <T_0 start> and <T_0 commit>. Hence, the redo (T_0) is executed and values of accounts A and B are restored to the same (or new values) 950 and 2050 respectively. But the log doesn't contain <T_1 commit> for <T_1 start>. Hence, the value of account C is restored to old value by undo (T_1) operation. Hence value of C is ₹700.

Case 3: If system crashes just after the log record <T_1 commit> has been written to log.

- The log and database contents are:

Log		**Database**
<T_0 start>		
<T_0 A, 1000, 950>		
<T_0 B, 2000, 2050>		
	→	A = 950
		B = 2050
<T_0 commit >		
<T_1 start >		
<T_1 C, 700, 600 >		
	→	C = 600
<T_1 commit>		

- Log contains <T_0 start>, <T_0 commit> and <T_1 start> <T_1 commit> operations. Hence the recovery system executes redo(T_0) and redo(T_1) operations. The values of accounts A, B and C are ₹ 950, ₹ 2050 and ₹ 600 respectively.

4.7 BUFFER MANAGEMENT (S-18)

- A database buffer is a temporary storage area in main memory used to hold a copy of a database block. Database buffers are grouped in an area of memory called the buffer pool.
- The assignment and management of the operating system that performs this task is called buffer management.
- The buffer manager of a DBMS is the software component responsible for using a limited amount of main storage as disk page buffers, thereby reducing the number of disk I/O per database transaction.
- The buffer manager is responsible for the efficient management of the database buffers that are used to transfer (flushing) pages between buffer and secondary storage.

1. **Steal versus No-Steal Buffer Management :**

- A page with modifications by an uncommitted transaction is a dirty page until either commit or rollback processing for that transaction has been completed. The buffer manager can either distinguish dirty pages from clean pages when deciding which page to remove from the buffer pool, or it can ignore the update status of a page.
- In the later case, the buffer manager uses a **Steal policy**, which means pages can be written out to disk even if the transaction having modified the pages is still active.

- The alternative is the **No-steal policy**, in which case all dirty pages are retained in the buffer pool until the final outcome of the transaction has been determined.

Advantages:

- The steal policy implies that rollback of a transaction requires access to pages on disk in order to re-establish their old state.
- With the no-steal policy, no page on disk ever has to be touched when rolling back a transaction.

2. Force versus No-Force Buffer Management :

- Force versus No-force concerns writing of clean pages from the buffer pool.
- There are two basic approaches:
  - **Force policy:** When a transaction commits, the buffer manager locates all pages modified by that transaction and immediately writes the pages to disk.
  - **No-force policy**: A page whether modified or not, stays in the buffer as long as it is still needed. With a no-force policy, when a transaction commits, the changes made to the actual pages are not "forced", that is, required to be written to disk in-place.

Advantage:

- It avoids any REDO recovery during restart. If a transaction is successfully committed, then, by definition, all its modified pages must be on disk.

4.7.1 Log Record Buffering

- We discussed earlier that every log record is output to stable storage at the time it is created. This assumption causes a high overhead on system execution for the following reasons:
  1. Typically, output to stable storage is in units of blocks. In many cases, a log record is much smaller than a block. Thus, the output of each log record translates to a much larger output at the physical level.
  2. Due to the use of log buffering, a log record may be stored in only main memory for a considerable time before it is output to stable storage.
  3. Since such log records are lost if the system crashes or fails we must impose additional requirements on the recovery techniques to ensure transaction atomicity.
  4. The following rules must be followed if log records are buffered:
     a. T_i Transaction enters that commit state after the <T_i commit> log record has been output to stable storage.
     b. Before the <T_i commit> log record can be output to stable storage, all log records belonging to transaction T_i must have been output to stable storage.

c. Before a block of data in main memory can be output to the database, all log records belonging to data in that block must have been output to stable storage.

- This rule is called the Write-Ahead Logging or WAL rule. WAL only requires undo information to be output.

4.7.2 Database Buffering

- The database is stored in non-volatile storage such as disk, tape etc. and blocks of data are brought into main memory as needed.
- Since, main memory is typically much smaller than the entire database, it may be necessary to overwrite B_1 block in main memory when another B_2 block needs to be brought into memory.
- If B_1 block has been changed, B_1 must be output before the input of B_2. This storage hierarchy is the standard operating system concept of virtual memory.
- The rules for the output of log records limit the freedom of the system to output blocks of data. If the input of B_2 block causes B_1 block to be chosen for output, all log records being brought to data in B_1 block must be output to stable storage before B_1 block is output. Thus, the sequence of actions by the system would be as given below:
  - Output log records to stable storage until all log records belonging to B_1 block have been output.
  - Output B_1 block to disk.
  - Input B_2 block from disk to main memory.
- It is very important that no writes to the block B_1 be in progress while the preceding sequence of actions is carried out. Users can ensure that there are no writes in progress by using a special means of locking.

4.8 RELATIONSHIP BETWEEN RECOVERY MANAGEMENT AND BUFFER MANAGEMENT (W-18)

- DBMS application program requires Input/Output (I/O) operations which are performed by a component of the operating system. These I/O operations normally use buffers.
- The recovery management system of the DBMS is responsible for recovery from hardware or software failures.
- The recovery manager ensures that the database remains in a consistent state in the presence of failures. It is responsible for transaction commit and abort operations, maintaining a log, and restoring the system to a consistent state after a crash.
- The buffer management effectively provides a temporary copy of a database page. Therefore, it is used in database recovery systems in which the modifications are

done in this temporary copy and the original page remains unchanged in the secondary storage.

- Fig. 4.1 shows layers of DBMS related to buffer management and recovery management.

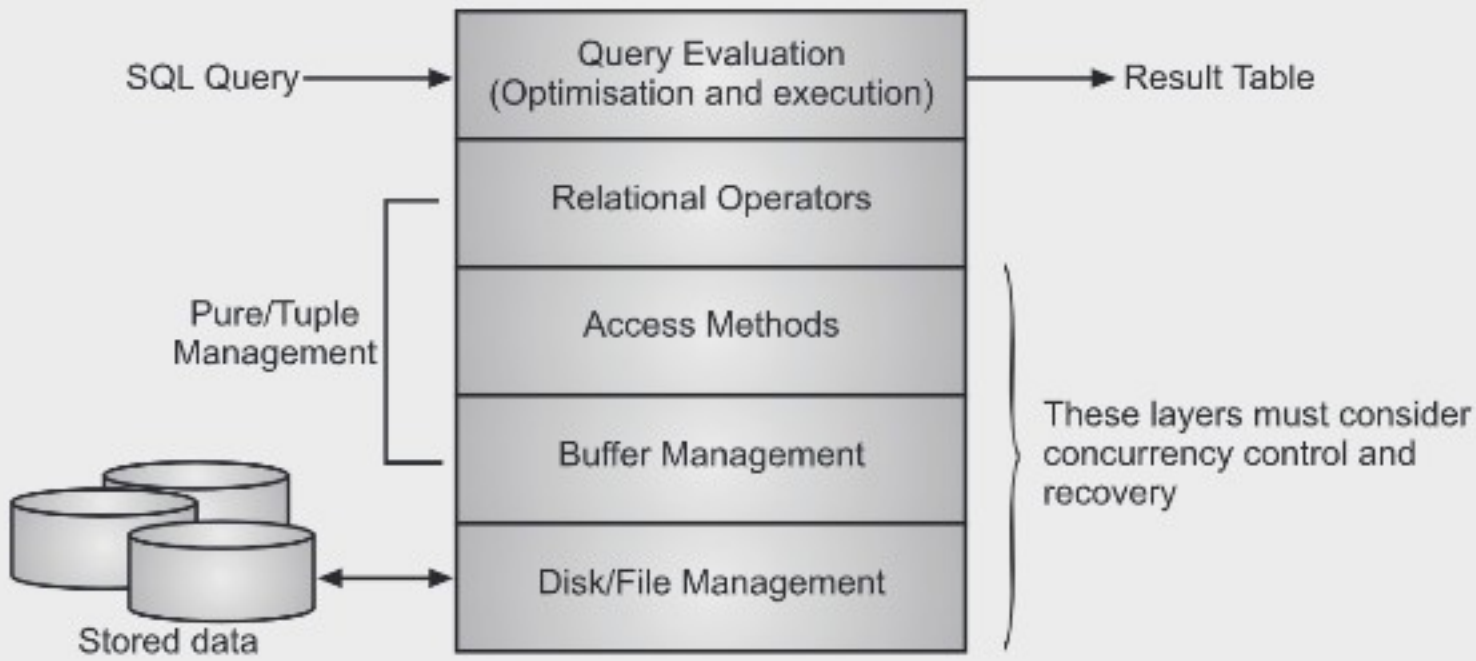


Fig. 4.1: Layers of DBMS related to Buffer and Recovery Management

4.9 ARIES ALGORITHM (S-18, 19)

- ARIES (Algorithms for Recovery and Isolation Exploiting Semantics) uses a steal/no-force approach for writing, and it is based on following three concepts:
  1. **Write-ahead Logging:** Any change to an object is first recorded in the log, and the log must be written to stable storage before changes to the objects are written to disk.
  2. **Repeating History During Redo:** ARIES will retrace all actions of the database system prior to the crash to reconstruct the database state when the crash occurred. Transactions that were uncommitted at the time of the crash (active transactions) are undone.
  3. **Logging Changes during Undo:** Will prevent ARIES from repeating the completed undo operations if a failure occurs during recovery, which causes a restart of the recovery process.

ARIES Recovery Procedure:

- The ARIES recovery procedure consists of three main steps (phases):

  Step 1: Analysis: The analysis phase 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.

  Step 2: REDO: 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.

Step 3. UNDO: 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.

4.10 DATABASE BACKUP AND RECOVERY FROM CATASTROPHIC FAILURE (S-18)

- The main technique used to handle catastrophic failures including disk crash is database backup.
- The whole database and the log are periodically copied onto a storage medium such as magnetic tapes. In case of a catastrophic system failure, the latest backup copy can be reloaded from the tape to the disk, and the system can be restarted.
- To avoid losing all the effects of transactions that have been executed since the last backup, it is normal to back up the system log by periodically copying it to magnetic tape.
- The system log is usually smaller than the database itself and hence can be backed up more frequently.
- When the system log is backed up, users do not lose all transactions they have performed since the last database backup.
- All committed transactions recorded in the portion of the system log that has been backed up can have their effect on the database reconstructed.
- A new system log is started after each database backup operation. Hence, to recover from disk failure, the database is first recreated on disk from its latest backup copy on tape.

4.11 SHADOW PAGING (W-18, S-19)

- Shadow paging is an alternative to log-based crash recovery technique. This is one possible form of indirect page allocation.
- Paging scheme is used in operating systems for virtual memory management. The memory that is addressed by a process is called Virtual Memory.

- In the shadow page scheme, the database is considered to be made up of logical units of storage called **pages** which are assumed to be of a certain size (1 KB or 4 KB). The virtual or logical pages are mapped onto physical memory blocks of same size. The mapping of pages is provided by table called a Page Table.

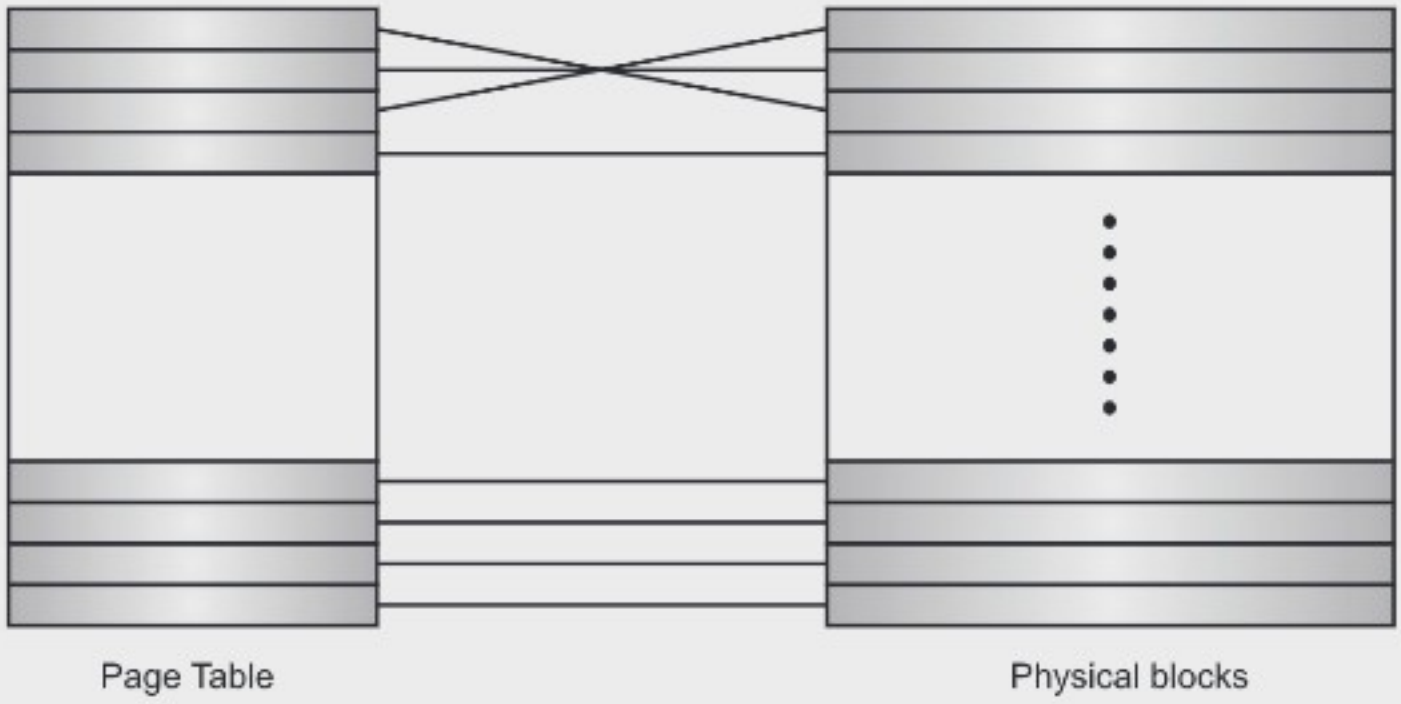


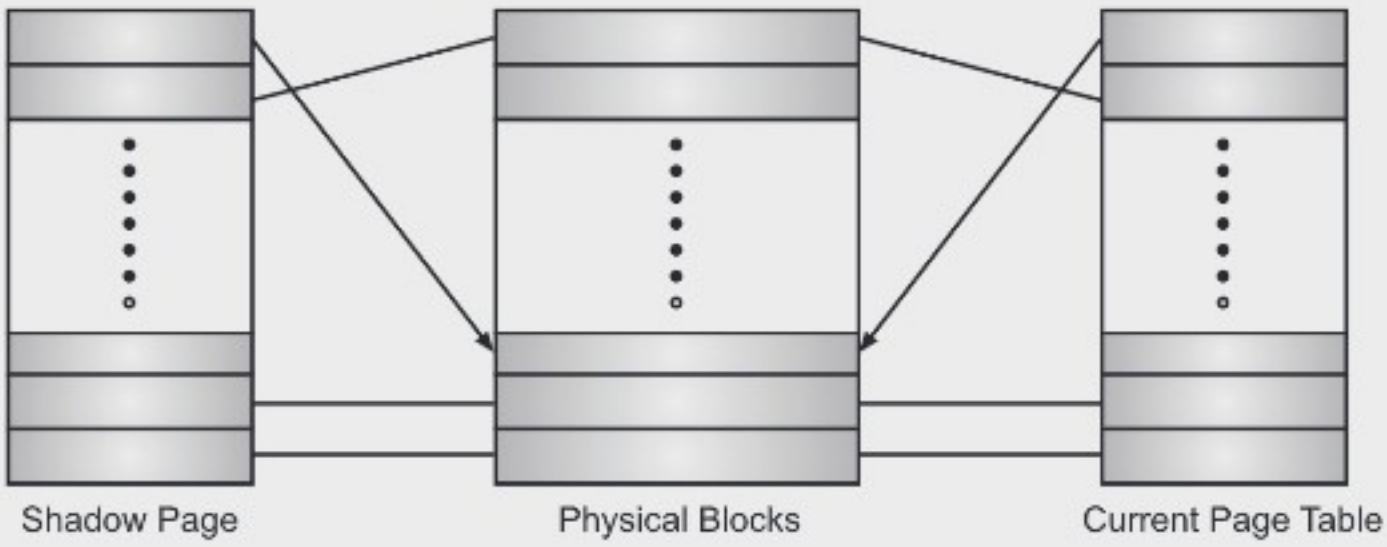
Fig. 4.2: Page Table

- Page table contains one entry for each logical page of the process's virtual address space. This entry contains the block number of physical storage where this page is stored. Page table is shown in the Fig. 4.2.
- The shadow page scheme uses two page tables.
  1. Current page table.
  2. Shadow page table.
- Transaction addresses the database using the current page table. It may change the current page table entries. The changes are made whenever the transaction executes the write operation.
- To modify a page, it copies that page to new blocks of physical storage. The page table entry corresponding to that page is made to point to a new block of storage.
- The shadow page table is the original page table. It contains the entries that existed prior to the start of the transaction. It remains unaltered by the transaction and it is used for undoing the transaction. Now, let us see how the transaction accesses data.

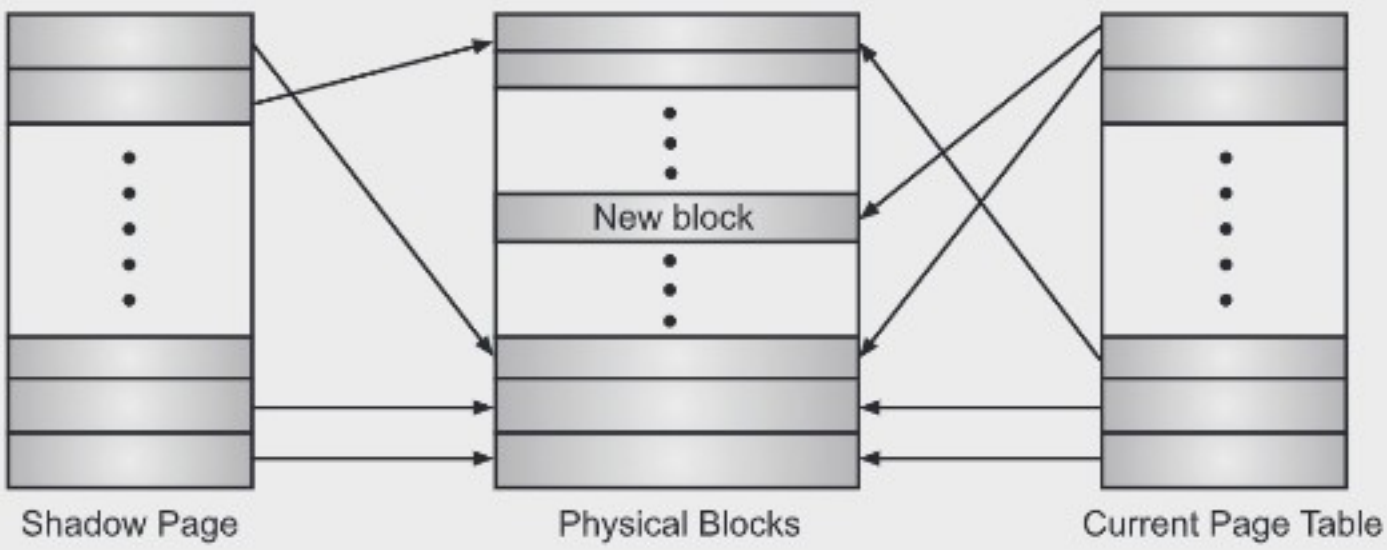
Data access by transaction:

- The transaction uses the current page table to access the database blocks. The shadow paging scheme handles a write operation of transaction as follows:
  1. A free block of non-volatile storage is located from the pool of free blocks accessible by the database system.
  2. The block to be modified is copied onto this block.

3. The original entry in the current page table is changed to point to this new block.
4. The updates are propagated to the block pointed to by the current page table which in this case would be a newly created block.



(a) Before Starting the Transaction



(b) After Executing Write Operation

Fig. 4.3

- Any changes made to the database are propagated to the blocks pointed to by the current page table. Once a transaction commits, all modifications made by the transaction and still in buffers are propagated to the physical database. It causes the current page table to be written to non-volatile storage.
- In case of a system crash before the transaction commits, the shadow page table and the corresponding blocks containing the old database will continue to be accessible.
- The old database is made accessible just by modifying a single pointer from shadow page table to current page table.
- Once, the transaction completes its execution successfully, the shadow block can be returned to the pool of available non-volatile storage blocks to be used for further transactions.

Advantages of Shadow Paging:

1. Recovery from system crash is relatively inexpensive and this is achieved without the overhead of logging.
2. In shadow paging, the overhead of maintaining the transaction log file is eliminated.
3. In shadow paging, no need for undo or redo operations, recovery are significantly faster.

Disadvantages of Shadow Paging:

1. Over a period of time the database will be scattered over the physical memory and related records may require a very long access time.
2. When the transaction completes its execution, shadow blocks have to be returned to the pool of free blocks. If this is not done successfully such blocks become inaccessible when a transaction commits. This is called the Garbage Collection Operation.
3. The commit of a single transaction using shadow paging requires multiple blocks to be output (the actual data blocks, the current page table, and disk address of the current page table log based schemes need to output only the log records.

EXAMPLES

Example 1: Following are the log entries at the time of system crash:

[start _transaction, T_1]

[write_item, T_1, A, 10]

[commit T_1]

[start_transaction, T_3]

[write_item T_3, B, 15]

[checkpoint]

[commit T_3]

[start_transaction, T_2]

[write_item T_2, B, 20]

[start_transaction, T_4]

[write_item, T_4, D, 25]

[write_item T_2, C, 30] ← system crash

If a deferred update technique with checkpoint is used, what will be the recovery procedure?

Solution:

Transactions T_2 and T_4 are ignored, because they did not reach their commit points. i.e. T_2 and T_4 do not have commit instructions before the system crash.

T_1 is committed before the last system checkpoint; hence it need not be redone. T_3 is redone because its commit point is after the last system checkpoint.

Example 2: Following are the log entries at the time of system crash:

[start_transaction, T_1)

[write_item, T_1, B, 100]

[commit, T_1]

[checkpoint]

[start_transaction, T_2]

[write_item, T_2, D, 100]

[commit, T_2]

[start_transaction, T_3]

[write_item, T_3, D, 200]

[write _item, T_3, B, 200] ← system crash.

If a deferred update technique with checkpoint is used, what will be the recovery procedure?

Solution:

Transaction T_3 is ignored, because it did not reach its commit point. i.e. T_3 does not have commit instruction before the system crash.

T_1 is committed before the last system checkpoint; hence it need not be redone. T_2 is redone because its commit point is after the last system checkpoint.

Example 3: Following are the log entries at the time of system crash:

[start_transaction, T_1]

[write_item, T_1, A, 10, 100]

[commit, T_1]

[checkpoint]

[start_transaction, T_2]

[write_item, T_2, B, 20, 200]

[commit, T_2]

[start_transaction, T_3]

[write_item, T_3, C, 30, 300] ← System crash

If immediate update technique with a checkpoint is used, what will be the recovery procedure?

Solution:

Log contains start instruction for T_3, but does not contain commit instructions for T_3 before the system crash. Hence, T_3 needs to be undone.

A log contains start and commits instruction for T_2 before the system crash. Hence, T_2 need to be redone.

T_1 is committed before the last system checkpoint. Hence, it need not be redone.

Example 4: Consider the log given below corresponding to a schedule S. Specify which transactions are rolled back, which operations in the log are redone and which (if any) are undone.

[start_transaction, T_1]
[read_item, T_1, A]
[read_item, T_1, D]
[write_item, T_1, D, 20]
[commit T_1]
[check point]
[start_transaction, T_2]
[read_item, T_2, B]
[write_item T_2, B, 12]
[start_transaction, T_4]
[read_item,T_4, B]
[write_item, T_4, B, 15]
[start_transaction, T_3]
[write_item T_3, A, 30]
[read_item, T_4, A]
[write_item, T_4, A, 20]
[commit, T_4]
[read_item, T_2, D]
[write_item, T_2, D, 25] ← System Crash

Solution:

We now describe the immediate update protocol:

Since, there is a checkpoint after {commit, T_1}, all redo and undo operations are done for the entry after this checkpoint.

In case of immediate method, the following transactions are redone → T_4, and follow transactions are undone → T_2, T_2.

In case of deferred update modification method, the following transactions are redone → T_4.

No transactions are undone, since in deferred update method, the actual updation to the database takes place only when transaction issues commit.

Example 5: Consider the following log image that is obtained during recovery after a crash:

<T_1, start>

<T_1, X, 10, 10>

<T_1, Y, 20, 5>

<T_2, start>

<T_2, X, 20, 200>

<T_1, commit>

<T_3, start>

<T_3, Z, 10, 20>

<Checkpoint>

<T_3, K, 20, 200>

<T_2, commit>

<T_4, start>

<T_4, X, 200, 100> ← System Crash

(I) List the contents in the list L.

(ii) List the contents in:

(a) Undo-list, and

(b) Redo-list.

Solution:

(i) The contents in the list L are: {T_2, T_3, T_4}

(ii) (a) Undo List = {T_4}

(b) Redo List = {T_2}.

Example 6: Consider the log image of Example 1. State how the recovery procedure takes place using the immediate update algorithm and using the deferred update algorithm.

Solution: The list L, the undo list and the redo list are given in the previous example.

L = {T_2, T_3, T_4}, Undo list = {T_4}, Redo List = {T_2}

Next the recovery manager updates undo and redo lists as per the contents in the list L.

List contains T_2, T_3 out of which T_2 is in redo list, but T_3 is not. Hence, it adds T_3 to undo-list, so now the undo list is = {T_3, T_4}.

Immediate Update Method: As per this method, actions of active transactions have to be undone. Hence, the recovery procedure is as follows:

(i) The recovery manager scans the log backwards and for each record of each transaction in the undo list, it undoes the action.

Here, Undoing <T_4, X, 200, 100>, leads

X = 200.

T_4 undo completes, since we reached <T_4, start>

Undoing <T_3, K, 20, 200> leads

K = 20.

Undoing <T_3, Z, 10, 20> leads

Z = 10.

At this point T_3's undo operations are completed, since we have reached <T_3, start> records.

(ii) Next recovery manager scans log forward from the most recent checkpoint record and redoes the actions of T_2, since only T_2 is in the redo list.

Deferred Update Method: In this method, only redo of committed transactions takes place. There is no undo of active transactions, since their actions don't affect the database at all. Hence here, the log is scanned forward from the most recent checkpoint, and actions of the transaction T_2 are redone.

Summary

- Recovery means restoring data into its previous state before the incident.
- Database recovery is the process of restoring the database to a correct (consistent) state in the event of a failure.
- Crash recovery is the process by which the database is moved back to a consistent and unable state.
- If strict two-phase locking is used for concurrency control, locks held by a transaction may be released only after the transaction has been rolled back.
- If a transaction executes all its operations successfully, it is said to be *committed*. All its effects are now permanently established on the database system.
- Commit command is used to permanently save any transaction into the database.
- The log-based recovery techniques maintain transaction logs to keep track of all update operations of the transaction.
- Deferred database modification technique stores the database modifications in the log.
- Log is a structure used to store the database modifications.
- Immediate database modification allows the database modifications to be output to the database while the transaction is still in active state.

- ARIES (Algorithms for Recovery and Isolation Exploiting Semantics) uses a steal/no-force approach for writing
- The main technique used to handle catastrophic failures including disk crash is database backup.
- Shadow paging is an alternative to log-based crash recovery technique. This is one possible form of indirect page allocation.

Check Your Understanding

1. Log records contain _____.
   (a) Old value (b) New value
   (c) Both A and B (d) Error value
2. An integral part of database that can restore the database to the consistent state of before failure is called _____.
   (a) Recovery scheme (b) Backup scheme
   (c) Restoring scheme (d) Transaction scheme
3. Which of the following is not a recovery technique?
   (a) Deferred update (b) Immediate update
   (c) Two-phase commit (d) Recovery management
4. _____ is an alternative of log based recovery.
   (a) Disk recovery (b) Shadow paging
   (c) Dish shadowing (d) Crash recovery
5. The subsystem responsible for the allocation of buffer space is called as _____
   (a) Buffer allocator (b) Buffer manager
   (c) Buffer enhancer (d) Buffer intermediary
6. The log is a sequence of _____ recording all the update activities in the database.
   (a) Log records (b) Records
   (c) Entries (d) Redo
7. In the _____ scheme, a transaction that wants to update the database first creates a complete copy of the database.
   (a) Shadow copy (b) Shadow Paging
   (c) Update log records (d) All of the mentioned
8. The _____ scheme uses a page table containing pointers to all pages; the page table itself and all updated pages are copied to a new location.
   (a) Shadow copy (b) Shadow Paging
   (c) Update log records (d) All of the mentioned

9. The current copy of the database is identified by a pointer, called _____ which is stored on disk.
   (a) Db-pointer (b) Update log
   (c) Update log records (d) All of the mentioned
10. If a transaction does not modify the database until it has committed, it is said to use the _____ technique.
   (a) Deferred-modification (b) Late-modification
   (c) Immediate-modification (d) Undo
11. _____ using a log record sets the data item specified in the log record to the old value.
   (a) Deferred-modification (b) Late-modification
   (c) Immediate-modification (d) Undo
12. In the _____ phase, the system replays updates of all transactions by scanning the log forward from the last checkpoint.
   (a) Repeating (b) Redo
   (c) Replay (d) Undo
13. The actions which are played in the order while recording it is called _____ history.
   (a) Repeating (b) Redo
   (c) Replay (d) Undo

ANSWER KEY

1. (c)	2. (a)	3. (c)	4. (b)	5. (b)	6. (a)	7. (a)	8. (b)
9. (a)	10. (a)	11. (d)	12. (b)	13. (a)			

Practice Questions

Q.I. Answer the following questions in short.

1. What is meant by recovery?
2. Enlist various types of errors.
3. Enlist various types of transaction failures.
4. What is meant by log record?
5. What is a system crash?
6. What are STEAL and NO-STEAL policies?
7. What are FORCE and NO-FORCE policies?

Q.II. Answer the following questions.

1. Explain failure classification in detail.
2. With a neat diagram explain shadow paging.
3. Explain ARIES algorithm.
4. Describe immediate database modification with example.
5. What is a system crash? Explain in detail.
6. With the help of diagrams explain the relation between recovery management and buffer management.
7. Write short notes on:
   (i) Checkpoint.
   (ii) Buffer management.
   (iii) Deferred update.

Q.III. Define following terms.

1. Commit
2. Rollback
3. Disk failure
4. Log
5. Log-based buffering.
6. Recovery and Atomicity
7. Transaction Failure
8. System and logical error
9. WAL rules
10. Database buffering

Previous Exams Questions

Summer 2019

1. List any two advantages of shadow paging. **(1M)**

Ans. Refer to Section 4.11

2. Define checkpoint. **(1M)**

Ans. Refer to Sections 4.5.2

3. Explain the concepts of Aries Algorithm. **(3M)**

Ans. Refer to Section 4.9

4. Consider the following log image, that is obtained during recovery after a crash : **(5M)**

< T1, Start >

< T1, X, 10, 10 >

< T1, Y, 20, 5 >

< T2, Start >

< T2, X, 20, 200 >

< T1, Commit >

< T3, Start >

< T3, Z, 10, 20 >

< Checkpoint >

< T3, K, 20, 200 >

< T2, Commit >

< T4, Start >

< T4, X, 200, 100 > ← system crash.

Ans. Refer to 'Examples' Section

5. Explain the concepts of Aries Algorithm. **(3M)**

Ans. Refer to Section 4.9

6. Explain Log based Recovery. **(4M)**

Ans. Refer to Section 4.6

Winter 2018

1. What is a log record ? **(1M)**

Ans. Refer to Section 4.4

2. What are checkpoints ? How are they useful in crash recovery? **(4M)**

Ans. Refer to Section 4.5.2

3. What is the relationship between Recovery manager and Buffer management?

Ans. Refer to Section 4.8 **(3M)**

4. Consider the following log image, state how the recovery procedure takes place using immediate update algorithm : **(5M)**

< T1, start >

< write_item T1 X 20 >

< T2, start >

< write_item T1 Y 300 >

< T1, commit >

< check point >

< T3, start >

< write_item T2 A 500 >

< write_item T3 X 82 >

< write_item T2 B 45 >

< T4, start >

< T2, commit >

< write_item T4 Y 50 > X ← system crash.

Ans. Refer to 'Examples 'Section'.

5. Explain different types of failures in detail. **(4M)**

Ans. Refer to Section 4.2

6. What is shadow paging ? State any two disadvantages of shadow paging. **(4M)**

Ans. Refer to Section 4.11

Summer 2018

1. Which technique is used to restore a database upto the last consistent state after system failure. **(1M)**

(a) Backup (b) Recovery

(c) Both (i) and (ii) (d) None

Ans. (b)

2. What are various entries in System log. **(1M)**

Ans. Refer to Section 4.10.

3. What do you mean by backup? **(1M)**

Ans. Refer to Section 4.10.

4. Write a short note on log based recovery. **(4M)**

Ans. Refer to Section 4.6.

5. Write a short note on checkpoints. **(3M)**

Ans. Refer to Section 4.5.2

6. Explain the recovery with concurrent transaction. **(5M)**

Ans. Refer to Section 4.5

7. Write a short note on Aries algorithm. **(4M)**

Ans. Refer to Section 4.9.

8. Write short note on Buffer Management. **(3M)**

Ans. Refer to Section 4.7.

❖❖❖

5... Database Security

Objectives...

- To understand Meaning of Database Security and Threats to Database Security
- To learn Methods for Database Security
- To know how Access Control can be represented
- To know how View can be Enforced as Security Device
- To learn Overview of Encryption Techniques
- To understand Importance of Statistical Database Security

5.1 INTRODUCTION

- Security is an important issue in database management because information stored in a database is valuable and sensitive resource.
- Due to the high value of data incorporated databases, there is strong motivation for unauthorized users to gain access to it, for instance, competitors or dissatisfied employees. So the data in a database management system needs to be protected from unauthorized access, alteration or deletion.

5.2 DATABASE SECURITY CONCEPTS (S-18)

- Database security can be defined as, "protecting a database from unauthorized access, malicious destruction and even any accidental loss or misuse."

Threats to Database Security:

- Threat is any intentional or accidental event that may adversely affect the system.
- For example, using another person's log-in name to access data, unauthorized copying data, program/data alteration, Illegal entry by hacker, viruses etc.
- The major database security threats are as follows:
  1. **Loss of Confidentiality:** Loss of confidentiality indicates that unauthorized users have been able to access information.
  2. **Loss of Privacy:** It includes loss of protection of an individual's data files.

3. **Loss of Integrity:** It refers to data corruption and invalid data.
4. **Loss of Availability:** It includes the damage of hardware, the network or applications that may cause data to become unavailable to users.
5. **Theft and Fraud:** It includes intentional security breaking of data and unauthorized data manipulation because of inadequate access controls, inadequate physical security of the database.
6. **Accidental Losses:** It includes losses as a result of human error, software problems, and hardware problems.

Levels of security measures:

- To protect the database, we must take security measures at several levels:
  1. **Physical:** The websites containing the computer systems must be secured against armed or surreptitious entry by intruders.
  2. **Human:** Users must be authorized carefully to reduce the chance of any such user giving access to an intruder in exchange for a bribe or other favors.
  3. **Operating System:** No matter how secure the database system is, weakness in the operating system's security may serve as a means of unauthorized access to the database.
  4. **Network:** Since almost all database systems allows remote access through terminals or networks, software-level security within the network software is as important as physical security, both on the Internet and in networks private to an enterprise.
  5. **Database System:** Some database system users may be authorized to access only a limited portion of the database. Other users may be allowed to issue queries, but may be not allowed to modify the data. It is responsibility of the database system to ensure that these authorization restrictions are not disturbed.
- Security at all these levels must be maintained if database security is to be ensured. A weakness at a low level of security (physical or human) allows circumvention of strict high level (database) security measures.
- There are three main objectives to consider while designing a secure database application:
  1. **Secrecy**: Information should not be disclosed to unauthorized users. For example, a student should not be allowed to examine other students' grades.
  2. **Integrity**: Only authorized users should be allowed to modify data. For example, students may be allowed to see their grades, yet not allowed (obviously!) to modify them.
  3. **Availability**: Authorized users should not be denied access. For example, an instructor who wishes to change a grade should be allowed to do so.

- To achieve these objectives, a clear and consistent **security policy** should be developed to describe what security measures must be enforced. Next, the **security mechanisms** of the underlying DBMS must be utilized to enforce the policy.

5.3 METHODS FOR DATABASE SECURITY (S-18, W-18)

- There are different techniques to implement the database security as follows:

1. **Authorization:** A DBMS typically includes a database security and authorization sub-system that is responsible for ensuring the security of portions of a database against unauthorized access. It is normally refers two types of database security mechanisms:
   (i) **Discretionary Security Mechanisms:** These are used to grant privileges to users, including the capability to access specific data files, records, or fields in a specified mode (such as read, insert, delete, or update).
   (ii) **Mandatory Security Mechanisms:** These are used to enforce multilevel security by classifying the data and users into various security classes (or levels) and then implementing the appropriate security policy of the organization.
2. **Access Control:** The security mechanism of a DBMS must include provisions for restricting access to the database as a whole. This function is called Access Control. An access control mechanism is a way to control the data that is accessible to a given user. It is handled by creating user accounts and passwords to control login process by the DBMS.
3. **Statistical Database Security:** The security problem associated with databases is that of controlling the access to a statistical database, which is used to provide statistical information or summaries of values based on various criteria. The counter measures to statistical database security problem are called Inference Control Measures.
4. **Database Encryption Techniques:** These are used to protect sensitive data (such as credit card numbers) that is being transmitted via some type communication network. The data is encoded using some encoding algorithm. An unauthorized user who access encoded data will have difficulty deciphering it, but authorized users are given decoding or decrypting algorithms (or keys) to decipher data.

Database Security and the DBA:

- The Database Administrator (DBA) is the central authority for managing a database system.
- The DBA's responsibilities include:
  1. Granting privileges to users who need to use the system.
  2. Classifying users and data in accordance with the policy of the organization.
- The DBA is responsible for the overall security of the database system. The DBA has a DBA account in the DBMS. Sometimes, these are called a System or Super User Account.

Capabilities of DBA account:

- DBA accounts provide powerful capabilities such as:

  Action 1. Account Creation: This action creates a new account and password for a user or a group of users to enable them to access the DBMS.

  Action 2. Privilege Granting: This action permits the DBA to grant certain privileges to certain accounts.

  Action 3. Privilege Revocation: This action permits the DBA to revoke (cancel) certain privileges that were previously given to certain accounts.

  Action 4. Security Level Assignment: This action consists of assigning user accounts to the appropriate security classification level.
- Action 1 is access control, whereas 2 and 3 are discretionary security mechanisms and 4 is used to control mandatory authorization security mechanisms.
- If any interfering with the database is suspected, a database audit is performed. A database audit consists of reviewing the log to examine all accesses and operations applied to the database during a certain time period.
- A database log that is used mainly for security purposes is sometimes called an audit trail.

5.3.1 Discretionary Access Control Method

- Discretionary Access Control (DAC) in a database is a type of security access control that is based on the granting and revoking privileges.
- A privilege is an action, such as creating, executing, reading, updating or deleting that a user is permitted to perform on database objects.
- A user who creates a database object such as a table or a view automatically gets all applicable privilege on that object. The DBMS keeps track of how these privileges are granted to other users.

5.3.1.1 Types of Discretionary Privileges

- The DBMS must provide selective access to each relation in the database based on specific accounts.
- Operations may also be controlled; thus having an account does not necessarily entitle the account holder to all the functionality provided by the DBMS.
- There are two levels for assigning privileges to use the database system:
  1. The **Account Level**: At this level, the DBA specifies the particular privileges that each account holds independently of the relations in the database.
  2. The **Relation Level** (or **Table Level**): At this level, the DBA can control the privilege to access each individual relation or view in the database.

- The privileges at the **account level** apply to the capabilities provided to the account itself and can include :
  a. The **CREATE SCHEMA** or **CREATE TABLE** privilege, to create a schema or base relation.
  b. The **CREATE VIEW** privilege.
  c. The **ALTER** privilege, to apply schema changes such adding or removing attributes from relations.
  d. The **DROP** privilege, to delete relations or views.
  e. The **MODIFY** privilege, to insert, delete, or update tuples.
  f. The **SELECT** privilege, to retrieve information from the database by using a SELECT query.
- The second level of privileges applies to the **relation level** which includes base relations and virtual (view) relations.

Access Matrix or **Authorization Matrix Model**:

- The granting and revoking of privileges generally follows an authorization model for discretionary privileges known as the Access Matrix or Authorization Matrix Model, where
  1. The rows of a matrix M represent subjects (users, accounts, programs).
  2. The columns represent objects (relations, records, columns, views, operations).
  3. Each position M (i, j) in the matrix represents the types of privileges (read, write, update) that subject i holds on object j.
- To control the granting and revoking of relation privileges, each relation R in a database is assigned and owner account, which is typically the account that was used when the relation was created in the first place.
- The owner of a relation is given all privileges on that relation. The DBA can assign and owner to a whole schema by creating the schema and associating the appropriate authorization identifier with that schema, using the CREATE SCHEMA command.
- The owner account holder can pass privileges on any of the owned relation to other users by granting privileges to their accounts.

Types of privileges in SQL :

- In SQL the following types of privileges can be granted on each individual relation R:
  1. **SELECT** (retrieval or read) privilege on R:
     - Gives the account retrieval privilege.
     - In SQL, this gives the account the privilege to use the SELECT statement to retrieve tuples from R.

2. **MODIFY** privilege on R:
   - This gives the account the capability to modify tuples of R.
   - In SQL, this privilege is further divided into UPDATE, DELETE, and INSERT privileges to apply the corresponding SQL command to R.
   - In addition, both the INSERT and UPDATE privileges can specify that only certain attributes can be updated by the account.
3. **REFERENCES** privilege on R:
   - This gives the account the capability to reference relation R when specifying integrity constraints.
   - The privilege can also be restricted to specific attributes of R.
   - To create a view, the account must have SELECT privilege on all relations involved in the view definition.

Specifying Privileges Using Views:

- The mechanism of views is an important discretionary authorization mechanism in its own right. For example,
  - If the owner A of a relation R wants another account B to be able to retrieve only some fields of R, then A can create a view V of R that includes only those attributes and then grant SELECT on V to B.
  - The same applies to limiting B to retrieving only certain tuples of R; a view V' can be created by defining the view by means of a query that selects only those tuples from R that A wants to allow B to access.

Granting Privileges:

SQL GRANT command:

- SQL GRANT command is used to provide access or privileges on the database objects to the users. Thus grant command is used to confer authorization.
- It has the following **syntax:**

```
GRANT privilege_name
ON  < tablename | viewname>
TO {user list |role_name}
[WITH GRANT OPTION];
```

Where, `privilege_name` is the access right or privilege granted to the user like SELECT, INSERT, UPDATE, DELETE etc.

`WITH GRANT OPTION` - allows a user to grant access rights to other users.

Propagation of Privileges using the GRANT OPTION:

- Whenever, the owner A of a relation R grants a privilege on R to another account B, privilege can be given to B with or without the GRANT OPTION. If the GRANT OPTION is given, this means that B can also grant that privilege on R to other accounts.

- Suppose that B is given the GRANT OPTION by A and that B then grants the privilege on R to a third account C, also with GRANT OPTION. In this way, privileges on R can propagate to other accounts without the knowledge of the owner of R.
- If the owner account A now revokes the privilege granted to B, all the privileges that B propagated based on that privilege should automatically be revoked by the system.

Revoking Privileges:

- In some cases, it is desirable to grant a privilege to a user temporarily. For example, the owner of a relation may want to grant the SELECT privilege to a user for a specific task and then revoke that privilege once the task is completed. Hence, a mechanism for revoking privileges is needed.

REVOKE command:

- In SQL, a REVOKE command is included for the purpose of canceling privileges.
- The revoke command is used to remove or cancel the given privileges from the user.

```
REVOKE privilege_name
ON < tablename | viewname>
FROM { user list |role_name} [RESTRICT |CASCADE ]
```

- If we use `RESTRICT` keyword, the privilege will be revoked only from the specified user. If the specified user granted had the `WITH GRANT OPTION` and granted the same privilege to other users, they will retain the privilege.
- If we use `CASCADE`, it will revoke the privilege and any dependent privileges as a result of our grant. A dependent privilege is one that could exist, if we granted the privilege that we were trying to revoke, which is what we are trying to achieve as a result of our REVOKE statement.
- **Example:** Suppose that A1 creates the two base relations EMPLOYEE and DEPARTMENT. A1 is then owner of these two relations and hence all the relation privileges on each of them.

EMPLOYEE Relation

NAME	SSN	BDATE	ADDRESS	SALARY	DNO

DEPARTMENT Relation

DNUMBER	DNAME	MGR_SSN

- Suppose that A1 wants to grant A2 the privilege to insert and delete tuples in both of these relations, but A1 does not want A2 to be able to propagate these privileges to additional accounts:

```
GRANT INSERT, DELETE ON
EMPLOYEE, DEPARTMENT TO A2;
```

- Suppose that A1 wants to allow A3 to retrieve information from either of the two tables and also to be able to propagate the SELECT privilege to other accounts. A1 can issue the command:

```
GRANT SELECT ON EMPLOYEE, DEPARTMENT
TO A3 WITH GRANT OPTION;
```

- A3 can grant the SELECT privilege on the EMPLOYEE relation to A4 by issuing:

```
GRANT SELECT ON EMPLOYEE TO A4;
```

- Notice that A4 can't propagate the SELECT privilege because GRANT OPTION was not given to A4
  - Suppose that A1 decides to revoke the SELECT privilege on the EMPLOYEE relation from A3; A1 can issue:

```
REVOKE SELECT ON EMPLOYEE FROM A3 CASCADE;
```

- The DBMS must now automatically revoke the SELECT privilege on EMPLOYEE from A4, too, because A3 granted that privilege to A4 and A3 does not have the privilege any more.
  - Suppose that A1 wants to give back to A3 a limited capability to SELECT from the EMPLOYEE relation and wants to allow A3 to be able to propagate the privilege. The limitation is to retrieve only the NAME, BDATE, and ADDRESS attributes and only for the tuples with DNO=5. Then A1 create the view as:

```
CREATE VIEW A3 EMPLOYEE AS
        SELECT NAME, BDATE, ADDRESS
        FROM EMPLOYEE
        WHERE DNO = 5;
```

  - After the view is created, A1 can grant SELECT on the view A3EMPLOYEE to A3 as follows:

```
GRANT SELECT ON A3EMPLOYEE TO A3 WITH GRANT OPTION;
```

  - Suppose that A1 wants to allow A4 to update only the SALARY attribute of EMPLOYEE; A1 can issue:

```
GRANT UPDATE ON EMPLOYEE (SALARY) TO A4;
```

- The UPDATE or INSERT privilege can specify particular attributes that may be updated or inserted in a relation. Other privileges (SELECT, DELETE) are not attribute-specific.

5.3.2 Mandatory Access Control and Role-based Access Control for Multilevel Security (W-18)

- The discretionary access control techniques of granting and revoking privileges on relations have traditionally been the main security mechanism for relational database systems. This is an all-or-nothing method because a user either has or does not have a certain privilege.

- In many applications, and additional security policy is needed that classifies data and users based on security classes. This approach as mandatory access control would typically be combined with the discretionary access control mechanisms.
- In this method:
  - Typical security classes are Top Secret (TS), Secret (S), Confidential (C), and Unclassified (U), where TS is the highest level and U the lowest: TS ≥ S ≥ C ≥ U.
  - The commonly used model for multilevel security, known as the Bell-LaPadula model, classifies each subject (user, account, program) and object (relation, tuple, column, view, operation) into one of the security classifications, T, S, C, or U.
  - Clearance (classification) of a subject S as class(S) and to the classification of an object O as class(O).
  - Two restrictions are enforced on data access based on the subject/object classifications:
    1. **Simple security property:** A subject S is not allowed read access to an object O unless class(S) ≥ class(O).
    2. **Star property (or * property):** A subject S is not allowed to write an object O unless class(S) ≤ class(O).
- The first restriction enforces the rule that no subject can read an object whose security classification is higher than the subject's security clearance.
- The second restriction is less intuitive. It prohibits a subject from writing an object at a lower security classification than the subject's security clearance. Violation of this rule would allow information to flow from higher to lower classifications, which violates a basic tenet of multilevel security.
- To incorporate multilevel security notions into the relational database model, it is common to consider attribute values and tuples as data objects. Hence, each attribute A is associated with a classification attribute C in the schema, and each attribute value in a tuple is associated with a corresponding security classification. In addition, in some models, a tuple classification attribute TC is added to the relation attributes to provide a classification for each tuple as a whole. Hence, a multilevel relation schema R with n attributes would be represented as $R(A_1,C_1,A_2,C_2, ..., A_n,C_n,TC)$ where each C_i represents the classification attribute associated with attribute A_i. The value of the TC attribute in each tuple t – which is the highest of all attribute classification values within t – provides a general classification for the tuple itself, whereas each C_i provides a finer security classification for each attribute value within the tuple.

Multilevel Relations and Polyinstantiation:

- The apparent key of a multilevel relation is the set of attributes that would have formed the primary key in a regular (single-level) relation. A multilevel relation will appear to contain different data to subjects (users) with different clearance levels.

- In some cases, it is possible to store a single tuple in the relation at a higher classification level and produce the corresponding tuple at a lower-level classification through a process known as filtering. In other cases, it is necessary to store two or more tuples at different classification levels with the same value for the apparent key. This leads to the concept of polyinstantiation where several tuples can have the same apparent key value but have different attribute values for users at different classification levels. The presence of data objects that appear to have different values to users with different clearances is called polyinstantiation.

Comparing Discretionary Access Control and Mandatory Access Control:

- Discretionary Access Control (DAC) policies are characterized by a high degree of flexibility, which makes them suitable for a large variety of application domains.
- The main drawback of DAC models is their vulnerability to malicious attacks, such as Trojan horses embedded in application programs.
- By contrast, mandatory policies ensure a high degree of protection in a way; they prevent any illegal flow of information. Mandatory policies have the drawback of being too rigid and they are only applicable in limited environments.
- In many practical situations, discretionary policies are preferred because they offer a better trade-off between security and applicability.

5.3.3 Role-Based Access Control

- Role-Based Access Control (RBAC) emerged rapidly in the 1990s as a proven technology for managing and enforcing security in large-scale enterprise wide systems.
- Its basic notion is that permissions are associated with roles, and users are assigned to appropriate roles.
- Roles can be created using the CREATE ROLE and delete or drop using DESTROY ROLE commands.
- The GRANT and REVOKE commands discussed under DAC can then be used to assign and revoke privileges from roles.
- RBAC appears to be a viable alternative to traditional discretionary and mandatory access controls; it ensures that only authorized users are given access to certain data or resources.
- Many DBMSs have allowed the concept of roles, where privileges can be assigned to roles. Role hierarchy in RBAC is a natural way of organizing roles to reflect the organization's lines of authority and responsibility.
- Another important consideration in RBAC systems is the possible temporal constraints that may exist on roles, such as time and duration of role activations, and timed triggering of a role by an activation of another role.

- Using an RBAC model is highly desirable goal for addressing the key security requirements of Web-based applications. In contrast, Discretionary Access Control (DAC) and Mandatory Access Control (MAC) models lack capabilities needed to support the security requirements emerging enterprises and Web-based applications.

5.4 USE OF VIEWS IN SECURITY ENFORCEMENT

- A view is a virtual relation that does not actually exist in the database, but is produced upon request by a particular user, at the time of request.
- It is the dynamic result of one or more relational operations operating on the base relations to produce another relation.
- The view is a widely used method for implementing access control in database applications.
- The view mechanism provides a powerful and flexible security mechanism by hiding parts of the database from certain users. The user is not aware of the existence of any attributes or rows that are missing from the view.
- Suppose that A1 wants to give a limited capability to A3 for SELECT from the EMPLOYEE relation and wants to allow A3 to be able to propagate the privilege. The limitation is to retrieve only the NAME, BDATE, and ADDRESS attributes and only for the tuples with DNO=5. A1 then create the view:

```
CREATE VIEW A3 EMPLOYEE AS
SELECT NAME, BDATE, ADDRESS
FROM EMPLOYEE
WHERE DNO = 5;
```

- After the view is created, A1 can grant SELECT on the view A3EMPLOYEE to A3 as follows:

```
GRANT SELECT ON A3EMPLOYEE TO A3 WITH GRANT OPTION
```

5.5 OVERVIEW OF ENCRYPTION TECHNIQUE FOR SECURITY (S-19)

- The primary purpose of encryption is to protect the confidentiality of data stored in database systems or transmitted via the Internet or other computer networks.
- Encryption is a means of maintaining secure data in an insecure environment.
- Encryption consists of applying an encryption algorithm to data using some pre-specified encryption key.
- The resulting data has to be decrypted using a decryption key to recover the original data.
- There are two advantages of encrypting data stored in the database:
  1. Encrypted data allows authorized users to access data without worrying about other users or the system administrator gaining any information.

2. Encryption of data may simplify or even strengthen other authorization mechanisms. For example, distribution of the cryptographic key amongst only trusted users is both, a simple way to control read access, and an added layer of security above that offered by views.

- Today's encryption algorithms are divided into two categories i.e., symmetric and asymmetric. In symmetric algorithm key ciphers use the same key, or secret, for encrypting and decrypting a message or file. The **Data Encryption Standard (DES)** is a **symmetric-key method** of data encryption.
- Asymmetric algorithm, also known as Public-Key Cryptography, uses two different but mathematically linked keys, one public and one private. RSA is Asymemetric algorithm.
- The **Data Encryption Standard (DES)** is a system developed by the U.S. government for use by the general public. It has been widely accepted as a cryptographic standard both in the United States and abroad.
- DES can provide end-to-end encryption on the channel between the sender A and receiver B.
- DES algorithm is a careful and complex combination of two of the fundamental building blocks of encryption: substitution and permutation (transposition).
- The DES algorithm derives its strength from repeated application of these two techniques for a total of 16 cycles.
- Plaintext (the original form of the message) is encrypted as blocks of 64 bits. After questioning the adequacy of DES, the National Institute of Standards (NIST) introduced the **Advanced Encryption Standards (AES)**. This algorithm has a block size of 128 bits and thus takes longer time to crack.
- In 1976, Diffie and Hellman proposed a new kind of cryptosystem, which they called **Public Key Encryption**. Public key algorithms are based on mathematical functions rather than operations on bit patterns.
- They also involve the use of two separate keys in contrast to conventional encryption, which uses only one key.
- The use of two keys can have profound consequences in the areas of confidentiality, key distribution, and authentication.
- The two keys used for public key encryption are referred to as the public key and the private key. The private key is kept secret, but it is referred to as private key rather than a secret key.
- A public key encryption scheme, or infrastructure, has six ingredients:-
  1. **Plaintext**: This is the data or readable message that is fed into the algorithm as input.
  2. **Encryption Algorithm**: The encryption algorithm performs various transformations on the plaintext.

3. **Public and Private Keys**: These are pair of keys that have been selected so that if one is used for encryption, the other is used for decryption. The exec transformations performed by the encryption algorithm depend on the public or private key that is provided as input.
4. **Ciphertext**: This is the scrambled message produced as output. It depends on the plaintext and the key. For a given message, two different keys will produce two different cipher texts.
5. **Decryption Algorithm:** This algorithm accepts the cipher text and the matching key and produces the original plaintext. Public key is made for public and private key is known only by owner. A general-purpose public key cryptographic algorithm relies on One key for encryption and a different but related key for decryption.

- The essential steps are as follows:
  1. Each user generates a pair of keys to be used for the encryption and decryption of messages.
  2. Each user places one of the two keys in a public register or other accessible file. This is the public key. The companion key is kept private (private key).
  3. If a sender wishes to send a private message to a receiver, the sender encrypts the message using the receiver's public key.
  4. When the receiver receives the message, he or she decrypts it using the receiver's private key. No other recipient can decrypt the message because only the receiver knows his or her private key.
- The RSA Public Key Encryption algorithm, one of the first public key schemes was introduced in 1978 by Ron Rivest (R), Adi Shamir (S), and Len Adleman (A) at MIT and is named after them. The RSA encryption algorithm incorporates results from number theory, such as the difficulty of determining the large prime factors of a large number.

5.6 STATISTICAL DATABASE SECURITY (S-18, 19; W-18)

- Statistical databases are mainly used to produce statistics on various populations. The database may contain confidential data on individuals, which should be protected from user access.
- Users are permitted to retrieve statistical information on the populations, such as averages, sums, counts, maximums, minimums, and standard deviations.
- A population is a set of tuples of a relation (table) that satisfy some selection condition.
- Statistical queries involve applying statistical functions to a population of tuples.
- For example, we may want to retrieve the number of individuals in a population or the average income in the population.

- However, statistical users are not allowed to retrieve individual data, such as the income of a specific person.
- For example, if a user is permitted statistical access to an employee database, he or she is able to write queries such as:

```
SELECT SUM(Salary)
FROM Employee
WHERE Dept = 10;
```

but not:

```
SELECT Salary
FROM Employee
WHERE empId = 'E101';
```

- Statistical database security techniques must prohibit the retrieval of individual data. This can be achieved by prohibiting queries that retrieve attribute values and by allowing only queries that involve statistical aggregate functions such as COUNT, SUM, MIN, MAX, AVERAGE, and STANDARD DEVIATION. Such queries are sometimes called statistical queries.
- It is DBMS's responsibility to ensure confidentiality of information about individuals, while still providing useful statistical summaries of data about those individuals to users. Provision of privacy protection of users in a statistical database is paramount.
- In some cases, it is possible to infer the values of individual tuples from sequence statistical queries. This is particularly true when the conditions result in a population consisting of a small number of tuples.

EXAMPLES

- Given the following database schemas:

EMPLOYEE Relation

FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	DNO

DEPARTMENT Relation

DNAME	DNUMBER	MGRSSN	MGRSTARTDATE

DEPT_LOCATION Relation

DNUMBER	DLOCATION

PROJECT Relation

PNAME	PNUMBER	PLOCATION	DNUM

DEPENDENT Relation

ESSN	DEP-NAME	SEX	BDATE	RELATIONSHIP

- Suppose that all the relations were created by (and hence are owned by) user X, who wants to grant the following privileges to user accounts A, B, C, D, and E:
  1. Account A can retrieve or modify any relation except DEPENDENT and can grant any of these privileges to other users.

     ```
     GRANT SELECT, UPDATE ON EMPLOYEE, DEPARTMENT, DEPT_LOCATIONS, PROJECT,
     WORKS_ON TO ACCOUNTA WITH GRANT OPTION;
     ```
  2. Account B can retrieve all the attributes of EMPLOYEE and DEPARTMENT except for SALARY, MGRSSN, and MGRSTARTDATE.

     ```
     CREATE VIEW EMPS AS SELECT FNAME, MINIT, LNAME, SSN, BDATE, ADDRESS,
     SEX, SUPERSSN, DNO FROM EMPLOYEE;
     GRANT SELECT ON EMPS TO ACCOUNTB;
     CREATE VIEW DEPTS AS SELECT DNAME, DNUMBER FROM DEPARTMENT;
     GRANT SELECT ON DEPTS TO ACCOUNTB;
     ```
  3. Account C can retrieve or modify WORKS_ON but can only retrieve the FNAME, MINIT, LNAME, SSN attributes of EMPLOYEE and PNAME, PNUMBER attributes of PROJECT.

     ```
     GRANT SELECT, UPDATE ON WORKS_ON TO ACCOUNTC;
     CREATE VIEW EMP1 AS SELECT FNAME, MINIT, LNAME, SSN FROM EMPLOYEE;
     GRANT SELECT ON EMP1 TO ACCOUNTC;
     CREATE VIEW PROJ1 AS SELECT PNAME, PNUMBER FROM PROJECT;
     GRANT SELECT ON PROJ1 TO ACCOUNTC;
     ```
  4. Account D can retrieve any attribute of EMPLOYEE or DEPENDENT and can modify DEPENDENT.

     ```
     GRANT SELECT ON EMPLOYEE, DEPENDENT TO ACCOUNTD;
     GRANT UPDATE ON DEPENDENT TO ACCOUNTD;
     ```
  5. Account E can retrieve any attribute of EMPLOYEE but only for EMPLOYEE tuples that have DNO = 3. Write SQL statements to grant these privileges. Use views where appropriate.

     ```
     CREATE VIEW DNO3_EMPLOYEES AS SELECT * FROM EMPLOYEE WHERE DNO=3;
     GRANT SELECT ON DNO3_EMPLOYEES TO ACCOUNTE;
     ```

Summary

- Security is an important issue in database management because information stored in a database is a valuable and sensitive resource.
- Database security can be defined as, protecting a database from unauthorized access, malicious destruction and even any accidental loss or misuse.
- Threat is any intentional or accidental event that may adversely affect the system.

- A DBMS typically includes a database security and authorization sub-system that is responsible for ensuring the security of portions of a database against unauthorized access.
- The security mechanism of a DBMS must include provisions for restricting access to the database as a whole. This function is called access control.
- An unauthorized user who access encoded data will have difficulty deciphering it, but authorized users are given decoding or decrypting algorithms (or keys) to decipher data.
- The database administrator (DBA) is the central authority for managing a database system. The DBA is responsible for the overall security of the database system.
- The typical method of enforcing discretionary access control in a database system is based on the granting and revoking privileges.
- A privilege is an action, such as creating, executing, reading, updating or deleting that a user is permitted to perform on database objects.
- The granting and revoking of privileges generally follow an authorization model for discretionary privileges known as the Access Matrix or Authorization Matrix Model.
- SQL GRANT command is used to provide access or privileges on the database objects to the users.
- In SQL, a REVOKE command is included for the purpose of canceling privileges.
- The discretionary access control techniques of granting and revoking privileges on relations have traditionally been the main security mechanism for relational database systems.
- The apparent key of a multilevel relation is the set of attributes that would have formed the primary key in a regular (single-level) relation. A multilevel relation will appear to contain different data to subjects (users) with different clearance levels.
- It is possible to store a single tuple in the relation at a higher classification level and produce the corresponding tuple at a lower-level classification through a process known as filtering.
- The presence of data objects that appear to have different values to users with different clearances is called polyinstantiation.
- Role-Based Access Control (RBAC) emerged rapidly in the 1990s as a proven technology for managing and enforcing security in large-scale enterprise wide systems.
- A view is a virtual relation that does not actually exist in the database, but is produced upon request by a particular user, at the time of request.
- The view is a widely used method for implementing access control in database applications.

- The primary purpose of encryption is to protect the confidentiality of data stored in database systems or transmitted via the Internet or other computer networks. Encryption is a means of maintaining secure data in an insecure environment.
- Today's encryption algorithms are divided into two categories i.e., symmetric and asymmetric.
- DES can provide end-to-end encryption on the channel between the sender A and receiver B.
- Plaintext (the original form of the message) is encrypted as blocks of 64 bits.
- Public key algorithms are based on mathematical functions rather than operations on bit patterns. They also involve the use of two separate keys in contrast to conventional encryption, which uses only one key.
- A general-purpose public key cryptographic algorithm relies on One key for encryption and a different but related key for decryption.
- The two keys used for public key encryption are referred to as the public key and the private key. The private key is kept secret, but it is referred to as private key rather than a secret key.
- Statistical databases are mainly used to produce statistics on various populations. A population is a set of tuples of a relation (table) that satisfy some selection condition. Statistical queries involve applying statistical functions to a population of tuples.
- Statistical database security techniques must prohibit the retrieval of individual data.

Check Your Understanding

1. The typical technique of discretionary access control in database system is based on the _____ of privilege.
   (a) Commit and rollback (b) Granting and revoking
   (c) Serial and non-serial (d) All of the above
2. _____ is responsible for using that the database remains in a consistent state despite system failure.
   (a) Storage manager (b) Transaction manager
   (c) End user (d) Sophisticated
3. Prevention of access to the database by unauthorized users is referred to as _____.
   (a) Integrity (b) Productivity
   (c) Security (d) Reliability
4. Data integrity means _____
   (a) providing first access to stored data.
   (b) ensuring correctness and consistency of data.

(c) providing data sharing.

(d) none of the above.

5. Authentication refers to _____.

(a) methods of restricting user access to system

(b) controlling access to portions of database

(c) controlling the operation on the data

(d) all of the above

ANSWER KEY

1. (b)	2. (b)	3. (c)	4. (b)	5. (d)

Practice Questions

Q.I. Answer the following questions in short.

1. Define the term database security.
2. What is meant by granting a privilege?
3. What is meant by revoking a privilege?
4. List the types of privileges available in SQL.
5. State different levels of security.

Q.II. Answer the following questions.

1. Discuss the database security threats.
2. Explain role of DBA with respect to database security.
3. Explain various methods for database security in brief.
4. Discuss the types of privileges at the account level and those at the relation level.
5. Which account is designated as the owner of a relation? What privileges does the owner of a relation have?
6. How is the view mechanism used as an authorization mechanism?
7. What is the difference between discretionary and mandatory access control?
8. What are the typical security classifications? Discuss the simple security property and the * property, and explain the justification behind these rules for enforcing multilevel security.
9. Describe the multilevel relational data model.
10. Explain encryptions techniques used for database security.
11. What are two advantages of encrypting data stored in the database?
12. What is a statistical database? Discuss the problem of statistical database security.

Q.III. Define following terms:

1. Apparent Key
2. Polyinstantiation
3. Filtering
4. Grant Command
5. Revoke Command
6. Ciphertext
7. Access Matrix
8. RSA Public Key Encryption Algorithm
9. View
10. Threat

Previous Exams Questions

Summer 2019

1. .....Algorithm uses same key for encrypting and decrypting a message. **(1M)**

(a) Symmetric key
(b) Asymmetric key
(c) Both (a) and (b)
(d) None of these

Ans. Refer to Section 5.5

2. Explain how encryption techniques are used in database security. **(4M)**

Ans. Refer to Section 5.6

3. Consider below scenario and write grant/revoke commands for the same : **(3M)**

Employee (ename, address, salary, Bdate) Suppose that the relation employee was created by user X, who wants to grant privileges to user A, that :

A can retrieve or modify employee relation and can grant any of these privileges to other users.

Ans. Refer to Section 5.3.1.1

Winter 2018

1. What is the encryption technique used for ? **(1M)**

Ans. Refer to Section 5.6

2. Explain how DBA is responsible for managing database security ? What are the privileged commands used by DBA ? **(4M)**

Ans. Refer to Section 5.3

3. Explain the mandatory access control mechanism in detail. **(5M)**

Ans. Refer to Section 5.3.2

4. Write a note on statistical database security. **(3M)**

Ans. Refer to Section 5.6

Summer 2018

1. Database security involves _____. **(1M)**

(a) Data encryption (b) a view

(c) finger print (d) all

Ans. (a)

2. State different levels of Security. **(1M)**

Ans. Refer to Section 5.2.

3. What are different privileges that can be granted, on a relation, to the database users. **(4M)**

Ans. Refer to Section 5.3.

4. Write a short note on encryption techniques in database security. **(5M)**

Ans. Refer to Section 5.6

❖❖❖

6... Database System Architectures

Objectives...

- To get information about database system architecture and its types such as Centralized and Client / Server Architectures.
- To get knowledge about parallel, distributed and object-based database systems.

6.1 INTRODUCTION TO DATABASE SYSTEMS ARCHITECTURES

- The architectures of database systems are mainly client/server, parallel systems and distributed systems.
- In the client/server database systems, the Networking of computers allows some tasks to be executed on a server system and some tasks to be executed on client systems.
- To speed up the database-system activities and allowing faster response to transactions, as well as more transactions per second, Parallel processing architecture is used. The need for parallel query processing has led to parallel database systems.
- A distributed database is basically a database that is not limited to one system; it is spread over different sites, i.e., on multiple computers or over a network of computers. A distributed database system is located on various sites that don't share physical components.
- In a distributed database, there are a number of databases that may be geographically distributed all over the world. A distributed DBMS manages the distributed database in a manner so that it appears as one single database to users.
- A distributed database is a collection of multiple interconnected databases, which are spread physically across various locations that communicate via a computer network.

- In this chapter, we will study the above architectures.

6.2 CENTRALIZED AND CLIENT/ SERVER ARCHITECTURES

- Historically, early database systems were based on a Centralized Model, in which the database resides on a single computer system that allows access by remote terminals. This is still the model used by many systems today.
- In this section, we will try to understand different aspects of centralized and client/server architectures.

6.2.1 Centralized Systems

- Centralized database systems will run on a single computer system and do not interact with other computer systems.
  1. **General-purpose computer system:** This system has one or more CPUs and a number of device controllers that are connected through a common bus that provides access to shared memory.
  2. **Single-user system (e.g., personal computer or workstation):** This system has Desktop unit, single user. It usually has only one CPU and one or two hard disks; the OS may support only one user.
  3. **Multi-user system:** This system has more disks, more memory, multiple CPUs, and a multi-user OS. Serve a large number of users who are connected to the system via terminals. Multi-user systems are often called server systems.
- Database systems designed for use by single users usually do not provide many of the facilities that a multiuser database provides. In particular, they may not support concurrency control, which is not required when only a single user can generate updates.

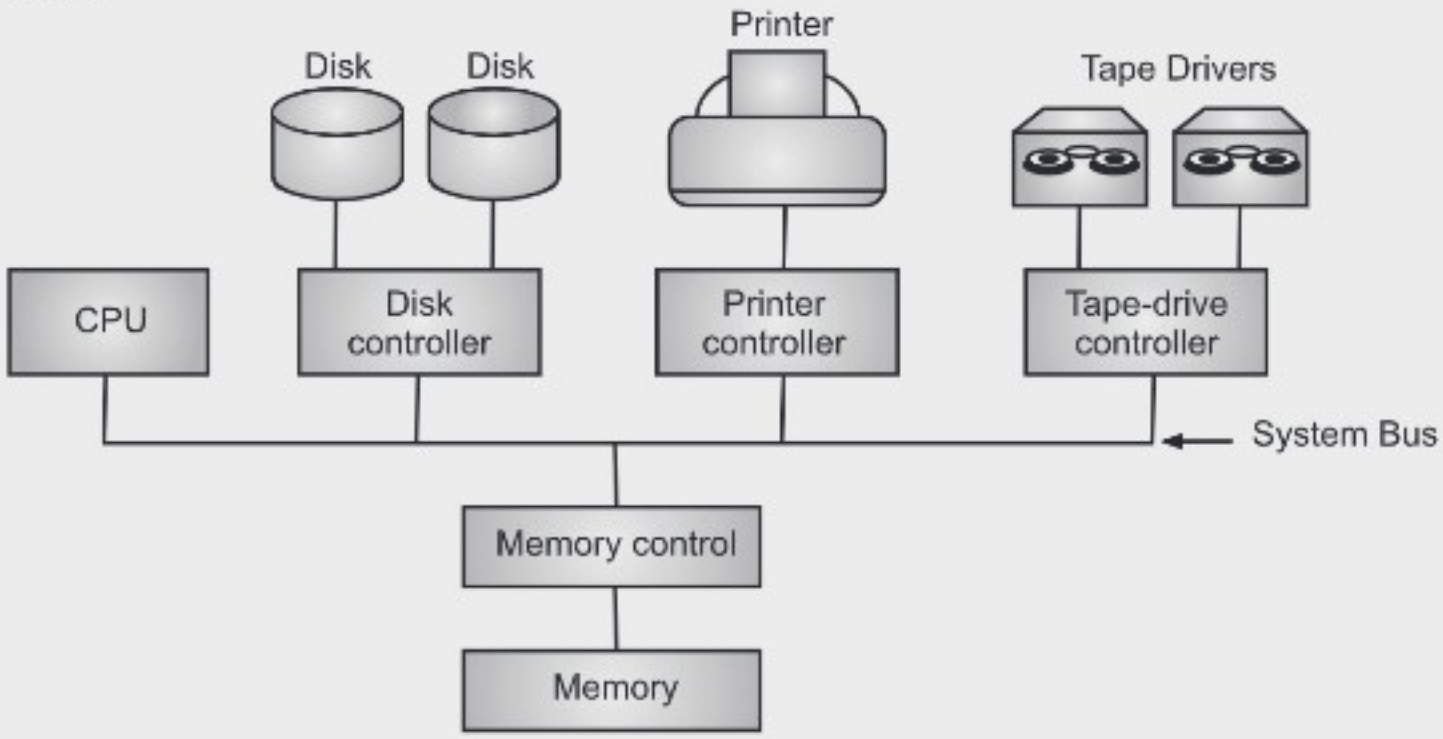


Fig. 6.1: A Centralized Computer System

- Provisions for crash recovery in such systems are either absent or primitive. For example, they may consist of simply making a backup of the database before any update. In contrast, database systems designed for multiuser systems support the full transactional features.

Disadvantages of Centralized Database System:

1. A centralized system is totally vulnerable to failure of the centralized site. Thus, for example, a power failure at the site where the computer is located can shut down all access to the database, even at remote sites unaffected by the failure.
2. As the volume of transactions handled by the system grows, it can become increasingly difficult for a single system to handle the demand.
3. Communication costs from the terminals to the central site can be expensive.

6.2.2 Client/Server Systems

- The database systems functionality can be broadly divided into two parts: the front end and the back end.
- The front end of a database system consists of tools such as the SQL user interface, forms interfaces, report generation tools, and data mining and analysis tools.
- The back end manages access structures, query evaluation and optimization, concurrency control, and recovery.
- The interface between the front end and the back end is through SQL, or through an application program.

6.2.3 Components of Client/Server Architecture

- There are three major components of client/server architecture:
  1. Server
  2. Client
  3. Network interface

1. Server:

- Server is DBMS itself. It consists of DBMS and supports all basic DBMS functions. Server components of DBMS are installed at the server. It acts as monitor of all of its clients. It distributes work-load to other computers. Clients must obey their servers.

  Functions of Server: The server performs various functions, which are as follows.

  1. It supports all basic DBMS functions.
  2. Monitor all his clients.
  3. Distribute work-load over clients.
  4. Solve problems which are not solved by clients.
  5. Maintain security and privacy.
  6. Avoiding unauthorized access of data.

2. Client:

- Client machine is a personal computer or workstation which provides services to both server and users. It must obey his server. Client components of DBMS are installed at the client site. Clients are taking instructions from the server and help them by taking their load. When any user wants to execute a query on a client, the client first takes data from the server then executes the query on his own hardware and returns the result to the server. As a result, servers are free to do more complex applications.

3. Network Interface:

- Clients are connected to the server by network interface. It is useful in connecting the server interface with the user interface so that the server can run his applications over its clients. In the Client/Server architecture, there are more than one server. Sometimes, a server is used as Database Server, other as Application Server, other as Backup Server etc.
- In Client/Server computing, the client requests a resource and the server provides that resource. A server may serve multiple clients at the same time while a client is in contact with only one server.

Types of Client/Server Architecture:

- Normally, there are three types of client/server architecture available in database management systems. These are listed below:
  (a) Single-tier client/server computing Model (One-tier Architecture)
  (b) Two-tier client/server computing Model
  (c) Three tier client/server computing Model

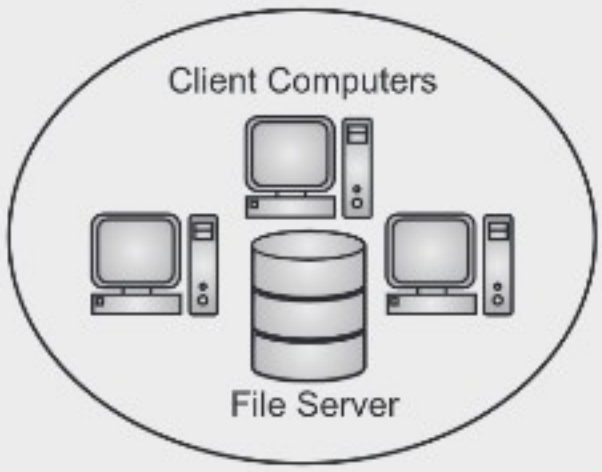


Fig. 6.2:Single-Tier Architecture

A. Single-Tier Client/server Architecture.

- Single-tier architecture is the first type of client/server computing model. In Single-tier, client/server computing model, the client/server database system used on a personal computer. In Single-tier system, the database is centralized, which means the DBMS software and the data in one location and the dumb terminals were used to access the database management system.

Advantage of Single-Tier:

1. The data is easily and quickly available since it is located in the same machine.

Disadvantage of Single-Tier:

1. This architecture is completely not scalable. Only one user can access the system at a given time through the local client.

B. Two-Tier Client/Server Architecture:

- The two-tier architecture primarily has two parts, a Client tier and a Server tier. The Client tier sends a request to the server tier and the server tier responds with the desired information.
- An example of a two-tier client/server structure is a web server. It returns the required web pages to the clients that requested them.

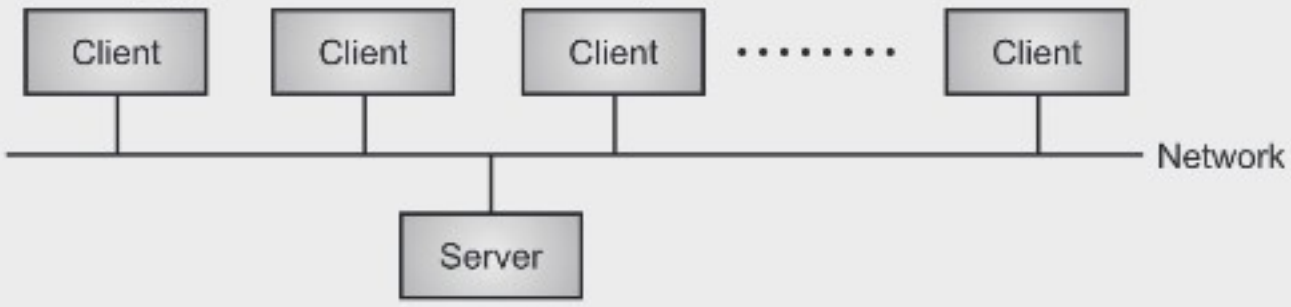


Fig. 6.3: Two-Tier Client/Server Architecture

Advantages of Two-Tier Client/Server Structure:

- This structure is quite easy to maintain and modify.
- The communication between the client and server in the form of request response messages is quite fast.

Disadvantage of Two-Tier Client/Server Architecture:

- If the client nodes are increased beyond capacity in the architecture, then the server is not able to handle the request overflow and performance of the system degrades.

C. Three-Tier Client/Server Architecture:

- The three-tier architecture has three layers namely client, application and data layer. The client layer is the one that requests the information. In this case it could be the GUI, web interface etc.

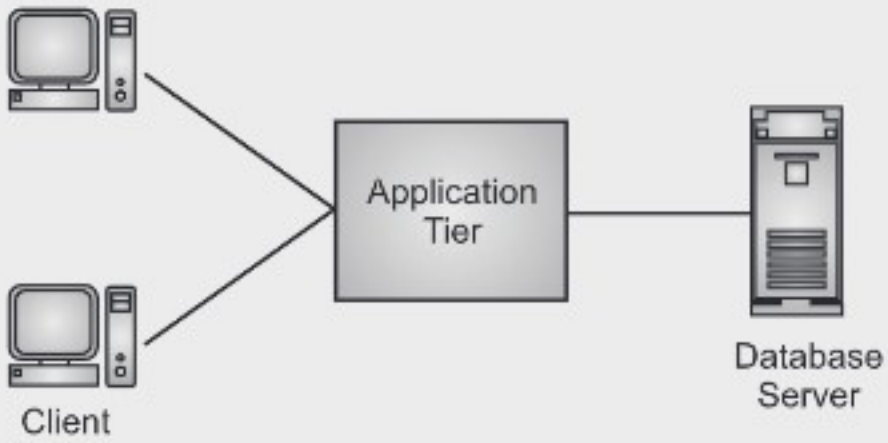


Fig. 6.4: Three-Tier Client/Server Architecture

- The application layer acts as an interface between the client and data layer. It helps in communication and also provides security. The data layer is the one that actually contains the required data.

Advantages of Three-Tier Client/Server architecture:

- The three-tier structure provides much better service and fast performance.
- The structure can be scaled according to requirements without any problem.
- Data security is much improved in the three- tier structure.

Disadvantage of Three-Tier Client/Server architecture:

- Three - tier client/server structure is quite complex due to advanced features.

6.3 SERVER SYSTEM ARCHITECTURES

- Server systems can be broadly categorized into two kinds:
  - **Transaction servers:** These servers are widely used in relational database systems.
  - **Data servers:** These servers used in object-oriented database systems.

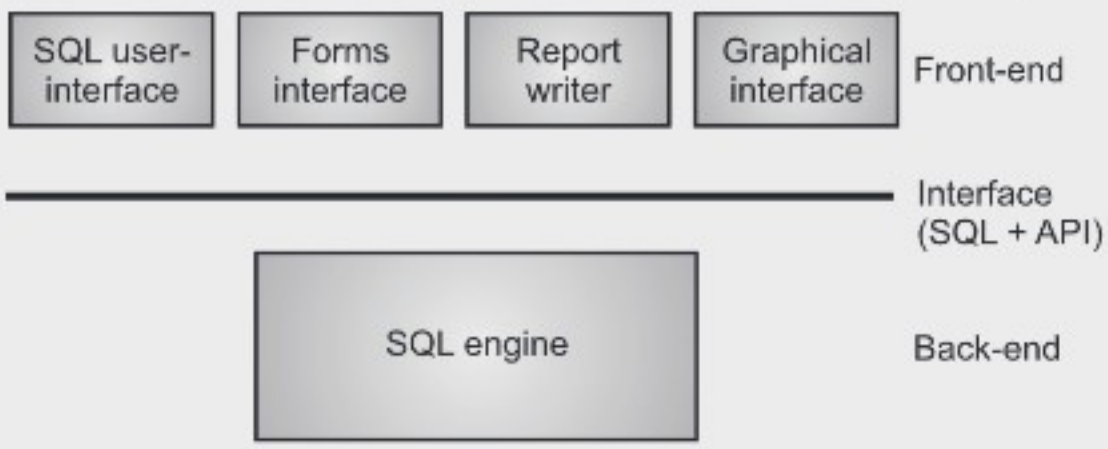


Fig. 6.5: Front-end and back-end functionality

Transaction Servers:

- This is also called **Query Server Systems** or **SQL Server Systems**.
  - Clients send requests to the server.
  - Transactions are executed at the server.
  - Results are shipped back to the client.
- Requests are specified in SQL, and communicated to the server through a Remote Procedure Call (RPC) mechanism.
- Transactional RPC allows many RPC calls to form a transaction.
- Open Database Connectivity (ODBC) is a C language application program interface standard from Microsoft for connecting to a server, sending SQL requests, and receiving results.
- JDBC standard is similar to ODBC, for Java.

6.3.1 Transaction Server Process Structure

- A typical transaction server consists of multiple processes accessing data in shared memory.
- **Server processes**
  - These receive user queries (transactions), execute them and send results back.
  - Processes may be multithreaded, allowing a single process to execute several user queries concurrently.
  - Typically multiple multithreaded server processes
- **Lock manager process** includes lock grant, lock release, and deadlock detection.
- **Database writer processes** that output modified buffer blocks back to disk on a continuous basis.
- **Log writer process**
  - Server processes simply add log records to log record buffer
  - Log writer process outputs log records to stable storage.
- **Checkpoint process**
  - Performs periodic checkpoints
- **Process monitor process**
  - Monitors other processes, and takes recovery actions if any of the other processes fail.
  - E.g. aborting any transactions being executed by a server process and restarting it.
- The shared memory contains all shared data, such as:
  - Buffer pool.
  - Lock table.
  - Log buffer, containing log records waiting to be output to the log on stable storage.
- All database processes can access shared memory.
- To ensure that no two processes are accessing the same data structure at the same time, databases systems implement mutual exclusion using either : Operating system semaphores or Atomic instructions such as test-and-set .
- To avoid overhead of inter-process communication for lock requests/grants, each database process operates directly on the lock table instead of sending requests to the lock manager process.
- Lock manager process still used for deadlock detection.

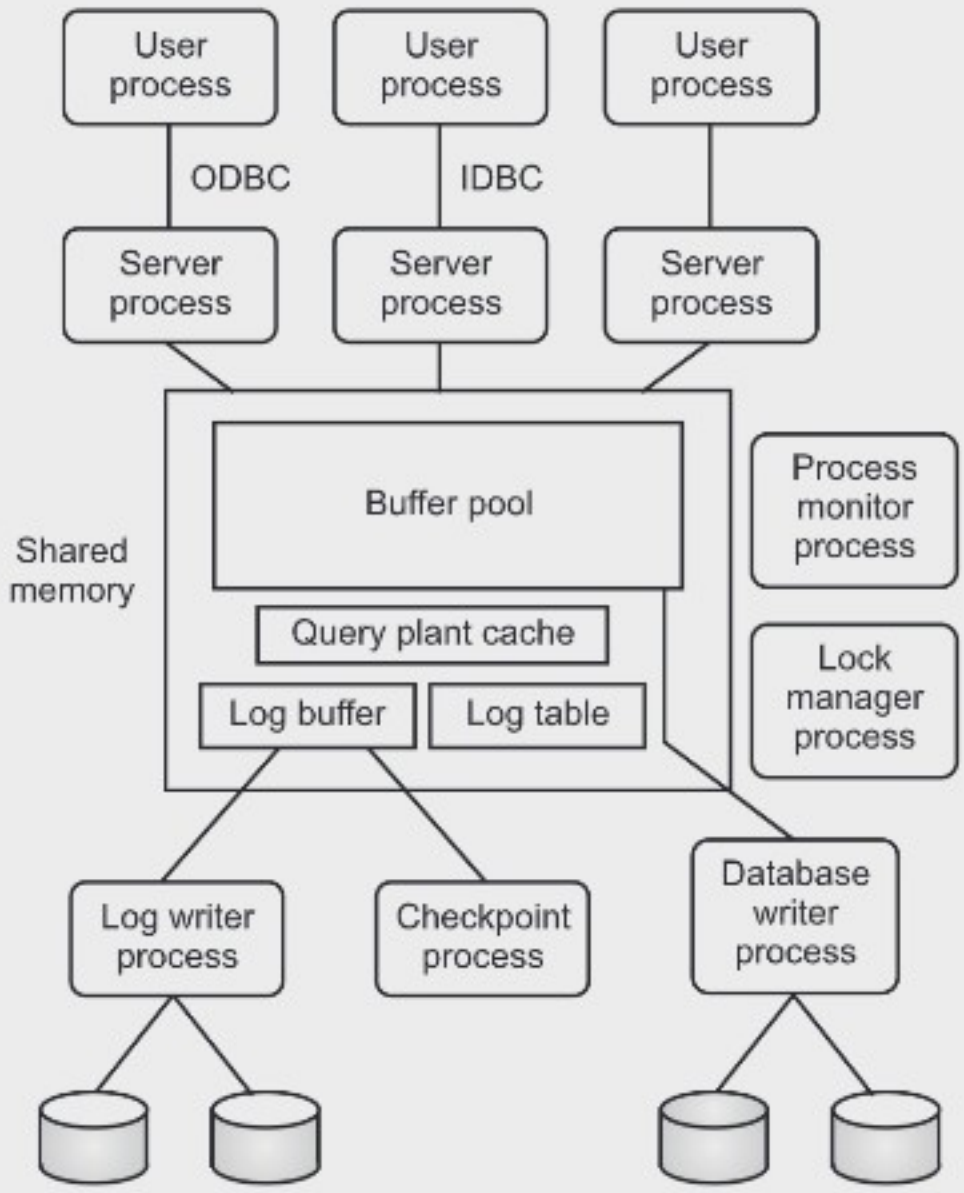


Fig. 6.6: Shared Memory and Process Structure.

6.3.2 Data Servers

- Used in high-speed LANs, in cases where,
  - The clients are comparable in processing power to the server.
  - The tasks to be executed are compute-intensive.
  - Data are shipped to clients where processing is performed, and then shipped results back to the server.
  - This architecture requires full back-end functionality at the clients.
  - Used in many object-oriented database systems.
  - Issues are:
    a) Page-Shipping versus Item-Shipping
    b) Locking
    c) Data Caching
    d) Lock Caching

(a) Page-shipping versus item-shipping

- Smaller unit of shipping ⇒ more messages
- Worth prefetching related items along with requested items.
- Page shipping can be thought of as a form of prefetching.

(b) Locking

- Overhead of requesting and getting locks from the server is high due to message delays.
- Can grant locks on requested and prefetched items; with page shipping, transaction is granted lock on whole page.

(c) Data Caching

- Data can be cached at client even in between transactions.
- But check that data is up-to-date before it is used (cache coherency).
- Check can be done when requesting lock on data item.

(d) Lock Caching

- Locks can be retained by the client system even in between transactions.
- Transactions can acquire cached locks locally, without contacting the server.
- Server calls back locks from clients when it receives conflicting lock requests. Client returns lock once no local transaction is using it.

6.4 INTRODUCTION TO PARALLEL SYSTEMS

- Parallel database systems consist of multiple processors and multiple disks connected by a fast interconnection network.
- A coarse-grain parallel machine consists of a small number of powerful processors.
- A massively parallel or fine grain parallel machine utilizes thousands of smaller processors.
- Two main performance measures of parallel systems are:
  1. **Throughput :** The number of tasks that can be completed in a given time interval.
  2. **Response time** : The amount of time it takes to complete a single task from the time it is submitted.

Advantges:

- Advantages of parallel databases are as follows:
  1. Parallel databases have increased the throughput (scale-up).
  2. Improved the response time (speed-up).
  3. Using several resources (CPU's and disks) in parallel can significantly improve performance.
  4. Increased availability.
  5. It is possible to serve large number of users.

Disadvantages:

1. The start-up costs are comparatively high.
2. Inference problem: Existing CPU's get slow down, as more CPU's are added; due to increased contention of memory and network bandwidth.
3. Data skew problem: Multiple disks contain partitions with widely varying number of tuples. Skew causes CPU's dealing with large partitions to become performance bottleneck.

6.4.1 Parallel Database Architectures

- Parallel databases can be roughly divided into two groups; the first group of architecture is the multiprocessor architecture, the alternatives of which are the following:
  1. Shared memory -- processors share a common memory.
  2. Shared disk -- processors share a common disk.
  3. Shared nothing -- processors share neither a common memory nor common disk.
- The other architecture group is,
  4. Hierarchical -- hybrid of the above architectures.

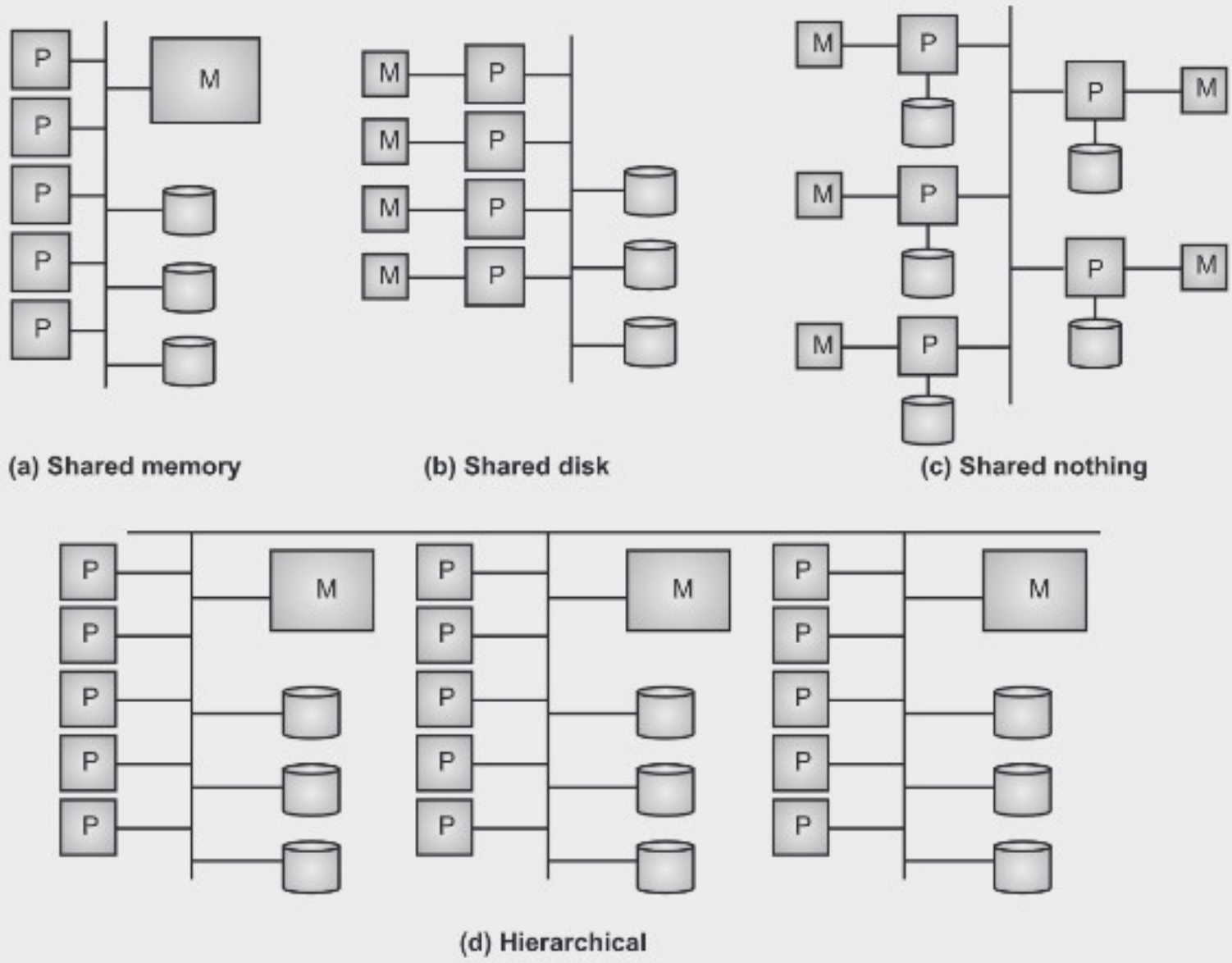


Fig. 6.7: Parallel Database Architectures

1. **Shared Memory**

- Processors and disks have access to a common memory, typically via a bus or through an interconnection network.
- Extremely efficient communication between processors — data in shared memory can be accessed by any processor without having to move it using software.
- **Disadvantage** – architecture is not scalable beyond 32 or 64 processors since the bus or the interconnection network becomes a bottleneck.
- Widely used for lower degrees of parallelism (4 to 8).

2. **Shared Disk**

- All processors can directly access all disks via an interconnection network, but the processors have private memories. The memory bus is not a bottleneck.
- Architecture provides a degree of fault-tolerance — if a processor fails, the other processors can take over its tasks since the database is resident on disks that are accessible from all processors.
- **Examples:** IBM Sysplex and DEC clusters (now part of Compaq) running Rdb (now Oracle Rdb) were early commercial users.
- **Disadvantage:** bottleneck now occurs at interconnection to the disk subsystem.
- Shared-disk systems can scale to a somewhat larger number of processors, but communication between processors is slower.

3. **Shared Nothing**

- Node consists of a processor, memory, and one or more disks. Processors at one node communicate with another processor at another node using an interconnection network. A node functions as the server for the data on the disk or disks the node owns.
- **Examples:** Teradata, Tandem, Oracle-n CUBE
- Data accessed from local disks (and local memory accesses) do not pass through interconnection networks, thereby minimizing the interference of resource sharing.
- Shared-nothing multiprocessors can be scaled up to thousands of processors without interference.
- **Disadvantage:** cost of communication and non-local disk access; sending data involves software interaction at both ends.

4. **Hierarchical**

- It combines characteristics of shared-memory, shared-disk, and shared nothing architectures.
- Top level is a shared-nothing architecture – nodes connected by an interconnection network, and do not share disks or memory with each other.
- Each node of the system could be a shared-memory system with a few processors.
- Alternatively, each node could be a shared-disk system, and each of the systems sharing a set of disks could be a shared-memory system.

- Reduce the complexity of programming such systems by distributed virtual-memory architectures.
- Also called Non-Uniform Memory Architecture (NUMA).

6.5 INTRODUCTION TO DISTRIBUTED SYSTEMS

- A distributed database is a collection of multiple interconnected databases, which are spread physically across various locations that communicate via a computer network.
- In the distributed systems,
  1. Data spread over multiple machines (also referred to as Sites or Nodes).
  2. Network interconnects the machines.
  3. Data shared by users on multiple machines.

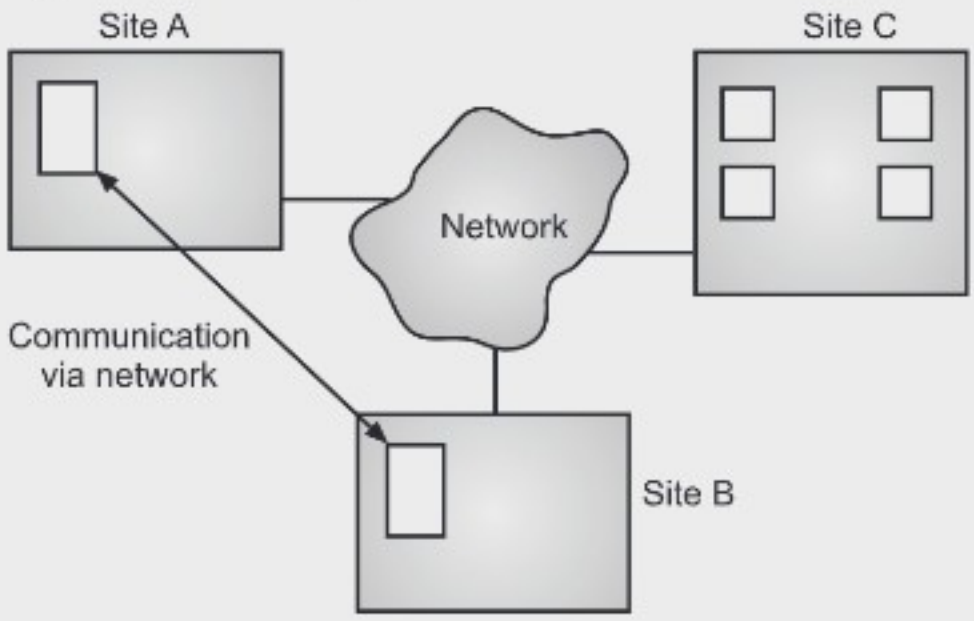


Fig. 6.8: Distributed Databases

6.5.1 Architecture of Distributed Databases

Types of distributed databases:

1. **Homogeneous distributed databases**

- Homogeneous distributed database system is a network of two or more databases (With the same type of DBMS software) which can be stored on one or more machines.
- So, in this system data can be accessed and modified simultaneously on several databases in the network. Homogeneous distributed systems are easy to handle.
- **Example:** Consider that we have three departments using Oracle-9i for DBMS. If some changes are made in one department then it would update the other department also.

2. **Heterogeneous distributed databases**

- Heterogeneous distributed database system is a network of two or more databases with different types of DBMS software, which can be stored on one or more machines.
- In this system data can be accessible to several databases in the network with the help of generic connectivity (ODBC and JDBC).

Difference between Homogeneous and Heterogeneous Distributed databases:

Homogeneous Distributed databases:

- Same software/schema on all sites, data may be partitioned among sites.
- Goal: provide a view of a single database, hiding details of distribution.

Heterogeneous Distributed databases:

- Different software/schema on different sites.
- Goal: integrate existing databases to provide useful functionality.

Differentiate between local and global transactions :

- A local transaction accesses data in the single site at which the transaction was initiated.
- A global transaction either accesses data in a site different from the one at which the transaction was initiated or accesses data in several different sites.

6.5.2 Distributed Database Management System

- A Distributed Database Management System (DDBMS) is a centralized software system that manages a distributed database in a manner as if it were all stored in a single location.

Features:

- It is used to create, retrieve, update and delete distributed databases.
- It synchronizes the database periodically and provides access mechanisms by the virtue of which the distribution becomes transparent to the users.
- It ensures that the data modified at any site is universally updated.
- It is used in application areas where large volumes of data are processed and accessed by numerous users simultaneously.
- It is designed for heterogeneous database platforms.
- It maintains confidentiality and data integrity of the databases.

6.5.3 Distributed Database Design

- The methodology used for the logical design of a centralized database applies to the design of the distributed one as well. However, for a distributed database three additional factors have to be considered.
  1. **Data Fragmentation:** Before we decide how to distribute the data we must determine the logical units of distribution. The database may be broken up into logical units called fragments which will be stored at different sites. The simplest logical units are the tables themselves.
  2. **Horizontal fragmentation:** A horizontal fragment of a table is a subset of rows in it. So horizontal fragmentation divides a table 'horizontally' by selecting the relevant rows and these fragments can be assigned to different sides in the

distributed system (for example, Tilak Road branch gets the fragment where myTable.branch ='Tilak Road').

3. **Vertical fragmentation:** A vertical fragment of a table keeps only certain attributes of it. It divides a table vertically by columns. It is necessary to include the primary key of the table in each vertical fragment so that the full table can be reconstructed if needed.
4. **Mixed fragmentation:** In a mixed fragmentation, each fragment can be specified by a SELECT-PROJECT combination of operations. In this case the original table can be reconstructed by applying union and natural join operations in the appropriate order.
5. **Data Allocation:** Each fragment has to be allocated to one or more sites, where it'll be stored.
6. **Data Replication:** A copy of each fragment can be maintained at several sites. Data replication is the design process of deciding which fragments will be replicated.

6.5.4 Advantages and Disadvantages of Distributed Database

Advantages:

- **Sharing data:** users at one site able to access the data residing at some other sites.
- **Autonomy:** Each site is able to retain a degree of control over data stored locally.
- **Higher system availability through redundancy:** Data can be replicated at remote sites, and systems can function even if a site fails.

Disadvantages:

- **Added complexity:** Required to ensure proper co-ordination among sites.
- **Cost:** Extra Software development cost. This database is more expensive as it is complex and hence, difficult to maintain.
- **Security:** Greater potential for bugs.
- **Processing overhead:** Even simple operations may require a large number of communications and additional calculations to provide uniformity in data across the sites.
- **Lack of standard:** Although it provides effective communication and data sharing, still there are no standard rules and protocols to convert a centralized DBMS to a large Distributed DBMS. Lack of standards decreases the potential of Distributed DBMS.

6.6 INTRODUCTION TO OBJECT BASED DATABASES

- An Object-oriented database is a collection of object-oriented programming and relational databases. There are various items which are created using object-oriented programming languages like C++, Java which can be stored in relational databases, but object-oriented databases are well-suited for those items.

- An object-oriented database is organized around objects rather than actions, and data rather than logic. For example, a multimedia record in a relational database can be a definable data object, as opposed to an alphanumeric value.
- The idea of object databases was originated in 1985 and today has become common for various common OOP languages, such as C++, Java, C#, Smalltalk, and LISP. Common examples are Smalltalk is used in GemStone, LISP is used in Gbase, and COP is used in Vbase. Today, other many popular OOP languages are such as Ruby, Python, JavaScript, and Perl.

Features:

- Object Database Management Systems (ODBMSs) are based on objects in Object-Oriented Programming (OOP).
- In OOP, an entity is represented as an object and objects are stored in memory.
- Objects have members such as fields, properties, and methods.
- Objects also have a life cycle that includes the creation of an object, use of an object, and deletion of an object.
- OOP has key characteristics such as encapsulation, inheritance, and polymorphism.

Applications:

- Object databases are commonly used in applications that require high performance, calculations, and faster results. Some of the common applications that use object databases are real-time systems, architectural & engineering for 3D modeling, telecommunications, and scientific products, molecular science, and astronomy.

Advantages of Object Databases:

- ODBMS provide persistent storage to objects. Imagine creating objects in your program and saving them as it is in a database and reading back from the database.
- In a typical relational database, the program data is stored in rows and columns. To store and read that data and convert it into program objects in memory requires reading data, loading data into objects, and storing it in memory. Imagine creating a class in your program and saving it as it is in a database, reading back and starting to use it again.
- Object databases bring permanent persistence to objects. Objects can be stored in persistent storage forever.
- In typical RDBMS, there is a layer of object-relational mapping that maps database schemas with objects in code. Reading and mapping an object database data to the objects is direct without any API or OR tool. Hence faster data access and better performance.
- Some object databases can be used in multiple languages. For example, the Gemstone database supports C++, Smalltalk and Java programming languages.

Disadvantages of Object Databases:

- Object databases are not as popular as RDBMS. It is difficult to find object DB developers.
- Not many programming languages support object databases.
- RDBMS have SQL as a standard query language. Object databases do not have a standard.
- Object databases are difficult to learn for non-programmers.

List of popular Object Databases:

- **ObjectDB :** ObjectDB is an object database for Java.
- **ObjectDatabase++ (ODBPP):** ObjectDatabase++ (ODBPP) is an embeddable object-oriented database designed for server applications that require minimal external maintenance.
- **Db4o(database for objects):** db4o (database for objects) was an embeddable open-source object database for Java and .NET developers
- **Cache:** InterSystems Caché is a commercial operational database management system from InterSystems, used to develop software applications for healthcare management, banking and financial services, government, and other sectors.
- **ObjectStore:** ObjectStore is an enterprise object-oriented database management system for C++ and Java.
- **Objectivity/DB :** Objectivity/DB is a commercial object database produced by Objectivity, Inc.
- **WakandaDB:** This is a JavaScript platform to develop and run web or mobile apps.

Summary

- Centralized database systems run entirely on a single computer.
- General-purpose computer system: one to a few CPUs and a number of device controllers that are connected through a common bus that provide access to shared memory.
- Single-user system (e.g., personal computer or workstation): desk-top unit, single user, usually has only one CPU and one or two hard disks; the OS may support only one user.
- Multi-user system: more disks, more memory, multiple CPUs and a multi-user OS. Serve a large number of users who are connected to the system via terminals. Often called server systems.
- Client/server interface protocols helped the growth of client/server database systems.

- Servers can be either transaction servers or data servers, although the use of transaction servers greatly exceeds the use of data servers for providing database services.
- Transaction servers have multiple processes, possibly running on multiple processors. So that these processes have access to common data, such as the database buffer, systems store such data in shared memory.
- Data-server systems supply raw data to clients. Such systems strive to minimize communication between clients and servers by caching data and locks at the clients.
- Parallel database systems use similar optimizations. Parallel database systems consist of multiple processors and multiple disks connected by a fast interconnection network. Speedup measures how much we can increase processing speed by increasing parallelism for a single transaction.
- Parallel database architectures include the shared-memory, shared-disk, shared-nothing, and hierarchical architectures.
- A distributed database system is a collection of partially independent database systems that (ideally) share a common schema, and coordinate processing of transactions that access nonlocal data. The systems communicate with one another through a communication network.
- An object-oriented database is a collection of object-oriented programming and relational databases. There are various items which are created using object-oriented programming languages like C++, Java which can be stored in relational databases, but object-oriented databases are well-suited for those items.

Check Your Understanding

1. Which is not type of parallel database?
   (a) Shared memory (b) Shared disk
   (c) Shared nothing (d) Shared processor
2. This system has only one CPU and supports only one user.
   (a) Multi-user OS (b) Single user OS
   (c) Multi-tasking OS (d) Multi-processor OS
3. In client/server architecture _____requests a resource.
   (a) Client (b) Server
   (c) Processor (d) Client and server
4. Which system depends on objects rather than actions?
   (a) Distributed system (b) Client/server system
   (c) Object-based system (d) Parallel system
5. Which is not a type of object oriented programming language?
   (a) C++ (b) Perl
   (c) Java (d) COBOL

ANSWER KEY

1. (d)	2. (b)	3. (a)	4. (c)	5. (d)

Practice Questions

Q.I. Answer the following questions in short.

1. What is the Client/server database system?
2. List types of server systems.
3. What is the use of data caching and lock caching?
4. What is the shared memory?
5. List the types of Object databases
6. List the types of distributed databases.

Q.II. Answer the following questions.

1. Explain Client/Server System architecture.
2. Explain two-tier Client/Server structure. Also state the advantage and disadvantage of it.
3. Explain three-tier Client/Server structure. Also state the advantage and disadvantage of it.
4. Explain Transaction server's process structure with diagram.
5. Explain parallel database architecture.
6. State the Advantages and disadvantages of distributed Databases.
7. State the features of distributed Databases.
8. State the Drawbacks of Object Databases.
9. State the advantages of Object Databases.

Q.III. Define following terms:

1. Centralized systems
2. Distributed database system
3. Object based database
4. Parallel database
5. Data server
6. Shared disk architecture
7. Two-tier client/server architecture
8. Three-tier client/server architecture
9. Shared- nothing architecture
10. Front-end and back-end system.

❖❖❖

Assignments

1. BANK DATABASE

- Consider the following database maintained by a Bank. The Bank maintains information about its branches, customers and their loan applications.
- Following are the tables:

  branch (bid integer, brname char (30), brcity char (10))

  customer (cno integer, cname char (20), caddr char (35), city (20))

  loan_application (lno integer, lamtrequired money, lamtapproved money, l_date date)
- The relationship is as follows:

  branch, customer, loan_application are related with ternary relationship.

  ternary (bid integer, cno integer, lno integer).

Execution of Query:

```
Postgres = # select * from branch;
```

Output:

bid	brname	brcity
1	CAMP	PUNE
2	MGROAD	PUNE

(2 rows)

Execution of Query:

```
postgres=# select * from customer ;
```

Output:

cno	cname	caddr	city
1	KIRAN	CAMP	PUNE
2	AVINASH	MGROAD	PUNE
3	SEEMA	KOTHRUD	PUNE

(3 rows)

Execution of Query:

```
Postgres = # select * from loan_application;
```

Output:

lno	lamtrequired	lamtapproved	l_date
1	$200,000.00	$100,000.00	2016-03-01
2	$500,000.00	$300,000.00	2015-03-01
3	$300,000.00	$100,000.00	2017-02-01

(3 rows)

Execution of Query:

```
Postgres = # select * from ternary;
```

Output:

bid	cno	lno
1	1	1
2	2	2
2	2	2

(3 rows)

Solve the following Queries:

1. Find the names of the customers for the "MGROAD" branch.

```
select cname from customer a, branch b, ternary c
where b.brname='MGROAD'
and a.cno=c.cno
and b.bid=c.bid;
```

Execution of Query:

```
postgres=# \i q1.sql
```

Output:

cname
AVINASH

(1 row)

2. List the names of the customers who have received loan less than their requirement.

```
select cname from customer a, loan_application b, ternary c
where b.lamtapproved< lamtrequired
and a.cno=c.cno
and b.lno=c.lno;
```

Execution of Query:

```
postgres-# \i q1.sql
```

Output:

```
cname
----------------
 KIRAN
 AVINASH
```

(2 rows)

3. Find the maximum loan amount approved.

```
select max(lamtapproved) from loan_application ;
```

Execution of Query:

```
postgres-# \i q1.sql
```

Output:

```
   max
-------------
 $300,000.00
```

(1 row)

4. Find out the total loan amount sanctioned by "Camp "branch.

```
select c.bid, sum(lamtapproved) from branch a, loan_application b, ternary
c
where a.brname='CAMP'
and a.bid=c.bid
and b.lno=c.lno
group by c.bid;
```

Execution of Query:

```
postgres=# \i q1.sql
```

Output:

```
bid |    sum
----- + ---------------
 1  |  100,000.00
```

(1 row)

5. Count the number of loan applications received by "MGROAD" branch.

```
select c.bid, Count(*) from branch a, loan_application b, ternary c
where a.brname='MGROAD'
and a.bid=c.bid
and b.lno=c.lno
group by c.bid;
```

Execution of Query:

```
postgres=# \i q1.sql
```

Output:

```
 bid | count
-----+----------
  2  |    1
(1 row)
```

Views:

1. Create a view which contains the details of all customers who have applied for a loan more than ₹ 100000.

```
create view V1 as
select cname from customer a, loan_application b , ternary c
where cast(lamtrequired as numeric) > 100000
and a.cno=c.cno
and b.lno=c.lno
```

Execution of View:

```
postgres=# \i view1.sql
CREATE VIEW
postgres=# select * from v1;
```

Output:

```
 cname
----------------
 KIRAN
 AVINASH
(2 rows)
```

2. Create a view which contains details of all loan applications from the 'Shivajinagar' branch.

```
create view V2 as
select  a.brname,b.cname,c.lamtrequired,c.lamtapproved  from  branch  a,
customer b,loan_application c, ternary d
where  a.bid=d.bid
and b.cno=d.cno
and c.lno=d.lno
```

Execution of View:

```
postgres=# \i view2.sql
CREATE VIEW
```

3. Write the following Queries, on the above created views:
   (a) List the details of customers who have applied for a loan of ₹ 100000.

Execution of View:

```
Postgres = # select * from v2 where cast(lamtrequired as numeric) >
                                                          100000;
```

Output:

brname	cname	lamtrequired	lamtapproved
CAMP	KIRAN	$200,000.00	$100,000.00
MGROAD	AVINASH	$500,000.00	$300,000.00

(2 rows)

(b) List the details of loan applications from 'MGROAD', where loan amount is > ₹ 10000.

```
select * from v2 where brname='MGROAD'
and cast(lamtapproved as numeric) > 100000;
```

Execution of View:

```
postgres=# \i v2.sql
```

Output:

brname	cname	lamtrequired	lamtapproved
MGROAD	AVINASH	$500,000.00	$300,000.00

(1 row)

(c) List the details of Loan applications, with the same loan amount.

```
select cname, lamtapproved from v2 where lamtapproved in
(select lamtapproved from v2 group by  lamtapproved
having COUNT(*) > 1)
```

Execution of View:

```
postgres=# \i v3.sql
```

Output:

cname	lamtapproved
KIRAN	$300,000.00
SEEMA	$100,000.00

(2 rows)

Stored Functions:

1. Write a function that returns the total number of customers of a particular branch. (Accept branch name as input parameter).

```
create or replace function f1(name1 text) returns integer as '
declare
 cnt integer;
begin
select into cnt count(*) from branch a, customer b , ternary c
```

```
where a.brname=name1
and a.bid=c.bid
and b.cno=c.cno;

return cnt;
end;
'LANGUAGE 'plpgsql';
```

Execution of Code:

```
postgres=# \i f1.sql
CREATE FUNCTION
postgres=# select f1('CAMP');
```

Output:

```
 f1
---------
 1
```

(1 row)

Execution of Code:

```
postgres=# select f1('MGROAD');
```

Output:

```
 f1
----------
 2
```

(1 row)

2. Write a function to find the maximum loan amount approved.

```
create or replace function f2() returns text as '
declare
 name text;
 amt money;
begin
select  into  name  ,amt  a.brname,max(lamtapproved)  from  branch  a,
loan_application b , ternary c
where  a.bid=c.bid and b.lno=c.lno group by a.brname;
name := name || amt;
return name;
end;
'LANGUAGE 'plpgsql';
```

Execution of Code:

```
postgres=# \i f1.sql
CREATE FUNCTION
postgres=# select f2();
```

Output:

```
      f2
-----------------------------------------------
MGROAD              $300,000.00
(1 row)
```

Errors and Exceptions:

1. Write a stored function to print the total number of customers of a particular branch. (Accept branch name as input parameter). In case the branch name is invalid, raise an exception for the same.

```
create or replace function f6(name1 text) returns integer as '
declare
 cnt integer;
begin
select into cnt  count(*) from branch where brname=name1;
       IF cnt = 0 THEN
            RAISE EXCEPTION ''Invalid Branch Name %'',name1;
        END IF;
select into cnt count(*) from branch a, customer b , ternary c
where a.brname=name1
and a.bid=c.bid
and b.cno=c.cno;
return cnt;
end;
'LANGUAGE 'plpgsql';
```

Execution of Code:

```
postgres=# \i h2.sql
CREATE FUNCTION
postgres=# select f6('pune');
```

Output:

```
ERROR:  Invalid Branch Name pune
```

2. Write a stored function to increase the loan approved amount for all loans by 20%. In case the initial loan approved amount was less than Rs 10000, then print a notice to the user, before updating the amount .

```
create or replace function f7() returns integer as '
declare
 amt money;
begin
select into amt lamtapproved from loan_application b , ternary c
where b.lno=c.lno;
       IF cast(amt as numeric) < 10000 THEN
            RAISE NOTICE ''Amount less than 10000'';
            Exit;
       END IF;
    update loan_application
    set lamtapproved = lamtapproved + lamtapproved *0.2 ;
return 0;
end;
'LANGUAGE 'plpgsql';
```

Execution of Code:

```
postgres=# \i u1.sql
CREATE FUNCTION
postgres=# select f7();
```

Output:

```
   f7
-----------
   0
```

(1 row)

Execution of Code:

```
postgres=# select * from loan_application;
```

Output:

lno	lamtrequired	lamtapproved	l_date
1	$200,000.00	$100,000.00	2016-03-01
2	$500,000.00	$300,000.00	2015-03-01
3	$300,000.00	$100,000.00	2017-02-01

(3 rows)

Triggers:

1. Write a trigger before deleting a customer record from the customer table. Raise a notice and display the message "customer record is being deleted".

```
create or replace function print_notice() returns trigger as'
declare
begin
       RAISE NOTICE ''deleteing Customer data ..'';
return NULL;
end;
'LANGUAGE 'plpgsql';
```

Execution of Code:

```
CREATE TRIGGER check_customers
    BEFORE DELETE  ON customer
    FOR EACH ROW
    EXECUTE PROCEDURE print_notice();
postgres=# \i t3.sql
CREATE FUNCTION
postgres=# \i t4.sql
CREATE TRIGGER
```

Output:

```
postgres=# delete  from customer where cno=1;
NOTICE:  deleteing Customer data ..
DELETE 0
postgres=#
```

2. Write a trigger to ensure that the loan amount entered never < 1000 and greater than 1000000.

```
create or replace function loan_data() returns trigger as'
declare
begin
if cast(NEW.lamtapproved as numeric) < 1000 or cast(NEW.lamtapproved as
numeric) > 1000000 then
   raise exception '' LOAN AMT is NOT VALID'';
   end if;
return NEW;
end;
'LANGUAGE 'plpgsql';
    CREATE TRIGGER check_loan
    BEFORE INSERT ON loan_application
    FOR EACH ROW
    EXECUTE PROCEDURE loan_data();
```

Execution of Code:

```
postgres=# \i l1.sql
CREATE FUNCTION
postgres-# \i t2.sql
CREATE TRIGGER
```

Output:

```
postgres=#   insert   into   loan_application   values(6,500,500,'2017-4-
                                                                    14');
ERROR:   LOAN AMT is NOT VALID
postgres=#
```

2. STUDENT-TEACHER DATABASE

- Consider the following database maintained by a college. It gives information about students and the teachers along with the subject taught by the teacher and the marks obtained by the student in the subject.
- Following are the tables:
  student (sno integer, sname char(30), sclass char(10), saddr char(50))
  teacher (tno integer, tname char (20), qualification char (15),experience integer)
- The relationship is as follows:
  Student_teacher: M-M relation with descriptive attribute as subject name and marks.

```
create table student
(
 sno integer primary key,
 sname char(30),
 sclass char(10),
 saddr char(50)
);
create table teacher
(
 tno integer primary key,
 tname char(30),
 qualification  char(15),
 experience integer
);
create table student_teacher
(
 sno integer references student(sno),
 tno integer  references teacher(tno),
 subject  char(30),
 marks integer
);
```

Query:

```
postgres=# \i ass2.sql
CREATE TABLE

insert into student values(1,'Amol','fybca','pune');
insert into student values(2,'Archana','fybca','pune');
insert into student values(3,'kiran','sybca','pune');
insert into student values(4,'Amar','sybca','pune');
insert into student values(5,'Suresh','tybca','pune');

insert into teacher values(1,'Kumar','PhD',10);
insert into teacher values(2,'Veena','MPhil',16);
insert into teacher values(3,'Ketaki','MCS',2);
insert into teacher values(4,'Manisha','MCA',5);
insert into teacher values(5,'Prakash','MCA',3);

insert into student_teacher values(5,1,'c',60);
insert into student_teacher values(4,1,'c++',75);
insert into student_teacher values(2,1,'c',80);
insert into student_teacher values(3,2,'DS',78);
insert into student_teacher values(4,3,'ADBMS',70);
insert into student_teacher values(5,3,'ADBMS',50);
insert into student_teacher values(1,4,'cn',66);
insert into student_teacher values(2,5,'se',73);
insert into student_teacher values(3,5,'se',85);
```

Execution of Query:

```
postgres=# select * from student;
```

Output:

sno	sname	sclass	saddr
1	Amol	fybca	Pune
2	Archana	fybca	Pune
3	Kiran	sybca	Pune
4	Amar	sybca	Pune
5	Suresh	tybca	Pune

(5 rows)

Execution of Query:

```
postgres=# select * from teacher;
```

Output:

tno	tname	qualification	experience
1	Kumar	PhD	10
2	Veena	MPhil	16
3	Ketaki	MCS	2
4	Manisha	MCA	5
5	Prakash	MCA	3

(5 rows)

Execution of Query:

```
postgres=# select * from student_teacher;
```

Output:

sno	tno	subject	marks
5	1	C	60
4	1	C++	75
2	1	C	80
3	2	DS	78
4	3	ADBMS	70
5	3	ADBMS	50
1	4	CN	66
2	5	SE	73
3	5	SE	85

(9 rows)

Solve the following queries:

1. Find the minimum experienced teacher.

```
select tname, experience from teacher
where experience = ( select min(experience) from teacher);
```

Execution of Query:

```
postgres=# \i a3.sql
```

Output:

tname	experience
Ketki	10

(1 row)

2. Find the number of teachers having qualification "Ph. D.".

```
select count(*) from teacher
where qualification='PhD';
```

Execution of Query:

```
postgres=# \i a3.sql
```

Output:

```
 count
----------
   1
(1 row)
```

3. List the names of the students taught by "Mr. Kumar" along with the subjects taught.

```
select sname,subject
from student a,teacher b, student_teacher c
where tname='Kumar'
and a.sno=c.sno
and b.tno=c.tno;
```

Execution of Query:

```
postgres=# \i a3.sql
```

Output:

```
     sname        | subject
--------------------+--------------------
     Suresh       |    C
     Amar         |    C++
     Archana      |    C
(3 rows)
```

4. Find the subjects taught by each teacher.

```
select distinct c.tno,tname,subject
from teacher b, student_teacher c
where b.tno=c.tno
order by c.tno,tname,subject;
```

Execution of Query:

```
postgres=# \i a3.sql
```

Output:

```
tno  |   tname      |  subject
------+--------------+----------------
  1  | Kumar        |   C
  1  | Kumar        |   C++
  2  | Veena        |  DS
  3  | Ketaki       |  ADBMS
  4  | Manisha      |  CN
  5  | Prakash      |  SE
(6 rows)
```

5. List the names of the teachers who are teaching to a student named "Suresh".

```
select tname
from student a, teacher b, student_teacher c
```

```
where sname='Suresh'
and a.sno=c.sno
and b.tno=c.tno;
```

Execution of Query:

```
postgres=# \i a3.sql
```

Output:

```
    tname
-----------------
  Kumar
  Ketaki
(2 rows)
```

6. List the names of all teachers along with the total number of students they are teaching.

```
select c.tno,b.tname, count(c.tno)
from  teacher b, student_teacher c
where b.tno=c.tno
group by c.tno , b.tname;
```

Execution of Query:

```
postgres=# \i a3.sql
```

Output:

tno	tname	count
5	Prakash	2
3	Ketaki	2
4	Manisha	1
1	Kumar	3
2	Veena	1

(5 rows)

7. Find the student having maximum marks in the subjects taught by "Mr. Kumar"

```
select sname ,marks
from student a,teacher b, student_teacher c
where marks = ( select max(marks)
                from student a,teacher b, student_teacher c
      where tname='Kumar'
      and a.sno=c.sno
      and b.tno=c.tno)
 and a.sno=c.sno
and b.tno=c.tno;
```

Execution of Query:

```
postgres=# \i a4.sql
```

Output:

sname	marks
Archana	80

(1 row)

Views :

1. Create a view containing details of all the teachers teaching the subject 'Mathematics'.

```
create view ass1 as
select distinct b.tname,b.experience
from teacher b, student_teacher c
where subject='c'
and b.tno=c.tno;
```

Execution of View:

```
postgres=# \i a4.sql
CREATE VIEW
postgres=# select * from ass1;
```

Output:

tname	experience
Kumar	10

(1 row)

2. Create a view to list the details of all the students who are taught by a teacher having experience of more than 3 years.

```
create view ass3 as
select distinct a.sname,a.sclass
from student a, teacher b, student_teacher c
where a.sno=c.sno
and b.tno=c.tno
group by sname,sclass,b.experience
having b.experience > 3;
```

Execution of View:

```
postgres=# \i a4.sql
CREATE VIEW
postgres=# select * from ass3;
```

Output:

sname	sclass
Amar	sybca
Amol	fybca
Archana	fybca
Kiran	sybca
Suresh	tybca

(5 rows)

3. Write the following Queries, on the above created views :

(a) List the name of the most experienced teacher for "Mathematics".

```
select max(experience)
from ass1;
```

Execution of View:

```
postgres=# \i v4.sql
```

Output:

max
10

(1 row)

(b) List the names of students of 'S.Y.B.C.A.' class, who are taught by a teacher having more than 3 years experience.

```
select sname from ass3
where sclass = 'sybca';
```

Execution of View:

```
postgres=# \i v4.sql
```

Output:

sname
Amar
kiran

(2 rows)

Stored Functions:

1. Write a function to accept teacher name as input and returns the number of students taught by the teacher.

```
create or replace function tech_cnt(name text) returns int as '
declare
```

```
 cnt integer;
begin
select into cnt count(*)
from student a, teacher b, student_teacher c
where tname= name
and a.sno=c.sno
and b.tno=c.tno;
return cnt;
end;
'LANGUAGE 'plpgsql';
```

Execution of Code:

```
postgres=# \i f1.sql
CREATE FUNCTION
postgres=# select tech_cnt('Kumar');
```

Output:

```
tech_cnt
---------------
     3
(1 row)
```

```
postgres=# select tech_cnt('Veena');
```

Output:

```
tech_cnt
--------------
   1
(1 row)
```

2. Write a function to accept name of subject and count the number of teachers who teach that subject.

```
create or replace function sub_cnt(name text) returns int as '
declare
 no integer ;
 cnt integer;
begin
select into cnt  distinct count(tno)
from  student_teacher
where subject= name
group by tno;
return cnt;
end;
'LANGUAGE 'plpgsql';
```

Execution of Code:

```
postgres=# select sub_cnt('DS');
```

Output:

```
 sub_cnt
--------------
     1
(1 row)
```

Execution of Code:

```
postgres=# \i f1.sql
CREATE FUNCTION
postgres=# select sub_cnt('c');
```

Output:

```
 sub_cnt
--------------
     2
(1 row)
```

3. Write a function to accept student name and calculate the total marks obtained by that student.

```
create or replace function sub_sum(name text) returns int as '
declare
 cnt integer;
begin
select into cnt  sum(marks)
from  student a , teacher b, student_teacher c
where sname= name
and a.sno=c.sno
and b.tno=c.tno
group by c.sno;
return cnt;
end;
'LANGUAGE 'plpgsql';
```

Output:

```
postgres=# \i f1.sql
CREATE FUNCTION
postgres=# select sub_sum('Suresh');
```

```
 sub_sum
------------
   110
(1 row)
```

Cursors:

1. Write a stored function using cursors to accept student name from the user and find the names of all teachers and subjects taught to the student.

```
create or replace function tech_name(name text) returns int as '
declare
 c1 cursor for select tname , subject
from  student a , teacher b, student_teacher c
where sname= name
and a.sno=c.sno
and b.tno=c.tno;
name1 text;
sub text;
begin
open c1;
loop
fetch c1 into name1,sub;
EXIT WHEN NOT FOUND;
raise notice ''% %'',name1,sub;
end loop;
close c1;
return 1;
end;
'LANGUAGE 'plpgsql';
```

Execution of Code:

```
postgres=# select tech_name('Suresh');
```

Output:

```
NOTICE:  Kumar        c
NOTICE:  Ketaki       ADBMS
```

tech_name
1

(1 row)

2. Write a stored function using cursors which will calculate total number of subjects taught by each teacher.

```
create or replace function tech_cnt() returns int as '
declare
 c2 cursor select distinct subject,tname,count(distinct(b.tno))
from teacher a, student_teacher b
where a.tno=b.tno
```

```
group by subject,tname;
name1 text;
sub1 text;
cnt integer;
begin
open c2;
loop
fetch c2 into name1,cnt;
EXIT WHEN NOT FOUND;
raise notice ''% % '',name1,cnt;
end loop;
close c2;
return 1;
end;
'LANGUAGE 'plpgsql';
```

Errors and Exceptions:

1. Write a stored function to accept teacher name as input parameter and print the total number of students taught by the teacher. In case the teacher name is invalid, raise an exception for the same.

```
create or replace function tech_cnt(name text) returns int as '
declare
 cnt integer;
begin
select into cnt count(*)
from student a, teacher b, student_teacher c
where tname= name
and a.sno=c.sno
and b.tno=c.tno;
if cnt=0 then
raise exception '' Invalid teacher '';
end if;
return cnt;
end;
'LANGUAGE 'plpgsql';
```

Execution of Code:

```
postgres=# \i f3.sql
CREATE FUNCTION
postgres=# select tech_cnt('Kumar');
```

Output :

```
 tech_cnt
----------
    3
```

(1 row)

Execution of Code:

```
postgres=# select tech_cnt('Kk');
```

Output :

```
ERROR:   Invalid teacher
```

2. Write a stored function to increase the marks of each student to 40 if the marks are between 35 and 40. Print a notice to the user, before updating the marks.

```
create or replace function upd_mk() returns int as '
 declare
 c2 cursor for select sno, marks from student_teacher where marks between
35 and 40;
rno integer;
mks  integer;
begin
open c2;
loop
fetch c2 into rno,mks;
EXIT WHEN NOT FOUND;
update student_teacher
set marks=40
where sno=rno;
raise notice ''% marks are updated '',rno;
end loop;
close c2;
return 1;
end;
'LANGUAGE 'plpgsql';
postgres=# insert into student_teacher values(1,1,'c',35);
INSERT 0 1
postgres=# insert into student_teacher values(1,2,'DS',38);
INSERT 0 1
```

Execution of Code:

```
postgres=# \i cur1.sql
CREATE FUNCTION
postgres=# select upd_mk();
```

Output :

```
NOTICE:  1 marks are updated
NOTICE:  1 marks are updated
```

upd_mk
1

(1 row)

Execution of Code:

```
postgres=# select * from student_teacher ;
```

Output :

sno	tn0	subject	marks
5	1	C	60
4	1	C++	75
2	1	C	80
3	2	DS	78
4	3	ADBMS	70
5	3	ADBMS	50
2	5	SE	73
3	5	SE	85
1	4	CN	40
1	1	C	40
1	2	DS	40

(11 rows)

Triggers:

1. Write a trigger before deleting a student record from the student table. Raise a notice and display the message "student record is being deleted" .

```
CREATE TRIGGER  del_student
    BEFORE DELETE  ON student
    FOR EACH ROW
    EXECUTE PROCEDURE print_notice_stud();
create or replace function print_notice_stud() returns trigger as '
declare
begin
            RAISE NOTICE ''deleting Student data ..'';
return null;
end;
'LANGUAGE 'plpgsql';
```

Execution of Code:

```
postgres=# \i c4.sql
CREATE FUNCTION
postgres=# \i trg.sql
CREATE TRIGGER
```

2. Write a trigger to ensure that the marks entered for a student, with respect to a subject is never < 0 and greater than 100.

```
create or replace function chk_stud() returns trigger as '
declare
mk integer;
begin
if NEW.marks < 0 or NEW.marks > 100 then
raise notice ''Marks should be never < 0 or Marks should be never > 100'';
end if;
return null;
end;
'LANGUAGE 'plpgsql';
    CREATE TRIGGER  chk_marks
    BEFORE INSERT  ON student_teacher
    FOR EACH ROW
    EXECUTE PROCEDURE chk_stud();
```

Execution of Code:

```
postgres=# \i c5.sql
CREATE FUNCTION
postgres=# \i trg1.sql
CREATE TRIGGER
```

Output:

```
postgres=# insert into student_teacher values(1,3,'ADBMS',105);
NOTICE:  Marks should be never < 0 or Marks should be never > 100
postgres=# insert into student_teacher values(1,3,'ADBMS',105);
NOTICE:  Marks should be never < 0 or Marks should be never > 100
```

3. MOVIE DATABASE

- Consider the following Movie database.
  movies (mno integer, mname varchar (25), release_year integer, budget money)
  actor (ano integer, a_name char (30),role char (30), charges money, a_address varchar (30))
  producer(pno integer, name char (30), p_address varchar (30))

Each actor has acted in one or more movies. Each producer has produced many movies and each movie can be produced by more than one producers. Each movie has one or more actors acting in it, in different roles.

- Create the above database in PostGreSQL.

```
create table movies
(
 mno integer primary key,
 mname char(30),
 release_year integer ,
 budget money
);
create table actor
(
 ano integer primary key,
 a_name char(30),
 role  char(15),
 a_address char(30),
charges money
);
create table producer
(
 pno integer primary key,
 pname char(30),
 p_address char(30)
);
create table mov_act_pro
(
 mno integer references movies(mno),
 ano integer  references actor(ano),
 pno integer  references producer(pno)
);
insert into movies values(1,'PK',2015,1000000);
insert into movies values(2,'piku',2014,20000000);
insert into movies values(3,'Bajirao mastani',2016,30000000);
insert into movies values(4,'Dangal',2016,50000000);
insert into movies values(5,'Student of Year',2015,25000000);

insert into actor values(1,'Amitabh','Hero', 'Mumbai',100000000);
insert into actor values(2,'Amir','Hero','Delhi',200000000);
insert into actor values(3,'Deepika','Heroine','Pune',10000000);
insert into actor values(4,'Anushka','Heroine','Chennai',5000000);
insert into actor values(5,'Aliya','Heroine','Delhi',6000000);
insert into actor values(6,'Varun','Villan','Pune',30000000);
```

```
insert into producer values(1,'Karan','Mumbai');
insert into producer values(2,'Shetty','Pune');
insert into producer values(3,'Khan','Mumbai');
insert into producer values(4,'Salman','Mumbai');
insert into producer values(5,'Amir','Delhi');

insert into mov_act_pro values(2,1,1);
insert into mov_act_pro values(2,3,2);
insert into mov_act_pro values(1,2,5);
insert into mov_act_pro values(1,4,5);
insert into mov_act_pro values(3,3,4);
insert into mov_act_pro values(3,5,4);
insert into mov_act_pro values(4,6,3);
insert into mov_act_pro values(4,4,5);
```

Execution of Query:

```
postgres=# select * from movies;
```

Output:

mno	mname	release_year	budget
1	PK	2015	$1,000,000.00
2	Piku	2014	$20,000,000.00
3	Bajirao mastani	2016	$30,000,000.00
4	Dangal	2016	$50,000,000.00
5	Student of Year	2015	$25,000,000.00

(5 rows)

Execution of Query:

```
postgres=# select * from actor;
```

Output:

ano	a_name	role	a_address	charges
1	Amitabh	Hero	Mumbai	$100,000,000.00
2	Amir	Hero	Delhi	$200,000,000.00
3	Deepika	Heroine	Pune	$10,000,000.00
4	Anushka	Heroine	Chennai	$5,000,000.00
5	Aliya	Heroine	Delhi	$6,000,000.00
6	Varun	Villan	Delhi	$30,000,000.00

(6 rows)

Execution of Query:

```
postgres=# select * from producer;
```

Output:

pno	pname	p_address
1	Karan	Mumbai
2	Shetty	Pune
3	Khan	Mumbai
4	Salman	Mumbai
5	Amir	Delhi

(5 rows)

Execution of Query:

```
postgres=# select * from mov_act_pro;
```

Output:

mno	ano	pno
2	1	1
2	3	2
1	2	5
1	4	5
3	3	4
3	5	4
4	6	3
4	4	5

(8 rows)

Execute the following queries in PostgreSQL:

1. List the names of actors who have acted in at least one movie, in which Mr. Amir has acted.

```
select a_name, mname from movies a,actor b , mov_act_pro c
where c.mno = (select c.mno from movies a,actor b , mov_act_pro c
where a_name='Amir'
and a.mno=c.mno
and b.ano=c.ano)
and a.mno=c.mno
and b.ano=c.ano
group by  a_name,mname
having count(*)>= 1;
```

Execution of Query:

```
postgres=# \i as3.sql
```

Output:

a_name	mname
Anushka	PK
Amir	PK

(2 rows)

2. List the names of the actors and their movie names.

```
select a_name, mname from movies a,actor b , mov_act_pro c
where  a.mno=c.mno
and b.ano=c.ano;
```

Execution of Query:

```
postgres=# \i as3.sql
```

Output:

a_name	mname
Amitabh	Piku
Amir	PK
Anushka	PK
Deepika	Bajirao mastani
Aliya	Bajirao mastani
Varun	Dangal
Anushka	Dangal

(8 rows)

3. List the names of movies whose producer is "Mr. Khan"

```
select pname, mname from movies a,producer c, mov_act_pro d
where pname='Khan'
and a.mno=d.mno
and c.pno=d.pno;
```

Execution of Query:

```
postgres=# \i as3.sql
```

Output:

pname	mname
Khan	Dangal

(1 row)

4. List the names of the movies with the highest budget.

```
select   mname , budget
from movies where budget =(select max(budget) from movies);
```

Execution of Query:

```
postgres=# \i as3.sql
```

Output:

mname	budget
Dangal	$50,000,000.00

(1 row)

5. List the names of movies released after 2000.

```
select   mname , release_year
from movies where release_year > 2000 ;
```

Execution of Query:

```
postgres=# \i as3.sql
```

Output:

mname	release_year
PK	2015
Piku	2014
Bajirao mastani	2016
Dangal	2016
Student of the Year	2015

(5 rows)

6. List the names of actors who played the role of 'Villan'.

```
select   a_name , role
from actor
where role = 'Villan' ;
```

Execution of Query:

```
postgres=# \i as3.sql
```

Output:

a_name	role
Varun	Villan

(1 row)

7. List the names of actors who are given the maximum charges for their movie along with movie name and release year.

```
select   a_name , mname , release_year, charges
```

```
from movies  a, actor b, mov_act_pro c
where  charges = (select max(charges) from actor )
and a.mno=c.mno
and b.ano=c.ano;
```

Execution of Query:

```
postgres=# \i as3.sql
```

Output:

a_name	mname	release_year	charge
Amir	PK	2015	$200,000,000.00

(1 row)

8. Display count and total budget of all movies released in year 2015.

```
select   count(mno), sum(budget)
from movies
where  release_year = 2015 ;
```

Execution of Query:

```
postgres=# \i as3.sql
```

Output:

count	sum
2	$26,000,000.00

(1 row)

Cursor and Triggers:

1. Write a trigger before inserting into a movie table to check budget. Budget should be minimum 50 lakh. Display appropriate message.

```
create or replace function chk_budget() returns trigger as '
declare
begin
if cast(NEW.budget as numeric)  <  50000 then
   raise notice ''Budget should be never < 50000 '';
end if;
return null;
end;
'LANGUAGE 'plpgsql';
CREATE TRIGGER  Trg_budget1
    BEFORE INSERT  ON movies
```

```
FOR EACH ROW
EXECUTE PROCEDURE chk_budget();
```

Execution of Code:

```
postgres=# \i as3.sql
CREATE FUNCTION
postgres=# \i trg2.sql
CREATE TRIGGER
```

Output:

```
postgres=# insert into movies values(6,'Dhoom 3',2013,2000);
NOTICE:  Budget should be never < 50000
INSERT 0 0
```

2. Write a stored function using cursors to display the names of actors who have acted in the maximum number of movies.

```
postgres=# insert into mov_act_pro values(4,3,2);
INSERT 0 1
```

Execution of Code:

```
postgres=# select * from mov_act_pro;
```

Output:

mno	ano	pno
2	1	1
2	3	2
1	2	5
1	4	5
3	3	4
3	5	4
4	6	3
4	4	5
4	3	2

(9 rows)

```
create or replace function print_act() returns int as '
declare
 c2 cursor for select a_name , count(b.mno)
from actor a, mov_act_pro b where a.ano=b.ano
group by a_name order by count(b.mno) desc Limit 1;
name char(30);
cnt integer;
```

```
begin
   open c2;
loop
fetch c2 into name,cnt;
EXIT WHEN NOT FOUND;
raise notice ''% %'',name, cnt ;
end loop;
close c2;
return 1;
end;
'LANGUAGE 'plpgsql';
```

Execution of Code:

```
postgres=# \i cur3.sql
CREATE FUNCTION
postgres=# select print_act();
NOTICE:  Deepika  3
```

Output:

```
 print_act
----------------
    1
(1 row)
```

Stored Functions:

1. Write a function to list moviewise charges of 'Amitabh Bachchan'.

```
create or replace function print_chrge(act_nm text) returns int as '
declare
   rec record;
begin
for rec in select mname, charges
from  movies a , actor b,  mov_act_pro c
where a_name= act_nm and a.mno=c.mno and b.ano=c.ano
loop
   raise notice ''% %'',rec.mname,rec.charges;
end loop;
return 1;
end;
'LANGUAGE 'plpgsql';
```

Execution of Code:

```
postgres=# \i try4.sql
CREATE FUNCTION
postgres=# select print_chrge('Amitabh');
```

Output:

```
NOTICE: piku $100,000,000.00
 print_chrge
----------------
    1
(1 row)
```

2. Write a stored function to accept producer name as input and print the names of movies produced by him/her. Also print the total number of actors in that movie.

```
create or replace function print_prod(act_nm text) returns int as '
declare
   rec record;
   rec1 record;
begin
for rec in select distinct c.mno, mname ,pname
from  movies a , mov_act_pro c , producer d
where pname= act_nm and c.pno=d.pno and a.mno=c.mno
loop
for rec1 in select count(*) as cnt from mov_act_pro where mno=rec.mno
loop
   raise notice ''% %'',rec.mname,rec1.cnt;
end loop;
end loop;
return 1;
end;
'LANGUAGE 'plpgsql';
```

Execution of Code:

```
postgres=# \i t4.sql
CREATE FUNCTION
postgres=# select print_prod('Amir');
```

Output:

```
NOTICE:  PK    2
NOTICE:  Dangal 3
 print_prod
----------------
    1
(1 row)
```

3. Write a stored function to accept movie name as input and print the names of actors working in the movie.

```
create or replace function print_name(mov_nm text) returns int as '
```

```
declare
   rec record;
begin
for rec in select a_name
from  movies a ,  actor b, mov_act_pro c
where mname= mov_nm and b.ano=c.ano and a.mno=c.mno
loop
   raise notice ''% '',rec.a_name;
end loop;
return 1;
end;
'LANGUAGE 'plpgsql';
```

Execution of Code:

```
postgres=# \i t4.sql
CREATE FUNCTION
postgres=# select print_name('Dangal');
```

Output:

```
NOTICE:  Varun
NOTICE:  Anushka
NOTICE:  Deepika
```

print_name
1

(1 row)

Views:

1. List the names of producers who produce the same movie as 'Karan Johar' has produced.

```
create view V1 as
select distinct pname
from  movies a ,   mov_act_pro c , producer d
where  c.mno=  (  select  distinct  c.mno  from  movies  a  ,  mov_act_pro  c,
                                                                  producer d
where pname= 'Karan' and c.pno=d.pno and a.mno=c.mno)
and c.pno=d.pno;
```

Execution of View:

```
postgres=# \i v1.sql
CREATE VIEW
postgres=# select * from v1;
```

Output:

```
   pname
-------------------
  Karan
  Shetty
```

(2 rows)

2. List the names of actors who do not live in Mumbai or Pune city.

```
create view V2 as
select  a_name
from  actor
where a_address not in ('Mumbai','Pune');
```

Execution of View:

```
postgres=# \i v1.sql
CREATE VIEW
postgres=# select * from v2;
```

Output:

```
      a_name
---------------------------
Amir
Anushka
Aliya
```

(3 rows)

3. List the actors in each movie.

```
create view V3 as
select  mname, a_name
from  movies a ,actor b, mov_act_pro c
where b.ano=c.ano and a.mno=c.mno group by mname,a_name
order by mname,a_name;
```

Execution of Code:

```
postgres=# \i v1.sql
CREATE VIEW
postgres=# select * from v3;
```

Output:

```
     mname            |  a_name
---------------------- + ----------------------------
  Bajirao mastani     |    Aliya
  Bajirao mastani     |    Deepika
  Dangal              |    Anushka
```

Dangal	|	Deepika
Dangal	|	Varun
Piku	|	Amitabh
Piku	|	Deepika
PK	|	Amir
PK	|	Anushka

(9 rows)

Exceptions:

1. Write a stored function to accept movie name as input and print the names of actors working in the movie. Also print the total number of actors working in that movie. Raise an exception for an invalid movie name.

```
create or replace function print_name1(mov_nm text) returns int as '
declare
rec record;
total integer:=0;
begin
select into total count(*)
from  movies a ,  actor b, mov_act_pro c
where mname= mov_nm
and b.ano=c.ano
and a.mno=c.mno
group by a_name;
IF NOT FOUND  THEN
   raise exception  ''Movie  % not found'', mov_nm;
END IF;
total:=0;
for rec in select a_name ,count(*)  as cnt
from  movies a ,  actor b, mov_act_pro c
where mname= mov_nm and b.ano=c.ano and a.mno=c.mno
group by a_name
loop
   raise notice ''% %'',rec.a_name,rec.cnt;
   total :=total + rec.cnt;
end loop;
raise notice ''Total actors=%'',total;
return 1;
end;
'LANGUAGE 'plpgsql';
```

Execution of Code:

```
postgres=# \i t4.sql
CREATE FUNCTION
postgres=# select print_name1('kll');
```

Output:

```
ERROR:  Movie  kll not found
postgres=# select print_name1('Dangal');
NOTICE:  Deepika    1
NOTICE:  Varun      1
NOTICE:  Anushka    1
NOTICE:  Total actors=3
```

print_name1
1

(1 row)

2. Write a stored function to accept producer name as input and print the names of Movies produced by him/her..Also print the total number of actors in that movie. Raise an exception for an invalid producer name.

```
create or replace function print_prod1(act_nm text) returns int as '
declare
rec record;
rec1 record;
total integer :=0;
begin
select into total distinct c.mno
from  movies a ,   mov_act_pro c , producer d
where pname= act_nm
and c.pno=d.pno and a.mno=c.mno;
IF NOT FOUND  THEN
   raise exception  ''Invalid Producer Name'';
END IF;
total:=0;
for rec in select distinct c.mno, mname ,pname
from  movies a ,   mov_act_pro c , producer d
where pname= act_nm
and c.pno=d.pno and a.mno=c.mno
```

```
loop
for rec1 in select count(*) as cnt from mov_act_pro where mno=rec.mno
   loop
   raise notice ''% %'',rec.mname,rec1.cnt;
   end loop;
end loop;
return 1;
end;
'LANGUAGE 'plpgsql';
```

Execution of Code:

```
postgres=# \i t5.sql
CREATE FUNCTION
postgres=# select print_prod1('gg');
```

Output:

```
ERROR:  Invalid Producer Name
postgres=# select print_prod1('Amir');
NOTICE:  Dangal   3
NOTICE:  PK       2
```

print_prod1
1

(1 row)

4. STUDENTS-MARKS DATABASE

- Consider the following Student-Marks database:
  Student(rollno integer, name varchar(30), address varchar(50), class varchar(10))
  Subject(scode varchar(10), subject_name varchar(20))
  Student-Subject are related with M-M relationship with attributes marks_scored.
  Create the above database in PostGreSQL.

```
create table student
(
 sno integer primary key,
 sname char(30),
 sclass char(10),
 saddr char(50)
);
create table subject(
 scode varchar(10)  primary key,
```

```
 sub_name  varchar(30)
);
create table stud_sub
(
sno integer references student(sno),
scode varchar(10)  references subject(scode),
marks  integer
);
insert into stud values(1,'Amol','fybca','pune');
insert into stud values(2,'Archana','fybca','pune');
insert into stud values(3,'kiran','sybca','pune');
insert into stud values(4,'Omkar','sybca','pune');
insert into stud values(5,'Suresh','tybca','pune');
insert into stud values(6,'Reena','tybca','Mumbai');

insert into subject values('BCA-301','DS');
insert into subject values('BCA-302','ARDBMS');
insert into subject values('BCA-303','SE');
insert into subject values('BCA-304','CN');
insert into subject values('BCA-305','LAB-I');
insert into subject values('BCA-306','LAB-II');
insert into subject values('BCA-101','C');
insert into subject values('BCA-501','JAVA');

insert into stud_sub values(3,'BCA-301',60);
insert into stud_sub values(3,'BCA-302',80);
insert into stud_sub values(3,'BCA-303',50);
insert into stud_sub values(3,'BCA-304',70);
insert into stud_sub values(3,'BCA-305',86);
insert into stud_sub values(3,'BCA-306',76);
insert into stud_sub values(4,'BCA-301',60);
insert into stud_sub values(4,'BCA-302',40);
insert into stud_sub values(4,'BCA-303',50);
insert into stud_sub values(4,'BCA-304',70);
insert into stud_sub values(4,'BCA-305',46);
insert into stud_sub values(4,'BCA-306',56);
insert into stud_sub values(1,'BCA-101',60);
insert into stud_sub values(2,'BCA-101',80);
insert into stud_sub values(5,'BCA-501',70);
insert into stud_sub values(6,'BCA-501',80);
```

Execution of Query:

```
postgres=# select * from student;
```

Output:

sno	sname	sclass	saddr
1	Amol	fybca	Pune
2	Archana	fybca	Pune
3	Kiran	sybca	Pune
4	Amar	sybca	Pune
5	Suresh	tybca	Pune
6	Reena	tybca	Mumbai

(6 rows)

Execution of Query:

```
postgres=# select * from subject;
```

Output:

scode	sub_name
BCA-301	DS
BCA-302	ARDBMS
BCA-303	SE
BCA-304	CN
BCA-305	LAB-I
BCA-306	LAB-II
BCA-101	C
BCA-501	JAVA

(8 rows)

Execution of Query:

```
postgres=# select * from stud_sub;
```

Output:

sno	scode	marks
3	BCA-301	60
3	BCA-302	80
3	BCA-303	50
3	BCA-304	70
3	BCA-305	86
3	BCA-306	76
4	BCA-301	60
4	BCA-302	40
4	BCA-303	50
4	BCA-304	70
4	BCA-305	46
4	BCA-306	56

4	BCA-306	56
1	BCA-101	60
2	BCA-101	80
5	BCA-501	70
6	BCA-501	80

(16 rows)

Execute the Following Queries:

1. Display the names of students scoring the maximum total marks.

```
select sname , sum(marks)
from student a,subject b,stud_sub c
where a.sno=c.sno
and b.scode=c.scode
group by sname
order by sum(marks) desc
limit 1;
```

Execution of Query:

```
postgres=# \i a4.sql
```

Output:

sname	sum
Kiran	422

(1 row)

2. List the distinct names of all the subjects.

Execution of Query:

```
postgres=# select * from subject;
```

Output:

scode	sub_name
BCA-301	DS
BCA-302	ARDBMS
BCA-303	SE
BCA-304	CN
BCA-305	LAB-I
BCA-306	LAB-II
BCA-101	C
BCA-501	JAVA

(8 rows)

3. Display class wise &subject wise student list.

```
select sclass, sname from student
group by sclass, sname
order by sclass, sname;
```

Execution of Query:

```
postgres=# \i a4.sql
```

Output:

sclass	sname
fybca	Amol
fybca	Archana
sybca	Amar
sybca	Kiran
tybca	Reena
tybca	Suresh

(6 rows)

```
select sclass,sub_name,sname
from student a,subject b,stud_sub c
where a.sno=c.sno
and b.scode=c.scode
group by sclass,sub_name,sname
order by sclass,sub_name,sname;
```

Execution of Query:

```
postgres=# \i a4.sql
```

Output:

sclass	sub_name	sname
fybca	C	Amol
fybca	C	Archana
sybca	ARDBMS	Amar
sybca	ARDBMS	Kiran
sybca	CN	Amar
sybca	CN	Kiran
sybca	DS	Amar
sybca	DS	Kiran
sybca	LAB-I	Amar
sybca	LAB-I	Kiran

sybca	LAB-II	Amar
sybca	LAB-II	Kiran
sybca	SE	Amar
sybca	SE	Kiran
tybca	JAVA	Reena
tybca	JAVA	Suresh

(16 rows)

Cursor and Triggers:

1. Write a stored function using cursors, to accept a address from the user and display the name, subject and the marks of the students staying at that address.

```
create or replace function print_mk(adr text) returns int as '
declare
c2 cursor for select sname,sub_name, marks
from student a,subject b,stud_sub c
where saddr=adr
and a.sno=c.sno
and b.scode=c.scode ;
name varchar(30);
sname varchar(30);
mks  integer;
begin
open c2;
loop
fetch c2 into name,sname,mks;
EXIT WHEN NOT FOUND;
raise notice ''% %  % '',name,sname,mks;
end loop;
close c2;
return 1;
end;
'LANGUAGE 'plpgsql';
```

Execution of Code:

```
postgres=# \i cur1.sql
CREATE FUNCTION
postgres=# select print_mk('pune');
```

Output:

```
NOTICE:  Amol          C  60
NOTICE:  Archana       C  80
NOTICE:  kiran         DS  60
```

```
NOTICE:  kiran          ARDBMS  80
NOTICE:  kiran          SE  50
NOTICE:  kiran          CN  70
NOTICE:  kiran          LAB-I  86
NOTICE:  kiran          LAB-II  76
NOTICE:  Amar           DS  60
NOTICE:  Amar           ARDBMS  40
NOTICE:  Amar           SE  50
NOTICE:  Amar           CN  70
NOTICE:  Amar           LAB-I  46
NOTICE:  Amar           LAB-II  56
NOTICE:  Suresh         JAVA  70
```

print_mk
1

(1 row)

Execution of Code:

```
postgres=# select print_mk('Mumbai');
NOTICE:  Reena   JAVA  80
```

Output:

print_mk
1

(1 row)

2. Write a stored function using cursors which will calculate total marks of each student.

```
create or replace function print_perc() returns int as '
declare
c2 cursor for select sclass,sname, sum(marks)
from student a,subject b,stud_sub c
where  a.sno=c.sno
and b.scode=c.scode
group by sclass,sname
order by sclass,sname ;
name varchar(30);
class varchar(30);
mks  integer;
begin
open c2;
```

```
raise notice ''Class       Name          Total Marks'';
raise notice ''-----------------------------------------------'';
loop
fetch c2 into class,name,mks;
EXIT WHEN NOT FOUND;
raise notice ''% %  % '',class,name,mks;
end loop;
close c2;
return 1;
end;
'LANGUAGE 'plpgsql';
```

Execution of Code:

```
postgres=# \i cur4.sql
CREATE FUNCTION
postgres=# select print_perc();
```

Output:

```
NOTICE:  Class      Name        Total Marks
NOTICE:  ----------------------------------------
NOTICE:  fybca      Amol           60
NOTICE:  fybca      Archana        80
NOTICE:  sybca      Amar           322
NOTICE:  sybca      kiran          422
NOTICE:  tybca      Reena          80
NOTICE:  tybca      Suresh         70
 print_perc
----------------
      1
(1 row)
```

3. Write a trigger before deleting a student record from the student table. Raise a notice.and display the message "student record is being deleted".

```
create or replace function print_notice_stud() returns trigger as '
declare
begin
     RAISE NOTICE ''deleting Student data ..'';
return null;
end;
'LANGUAGE 'plpgsql';
CREATE TRIGGER  del_student2
    BEFORE DELETE  ON student
    FOR EACH ROW
    EXECUTE PROCEDURE print_notice_stud();
```

Execution of Code:

```
postgres=# \i tr.sql
CREATE FUNCTION
CREATE TRIGGER
postgres=# delete from student where sno=6;
```

Output:

```
NOTICE:  deleting Student data ..
DELETE 0
postgres=#
```

4. Write a trigger to ensure that the marks entered for a student, with respect to a subject is never < 0 and greater than 100.

```
create or replace function chk_stud1() returns trigger as '
declare
begin
if NEW.marks < 0 or NEW.marks > 100 then
raise notice ''Marks should be never < 0 or Marks should be never > 100'';
end if;
return null;
end;
'LANGUAGE 'plpgsql';
    CREATE TRIGGER  TRG_marks
    BEFORE INSERT  ON stud_sub
    FOR EACH ROW
    EXECUTE PROCEDURE chk_stud1();
```

Execution of Code:

```
postgres=# \i trg4.sql
CREATE FUNCTION
CREATE TRIGGER
postgres=# insert into stud_sub values(3,'BCA-101',150);
```

Output:

```
NOTICE:  Marks should be never < 0 or Marks should be never > 100
INSERT 0 0
postgres=#
```

Views:

1. To list student name, class &total marks scored by each student ,sorted by student name.

```
create view v4 as
select sname ,sclass,sum(marks)
from student a,subject b,stud_sub c
where a.sno=c.sno
and b.scode=c.scode
group by sname,sclass
order by sname,sclass
```

Execution of View:

```
postgres=# \i a4.sql
CREATE VIEW
postgres=# select * from v4;
```

Output:

sname	sclass	sum
Amar	sybca	322
Amol	fybca	60
Archana	fybca	80
Kiran	sybca	422
Reen	tybca	80
Suresh	tybca	70

(6 rows)

2. To list student names along with subject name and markswho scored more than 60 marks.

```
create view v5 as
select sname ,sub_name,marks
from student a,subject b,stud_sub c
where a.sno=c.sno
and b.scode=c.scode
and marks > 60
order by sname,sclass
```

Execution of View:

```
postgres=# \i a4.sql
CREATE VIEW
postgres=# select * from v5;
```

Output:

sname	sub_name	sum
Amar	CN	70
Archana	C	80
Kiran	LAB-I	86
Kiran	LAB-II	76
Kiran	ARDBMS	80
Kiran	CN	70
Reen	JAVA	80
Suresh	JAVA	70

(8 rows)

3. Containing all the details of student named 'Amar'.

```
create view v8 as
select sname ,sclass,saddr,sub_name,marks
from student a,subject b,stud_sub c
where sname='Amar'
and a.sno=c.sno
and b.scode=c.scode ;
```

Execution of View:

```
postgres=# \i a4.sql
CREATE VIEW
postgres=# select * from v8;
```

Output:

sname	sclass	saddr	sub-name	marks
Amar	sybca	pune	DS	60
Amar	sybca	pune	ARDBMS	40
Amar	sybca	pune	SE	50
Amar	sybca	pune	CN	70
Amar	sybca	pune	LAB-I	46
Amar	sybca	pune	LAB-II	56

(6 rows)

5. BUS TRANSPORT DATABASE

- Consider the following Bus transport Database.
  bus (bus_no integer , capacity integer , depot_name varchar(20), route_no integer)
  route (route_no integer, source char(20), destination char(20), No_of_stations integer)
  driver (driver_no integer, driver_name char(20), license_no integer, address Char(20), d_age integer, salary float)
  bus_route : M-1 and Bus_Driver : M-M with descriptive attributes date of duty allotted and shift can be 1 (Morning)or 2 (Evening).
  Constraints:1.License_no is unique. 2. Bus capacity is not null.
  Create the above database in PostGreSQL.

```
create table route(
 route_no integer  primary key,
 src  varchar(30),
dest varchar(30),
no_of_station integer
);
create table bus
(
 bus_no integer primary key,
```

```
capacity integer not null,
depot_name char(30),
route_no integer references route(route_no)
);
create table driver
(
  dno integer primary key,
  dname varchar(30),
  lic_no integer unique ,
  address varchar(30),
  age integer ,
  salary float
);
create table bus_driver
(
bus_no integer references bus(bus_no),
dno  integer  references driver(dno),
date_of_duty date,
shift integer CHECK(shift in (1 ,2))
);
insert into route values(1,'Hadapsar','Katraj', 20);
insert into route values(2,'Hadapsar','Kothrud', 25);
insert into route values(3,'Hadapsar','Chinchwad', 30);
insert into route values(4,'Nigadi','Hadapsar', 30);
insert into route values(5,'Kothrud','Katraj', 22);

insert into bus values(101,35,'Hadapsar',1);
insert into bus values(102,20,'Hadapsar',2);
insert into bus values(103,32,'Hadapsar',3);
insert into bus values(104,25,'Nigadi',4);
insert into bus values(105,5,'Kothrud',5);
insert into bus values(106,25,'Kothrud',5);
insert into bus values(107,35,'Hadapsar',3);

insert into driver values(1,'Amol', 1001,'Pune',55,15000);
insert into driver values(2,'Ashok', 2001,'Karad',45,12000);
insert into driver values(3,'Bhushan', 3001,'Pune',35,10000);
insert into driver values(4,'Kiran', 4001,'Satara',25,8000);
insert into driver values(5,'Ram', 5001,'Pune',32,9000);

insert into bus_driver values(101,1,'2014-04-20',1);
insert into bus_driver values(102,2,'2014-04-20',2);
insert into bus_driver values(103,3,'2015-04-20',1);
```

```
insert into bus_driver values(104,4,'2016-04-20',2);
insert into bus_driver values(105,5,'2017-04-20',1);
insert into bus_driver values(106,3,'2015-04-20',2);
insert into bus_driver values(107,4,'2016-04-20',1);
```

Execution of Query:

```
postgres=# select * from route;
```

Output:

route_no	src	dest	no_of_station
1	Hadapsar	Katraj	20
2	Hadapsar	Kothrud	25
3	Hadapsar	Chinchwad	30
4	Nigadi	Hadapsar	30
5	Kothrud	Katraj	22

(5 rows)

Execution of Query:

```
postgres=# select * from bus;
```

Output:

bus_no	src	dest	route_no
101	35	Hadapsar	1
102	20	Hadapsar	2
103	32	Hadapsar	3
104	25	Nigadi	4
105	5	Kothrud	5
106	25	Kothrud	5
107	35	Hadapsar	3

(7 rows)

Execution of Query:

```
postgres=#
postgres=# select * from driver;
```

Output:

dno	dname	lic_no	address	age	salary
1	Amol	1001	Pune	55	15000
2	Archana	2001	Pune	45	12000
3	Kiran	3001	Pune	35	10000
4	Amar	4001	Pune	25	8000
5	Suresh	5001	Pune	32	9000

(5 rows)

Execution of Query:

```
postgres=# select * from bus_driver;
```

Output:

bus_no	dno	date_of_duty	shift
101	1	2014-04-20	1
102	2	2014-04-20	2
103	3	2015-04-20	1
104	4	2016-04-20	2
105	5	2017-04-20	1
106	3	2015-04-20	2
107	4	2016-04-20	1

(7 rows)

Execute the Following Queries:

1. Find out the name of the driver having maximum salary.

```
select dname, salary
from driver
where salary = (select max(salary) from driver );
```

Execution of Query:

```
postgres=# \i as51.sql
```

Output:

dname	salary
Amol	15000

(1 row)

2. Delete the record of bus having capacity< 10

```
postgres=# delete from bus where capacity < 10;
```

3. Increase the salary of all drivers by5% if driver's age> 45.

```
update driver
set salary = salary+salary * 0.05
where age > 45;
```

Execution of Query:

```
postgres=# \i as51.sql
UPDATE 1
postgres=# select * from driver;
```

Output:

dno	dname	lic_no	address	age	salary
2	Ashok	2001	Karad	45	12000
3	Bhushan	3001	Pune	35	10000
4	Kiran	4001	Satara	25	8000
5	Ram	5001	Pune	32	9000
1	Amol	1001	Pune	55	15750

(5 rows)

4. Find out the route details on which buses of capacity 20 run.

```
select a.route_no , src, dest , capacity from route a, bus b
where b.capacity=20 and a.route_no = b.route_no;
```

Execution of Query:

```
postgres=# \i as51.sql
```

Output:

route_no	src	dest	capacity
2	Hadapsar	Kothrud	20

(1 row)

5. Print the names and license nos. of drivers working on in both shifts.

```
select dname,lic_no from driver  a, bus_driver b
where b.shift = 1 and a.dno = b.dno
intersect
select dname,lic_no from driver  a, bus_driver b
where b.shift = 2 and a.dno = b.dno;
```

Execution of Query:

```
postgres=# \i as51.sql
```

Output:

dname	lic_no
Kiran	4001
Bhushan	3001

(2 rows)

Cursor and Trigger:

1. Define a trigger after insert or update the record of driver if the age is between 18 and 50 give the message "valid entry" otherwise give appropriate message.

```
create or replace function chk_age() returns trigger as '
declare
begin
if NEW.age > 18 and  NEW.age < 50 then
raise notice ''Valid Entry'';
else
raise exception '' Age is not appropriate'';
end if;
return null;
end;
'LANGUAGE 'plpgsql';
    CREATE TRIGGER  TRG_marks1
    AFTER INSERT  or UPDATE ON driver
    FOR EACH ROW
    EXECUTE PROCEDURE chk_age();
```

Execution of Code:

```
postgres=# \i trg5.sql
CREATE FUNCTION
CREATE TRIGGER
```

Output:

```
postgres=# insert into driver values(7,'Paul', 7001,'Pune',17,5000);
ERROR:   Age is not appropriate
postgres=# select * from driver;
```

dno	dname	lic_no	address	age	salary
1	Amol	1001	Pune	55	15750
2	Ashok	2001	Karad	45	12000
3	Bhushan	3001	Pune	35	10000
4	Kiran	4001	Satara	25	8000
5	Ram	5001	Pune	32	9000

(5 rows)

2. Define a trigger after delete the record of bus having capacity< 10. Display the message accordingly.

```
create or replace function print_notice_bus() returns trigger as '
declare
begin
  if (old.capacity < 10)  then
  RAISE   NOTICE   ''deleting   BUS   NO   %   data   havingcapacity   <   10
                                                        ...'',old.bus_no;
  end if;
return null;
end;
'LANGUAGE 'plpgsql';

CREATE TRIGGER  del_bus_trg3
    AFTER DELETE  ON bus
      FOR EACH ROW
    EXECUTE PROCEDURE print_notice_bus();
```

Execution of Code:

```
postgres=# \i trg5.sql
CREATE FUNCTION
CREATE TRIGGER
```

3. Write a stored function using cursors to display the details of a driver. (Accept driver name as input parameter).

```
create or replace function print_driver(name text) returns int as '
declare
 c2 cursor for select dname,lic_no,age,salary
 from driver where dname=name;
dname varchar(30);
dlic integer;
dage  integer;
dsal float;
begin
open c2;
loop
   fetch c2 into dname,dlic,dage,dsal;
   EXIT WHEN NOT FOUND;
   raise notice ''% %  %  %'',dname,dlic,dage,dsal;
end loop;
```

```
close c2;
return 1;
end;
'LANGUAGE 'plpgsql';
```

Execution of Code:

```
postgres=# \i cur5.sql
CREATE FUNCTION
postgres=# select print_driver('Amol');
```

Output:

```
NOTICE:  Amol 1001  55  15750
 print_driver
-------------------
        1
(1 row)
```

Stored Functions:

1. Write a stored function to print the names of drivers working on both shifts on '20/04/2014'.

```
create or replace function print_dri_shift(dt date) returns int as '
declare
rec record;
begin
for rec in  select dname
from driver a , bus_driver b where  b.date_of_duty=dt
and a.dno=b.dno and b.shift=1
intersect
select dname from driver a , bus_driver b
where  b.date_of_duty=dt and a.dno=b.dno and b.shift=2
loop
   raise notice ''%'',rec.dname;
end loop;
return 1;
end;
'LANGUAGE 'plpgsql';
```

Execution of Code:

```
postgres=# insert into bus_driver values(106,1,'2014-04-20',2);
INSERT 0 1
postgres=# \i cur5.sql
CREATE FUNCTION
postgres=# select print_dri_shift('2014-04-20');
```

Output:

```
NOTICE:  Amol
 print_dri_shift
-----------------
       1
```

(1 row)

2. Write a stored function to display the details of a driver. (Accept driver name as input parameter).

```
create or replace function print_dri_dtl(name text) returns int as '
declare
rec record;
begin
for rec in  select dname,lic_no,age,salary
from driver where dname=name
loop
raise notice ''% %  %  %'',rec.dname,rec.lic_no,rec.age,rec.salary;
end loop;
return 1;
end;
'LANGUAGE 'plpgsql';
```

Execution of Code:

```
postgres=# \i cur5.sql
CREATE FUNCTION
postgres=# select print_dri_dtl('Amol');
```

Output:

```
NOTICE:  Amol 1001  55  15750
 print_dri_dtl
-----------------
       1
```

(1 row)

3. Write a function to accept the bus_no, date & print its allotted driver.

```
create or replace function print_dri_bus(no integer, dt date ) returns int
as '
declare
rec record;
begin
for rec in  select dname,bus_no
from driver a , bus_driver b where b.bus_no=no
```

```
and b.date_of_duty=dt and a.dno=b.dno
loop
   raise notice ''% % '',rec.dname,rec.bus_no;
end loop;
return 1;
end;
'LANGUAGE 'plpgsql';
```

Execution of Code:

```
postgres=# \i cur5.sql
CREATE FUNCTION
postgres=# select print_dri_bus(101,'2014-04-20');
```

Output:

```
NOTICE:  Amol 101
 print_dri_bus
--------------------
     1
(1 row)
```

Views:

1. Which contains details of bus no 101 along with details of all drivers who have driven that bus.

```
create view v51 as
select a.*
from driver a , bus_driver b
where b.bus_no=101
and a.dno=b.dno;
```

Execution of View:

```
postgres=# \i v5.sql
CREATE VIEW
select * from v51;
```

Output:

dno	dname	lic_no	address	age	salary
1	Amol	1001	Pune	55	15750

(1 row)

2. To display the details of the buses that run on routes 1 or 2.

```
create view v512 as
select distinct b.* from route a, bus b
where b.route_no=1 or b.route_no=2 and a.route_no=b.route_no;
```

Execution of Code:

```
postgres=# \i v5.sql
CREATE VIEW
select * from v512;
```

Output:

bus_no	capacity	depot_name	route_no
102	20	Hadapsar	2
101	35	Hadapsar	1

(1 row)

3. To find out the name of the driver having maximum salary.

```
create view v53 as
select dname ,salary from driver
where salary = (select max(salary) from driver );
```

Execution of View:

```
postgres=# \i v5.sql
CREATE VIEW
select * from v53;
```

Output:

dname	salary
Amol	15750

(1 row)

4. To accept the bus_no and date and print its allotted driver.

```
create view v54 as
select dname,bus_no from driver a, bus_driver b
where b.bus_no=102 and b.date_of_duty='2014-04-20'
and a.dno=b.dno;
```

Execution of Code:

```
postgres=# \i v5.sql
CREATE VIEW
select * from v54;
```

Output:

dname	bus_no
Ashok	102

(1 row)

Exceptions:

1. Write a stored function to accept the bus_no and date and print its allotted drivers. Raise an exception in case of invalid bus number.

```
create or replace function print_dri_bus(no integer, dt date ) returns int
as '
declare
rec record;
begin
select * into rec from bus_driver where bus_no = no and date_of_duty=dt;
if NOT FOUND then
raise exception ''Invalid bus no'';
end if;
for rec in  select dname,bus_no
from driver a , bus_driver b
where b.bus_no=no and b.date_of_duty=dt and a.dno=b.dno
loop
   raise notice ''% % '',rec.dname,rec.bus_no;
end loop;
return 1;
end;
'LANGUAGE 'plpgsql';
```

Execution of Code:

```
postgres=# \i fun5.sql
CREATE FUNCTION
postgres=# select print_dri_bus(11,'2014-04-20');
```

Output:

```
ERROR:  Invalid bus no
postgres=# select print_dri_bus(101,'2014-04-20');
NOTICE:  Amol 101
 print_dri_bus
----------------
       1
(1 row)
```

❖❖❖