



# PL/SQL

- PL/SQL is a combination of SQL along with the procedural features of programming languages. It was developed by Oracle Corporation in the early 90's to enhance the capabilities of SQL.

# PL/SQL sections

- **Declarations**

- This section starts with the keyword **DECLARE**. It is an optional section and defines all variables, cursors, subprograms, and other elements to be used in the program.

- **Executable Commands**

- This section is enclosed between the keywords **BEGIN** and **END** and it is a mandatory section. It consists of the executable PL/SQL statements of the program. It should have at least one executable line of code, which may be just a **NULL command** to indicate that nothing should be executed.

- **Exception Handling**

- This section starts with the keyword **EXCEPTION**. This optional section contains **exception(s)** that handle errors in the program.

- DECLARE
- <declarations section>
- BEGIN
- <executable command(s)>
- EXCEPTION
- <exception handling>
- END;

- DECLARE
- message varchar2(20):= 'Hello, World!';
- BEGIN
- dbms\_output.put\_line(message);
- END;
- /

# comments

- DECLARE
- -- variable declaration
- message varchar2(20):= 'Hello, World!';
- BEGIN
- /\*
- \* PL/SQL executable statement(s)
- \*/
- dbms\_output.put\_line(message);
- END;
- /

# Variables in PL/SQL

- Whenever you declare a variable, PL/SQL assigns it a default value of NULL. If you want to initialize a variable with a value other than the NULL value, you can do so during the declaration, using either of the following –
  - The DEFAULT keyword
  - The assignment operator
- `counter binary_integer := 0;`
- `greetings varchar2(20) DEFAULT 'Have a Good Day';`

- DECLARE
- a integer := 10;
- b integer := 20;
- c integer;
- f real;
- BEGIN
- c := a + b;
- dbms\_output.put\_line('Value of c: ' || c);
- f := 70.0/3.0;
- dbms\_output.put\_line('Value of f: ' || f);
- END;
- /



- DECLARE
- -- Global variables
- num1 number := 95;
- num2 number := 85;
- BEGIN
- 
- DECLARE
- -- Local variables
- num1 number := 195;
- num2 number := 185;
- BEGIN
- dbms\_output.put\_line('Inner Variable num1: ' || num1);
- dbms\_output.put\_line('Inner Variable num2: ' || num2);
- END;
- dbms\_output.put\_line('Outer Variable num1: ' || num1);
- dbms\_output.put\_line('Outer Variable num2: ' || num2);
- END;
- /

# Assigning SQL Query Results to PL/SQL Variables

- CREATE TABLE CUSTOMERS(
  - ID INT NOT NULL,
  - NAME VARCHAR (20) NOT NULL,
  - AGE INT NOT NULL,
  - ADDRESS CHAR (25),
  - SALARY DECIMAL (18, 2),
  - PRIMARY KEY (ID)
- );

- INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
- VALUES (1, 'Ramesh', 32, 'Ahmedabad', 2000.00 );
  
- INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
- VALUES (2, 'Khilan', 25, 'Delhi', 1500.00 );
  
- INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
- VALUES (3, 'kaushik', 23, 'Kota', 2000.00 );
- 
- INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
- VALUES (4, 'Chaitali', 25, 'Mumbai', 6500.00 );
- 
- INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
- VALUES (5, 'Hardik', 27, 'Bhopal', 8500.00 );
  
- INSERT INTO CUSTOMERS (ID,NAME,AGE,ADDRESS,SALARY)
- VALUES (6, 'Komal', 22, 'MP', 4500.00 );

- DECLARE
- c\_id customers.id%type := 1;
- c\_name customers.name%type;
- c\_addr customers.address%type;
- c\_sal customers.salary%type;
- BEGIN
- SELECT name, address, salary INTO c\_name, c\_addr, c\_sal
- FROM customers
- WHERE id = c\_id;
- dbms\_output.put\_line
- ('Customer ' || c\_name || ' from ' || c\_addr || ' earns ' || c\_sal);
- END;
- /

# Loops

- DECLARE
- i number(1);
- j number(1);
- BEGIN
- << outer\_loop >>
- FOR i IN 1..3 LOOP
- << inner\_loop >>
- FOR j IN 1..3 LOOP
- dbms\_output.put\_line('i is: ' || i || ' and j is: ' || j);
- END loop inner\_loop;
- END loop outer\_loop;
- END;
- /

# Procedure vs Function

- A **subprogram** is a program unit/module that performs a particular task.
- **Procedures** – These subprograms do not return a value directly; mainly used to perform an action.
- **Functions** – These subprograms return a single value; mainly used to compute and return a value.

# Procedures

- DECLARE
- a number;
- b number;
- c number;
- PROCEDURE findMin(x IN number, y IN number, z OUT number) IS
- BEGIN
- IF x < y THEN
- z:= x;
- ELSE
- z:= y;
- END IF;
- END;
- BEGIN
- a:= 61;
- b:= 15;
- findMin(a, b, c);
- dbms\_output.put\_line(' Minimum of (61, 15) : ' || c);
- END;
- /

- DECLARE
- a number;
- PROCEDURE squareNum(x IN OUT number) IS
- BEGIN
- x := x \* x;
- END;
- BEGIN
- a:= 13;
- squareNum(a);
- dbms\_output.put\_line(' Square of (13): ' || a);
- END;
- /



# Functions

- CREATE OR REPLACE FUNCTION totalCustomers
- RETURN number IS
- total number(2) := 0;
- BEGIN
- SELECT count(\*) into total
- FROM customers;
- 
- RETURN total;
- END;
- /

- DECLARE
- c number(2);
- BEGIN
- c := totalCustomers();
- dbms\_output.put\_line('Total no. of Customers: ' || c);
- END;
- /

- DECLARE
- a number;
- b number;
- c number;
- FUNCTION findMax(x IN number, y IN number)
- RETURN number
- IS
- z number;
- BEGIN
- IF x > y THEN
- z:= x;
- ELSE
- z:= y;
- END IF;
- RETURN z;
- END;
- BEGIN
- a:= &a;
- b:= &b;
- c := findMax(a, b);
- dbms\_output.put\_line(' Maximum of (a,b): ' || c);
- END;
- /

# Cursor

- A cursor is a pointer to the context area. PL/SQL controls the context area through a cursor. A cursor holds the rows (one or more) returned by a SQL statement. The set of rows the cursor holds is referred to as the active set.
- Implicit Cursors
- %FOUND, %ISOPEN, %NOTFOUND, and %ROWCOUNT

- DECLARE
- total\_rows number(2);
- BEGIN
- UPDATE customers
- SET salary = salary + 500;
- IF sql%notfound THEN
- dbms\_output.put\_line('no customers selected');
- ELSIF sql%found THEN
- total\_rows := sql%rowcount;
- dbms\_output.put\_line( total\_rows || ' customers selected ');
- END IF;
- END;
- /

- Explicit Cursors
- DECLARE
  - c\_id customers.id%type;
  - c\_name customers.name%type;
  - c\_addr customers.address%type;
  - CURSOR c\_customers IS
    - SELECT id, name, address FROM customers;
- BEGIN
  - OPEN c\_customers;
  - LOOP
    - FETCH c\_customers into c\_id, c\_name, c\_addr;
    - EXIT WHEN c\_customers%notfound;
    - dbms\_output.put\_line(c\_id || ' ' || c\_name || ' ' || c\_addr);
  - END LOOP;
  - CLOSE c\_customers;
- END;
- /

# Exception Handling

- DECLARE
- c\_id customers.id%type := 4;
- c\_name customerS.Name%type;
- c\_addr customers.address%type;
- BEGIN
- SELECT name, address INTO c\_name, c\_addr
- FROM customers
- WHERE id = c\_id;
- DBMS\_OUTPUT.PUT\_LINE ('Name: ' || c\_name);
- DBMS\_OUTPUT.PUT\_LINE ('Address: ' || c\_addr);
- EXCEPTION
- WHEN no\_data\_found THEN
- dbms\_output.put\_line('No such customer!');
- WHEN others THEN
- dbms\_output.put\_line('Error!');
- END;
- /

# User defined exception

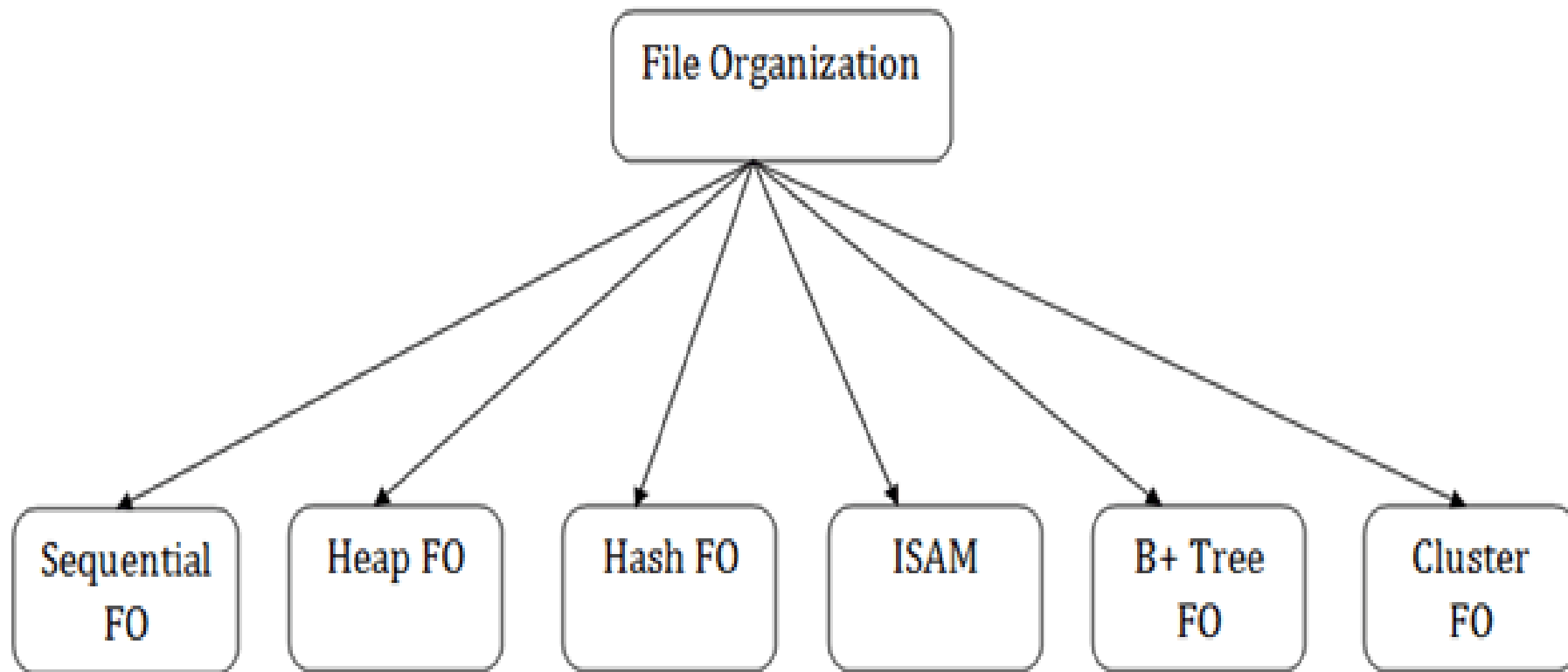
```
• DECLARE
•     c_id customers.id%type := &id;
•     c_name customerS.Name%type;
•     c_addr customers.address%type;
•     -- user defined exception
•     ex_invalid_id EXCEPTION;
• BEGIN
•     IF c_id <= 0 THEN
•         RAISE ex_invalid_id;
•     ELSE
•         SELECT name, address INTO c_name, c_addr
•         FROM customers
•         WHERE id = c_id;
•         DBMS_OUTPUT.PUT_LINE ('Name: ' || c_name);
•         DBMS_OUTPUT.PUT_LINE ('Address: ' || c_addr);
•     END IF;
• EXCEPTION
•     WHEN ex_invalid_id THEN
•         dbms_output.put_line('ID must be greater than zero!');
•     WHEN no_data_found THEN
•         dbms_output.put_line('No such customer!');
•     WHEN others THEN
•         dbms_output.put_line('Error!');
• END;
• /
```





# File/Record Organization

- File/record organization is used to describe the way in which the records are stored in terms of blocks, and the blocks are placed on the storage medium.



# B+ Tree

- The B+ tree is a balanced binary search tree. It follows a multi-level index format.
- In the B+ tree, leaf nodes denote actual data pointers. B+ tree ensures that all leaf nodes remain at the same height.
- In the B+ tree, the leaf nodes are linked using a link list. Therefore, a B+ tree can support random access as well as sequential access.

# Advantages

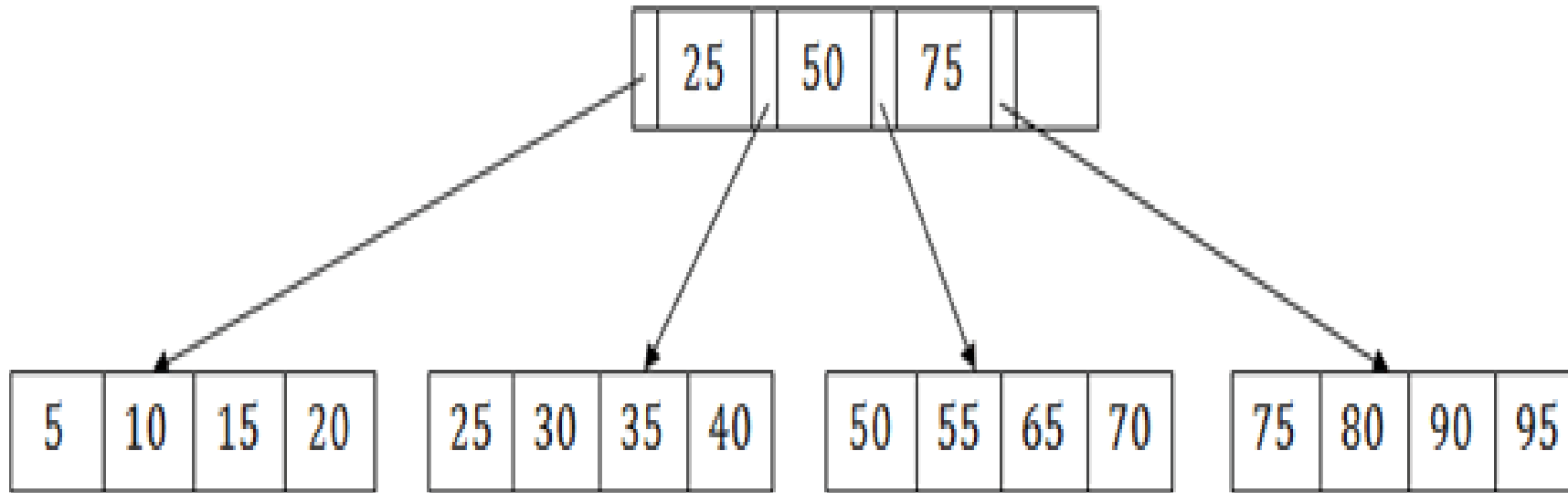
- Traversal of records is easy and quick in the B+ tree file organization.
- As all the records are stored in the leaf nodes, so searching for the records becomes very simple.
- It is the balanced tree structure. So, the operations like insert, update or delete do not affect the performance of the B+ tree.
- The structure of the B+ tree grows or shrinks automatically if the number of records increases or decreases. So, there are no limitations on its size

# Properties

- A B tree of order  $m$  contains the following properties.
- Every node in a B-Tree contains at most  $m$  children.
- Every node in a B-Tree except the root node and the leaf node contain at least  $m/2$  children.
- The root nodes must have at least 2 children.
- All leaf nodes must be at the same level.

# Searching

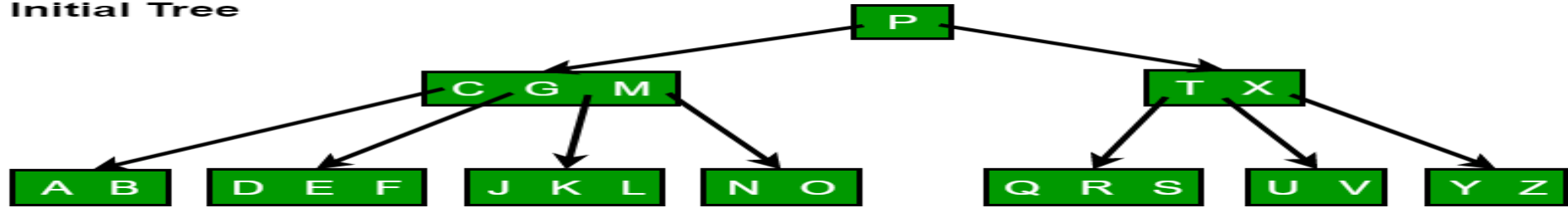
- Suppose we have to search 55 in the below B+ tree structure



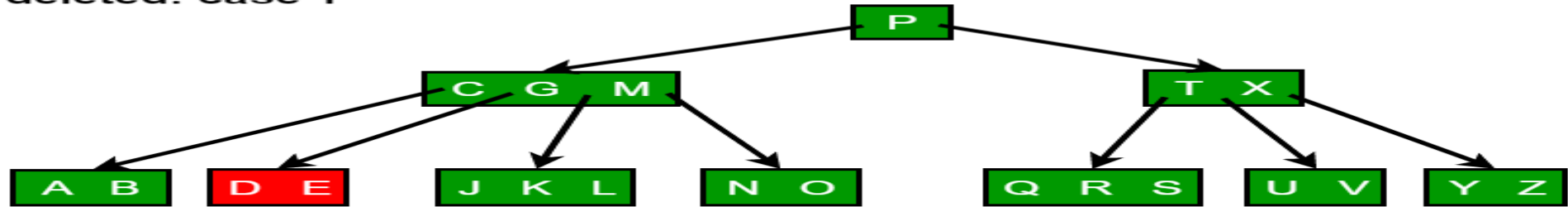
# B tree & B+ tree

- Insert 10, 20, 30, 40, 50, 60, 70, 80, 90.
- m (order) = 4

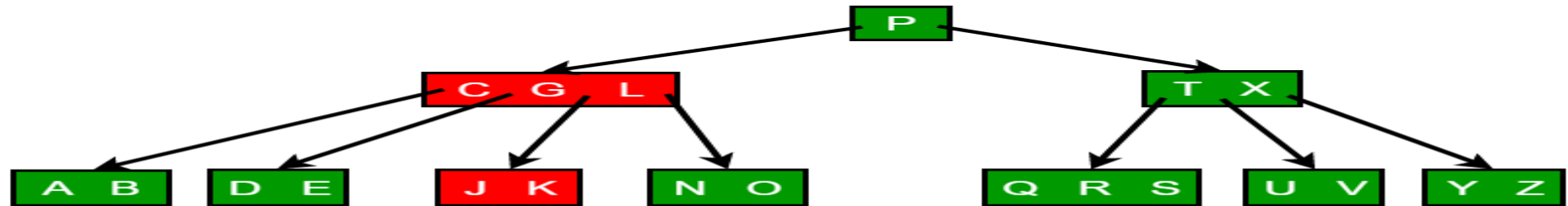
(a) Initial Tree



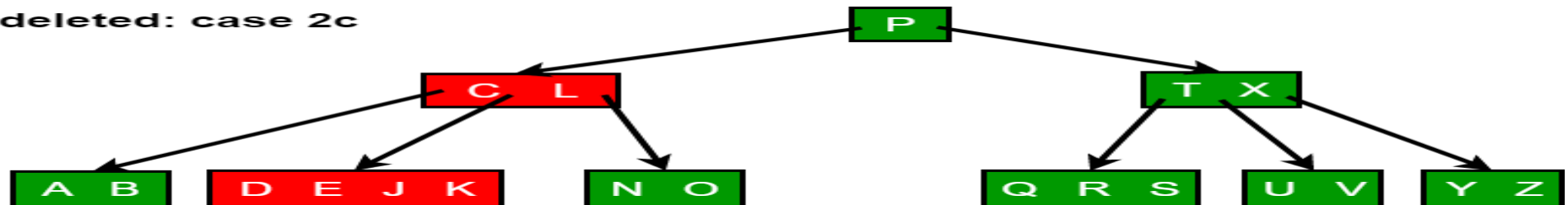
(b) F deleted: case 1



(c) M deleted: case 2a



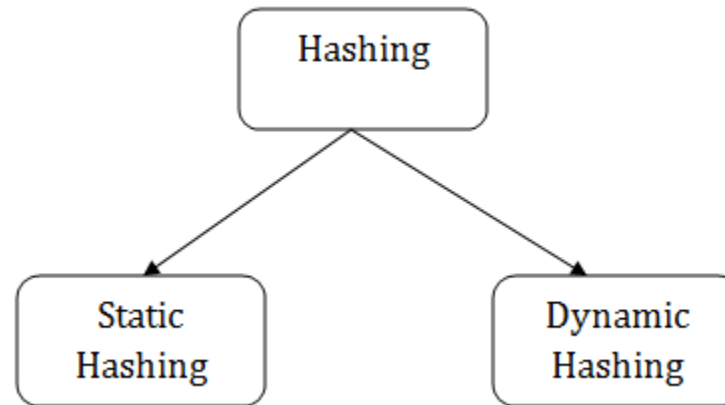
(d) G deleted: case 2c





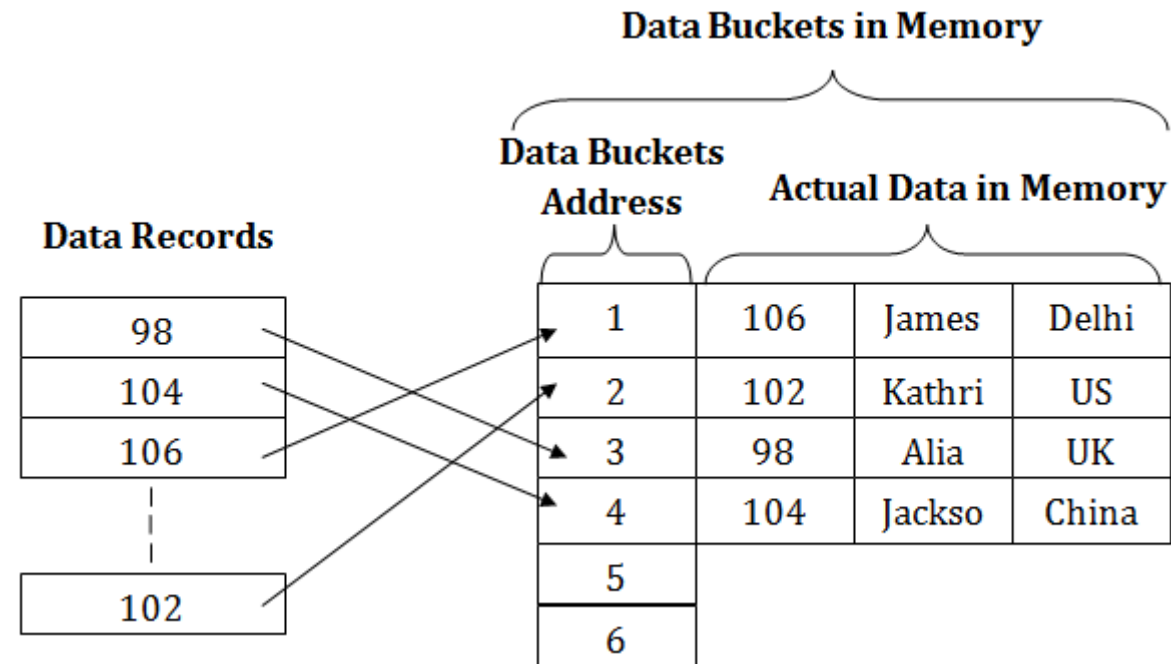
# Hashing

- In a huge database structure, it is very inefficient to search all the index values and reach the desired data. Hashing technique is used to calculate the direct location of a data record on the disk without using index structure.
- Hash function(Data)  $\rightarrow$  location to store



# Static Hashing

- In static hashing, the resultant data bucket address will always be the same. That means if we generate an address for EMP\_ID = 103 using the hash function  $\text{mod}(5)$  then it will always result in same bucket address 3. Here, there will be no change in the bucket address.
- Hence in this static hashing, the number of data buckets in memory remains constant throughout. In this example, we will have five data buckets in the memory used to store the data.

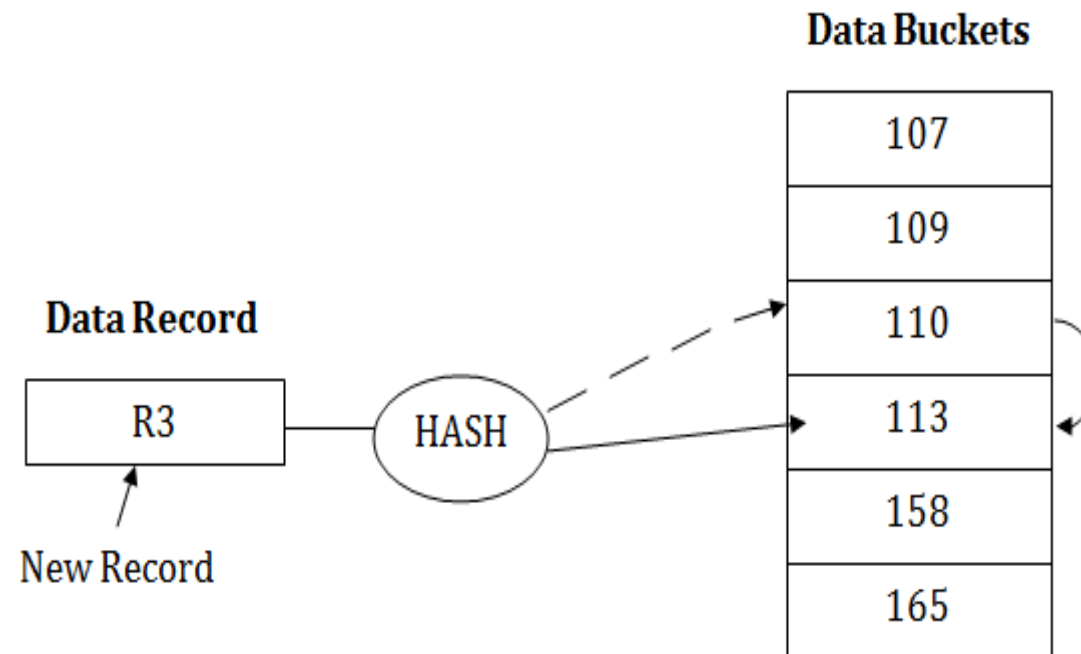


- Operations with hashing
  - -Insert
  - -Delete
  - -Searching
  - -Update
- All these operations can be done using the same hashing function.

- If we want to insert some new record into the file but the address of a data bucket generated by the hash function is not empty, or data already exists in that address. This situation in the static hashing is known as **bucket overflow**. This is a critical situation in this method.
- To overcome this situation, there are various methods.

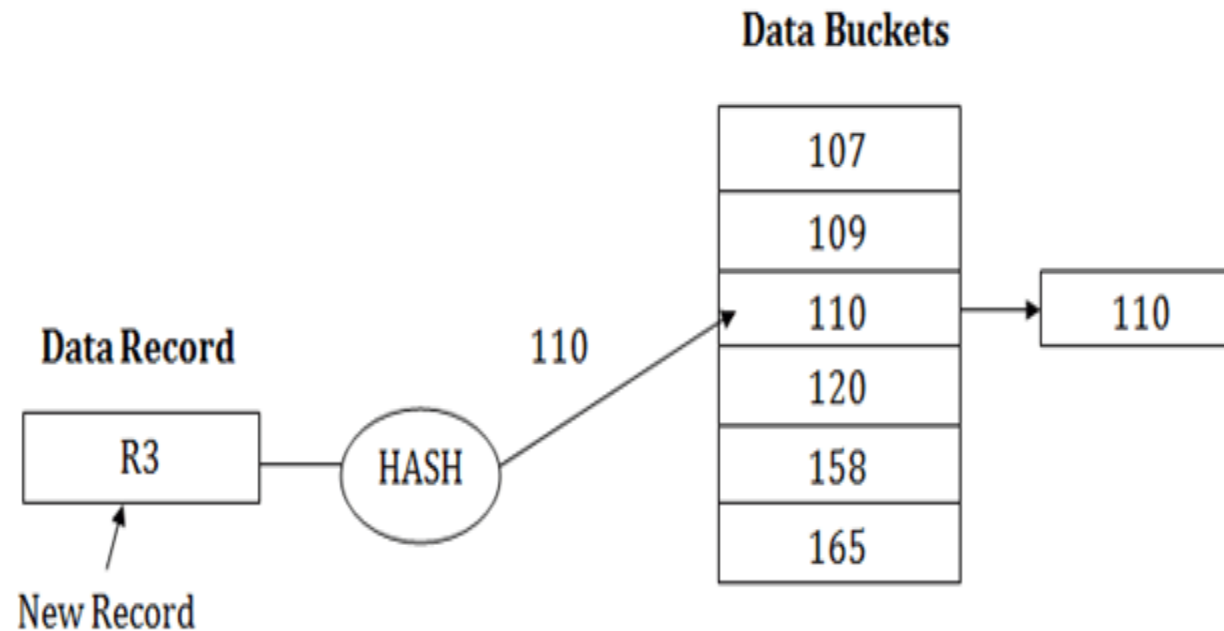
# Open Hashing

- When a hash function generates an address at which data is already stored, then the next bucket will be allocated to it. This mechanism is called as **Linear Probing**.



# Close Hashing

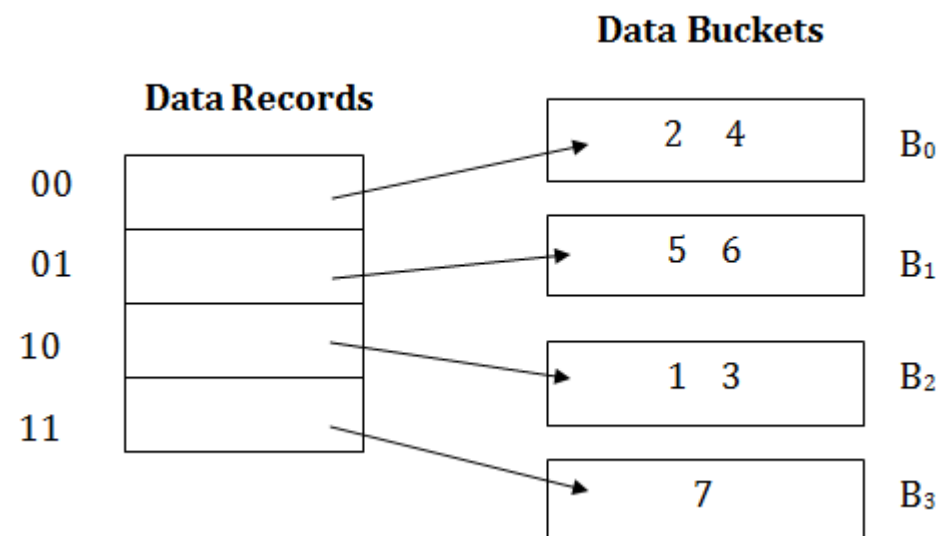
- When buckets are full, then a new data bucket is allocated for the same hash result and is linked after the previous one. This mechanism is known as **Overflow chaining**



# Dynamic Hashing

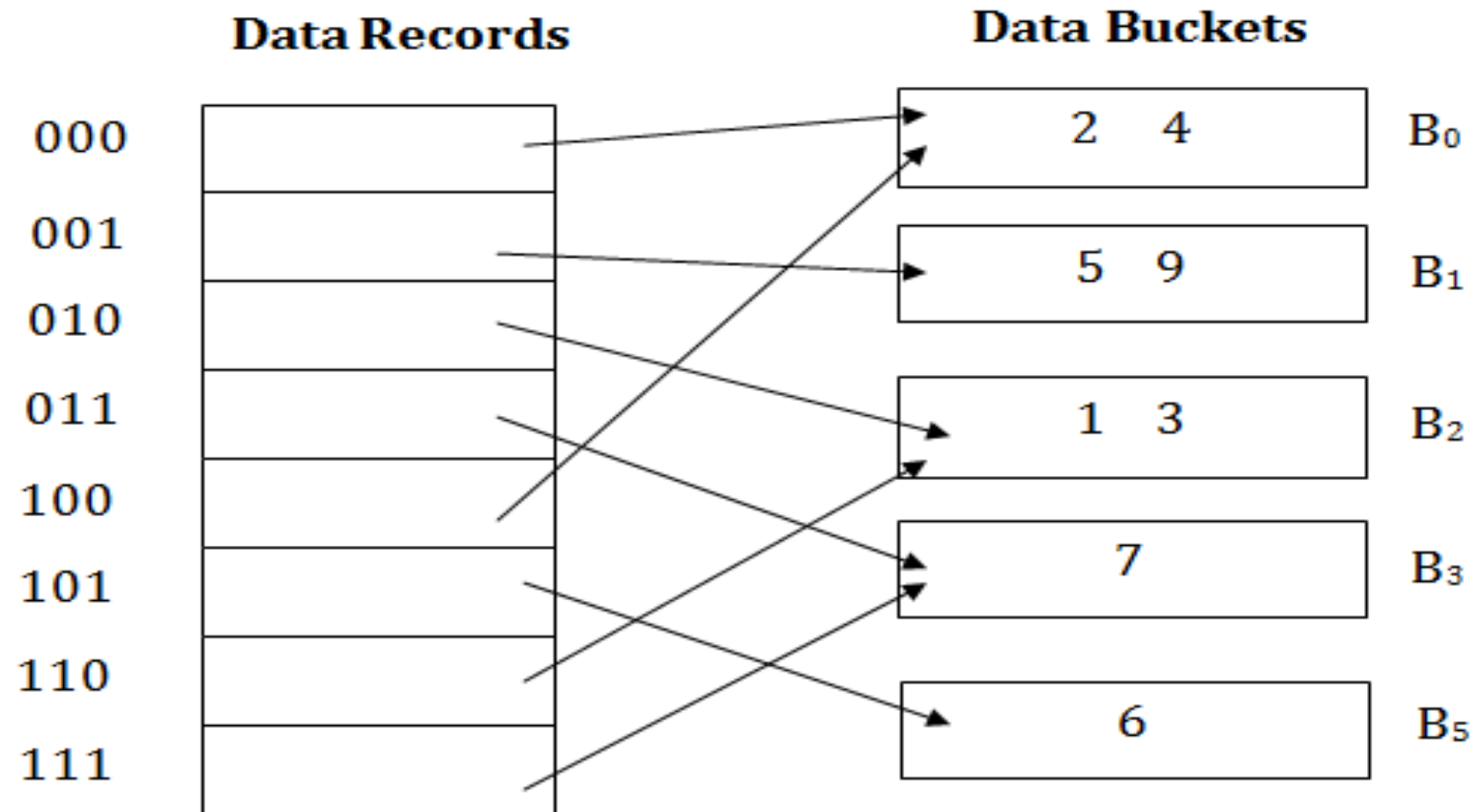
- The dynamic hashing method is used to overcome the problems of static hashing like bucket overflow.
- In this method, data buckets grow or shrink as the records increases or decreases.
- This method makes hashing dynamic, i.e., it allows insertion or deletion without resulting in poor performance.

Key	Hash address
1	11010
2	00000
3	11110
4	00000
5	01001
6	10101
7	10111





- Insert key 9 with hash address 10001





# Transaction management

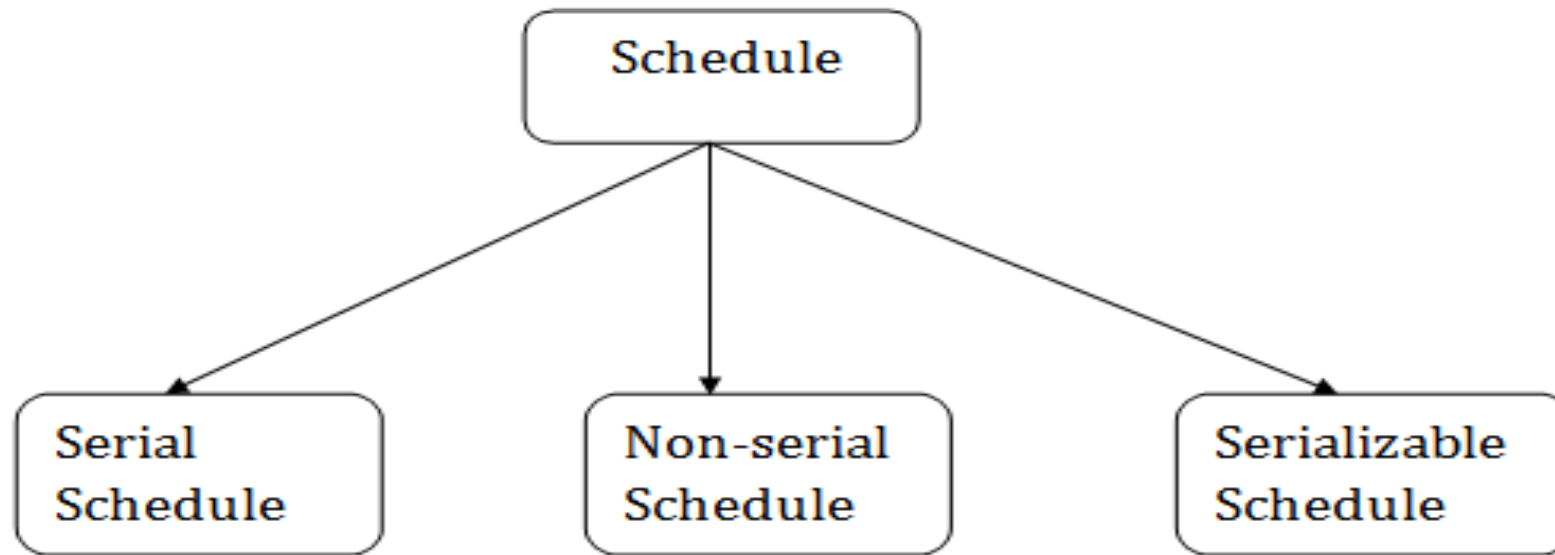
- A transaction is an action or series of actions. It is performed by a single user to perform operations for accessing the contents of the database.
- Suppose an employee of bank transfers Rs 800 from X's account to Y's account. This small transaction contains several low-level tasks:
  - **X's Account**
    - Open\_Account(X)
    - Old\_Balance = X.balance
    - New\_Balance = Old\_Balance - 800
    - X.balance = New\_Balance
    - Close\_Account(X)
  - **Y's Account**
    - Open\_Account(Y)
    - Old\_Balance = Y.balance
    - New\_Balance = Old\_Balance + 800
    - Y.balance = New\_Balance
    - Close\_Account(Y)

## Property of Transaction (ACID)


- Atomicity – It states that all operations of the transaction take place at once if not, the transaction is aborted.
- Consistency - The integrity constraints are maintained so that the database is consistent before and after the transaction. The execution of a transaction will leave a database in either its prior stable state or a new stable state.
- Isolation - It shows that the data which is used at the time of execution of a transaction cannot be used by the second transaction until the first one is completed.
- Durability - This property ensures that once the transaction has completed execution, the updates and modifications to the database are stored in and written to disk and they persist even if a system failure occurs.

# Schedule


- A series of operation from one transaction to another transaction is known as schedule. It is used to preserve the order of the operation in each of the individual transaction.



- Serial Schedule
- The serial schedule is a type of schedule where one transaction is executed completely before starting another transaction. In the serial schedule, when the first transaction completes its cycle, then the next transaction is executed.




T1	T2
READ(A)	
A=A-N	
WRITE(A)	
READ(B)	
B=B+N	
WRITE(B)	
	READ(A)
	A=A+M
	WRITE(A)
Schedule A	




T1	T2
	READ(A)
	A=A+M
	WRITE(A)
READ(A)	
A=A-N	
WRITE(A)	
READ(B)	
B=B+N	
WRITE(B)	
Schedule B	

- Non-serial Schedule
- It contains many possible orders in which the system can execute the individual operations of the transactions.



T1	T2
READ(A)	
A=A-N	
	READ(A)
	A=A+M
WRITE(A)	
READ(B)	
	WRITE(A)
B=B+N	
WRITE(B)	
Schedule C	



T1	T2
READ(A)	
A=A-N	
WRITE(A)	
	READ(A)
	A=A+M
	WRITE(A)
READ(B)	
B=B+N	
WRITE(B)	
Schedule D	

## Serializable schedule

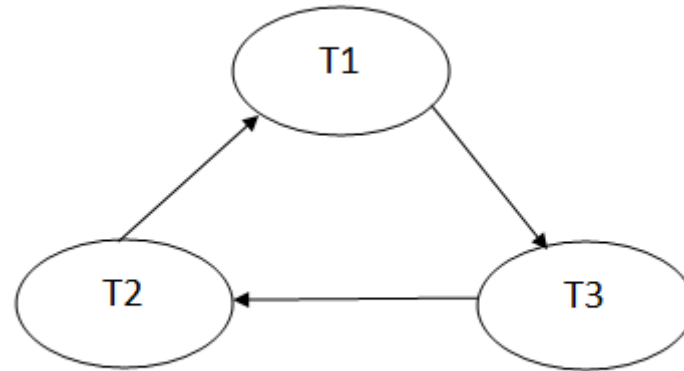
- The serializability of schedules is used to find non-serial schedules that allow the transaction to execute concurrently without interfering with one another.
- Serialization Graph is used to test the Serializability of a schedule.
- The set of edges is used to contain all edges  $T_i \rightarrow T_j$  for which one of the three conditions holds:
  - Create a node  $T_i \rightarrow T_j$  if  $T_i$  executes write (Q) before  $T_j$  executes read (Q).
  - Create a node  $T_i \rightarrow T_j$  if  $T_i$  executes read (Q) before  $T_j$  executes write (Q).
  - Create a node  $T_i \rightarrow T_j$  if  $T_i$  executes write (Q) before  $T_j$  executes write (Q).





T1	T2	T3
READ(A)		
	READ(B)	
A=f1(A)		
		READ(C)
	B=f2(B)	
	WRITE(B)	
		C=f3(C)
		WRITE(C)
WRITE(A)		
		READ(B)
	READ(A)	
	A=f4(A)	
READ(C)		
	WRITE(A)	
C=f5(C)		
WRITE(C)		
		B=f6(B)
		WRITE(B)
Schedule S1		

- The precedence graph for schedule contains a cycle that's why Schedule is non-serializable.



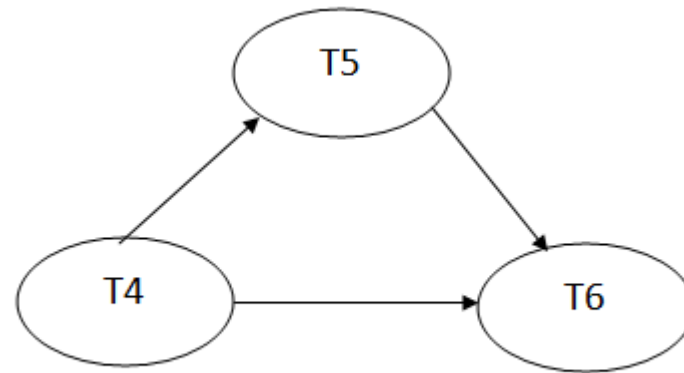
- **Read(A):** In T1, no subsequent writes to A, so no new edges
- Read(B):** In T2, no subsequent writes to B, so no new edges
- Read(C):** In T3, no subsequent writes to C, so no new edges
- Write(B):** B is subsequently read by T3, so add edge  $T2 \rightarrow T3$
- Write(C):** C is subsequently read by T1, so add edge  $T3 \rightarrow T1$
- Write(A):** A is subsequently read by T2, so add edge  $T1 \rightarrow T2$
- Write(A):** In T2, no subsequent reads to A, so no new edges
- Write(C):** In T1, no subsequent reads to C, so no new edges
- Write(B):** In T3, no subsequent reads to B, so no new edges



T4	T5	T6
READ(A)		
A=f1(A)		
READ(C)		
WRITE(A)		
A=f2(A)		
	READ(B)	
WRITE(C)		
	READ(A)	
		READ(C)
	B=f3(B)	
	WRITE(B)	
		C=f4(C)
		READ(B)
		WRITE(C)
	A=f5(A)	
	WRITE(A)	
		B=f6(B)
		WRITE(B)
Schedule S2		

- **Read(A):** In T4, no subsequent writes to A, so no new edges
- Read(C):** In T4, no subsequent writes to C, so no new edges
- Write(A):** A is subsequently read by T5, so add edge  $T4 \rightarrow T5$
- Read(B):** In T5, no subsequent writes to B, so no new edges
- Write(C):** C is subsequently read by T6, so add edge  $T4 \rightarrow T6$
- Write(B):** A is subsequently read by T6, so add edge  $T5 \rightarrow T6$
- Write(C):** In T6, no subsequent reads to C, so no new edges
- Write(A):** In T5, no subsequent reads to A, so no new edges
- Write(B):** In T6, no subsequent reads to B, so no new edges

- The precedence graph for schedule contains no cycle that's why Schedule is serializable.



# Conflict Serializable Schedule

- A schedule is called conflict serializability if after swapping of non-conflicting operations, it can transform into a serial schedule.

The two operations become conflicting if all conditions satisfy:

- Both belong to separate transactions.
- They have the same data item.
- They contain at least one write operation.

T1	T2
Read(A)	
	Read(A)
Schedule S1	



T1	T2
	Read(A)
Read(A)	
Schedule S2	



- Here,  $S1 \neq S2$ . That means it is conflict.

T1	T2
Read(A)	
	Write(A)
Schedule S1	



T1	T2
	Write(A)
Read(A)	
Schedule S2	

Non-serial schedule	
T1	T2
Read(A) Write(A)	
	Read(A) Write(A)
Read(B) Write(B)	
	Read(B) Write(B)
Schedule S1	

Serial schedule	
T1	T2
Read(A) Write(A)	
Read(A) Write(A)	
	Read(B) Write(B)
	Read(B) Write(B)
Schedule S2	

# Concurrency Control

## Problems with Concurrent Execution

- Problem 1: Lost Update Problems (W - W Conflict)

Time	Tx	Ty
t1	READ(A)	----
T2	A=A-50	----
t3	----	READ(A)
t4	----	A=A+100
t5	----	----
t6	WRITE(A)	----
t7	----	WRITE(A)

- Problem 2: Dirty Read Problems (W-R Conflict)

Time	Tx	Ty
t1	READ(A)	----
T2	A=A+50	----
t3	WRITE(A)	----
t4	----	READ(A)
t5	SERVER DOWN ROLLBACK	----

- Problem 3: Unrepeatable Read Problem (W-R Conflict)

Time	Tx	Ty
t1	READ(A)	----
t2	----	READ(A)
t3	----	A=A+100
t4	----	WRITE(A)
t5	READ(A)	----



# Concurrency Control Protocols

The concurrency control protocols ensure the *atomicity, consistency, isolation, durability* and *serializability* of the concurrent execution of the database transactions. Therefore, these protocols are categorized as:

- Lock Based Concurrency Control Protocol
- Time Stamp Concurrency Control Protocol
- Validation Based Concurrency Control Protocol

# Lock-Based Protocol

- In this type of protocol, any transaction cannot read or write data until it acquires an appropriate lock on it. There are two types of lock:
- **Shared lock:** It is also known as a Read-only lock. In a shared lock, the data item can only read by the transaction.
- It can be shared between the transactions because when the transaction holds a lock, then it can't update the data on the data item.
- **Exclusive lock:** In the exclusive lock, the data item can be both reads as well as written by the transaction.
- This lock is exclusive, and in this lock, multiple transactions do not modify the same data simultaneously.



## Simplistic lock protocol

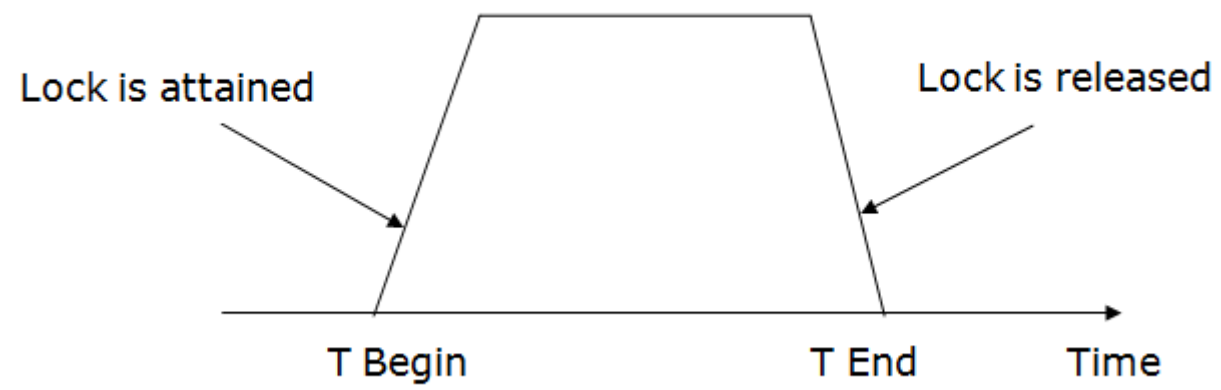
- It is the simplest way of locking the data while transaction. Simplistic lock-based protocols allow all the transactions to get the lock on the data before insert or delete or update on it. It will unlock the data item after completing the transaction.

## Pre-claiming Lock Protocol

- Pre-claiming Lock Protocols evaluate the transaction to list all the data items on which they need locks.
- If all the locks are granted then this protocol allows the transaction to begin. When the transaction is completed then it releases all the lock.

## Two-phase locking (2PL)

- The two-phase locking protocol divides the execution phase of the transaction into three parts.
- In the first part, when the execution of the transaction starts, it seeks permission for the lock it requires.
- In the second part, the transaction acquires all the locks. The third phase is started as soon as the transaction releases its first lock.
- In the third phase, the transaction cannot demand any new locks. It only releases the acquired locks.

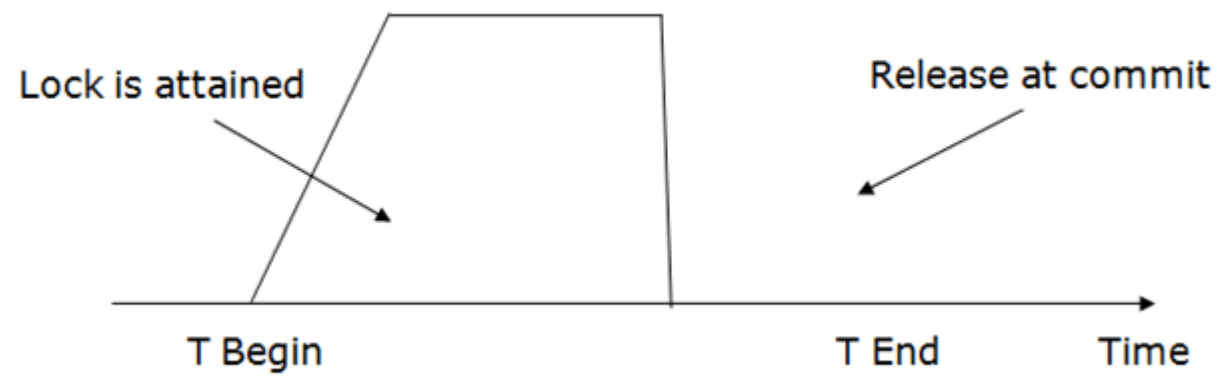


- **Transaction T1:**
- **Growing phase:** from step 1-3
- **Shrinking phase:** from step 5-7
- **Lock point:** at 3
- **Transaction T2:**
- **Growing phase:** from step 2-6
- **Shrinking phase:** from step 8-9
- **Lock point:** at 6

	T1	T2
0	LOCK-S(A)	
1		LOCK-S(A)
2	LOCK-X(B)	
3	----	----
4	UNLOCK(A)	
5		LOCK-X(C)
6	UNLOCK(B)	
7		UNLOCK(A)
8		UNLOCK(C)
9	----	----

# Strict Two-phase locking (Strict-2PL)

- The first phase of Strict-2PL is similar to 2PL. In the first phase, after acquiring all the locks, the transaction continues to execute normally.
- The only difference between 2PL and strict 2PL is that Strict-2PL does not release a lock after using it.
- Strict-2PL waits until the whole transaction to commit, and then it releases all the locks at a time.
- Strict-2PL protocol does not have shrinking phase of lock release.



# Timestamp Ordering Protocol

- Timestamp based Protocol in DBMS is an algorithm which uses the System Time or Logical Counter as a timestamp to serialize the execution of concurrent transactions. The Timestamp-based protocol ensures that every conflicting read and write operations are executed in a timestamp order.

# Timestamp Ordering Protocol

- 1. Check the following condition whenever a transaction  $T_i$  issues a **Read (X)** operation:
  - If  $W\_TS(X) > TS(T_i)$  then the operation is rejected.
  - If  $W\_TS(X) \leq TS(T_i)$  then the operation is executed.
  - Timestamps of all the data items are updated.
- 2. Check the following condition whenever a transaction  $T_i$  issues a **Write(X)** operation:
  - If  $TS(T_i) < R\_TS(X)$  then the operation is rejected.
  - If  $TS(T_i) < W\_TS(X)$  then the operation is rejected and  $T_i$  is rolled back otherwise the operation is executed.
- **Where,**
  - **$TS(T_i)$**  denotes the timestamp of the transaction  $T_i$ .
  - **$R\_TS(X)$**  denotes the Read time-stamp of data-item  $X$ .
  - **$W\_TS(X)$**  denotes the Write time-stamp of data-item  $X$ .



# Validation Based Protocol

- Validation based Protocol in DBMS also known as Optimistic Concurrency Control Technique is a method to avoid concurrency in transactions. In this protocol, the local copies of the transaction data are updated rather than the data itself, which results in less interference while execution of the transaction.
- The Validation based Protocol is performed in the following three phases:
  - Read Phase
  - Validation Phase
  - Write Phase
- Read Phase
  - In the Read Phase, the data values from the database can be read by a transaction but the write operation or updates are only applied to the local data copies, not the actual database.
- Validation Phase
  - In Validation Phase, the data is checked to ensure that there is no violation of serializability while applying the transaction updates to the database.
- Write Phase
  - In the Write Phase, the updates are applied to the database if the validation is successful, else; the updates are not applied, and the transaction is rolled back.