UNIT 10 TRANSACTIONS AND CONCURRENCY MANAGEMENT

(Adopted from MCS-023 Block-2 Unit-2)

Structure **Page Nos.**

10.0 INTRODUCTION

One of the main advantages of storing data in an integrated repository or a database is to allow sharing of it among multiple users. Several users access the database or perform transactions at the same time. What if a user's transactions try to access a data item that is being used /modified by another transaction? This unit attempts to provide details on how concurrent transactions are executed under the control of DBMS. However, in order to explain the concurrent transactions, first we must describe the term transaction.

Concurrent execution of user programs is essential for better performance of DBMS, as concurrent running of several user programs keeps utilizing CPU time efficiently, since disk accesses are frequent and are relatively slow in case of DBMS. Also, a user's program may carry out many operations on the data returned from DB, but DBMS is only concerned about what data is being read /written from/ into the database. This unit discusses the issues of concurrent transactions in more detail.

10.1 OBJECTIVES

After going through this unit, you should be able to:

- describe the term CONCURRENCY;
- define the term transaction and concurrent transactions;
- discuss about concurrency control mechanism;
- describe the principles of locking and serialisability, and
- describe concepts of deadlock & its prevention.

10.2 THE TRANSACTIONS

A transaction is defined as the unit of work in a database system. Database systems

that deal with a large number of transactions are also termed as transaction processing systems.

What is a transaction? Transaction is a unit of data processing. For example, some of the transactions at a bank may be withdrawal or deposit of money; transfer of money from A's account to B's account etc. A transaction would involve manipulation of one or more data values in a database. Thus, it may require reading and writing of database value. For example, the withdrawal transactions can be written in pseudocode as:

Example 1:

```
; Assume that we are doing this transaction for person
;  whose account number is X.

TRANSACTION WITHDRAWAL (withdrawal_amount)
Begin transaction
        IF X exist then
                READ X.balance
                IF      X.balance > withdrawal_amount
                        THEN SUBTRACT withdrawal_amount
                        WRITE X.balance
                        COMMIT
                ELSE
                DISPLAY "TRANSACTION CANNOT BE PROCESSED"
        ELSE  DISPLAY "ACCOUNT X DOES NOT EXIST"
End transaction;
```

Another similar example may be transfer of money from Account no x to account number y. This transaction may be written as:

Example 2:

```
; transfers transfer_amount from x's account to y's account
; assumes x&y both accounts exist

TRANSACTION (x, y, transfer_amount)
Begin transaction
        IF X AND Y exist then
                READ x.balance
                IF x.balance > transfer_amount THEN
                        x.balance = x.balance – transfer_amount
                        READ y.balance
                        y.balance = y.balance + transfer_amount
                        COMMIT
                ELSE DISPLAY ("BALANCE IN X NOT OK")
                        ROLLBACK
        ELSE DISPLAY ("ACCOUNT X OR Y DOES NOT EXIST")
End_transaction
```

Please note the use of two keywords here COMMIT and ROLLBACK. Commit makes sure that all the changes made by transactions are made permanent.
ROLLBACK terminates the transactions and rejects any change made by the transaction. Transactions have certain desirable properties. Let us look into those properties of a transaction.

Properties of a Transaction

A transaction has four basic properties. These are:

- Atomicity
- Consistency
- Isolation or Independence
- Durability or Permanence

Atomicity: It defines a transaction to be a single unit of processing. In other words either a transaction will be done *completely* or *not at all*. In the transaction example 1 & 2 please note that transaction 2 is reading and writing more than one data items, the atomicity property requires either operations on both the data item be performed or not at all.

Consistency: This property ensures that a complete transaction execution takes a database from one consistent state to another consistent state. If a transaction fails even then the database should come back to a consistent state.

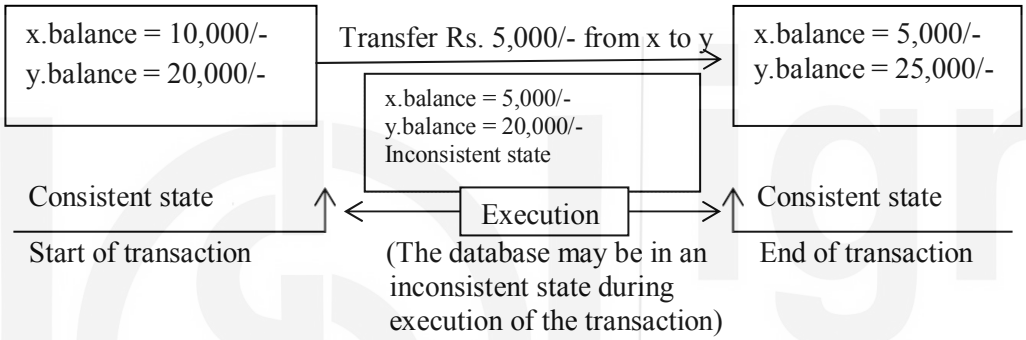


Figure 1: A Transaction execution

Isolation or Independence: The isolation property states that the updates of a transaction should not be visible till they are committed. Isolation guarantees that the progress of other transactions do not affect the outcome of this transaction. For example, if another transaction that is a withdrawal transaction which withdraws an amount of Rs. 5000 from X account is in progress, whether fails or commits, should not affect the outcome of this transaction. Only the state that has been read by the transaction last should determine the outcome of this transaction.

Durability: This property necessitates that once a transaction has committed, the changes made by it be never lost because of subsequent failure. Thus, a transaction is also a basic unit of recovery. The details of transaction-based recovery are discussed in the next unit.

A transaction has many states of execution. These states are displayed in *Figure 2*.

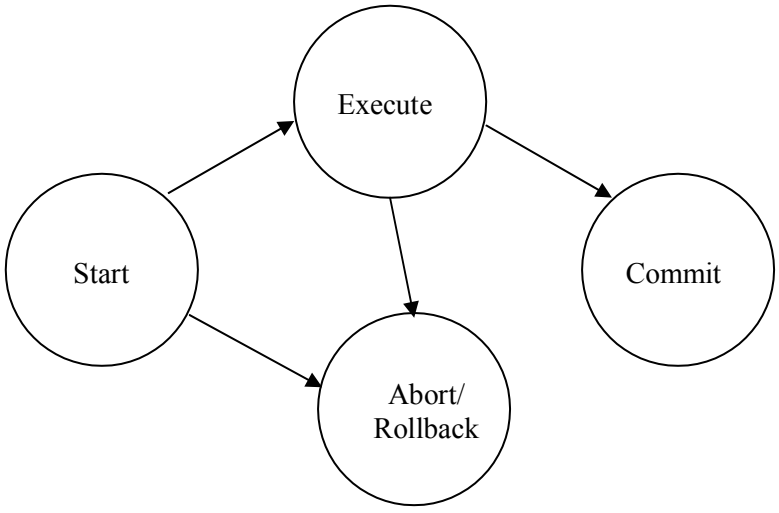


Figure 2: States of transaction execution

A transaction is started as a program. From the start state as the transaction is scheduled by the CPU it moves to the Execute state, however, in case of any system error at that point it may also be moved into the Abort state. During the execution transaction changes the data values and database moves to an inconsistent state. On successful completion of transaction it moves to the Commit state where the durability feature of transaction ensures that the changes will not be lost. In case of any error the transaction goes to Rollback state where all the changes made by the transaction are undone. Thus, after commit or rollback database is back into consistent state. In case a transaction has been rolled back, it is started as a new transaction. All these states of the transaction are shown in *Figure 2*.

10.3 THE CONCURRENT TRANSACTIONS

Almost all the commercial DBMS support multi-user environment. Thus, allowing multiple transactions to proceed simultaneously. The DBMS must ensure that two or more transactions do not get into each other's way, i.e., transaction of one user doesn't effect the transaction of other or even the transactions issued by the same user should not get into the way of each other. Please note that concurrency related problem may occur in databases only if **two transactions are contending for the same data item and at least one of the concurrent transactions wishes to update a data value in the database**. In case, the concurrent transactions only read same data item and no updates are performed on these values, then it does NOT cause any concurrency related problem. Now, let us first discuss why you need a mechanism to control concurrency.

Consider a banking application dealing with checking and saving accounts. A Banking Transaction T1 for Mr. Sharma moves Rs.100 from his checking account balance X to his savings account balance Y, using the transaction T1:

Transaction T1:

```
A:Read X
  Subtract 100
  Write X
B:Read Y
  Add 100
  Write Y
```

Let us suppose an auditor wants to know the total assets of Mr. Sharma. He executes the following transaction:

Transaction T2:

```
Read X
Read Y
Display X+Y
```

Suppose both of these transactions are issued simultaneously, then the execution of these instructions can be mixed in many ways. This is also called the **Schedule**. Let us define this term in more detail.

A schedule S is defined as the sequential ordering of the operations of the 'n' interleaved transactions. A schedule maintains the order of operations within the individual transaction.

Conflicting Operations in Schedule: Two operations of different transactions conflict if they access the same data item AND one of them is a write operation.

For example, the two transactions TA and TB as given below, if executed in parallel, may produce a schedule:

TA
READ X
WRITE X

TB
READ X
WRITE X

SCHEDULE	TA	TB
READ X	READ X	
READ X		READ X
WRITE X		WRITE X
WRITE X	WRITE X	

One possible schedule for interleaved execution of TA and TB

Let us show you three simple ways of interleaved instruction execution of transactions T1 and T2. Please note that in the following tables the first column defines the sequence of instructions that are getting executed, that is the schedule of operations.

a) T2 is executed completely before T1 starts, then sum X+Y will show the correct assets:

Schedule	**Transaction T1**	**Transaction T2**	**Example Values**
Read X		Read X	X = 50000
Read Y		Read Y	Y= 100000
Display X+Y		Display X+Y	**150000**
Read X	Read X		X = 50000
Subtract 100	Subtract 100		49900
Write X	Write X		X = 49900
Read Y	Read Y		Y= 100000
Add 100	Add 100		100100
Write Y	Write Y		Y= 100100

b) T1 is executed completely before T2 starts, then sum X+Y will still show thecorrect assets:

Schedule	**Transaction T1**	**Transaction T2**	**Example Values**
Read X	Read X		X = 50000
Subtract 100	Subtract 100		49900
Write X	Write X		X = 49900
Read Y	Read Y		Y= 100000
Add 100	Add 100		100100
Write Y	Write Y		Y= 100100
Read X		Read X	X = 49900
Read Y		Read Y	Y= 100100
Display X+Y		Display X+Y	**150000**

c) Block A in transaction T1 is executed, followed by complete execution of T2, followed by the Block B of T1.

Schedule	Transaction T1	Transaction T2	Example Values
Read X	Read X		X = 50000
Subtract 100	Subtract 100		49900
Write X	Write X		X = 49900
Read X		Read X	X = 49900
Read Y		Read Y	Y= 100000
Display X+Y		Display X+Y	**149900**
Read Y	Read Y		Y= 100000
Add 100	Add 100		100100
Write Y	Write Y		Y= 100100

In this execution an incorrect value is being displayed. This is because Rs.100 although removed from X, has not been put in Y, and is thus missing. Obviously, if T1 had been written differently, starting with block B and following up with block A, even then such an interleaving would have given a different but incorrect result.

Please note that for the given transaction there are many more ways of this interleaved instruction execution.

Thus, there may be a problem when the transactions T1 and T2 are allowed to execute in parallel. Let us define the problems of concurrent execution of transaction more precisely.

Let us assume the following transactions (assuming there will not be errors in data while execution of transactions)

Transaction T3 and T4: T3 reads the balance of account X and subtracts a withdrawal amount of Rs. 5000, whereas T4 reads the balance of account X and adds an amount of Rs. 3000

T3
READ X
SUB 5000
WRITE X

T4
READ X
ADD 3000
WRITE X

Problems of Concurrent Transactions

1. **Lost Updates:** Suppose the two transactions T3 and T4 run concurrently and they happen to be interleaved in the following way (assume the initial value of X as10000):

T3	T4	Value of X	
		T3	T4
READ X		10000	
	READ X		10000
SUB 5000		5000	
	ADD 3000		13000
WRITE X		5000	
	WRITE X		13000

After the execution of both the transactions the value X is 13000 while the

semantically correct value should be 8000. The problem occurred as the update made by T3 has been overwritten by T4. The root cause of the problem was the fact that both the transactions had read the value of X as 10000. Thus one of the two updates has been lost and we say that a **lost update** has occurred.

There is one more way in which the lost updates can arise. Consider the following part of some transactions:

T5	T6	Value of x originally2000	
		T5	T6
UPDATE X		3000	
	UPDATE X		4000
ROLLBACK		2000	

Here T5 & T6 updates the same item X. Thereafter T5 decides to undo its action and rolls back causing the value of X to go back to the original value that was 2000. In this case also the update performed by T6 had got lost and a lost update is said to have occurred.

2. **Unrepeatable reads:** Suppose T7 reads X twice during its execution. If it did not update X itself it could be very disturbing to see a different value of X in its next read. But this could occur if, between the two read operations, another transaction modifies X.

T7	T8	Assumed value ofX=2000	
		T7	T8
READ X		2000	
	UPDATE X		3000
READ X		3000	

Thus, the inconsistent values are read and results of the transaction may be in error.

3. **Dirty Reads:** T10 reads a value which has been updated by T9. This update hasnot been committed and T9 aborts.

T9	T10	Value of x old value =200	
		T9	T10
UPDATE X		500	
	READ X		500
ROLLBACK		200	?

Here T10 reads a value that has been updated by transaction T9 that has been aborted. Thus T10 has read a value that would never exist in the database and hence the problem. Here the problem is primarily of isolation of transaction.

4. **Inconsistent Analysis:** The problem as shown with transactions T1 and T2 where two transactions interleave to produce incorrect result during an analysis by Audit is the example of such a problem. This problem occurs when more than one data items are being used for analysis, while another transaction has modified some of those values and some are yet to be modified. Thus, an analysis transaction reads values from the inconsistent state of the database that results in inconsistent analysis.

Thus, we can conclude that the prime reason of problems of concurrent transactions is that a transaction reads an inconsistent state of the database that has been created by

other transaction.

But how do we ensure that execution of two or more transactions have not resulted in a concurrency related problem?

Well one of the commonest techniques used for this purpose is to restrict access to data items that are being read or written by one transaction and is being written by another transaction. This technique is called locking. Let us discuss locking in more detail in the next section.

☞ **Check Your Progress 1**

1) What is a transaction? What are its properties? Can a transaction update more than on data values? Can a transaction write a value without reading it? Give anexample of transaction.

……

……

……

2) What are the problems of concurrent transactions? Can these problems occur intransactions which do not read the same data values?

……

……

……

3) What is a Commit state? Can you rollback after the transaction commits?

……

……

10.4 THE LOCKING PROTOCOL

To control concurrency related problems we use locking. A lock is basically a variable that is associated with a data item in the database. A lock can be placed by a transaction on a shared resource that it desires to use. When this is done, the data item is available for the exclusive use for that transaction, i.e., other transactions are locked out of that data item. When a transaction that has locked a data item does not desire to use it any more, it should unlock the data item so that other transactions can use it. If a transaction tries to lock a data item already locked by some other transaction, it cannot do so and waits for the data item to be unlocked. The component of DBMS that controls and stores lock information is called the Lock Manager. The locking mechanism helps us to convert a schedule into a serialisable schedule. We had defined what a schedule is, but what is a serialisable schedule? Let us discuss about it in more detail:

10.4.1 Serialisable Schedules

If the operations of two transactions conflict with each other, how to determine that no concurrency related problems have occurred? For this, serialisability theory has been developed. Serialisability theory attempts to determine the **correctness** of the schedules. The rule of this theory is:

"A schedule S of n transactions is serialisable if it is equivalent to some **serial schedule** of the same 'n' transactions".

A serial schedule is a schedule in which either transaction T_1 is completely done before T_2 or transaction T_2 is completely done before T_1. For example, the following figure shows the two possible serial schedules of transactions T_1 & T_2.

Schedule A: T2 followed by T1			Schedule B: T1 followed by T2		
Schedule	**T1**	**T2**	**Schedule**	**T1**	**T2**
Read X		Read X	Read X	Read X	
Read Y		Read Y	Subtract 100	Subtract 100	
Display X+Y		Display X+Y	Write X	Write X	
Read X	Read X		Read Y	Read Y	
Subtract 100	Subtract 100		Add 100	Add 100	
Write X	Write X		Write Y	Write Y	
Read Y	Read Y		Read X		Read X
Add 100	Add 100		Read Y		Read Y
Write Y	Write Y		Display X+Y		Display X+Y

Figure 3: Serial Schedule of two transactions

Schedule C: An Interleaved Schedule		
Schedule	**T1**	**T2**
Read X	Read X	
Subtract 100	Subtract 100	
Read X		Read X
Write X	Write X	
Read Y		Read Y
Read Y	Read Y	
Add 100	Add 100	
Display X+Y		Display X+Y
Write Y	Write Y	

Figure 4: An Interleaved Schedule

Now, we have to figure out whether this interleaved schedule would be performing read and write in the same order as that of a serial schedule. If it does, then it is equivalent to a serial schedule, otherwise not. In case it is not equivalent to a serial schedule, then it may result in problems due to concurrent transactions.

Serialisability

Any schedule that produces the same results as a serial schedule is called a serialisable schedule. But how can a schedule be determined to be serialisable or not? In other words, other than giving values to various items in a schedule and checking if the results obtained are the same as those from a serial schedule, is there an algorithmic way of determining whether a schedule is serialisable or not?

The basis of the algorithm for serialisability is taken from the notion of a serial schedule. There are two possible serial schedules in case of two transactions (T1- T2 OR T2 - T1). Similarly, in case of three parallel transactions the number of possible serial schedules is 3!, that is, 6. These serial schedules can be:

T1-T2-T3 T1-T3-T2 T2-T1-T3
T2-T3-T1 T3-T1-T2 T3-T2-T1

Using the notion of precedence graph, an algorithm can be devised to determine whether an interleaved schedule is serialisbale or not. In this graph, the transactions of the schedule are represented as the nodes. This graph also has directed edges. An edge

from the node representing transactions Ti to node Tj means that there exists a **conflicting operation** between Ti and Tj and Ti precedes Tj in some conflicting operations. It has been proved that a serialisable schedule is the one that contains no cycle in the graph.

Given a graph with no cycles in it, there must be a serial schedule corresponding to it.

The steps of constructing a precedence graph are:

1. Create a node for every transaction in the schedule.
2. Find the precedence relationships in conflicting operations. Conflicting operations are (read-write) or (write-read) or (write–write) on the same data item in two different transactions. But how to find them?

   2.1 For a transaction Ti which *reads* an item A, find a transaction Tj that *writes* A later in the schedule. If such a transaction is found, draw an edge from Ti to Tj.

   2.2 For a transaction Ti which has *written* an item A, find a transaction Tj later in the schedule that *reads* A. If such a transaction is found, draw an edge from Ti to Tj.

   2.3 For a transaction Ti which has *written* an item A, find a transaction Tj that *writes* A later than Ti. If such a transaction is found, draw an edge from Ti to Tj.

3. If there is any cycle in the graph, the schedule is not serialisable, otherwise, find the equivalent serial schedule of the transaction by traversing the transaction nodes starting with the node that has no input edge.

Let us use this algorithm to check whether the schedule as given in Figure 4 is Serialisable. *Figure 5* shows the required graph. Please note as per step 1, we draw the two nodes for T1 and T2. In the schedule given in *Figure 4*, please note that the transaction T2 reads data item X, which is subsequently written by T1, thus there is an edge from T2 to T1 (clause 2.1). Also, T2 reads data item Y, which is subsequently written by T1, thus there is an edge from T2 to T1 (clause 2.1). However, that edge already exists, so we do not need to redo it. Please note that there are no cycles in the graph, thus, the schedule given in *Figure 4* is serialisable. The equivalent serial schedule (as per step 3) would be T2 followed by T1.

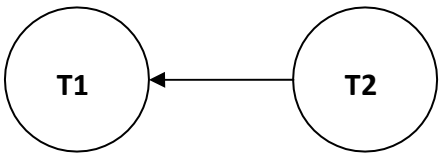


Figure 5: Test of Serialisability for the Schedule of Figure 4

Please note that the schedule given in part (c) of section 2.3 is not serialsable, because in that schedule, the two edges that exist between nodes T1 and T2 are:

- T1 writes X which is later read by T2 (clause 2.2), so there exists an edge from T1to T2.
- T2 reads X which is later written by T1 (clause 2.1), so there exists an edge fromT2 to T1.

Thus the graph for the schedule will be:

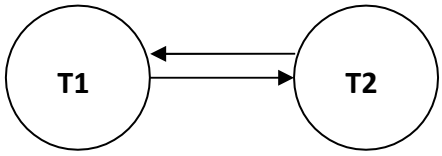


Figure 6: Test of Serialisability for the Schedule (c) of section 2.3

Please note that the graph above has a cycle T1-T2-T1, therefore it is not serialisable.

10.4.2 Locks

Serialisabilty is just a test whether a given interleaved schedule is ok or has a concurrency related problem. However, it does not ensure that the interleaved concurrent transactions do not have any concurrency related problem. This can be done by using locks. So let us discuss about what the different types of locks are, and then how locking ensures serialsability of executing transactions.

Types of Locks

There are two basic types of locks:

- Binary lock: This locking mechanism has two states for to a data item: locked or unlocked
- Multiple-mode locks: In this locking type each data item can be in three states read locked or shared locked, write locked or exclusive locked or unlocked.

Let us first take an example for binary locking and explain how it solves the concurrency related problems. Let us reconsider the transactions T1 and T2 for this purpose; however we will add to required binary locks to them.

Schedule	T1	T2
Lock X	Lock X	
Read X	Read X	
Subtract 100	Subtract 100	
Write X	Write X	
Unlock X	Unlock X	
Lock X		Lock X
Lock Y		Lock Y
Read X		Read X
Read Y		Read Y
Display X+Y		Display X+Y
Unlock X		Unlock X
Unlock Y		Unlock Y
Lock Y	Lock Y	
Read Y	Read Y	
Add 100	Add 100	
Write Y	Write Y	
Unlock Y	Unlock Y	

Figure 7: An incorrect locking implementation

Does the locking as done above solve the problem of concurrent transactions? No the same problems still remains. Try working with the old value. Thus, locking should be done with some logic in order to make sure that locking results in no concurrency related problem. One such solution is given below:

Schedule	T1	T2
Lock X	Lock X	
Lock Y	Lock Y	
Read X	Read X	
Subtract 100	Subtract 100	
Write X	Write X	
Lock X (issued by T2)	Lock X: denied as T1 holds the lock. The transaction T2 Waits and T1 continues.	
Read Y	Read Y	
Add 100	Add 100	
Write Y	Write Y	
Unlock X	Unlock X	
	The lock request of T2 on X can now be granted it can resumes by locking X	
Unlock Y	Unlock Y	
Lock Y		Lock Y
Read X		Read X
Read Y		Read Y
Display X+Y		Display X+Y
Unlock X		Unlock X
Unlock Y		Unlock Y

Figure 8: A correct but restrictive locking implementation

Thus, the locking as above when you obtain all the locks at the beginning of the transaction and release them at the end ensures that transactions are executed with no concurrency related problems. However, such a scheme limits the concurrency. We will discuss a two-phase locking method in the next subsection that provides sufficient concurrency. However, let us first discuss multiple mode locks.

Multiple-mode locks: It offers two locks: shared locks and exclusive locks. But why do we need these two locks? There are many transactions in the database system that never update the data values. These transactions can coexist with other transactions that update the database. In such a situation multiple reads are allowed on a data item, so multiple transactions can lock a data item in the shared or read lock. On the other hand, if a transaction is an updating transaction, that is, it updates the data items, it has to ensure that no other transaction can access (read or write) those data items that it wants to update. In this case, the transaction places an exclusive lock on the data items. Thus, a somewhat higher level of concurrency can be achieved in comparison to the binary locking scheme.

The properties of shared and exclusive locks are summarised below:

a) Shared lock or Read Lock

- It is requested by a transaction that wants to just read the value of data item.
- A shared lock on a data item does not allow an exclusive lock to be placed but permits any number of shared locks to be placed on that item.

b) Exclusive lock

- It is requested by a transaction on a data item that it needs to update.
- No other transaction can place either a shared lock or an exclusive lock on a data item that has been locked in an exclusive mode.

Let us describe the above two modes with the help of an example. We will once again consider the transactions T1 and T2 but in addition a transaction T11 that finds the total of accounts Y and Z.

Schedule	T1	T2	T11
S_Lock X		S_Lock X	
S_Lock Y		S_Lock Y	
Read X		Read X	
S_Lock Y			S_Lock Y
S_Lock Z			S_Lock Z
			Read Y
			Read Z
X_Lock X	X_Lock X. The exclusive lock request on X is denied as T2 holds the Read lock. The transaction T1 Waits.		
Read Y		Read Y	
Display X+Y		Display X+Y	
Unlock X		Unlock X	
X_Lock Y	X_Lock Y. The previous exclusive lock requeston X is granted as X is unlocked. But the new exclusive lock request on Y is not granted as Y is locked by T2 and T11 in read mode. Thus T1waits till both T2 and T11 will release the read lock on Y.		
Display Y+Z			Display Y+Z
Unlock Y		Unlock Y	
Unlock Y			Unlock Y
Unlock Z			Unlock Z
Read X	Read X		
Subtract 100	Subtract 100		
Write X	Write X		
Read Y	Read Y		
Add 100	Add 100		
Write Y	Write Y		
Unlock X	Unlock X		
Unlock Y	Unlock Y		

Figure 9: Example of Locking in multiple-modes

Thus, the locking as above results in a serialisable schedule. Now the question is can we release locks a bit early and still have no concurrency related problem? Yes, we can do it if we lock using two-phase locking protocol. This protocol is explained in the next sub-section.

10.4.3 Two Phase Locking (2PL)

The two-phase locking protocol consists of two phases:

Phase 1: The lock acquisition phase: If a transaction T wants to read an object, it needs to obtain the S (shared) lock. If T wants to modify an object, it needs to obtain X (exclusive) lock. No conflicting locks are granted to a transaction. **New locks on items can be acquired but no lock can be released till all the locks required by the transaction are obtained.**

Phase 2: Lock Release Phase: The existing locks can be released in any order but no new lock can be acquired **after a lock has been released**. The locks are held only till they are required.

Normally the locks are obtained by the DBMS. Any legal schedule of transactions that follows 2 phase locking protocol is guaranteed to be serialisable. The two phase locking protocol has been proved for it correctness. However, the proof of this protocol is beyond the scope of this Unit. You can refer to further readings for more details on this protocol.

There are two types of 2PL:

(1) The Basic 2PL
(2) Strict 2PL

The basic 2PL allows release of lock at any time after all the locks have been acquired. For example, we can release the locks in schedule of *Figure 8*, after we have Read the values of Y and Z in transaction 11, even before the display of the sum. This will enhance the concurrency level. The basic 2PL is shown graphically in *Figure 10*.

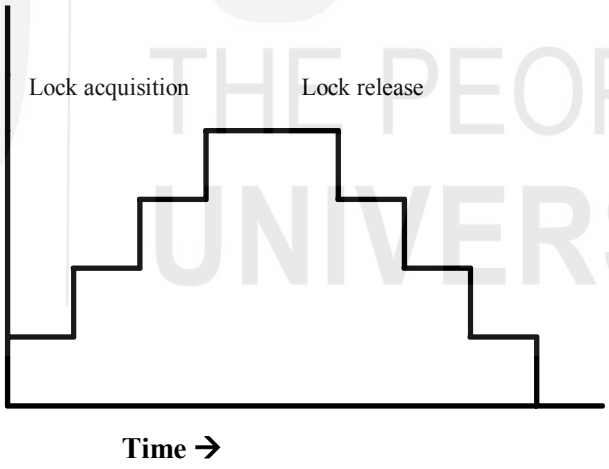


Figure 10: Basic Two Phase Locking

However, this basic 2PL suffers from the problem that it can result into loss of atomic / isolation property of transaction as theoretically speaking once a lock is released on data item it can be modified by another transaction before the first transaction commits or aborts.

To avoid such a situation we use strict 2PL. The strict 2PL is graphically depicted in *Figure 11*. However, the basic disadvantage of strict 2PL is that it restricts concurrency as it locks the item beyond the time it is needed by a transaction.

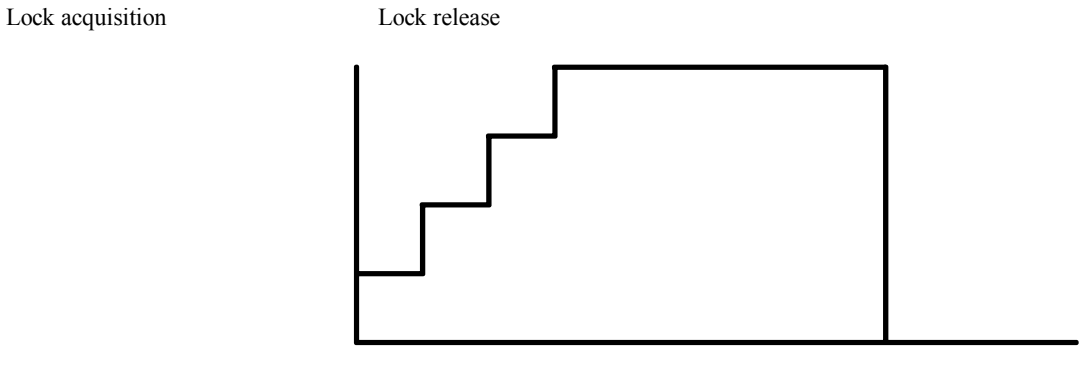


Figure 11: Strict Two Phase Locking

Does the 2PL solve all the problems of concurrent transactions? No, the strict 2PL solves the problem of concurrency and atomicity, however it introduces another problem: "Deadlock". Let us discuss this problem in next section.

☞ Check Your Progress 2

1) Let the transactions T1, T2, T3 be defined to perform the following operations:

   T1: Add one to A
   T2: Double A
   T3: Display A on the screen and set A to one.

   Suppose transactions T1, T2, T3 are allowed to execute concurrently. If A has initial value zero, how many possible correct results are there? Enumerate them.

……

……

2) Consider the following two transactions, given two bank accounts having a balance A and B.

   Transaction T1: Transfer Rs. 100 from A to B

   Transaction T2: Find the multiple of A and B.

   Create a non-serialisable schedule.

……

……

3) Add lock and unlock instructions (exclusive or shared) to transactions T1 and T2 so that they observe the serialisable schedule. Make a valid schedule.

……

……

10.5 DEADLOCK AND ITS PREVENTION

As seen earlier, though 2PL protocol handles the problem of serialisability, but it causes some problems also. For example, consider the following two transactions anda schedule involving these transactions:

TA	**TB**
X_lock A	X_lock A
X_lock B	X_lock B
:	:
:	:
Unlock A	Unlock A
Unlock B	Unlock B

Schedule

T1: X_lock A
T2: X_lock B
T1: X_lock B
T2: X_lock A

As is clearly seen, the schedule causes a problem. After T1 has locked A, T2 locks B and then T1 tries to lock B, but unable to do so waits for T2 to unlock B. Similarly, T2 tries to lock A but finds that it is held by T1 which has not yet unlocked it and thus waits for T1 to unlock A. At this stage, neither T1 nor T2 can proceed since both of these transactions are waiting for the other to unlock the locked resource.

Clearly the schedule comes to a halt in its execution. The important thing to be seen here is that both T1 and T2 follow the 2PL, which guarantees serialisability. So whenever the above type of situation arises, we say that a deadlock has occurred, since two transactions are **waiting for a condition that will never occur**.

Also, the deadlock can be described in terms of a directed graph called a "*wait for*" graph, which is maintained by the lock manager of the DBMS. This graph G is defined by the pair (V, E). It consists of a set of vertices/nodes V is and a set of edges/arcs E. Each transaction is represented by node and an arc from Ti → Tj, if Tj holds a lock and Ti is waiting for it. When transaction Ti requests a data item currently being held by transaction Tj then the edge Ti → Tj is inserted in the "wait for" graph. This edge is removed only when transaction Tj_j is no longer holding the data item needed by transaction Ti.

A deadlock in the system of transactions occurs, if and only if the wait-for graph contains a cycle. Each transaction involved in the cycle is said to be deadlocked.
To detect deadlocks, a periodic check for cycles in graph can be done. For example, the "wait-for" for the schedule of transactions TA and TB as above can be made as:

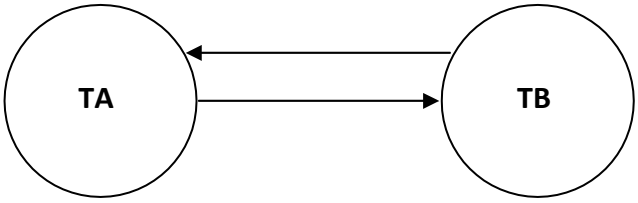


Figure 12: Wait For graph of TA and TB

In the figure above, TA and TB are the two transactions. The two edges are present between nodes TA and TB since each is waiting for the other to unlock a resource held by the other, forming a cycle, causing a deadlock problem. The above case shows a direct cycle. However, in actual situation more than two nodes may be there in a cycle.

A deadlock is thus a situation that can be created because of locks. It causes transactions to wait forever and hence the name deadlock. A deadlock occurs because of the following conditions:

a) Mutual exclusion: A resource can be locked in exclusive mode by only one transaction at a time.
b) Non-preemptive locking: A data item can only be unlocked by the transaction that locked it. No other transaction can unlock it.
c) Partial allocation: A transaction can acquire locks on database in a piecemeal fashion.
d) Circular waiting: Transactions lock part of data resources needed and then waitindefinitely to lock the resource currently locked by other transactions.

In order to prevent a deadlock, one has to ensure that at least one of these conditions does not occur.

A deadlock can be prevented, avoided or controlled. Let us discuss a simple method for deadlock prevention.

Deadlock Prevention

One of the simplest approaches for avoiding a deadlock would be to acquire all the locks at the start of the transaction. However, this approach restricts concurrency greatly, also you may lock some of the items that are not updated by that transaction (the transaction may have if conditions).Thus, better prevention algorithm have been evolved to prevent a deadlock having the basic logic: *not to allow circular wait to occur*. This approach rolls back some of the transactions instead of letting them wait.

There exist two such schemes. These are:

"Wait-die" scheme: The scheme is based on non-preventive technique. It isbased on a simple rule:

```
If Ti requests a database resource that is held by Tj
        then if Ti has a smaller timestamp than that of Tj
                it is allowed to wait;
        else Ti aborts.
```

A timestamp may loosely be defined as the system generated sequence number that is unique for each transaction. Thus, a smaller timestamp means an older transaction.
For example, assume that three transactions T1, T2 and T3 were generated in that sequence, then if T1requests for a data item which is currently held by transaction T2, it is allowed to wait as it has a smaller time stamping than that of T1. However, if T3 requests for a data item which is currently held by transaction T2, then T3 is rolled back (die).

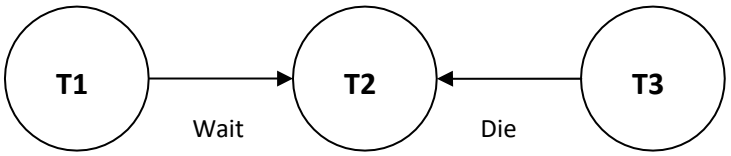


Figure 13: Wait-die Scheme of Deadlock prevention

"Wound-wait" scheme: It is based on a preemptive technique. It is based on a simple rule:

If Ti requests a database resource that is held by Tj
then if Ti has a larger timestamp (Ti is younger) than that of Tj
it is allowed to wait;
else Tj is wounded up by Ti.

For example, assume that three transactions T1, T2 and T3 were generated in that sequence, then if T1requests for a data item which is currently held by transaction T2, then T2 is rolled back and data item is allotted to T1 as T1 has a smaller time stamping than that of T2. However, if T3 requests for a data item which is currently held by transaction T2, then T3 is allowed to wait.

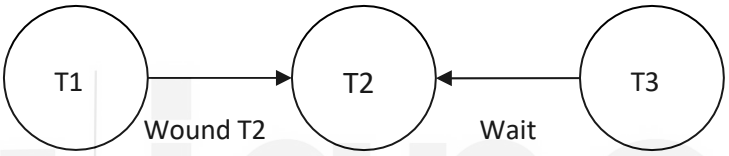


Figure 14: Wound-wait Scheme of Deadlock prevention

It is important to see that whenever any transaction is rolled back, it would not make a starvation condition, that is no transaction gets rolled back repeatedly and is never allowed to make progress. Also both "wait-die" & "wound-wait" scheme avoid starvation. The number of aborts & rollbacks will be higher in wait-die scheme than in the wound-wait scheme. But one major problem with both of these schemes is that these schemes may result in unnecessary rollbacks. You can refer to further readings for more details on deadlock related schemes.

2.6 OPTIMISTIC CONCURRENCY CONTROL

Is locking the only way to prevent concurrency related problems? There exist some other methods too. One such method is called an Optimistic Concurrency control. Let us discuss it in more detail in this section.

The basic logic in optimistic concurrency control is to allow the concurrent transactions to update the data items assuming that the concurrency related problem will not occur. However, we need to reconfirm our view in the validation phase. Therefore, the optimistic concurrency control algorithm has the following phases:

a) READ Phase: A transaction T reads the data items from the database into its private workspace. All the updates of the transaction can only change the local copies of the data in the private workspace.

b) VALIDATE Phase: Checking is performed to confirm whether the read values have changed during the time transaction was updating the local values. This is performed by comparing the current database values to the values that were read in the private workspace. In case, the values have changed the local copies

are thrown away and the transaction aborts.

c) WRITE Phase: If validation phase is successful the transaction is committed andupdates are applied to the database, otherwise the transaction is rolled back.

Some of the terms defined to explain optimistic concurrency contents are:

- write-set(T): all data items that are written by a transaction T
- read-set(T): all data items that are read by a transaction T
- Timestamps: for each transaction T, the start-time and the end time are kept for all the three phases.

More details on this scheme are available in the further readings. But let us show this scheme here with the help of the following examples: Consider the set for transaction

T1 and T2.

T1		**T2**	
Phase	**Operation**	**Phase**	**Operation**
-	-	Read	Reads the read set (T2). Let say variables X and Y and performs updating of local values
Read	Reads the read set (T1) lets say variable X and Y and performs updating of local values	-	-
Validate	Validate the values of (T1)	-	-
-	-	Validate	Validate the values of (T2)
Write	Write the updated values in the database and commit	-	-
-	-	Write	Write the updated values in the database and commit

In this example both T1 and T2 get committed. Please note that Read set of T1 and Read Set of T2 are both disjoint, also the Write sets are also disjoint and thus no concurrency related problem can occur.

T1	T2	T3
Operation	Operation	Operation
Read R(A)	--	--
--	Read R(A)	--
--	--	Read (D)
--	--	Update(D)
--	--	Update (A)
--	--	Validate (D,A) finds OK Write (D,A), COMMIT
--	Validate(A):Unsuccessful Value changed by T3	--
Validate(A):Unsuccessful Value changed by T3	--	--
ABORT T1	--	--
--	Abort T2	--

In this case both T1 and T2 get aborted as they fail during validate phase while only T3 is committed. Optimistic concurrency control performs its checking at

the transaction commits point in a validation phase. The serialization order is determinedby the time of transaction validation phase.

☞ Check Your Progress 3

1) Draw suitable graph for following locking requests, find whether thetransactions are deadlocked or not.

T1: S_lock A	--	--
--	T2: X_lock B	--
--	T2: S_lock C	--
--	--	T3: X_lock C
--	T2: S_lock A	--
T1: S_lock B	--	--
T1: S_lock A	--	--
--	--	T3: S_lock A
All the unlocking requests start from here		

……

……

……

……

……

2) What is Optimistic Concurrency Control?

……

……

……

……

10.7 SUMMARY

In this unit you have gone through the concepts of transaction and Concurrency Management. A transaction is a sequence of many actions. Concurrency control deals with ensuring that two or more users do not get into each other's way, i.e., updates of transaction one doesn't affect the updates of other transactions.

Serializability is the generally accepted criterion for correctness for the concurrency control. It is a concept related to concurrent schedules. It determines how to analyse whether any schedule is serialisable or not. Any schedule that produces the same results as a serial schedule is a serialisable schedule.

Concurrency Control is usually done via locking. If a transaction tries to lock a resource already locked by some other transaction, it cannot do so and waits for the resource to be unlocked.

Locks are of two type a) shared lock b) Exclusive lock. Then we move on to a method known as Two Phase Locking (2PL). A system is in a deadlock state if there exist a set of transactions such that every transaction in the set is waiting for another transaction in the set. We can use a deadlock prevention protocol to ensure that the system will never enter a deadlock state.

Finally we have discussed the method Optimistic Concurrency Control, another concurrency management mechanism.

10.8 SOLUTIONS/ANSWERS

Check Your Progress 1

1) A transaction is the basic unit of work on a Database management system. It defines the data processing on the database. IT has four basic properties:

   a. Atomicity: transaction is done completely or not at all.
   b. Consistency: Leaves the database in a consistent state
   c. Isolation: Should not see uncommitted values
   d. Durability: Once committed the changes should be reflected.

   A transaction can update more than one data values. Some transactions can do writing of data without reading a data value.

   A simple transaction example may be: Updating the stock inventory of an itemthat has been issued. Please create a sample pseudo code for it.

2) The basic problems of concurrent transactions are:

- Lost updates: An update is overwritten
- Unrepeatable read: On reading a value later again an inconsistent value is found.
- Dirty read: Reading an uncommitted value
- Inconsistent analysis: Due to reading partially updated value.

No these problems cannot occur if the transactions do not read the same data values. The conflict occurs only if one transaction updates a data value while another is reading or writing the data value.

Commit state is defined as when transaction has done everything correctly and shows the intent of making all the changes as permanent. No, you cannot rollback after commit.

Check Your Progress 2

1) There are six possible results, corresponding to six possible serial schedules:

Initially:	A = 0
T1-T2-T3:	A = 1
T1-T3-T2:	A = 2
T2-T1-T3:	A = 1
T2-T3-T1:	A = 2
T3-T1-T2:	A = 4
T3-T2-T1:	A = 3

2)

Schedule	T1	T2
Read A	Read A	
A = A - 100	A = A - 100	
Write A	Write A	
Read A		Read A
Read B		Read B
Read B	Read B	
Result = A * B		Result = A * B
Display Result		Display Result
B = B + 100	B = B + 100	
Write B	Write B	

Please make the precedence graph and find out that the schedule is not serialisable.3)

Schedule	T1	T2
Lock A	Lock A	
Lock B	Lock B	
Read A	Read A	
A = A - 100	A = A - 100	
Write A	Write A	
Unlock A	Unlock A	
Lock A		Lock A: Granted
Lock B		Lock B: Waits
Read B	Read B	
B = B + 100	B = B + 100	
Write B	Write B	
Unlock B	Unlock B	
Read A		Read A
Read B		Read B
Result = A * B		Result = A * B
Display Result		Display Result
Unlock A		Unlock A
Unlock B		Unlock B

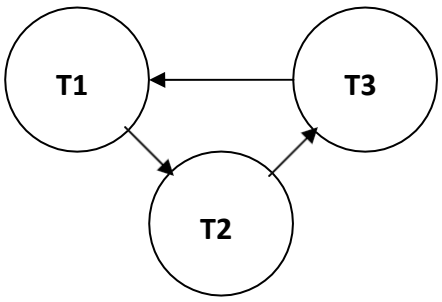
You must make the schedules using read and exclusive lock and a schedule in strict 2PL.

Check Your Progress 3

1) The transaction T1 gets the shared lock on A, T2 gets exclusive lock on B and Shared lock on A, while the transactions T3 gets exclusive lock on C.

- Now T2 requests for shared lock on C which is exclusively locked by T3, socannot be granted. So T2 waits for T3 on item C.
- T1 now requests for Shared lock on B which is exclusively locked by T2, thus, it waits for T2 for item B. The T1 request for shared lock on C is notprocessed.
- Next T3 requests for exclusive lock on A which is share locked by T1, so itcannot be granted. Thus, T3 waits for T1 for item A.

The Wait for graph for the transactions for the given schedule is:



Since there exists a cycle, therefore, the schedule is deadlocked.

2) The basic philosophy for optimistic concurrency control is the optimism that nothing will go wrong so let the transaction interleave in any fashion, but to avoid any concurrency related problem we just validate our assumption before we make changes permanent. This is a good model for situations having a low rate of transactions.

UNIT 11 DATABASE RECOVERY AND SECURITY (Adopted from MCS-023 Block-2 Unit-3)

11.0 INTRODUCTION

In the previous unit of this block, you have gone through the concepts of transactions and Concurrency management. In this unit we will introduce two important issues relating to database management systems.

A computer system suffers from different types of failures. A DBMS controls very critical data of an organisation and therefore must be reliable. However, the reliability of the database system is linked to the reliability of the computer system on which it runs. In this unit we will discuss recovery of the data contained in a database system following failure of various types and present the different approaches to database recovery. The types of failures that the computer system is likely to be subjected to include failures of components or subsystems, software failures, power outages, accidents, unforeseen situations and natural or man-made disasters. Database recovery techniques are methods of making the database consistent till the last possible consistent state. The aim of recovery scheme is to allow database operations to be resumed after a failure with minimum loss of information at an economically justifiable cost.

The second main issue that is being discussed in this unit is Database security. "Database security" is protection of the information contained in the database against unauthorised access, modification or destruction. The first condition for security is to have Database integrity. "Database integrity" is the mechanism that is applied to ensure that the data in the database is consistent.

Let us discuss all these terms in more detail in this unit.

11.1 OBJECTIVES

At the end of this unit, you should be able to:

- describe the terms RECOVERY and INTEGRITY;
- describe Recovery Techniques;

- define Error and Error detection techniques, and
- describe types of Authorisation

11.2 WHAT IS RECOVERY?

During the life of a transaction, that is, a after the start of a transaction but before the transaction commits, several changes may be made in a database state. The database during such a state is in an inconsistent state. What happens when a failure occurs at this stage? Let us explain this with the help of an example:

Assume that a transaction transfers Rs.2000/- from A's account to B's account. For simplicity we are not showing any error checking in the transaction. The transaction may be written as:

Transaction T1:

```
READ A
A = A – 2000
WRITE A
                Failure  ------------>
READ B
B = B + 2000
WRITE B
COMMIT
```

What would happen if the transaction fails after account A has been written back to database? As far as the holder of account A is concerned s/he has transferred the money but that has never been received by account holder B.

Why did this problem occur? Because although a transaction is considered to be atomic, yet it has a life cycle during which the database gets into an inconsistent state and failure has occurred at that stage.

What is the solution? In this case where the transaction has not yet committed the changes made by it, the partial updates need to be undone.

How can we do that? By remembering information about a transaction such as when did it start, what items it updated etc. All such details are kept in a log file. We will study about log in Section 3.3. But first let us analyse the reasons of failure.

Failures and Recovery

In practice several things might happen to prevent a transaction from completing. Recovery techniques are used to bring database, which does not satisfy consistency requirements, into a consistent state. If a transaction completes normally and commits then all the changes made by the transaction on the database are permanently registered in the database. They should not be lost (please recollect the durability property of transactions given in Unit 2). But, if a transaction does not complete normally and terminates abnormally then all the changes made by it should be discarded. An abnormal termination of a transaction may be due to several reasons, including:

a) user may decide to abort the transaction issued by him/ her
b) there might be a deadlock in the system

c) there might be a system failure.

The recovery mechanisms must ensure that a consistent state of database can be restored under all circumstances. In case of transaction abort or deadlock, the system remains in control and can deal with the failure but in case of a system failure the system loses control because the computer itself has failed. Will the results of such failure be catastrophic? A database contains a huge amount of useful information anD any system failure should be recognised on the restart of the system. The DBMS should recover from any such failures. Let us first discuss the kinds of failure for identifying how to recover.

11.2.1 Kinds of Failures

The kinds of failures that a transaction program during its execution can encounterare:

1) **Software failures:** In such cases, a software error abruptly stops the execution of the current transaction (or all transactions), thus leading to losing the state of program excution and the state/ contents of the buffers. But what is a buffer? A buffer is the portion of RAM that stores the partial contents of database that is currently needed by the transaction. The software failures can further be subdivided as:

   a) Statement or application program failure
   b) Failure due to viruses
   c) DBMS software failure
   d) Operating system failure

A Statement of program may cause abnormal termination if it does not execute completely. This happens if during the execution of a statement, an integrity constraint gets violated. This leads to abnormal termination of the transaction due to which any prior updates made by the transaction may still get reflected in the database leaving it in an inconsistent state.

A failure of transaction can occur if some code in a transaction program leads toits abnormal termination. For example, a transaction can go into an infinite loop. In such a case the only way to break the loop is to abort the program. Similarly, the failure can be traced to the operating system or DBMS and transactions are aborted abruptly. Thus part of the transaction that was executed before abort may cause some updates in database, and hence the database is updated only partially which leads to an inconsistent state of database.

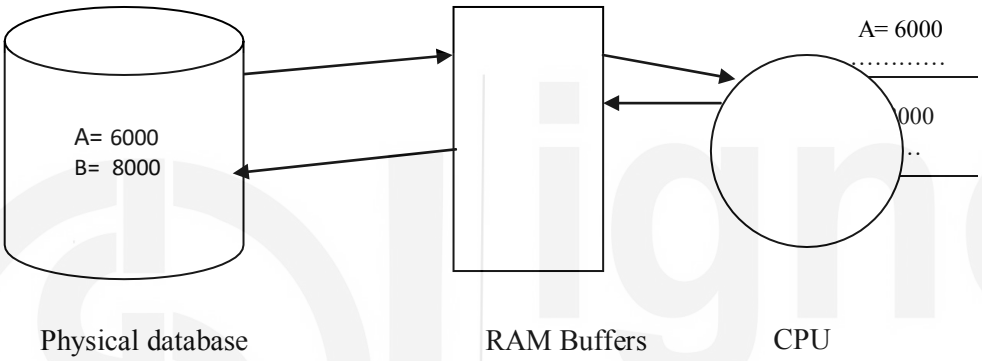
2) **Hardware failure:** Hardware failures are those failures when some hardware chip or disk fails. This may result in loss of data. One such problem can be that a disk gets damaged and cannot be read any more. This may be due to many reasons. For example, a voltage fluctuation in the power supply to the computer makes it go off or some bad sectors may come on disk or there is a disk crash. In all these cases, the database gets into an inconsistent state.

3) **External failure:** A failure can also result due to an external cause, such as fire, earthquakes, floods, etc. The database must be duly backed up to avoid problems occurring due to such failures.

In practice software failures are more common than hardware failures. Fortunately, recovery from software failures is much quicker.

The basic unit of recovery is a transaction. But, how are the transactions handled during recovery? Consider that some transactions are deadlocked, then at least one of

these transactions has to be restarted to break the deadlock and thus the partial updates made by such restarted program in the database need to be **undone** so that the database does not go to an inconsistent state. So the transaction may have to be rolled back which makes sure that the transaction does not bring the database to an inconsistent state. This is one form of recovery. Let us consider a case when a transaction has committed but the changes made by the transaction have not been communicated to permanently stored physical database. A software failure now occurs and the contents of the CPU/ RAM are lost. This leaves the database in an inconsistent state. Such failure requires that on restarting the system the database be brought to a consistent state using **redo** operation. The redo operation makes the changes made by the transaction again to bring the system to a consistent state. The database system then can be made available to the users. The point to be noted here is that the database updates are performed in the buffer in the memory. *Figure 1* shows some cases of undo and redo. You can create more such cases.



	Physical Database	RAM	Activity
Case 1	A=6000 B=8000	A=4000 B=8000	Transaction T1 has just changed the value in RAM. Now it aborts, value in RAM is lost. No problem. But we are not sure that the physical database has been written back, so must undo.
Case 2	A=4000 B=8000	A=4000 B=8000	The value in physical database has got updated due to buffer management, now the transaction aborts. The transaction must be undone.
Case 3	A=6000 B=8000	A=4000 B=10000 Commit	The value B in physical database has not got updated due to buffer management. In case of failure now when the transaction has committed. The changes of transaction must be redone to ensure transfer of correct values to physical database.

Figure 1: Database Updates And Recovery

11.2.2 Failure Controlling Methods

Failures can be handled using different recovery techniques that are discussed later in

the unit. But the first question is do we really need recovery techniques as a failure control mechanism? The recovery techniques are somewhat expensive both in terms of time and in memory space for small systems. In such a case it is more beneficial to better avoid the failure by some checks instead of deploying recovery technique to make database consistent. Also, recovery from failure involves manpower that can be used in some other productive work if failure can be avoided. It is, therefore, important to find out some general precautions that help in controlling failure. Some of these precautions may be:

- having a regulated power supply.
- having a better secondary storage system such as RAID.
- taking periodic backup of database states and keeping track of transactions aftereach recorded state.
- properly testing the transaction programs prior to use.
- setting important integrity checks in the databases as well as user interfaces etc.

However, it may be noted that if the database system is critical it must use a DBMS that is suitably equipped with recovery procedures.

11.2.3 Database Errors

An error is said to have occurred if the execution of a command to manipulate the database cannot be successfully completed either due to inconsistent data or due to state of program. For example, there may be a command in program to store data in database. On the execution of command, it is found that there is no space/place in database to accommodate that additional data. Then it can be said that an error has occurred. This error is due to the physical state of database storage.

Broadly errors are classified into the following categories:

1) **User error:** This includes errors in the program (e.g., Logical errors) as well as errors made by online users of database. These types of errors can be avoided by applying some check conditions in programs or by limiting the access rights of online users e.g., read only. So only updating or insertion operation require appropriate check routines that perform appropriate checks on the data being entered or modified. In case of an error, some prompts can be passed to user to enable him/her to correct that error.

2) **Consistency error:** These errors occur due to the inconsistent state of database caused may be due to wrong execution of commands or in case of a transaction abort. To overcome these errors the database system should include routines that check for the consistency of data entered in the database.

3) **System error:** These include errors in database system or the OS, e.g., deadlocks. Such errors are fairly hard to detect and require reprogramming the erroneous components of the system software.

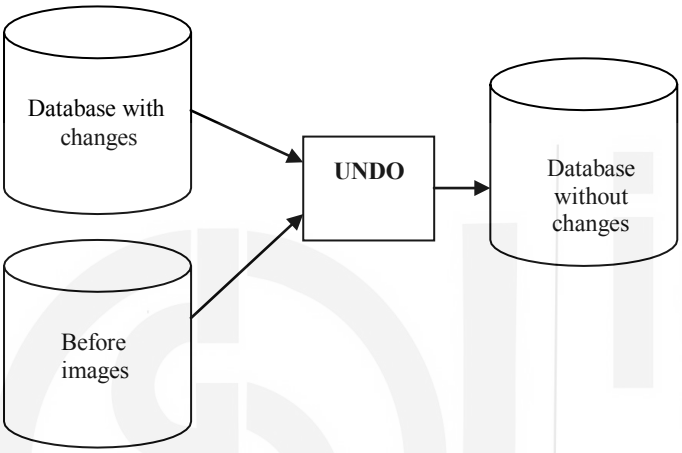
Database errors can result from failure or can cause failure and thus will require recovery. However, one of the main tasks of database system designers is to make sure that errors minimised. These concepts are also related to database integrity and have also been discusses in a later section.

11.3 RECOVERY TECHNIQUES

After going through the types of failures and database errors, let us discuss how to recover from the failures. Recovery can be done using/restoring the previous consistent state (backward recovery) or by moving forward to the next consistent state as per the committed transactions (forward recovery) recovery. Please note that a system can recover from software and hardware failures using the forward and backward recovery only if the system log is intact. What is system log? We will discuss it in more detail, but first let us define forward and backward recovery.

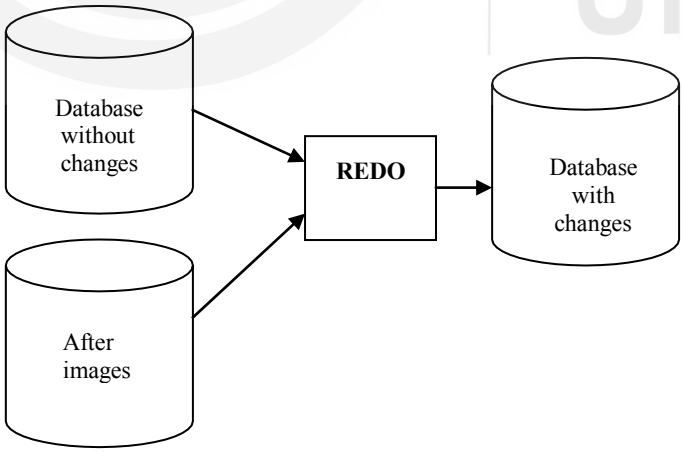
1) Backward Recovery (UNDO)

In this scheme the uncommitted changes made by a transaction to a database are undone. Instead the system is reset to the previous consistent state of database that is free from any errors.



2) Forward Recovery (Redo)

In this scheme the committed changes made by a transaction are reapplied to anearlier copy of the database.



In simpler words, when a particular error in a system is detected, the recovery system makes an accurate assessment of the state of the system and then makes the appropriate adjustment based on the anticipated results - had the system been error free.

One thing to be noted is that the Undo and Redo operations must be idempotent, i.e., executing them several times must be equivalent to executing them once. This

characteristic is required to guarantee correct behaviour of database even if a failure occurs during the recovery process.

Depending on the above discussed recovery scheme, several types of recovery methods have been used. However, we define the most important recovery schemes used in most of the commercial DBMSs

Log based recovery

Let us first define the term transaction log in the context of DBMS. A transaction log is a record in DBMS that keeps track of all the transactions of a database system that update any data values in the database. A log contains the following information about a transaction:

- A transaction begin marker
- The transaction identification: The transaction id, terminal id or user id etc.
- The operations being performed by the transaction such as update, delete, insert.
- The data items or objects that are affected by the transaction including name of the table, row number and column number.
- The before or previous values (also called UNDO values) and after or changed values (also called REDO values) of the data items that have been updated.
- A pointer to the next transaction log record, if needed.
- The COMMIT marker of the transaction.

In a database system several transactions run concurrently. When a transaction commits, the data buffers used by it need not be written back to the physical database stored on the secondary storage as these buffers may be used by several other transactions that have not yet committed. On the other hand, some of the data buffers that may have updates by several uncommitted transactions might be forced back to the physical database, as they are no longer being used by the database. So the transaction log helps in remembering which transaction did which changes. Thus the system knows exactly how to separate the changes made by transactions that have already committed from those changes that are made by the transactions that did not yet commit. Any operation such as begin transaction, insert /delete/update and end transaction (commit), adds information to the log containing the transaction identifier and enough information to undo or redo the changes.

But how do we recover using log? Let us demonstrate this with the help of an example having three concurrent transactions that are active on ACCOUNTS table as:

Transaction T1	**Transaction T2**	**Transaction T3**
Read X	Read A	Read Z
Subtract 100	Add 200	Subtract 500
Write X	Write A	Write Z
Read Y		
Add 100		
Write Y		

Figure 2: The sample transactions

Assume that these transactions have the following log file (hypothetical) at a point:

Transaction Begin Marker	Transaction Id	Operation on ACCOUNTS table	UNDO values (assumed)	REDO values	Transaction Commit Marker
Y	T1	Sub on X Add on Y	500 800	400 Not done yet	N
Y	T2	Add on A	1000	1200	N
Y	T3	Sub on Z	900	400	Y

Figure 3: A sample (hypothetical) Transaction log

Now assume at this point of time a failure occurs, then how the recovery of the database will be done on restart.

Values	Initial	Just before the failure	Operation Required for recovery	Recovered DatabaseValues
X	500	400 (assuming update has been done in physical database also)	UNDO	500
Y	800	800	UNDO	800
A	1000	1000 (assuming update has not been done in physical database)	UNDO	1000
Z	900	900 (assuming update has not been done in physical database)	REDO	400

Figure 4: The database recovery

The selection of REDO or UNDO for a transaction for the recovery is done on the basis of the state of the transactions. This state is determined in two steps:

- Look into the log file and find all the transactions that have started. For example,in *Figure 3*, transactions T1, T2 and T3 are candidates for recovery.
- Find those transactions that have committed. REDO these transactions. All other transactions have not committed so they should be rolled back, so UNDO them. For example, in *Figure 3* UNDO will be performed on T1 and T2; and REDO willbe performed on T3.

Please note that in Figure 4 some of the values may not have yet been communicated to database, yet we need to perform UNDO as we are not sure what values have been written back to the database.
But how will the system recover? Once the recovery operation has been specified, the system just takes the required REDO or UNDO values from the transaction log and changes the inconsistent state of database to a consistent state. (Please refer to *Figure 3* and *Figure 4*).

Let us consider several transactions with their respective start & end (commit) times as shown in *Figure 5*.

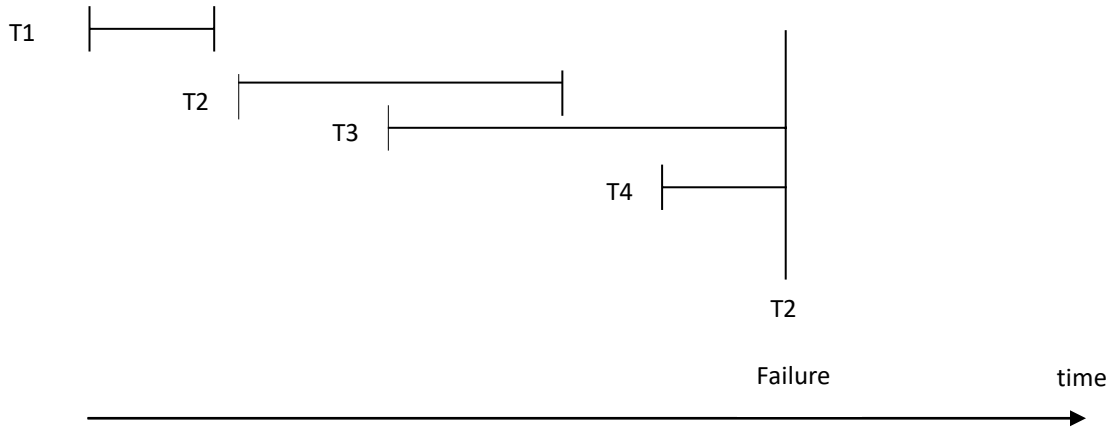


Figure 5: Execution of Concurrent Transactions

In the figure above four transactions are executing concurrently, on encountering a failure at time t2, the transactions T1 and T2 are to be REDONE and T3 and T4 will be UNDONE. But consider a system that has thousands of parallel transactions then all those transactions that have been committed may have to be redone and all uncommitted transactions need to be undone. That is not a very good choice as it requires redoing of even those transactions that might have been committed even hours earlier. So can we improve on this situation? Yes, we can take checkpoints. *Figure 6* shows a checkpoint mechanism:

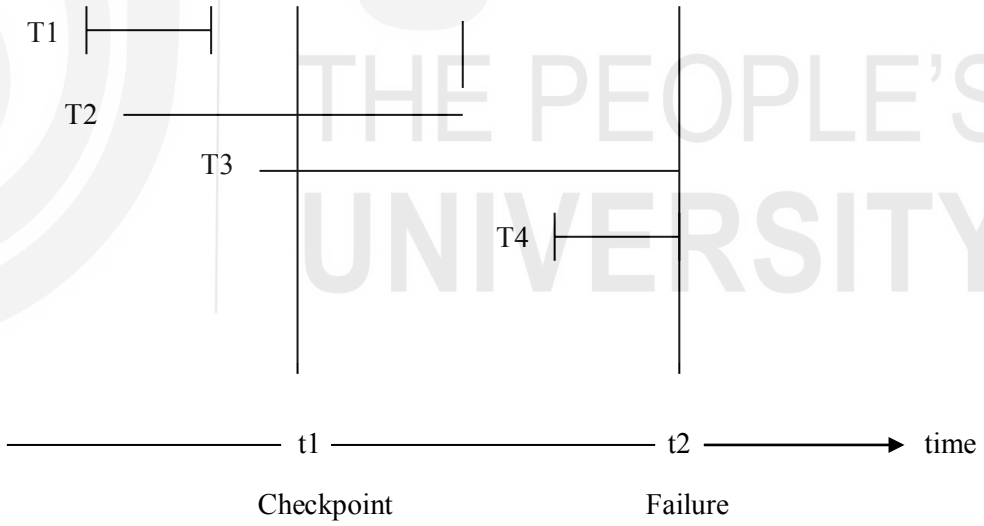


Figure 6: Checkpoint In Transaction Execution

A checkpoint is taken at time t_1 and a failure occurs at time t_2. Checkpoint transfers all the committed changes to database and all the system logs to stable storage (it is defined as the storage that would not be lost). At restart time after the failure the stable check pointed state is restored. Thus, we need to only REDO or UNDO those transactions that have completed or started after the checkpoint has been taken. The only possible disadvantages of this scheme may be that during the time of taking the checkpoint the database would not be available and some of the uncommitted values may be put in the physical database. To overcome the first problem the checkpoints should be taken at times when system load is low. To avoid the second problem some systems allow some time to the ongoing transactions to complete without restarting

new transactions.

In the case of *Figure 6* the recovery from failure at t_2 will be as follows:

- The transaction T1 will not be considered for recovery as the changes made by it have already been committed and transferred to physical database at checkpoint t_1.
- The transaction T2 since it has not committed till the checkpoint t_1 but have committed before t_2, will be REDONE.
- T3 must be UNDONE as the changes made by it before checkpoint (we do not know for sure if any such changes were made prior to checkpoint) must have been communicated to the physical database. T3 must be restarted with a new name.
- T4 started after the checkpoint, and if we strictly follow the scheme in which the buffers are written back only on the checkpoint, then nothing needs to be done except restarting the transaction T4 with a new name.

The restart of a transaction requires the log to keep information of the new name of the transaction and maybe give higher priority to this newer transaction.

But one question is still unanswered that is during a failure we lose database information in RAM buffers, we may also lose log as it may also be stored in RAM buffers, so how does log ensure recovery?

The answer to this question lies in the fact that for storing transaction log we follow a **Write Ahead Log Protocol.** As per this protocol, the transaction logs are written to stable storage before any item is updated. Or more specifically it can be stated as; **the undo potion of log is written to stable storage prior to any updates and redo portion of log is written to stable storage prior to commit.**

Log based recovery scheme can be used for any kind of failure provided you have stored the most recent checkpoint state and most recent log as per write ahead log protocol into the stable storage. Stable storage from the viewpoint of external failure requires more than one copy of such data at more than one location. You can refer to the further readings for more details on recovery and its techniques.

Check Your Progress 1

1) What is the need of recovery? What is it the basic unit of recovery?

……

……

2) What is a checkpoint? Why is it needed? How does a checkpoint help in recovery?

……

……

3) What are the properties that should be taken into consideration while selecting recovery techniques?

……

……..……..……..

11.4 SECURITY AND INTEGRITY

After going through the concepts of database recovery in the previous section, let us now deal with an important concept that helps in minimizing consistency errors in database systems. These are the concepts of database security and integrity.

Information security is the protection of information against unauthorised disclosure, alteration or destruction. Database security is the protection of information that is maintained in a database. It deals with ensuring only the "right people" get the right to access the "right data". By right people we mean those people who have the right to access/update the data that they are requesting to access/update with the database. This should also ensure the confidentiality of the data. For example, in an educational institution, information about a student's grades should be made available only to that student, whereas only the university authorities should be able to update that information. Similarly, personal information of the employees should be accessible only to the authorities concerned and not to everyone. Another example can be the medical records of patients in a hospital. These should be accessible only to health care officials.

Thus, one of the concepts of database security is primarily a specification of access rules about who has what type of access to what information. This is also known as the problem of Authorisation. These access rules are defined at the time database is defined. The person who writes access rules is called the authoriser. The process of ensuring that information and other protected objects are accessed only in authorised ways is called access control. There may be other forms of security relating to physical, operating system, communication aspects of databases. However, in this unit, we will confine ourselves mainly to authorisation and access control using simple commands.

The term integrity is also applied to data and to the mechanism that helps to ensure its consistency. Integrity refers to the avoidance of accidental loss of consistency.
Protection of database contents from unauthorised access includes legal and ethical issues, organization policies as well as database management policies. To protect database several levels of security measures are maintained:

1) **Physical:** The site or sites containing the computer system must be physically secured against illegal entry of unauthorised persons.

2) **Human**: An Authorisation is given to a user to reduce the chance of any information leakage and unwanted manipulations.

3) **Operating System:** Even though foolproof security measures are taken to secure database systems, weakness in the operating system security may serve as a means of unauthorised access to the database.

4) **Network:** Since databases allow distributed or remote access through terminals or network, software level security within the network software is an important issue.

5) **Database system:** The data items in a database need a fine level of access control. For example, a user may only be allowed to read a data item and is allowed to issue queries but would not be allowed to deliberately modify the data. It is the responsibility of the database system to ensure that these access restrictions are not violated. Creating database views as discussed in Unit 1 Section 1.6.1 of this block is a very useful mechanism of ensuring database security.

To ensure database security requires implementation of security at all the levels as

above. The Database Administrator (DBA) is responsible for implementing the database security policies in a database system. The organisation or data owners create these policies. DBA creates or cancels the user accounts assigning appropriate security rights to user accounts including power of granting and revoking certain privileges further to other users.

11.4.1 Relationship between Security and Integrity

Database security usually refers to access, whereas database integrity refers to avoidance of accidental loss of consistency. But generally, the turning point or the dividing line between security and integrity is not always clear. *Figure 7* shows the relationship between data security and integrity.

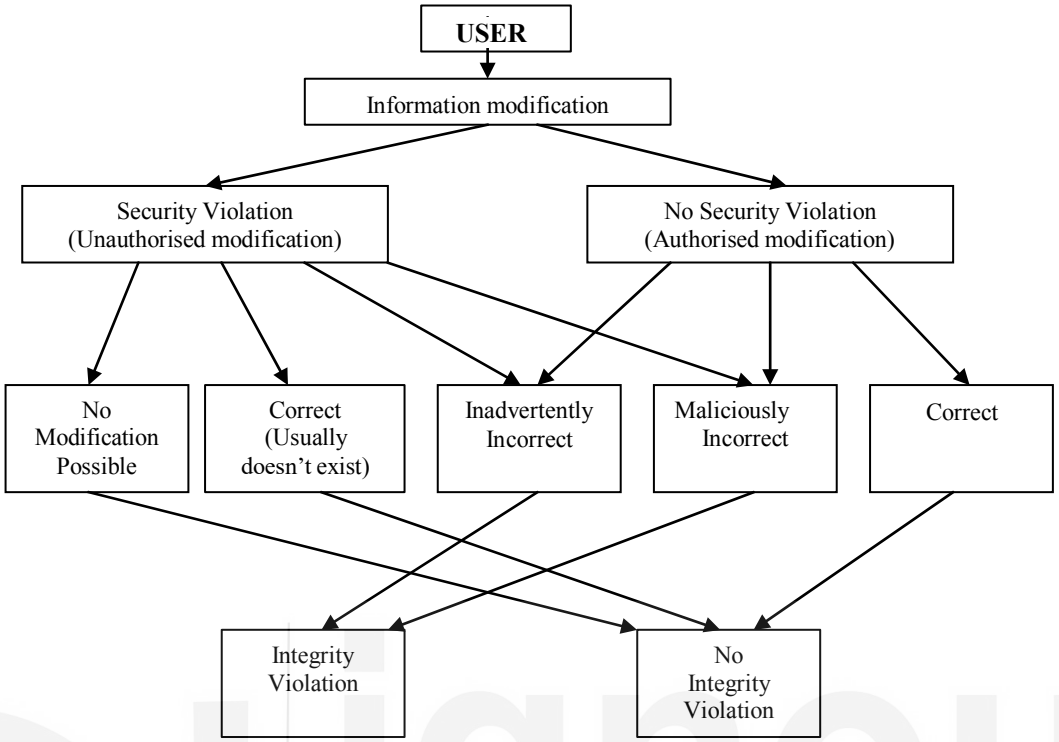


Figure 7: Relationship between security and Integrity

11.4.2 Difference between Operating System and Database Security

Security within the operating system can be implemented at several levels ranging from passwords for access to system, to the isolation of concurrent executing processes with the system. However, there are a few differences between security measures taken at operating system level as compared to those that of database system. These are:

- Database system protects more objects, as the data is persistent in nature. Also database security is concerned with different levels of granularity such as file, tuple, an attribute value or an index. Operating system security is primarily concerned with the management and use of resources.
- Database system objects can be complex logical structures such as views, a number of which can map to the same physical data objects. Moreover different architectural levels viz. internal, conceptual and external levels, have different security requirements. Thus, database security is concerned with the semantics – meaning of data, as well as with its physical representation. Operating system can provide security by not allowing any operation to be performed on the database unless the user is authorized for the operation concerned.

Figure 8 shows the architecture of a database security subsystem that can be found in any commercial DBMS.

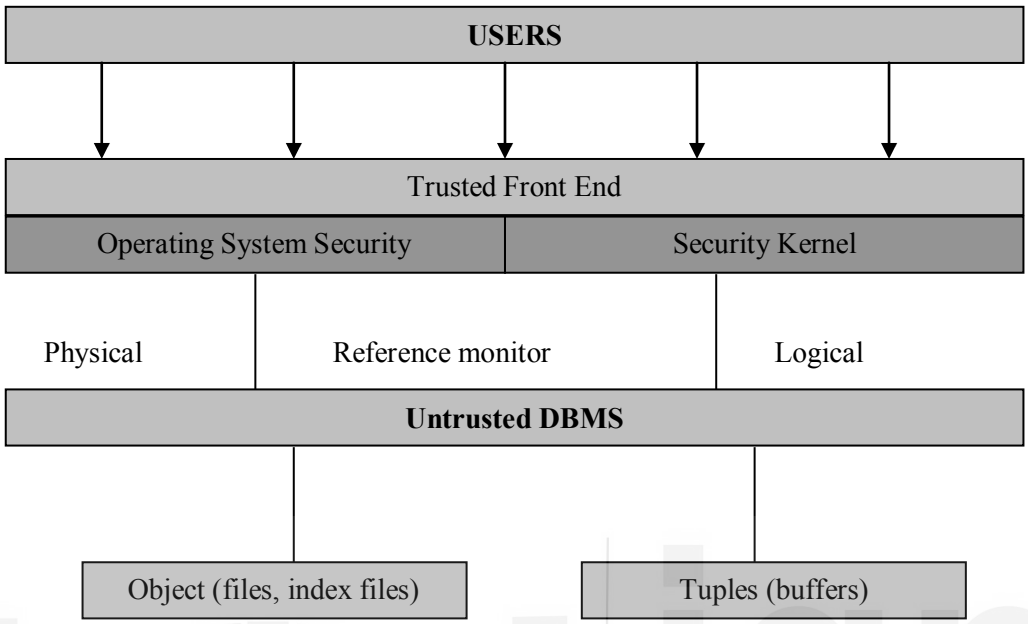


Figure 8: Database Security subsystem

11.5 AUTHORISATION

Authorisation is the culmination of the administrative policies of the organisation. As the name specifies, authorisation is a set of rules that can be used to determine which user has what type of access to which portion of the database. The following forms of authorisation are permitted on database items:

1) **READ**: it allows reading of data object, but not modification, deletion or insertion of data object.
2) **INSERT**: allows insertion of new data, but not the modification of existing data,e.g., insertion of tuple in a relation.
3) **UPDATE**: allows modification of data, but not its deletion. But data items likeprimary-key attributes may not be modified.
4) **DELETE**: allows deletion of data only.

A user may be assigned all, none or a combination of these types of Authorisation, which are broadly called access authorisations.

In addition to these manipulation operations, a user may be granted control operations like

1) Add: allows adding new objects such as new relations.

2) Drop: allows the deletion of relations in a database.

3) Alter: allows addition of new attributes in a relations or deletion of existing

attributes from the database.

4) Propagate Access Control: this is an additional right that allows a user to propagate the access control or access right which s/he already has to some other, i.e., if user A has access right R over a relation S, then if s/he has propagate access control, s/he can propagate her/his access right R over relation S to another user B either fully or part of it. In SQL you can use WITH GRANT OPTION for this right.

You must refer to Section 1.5 of Unit 1 of this block for the SQL commands relating to data and user control.

The ultimate form of authority is given to the database administrator. He is the one who may authorize new users, restructure the database and so on. The process of Authorisation involves supplying information known only to the person the user has claimed to be in the identification procedure.

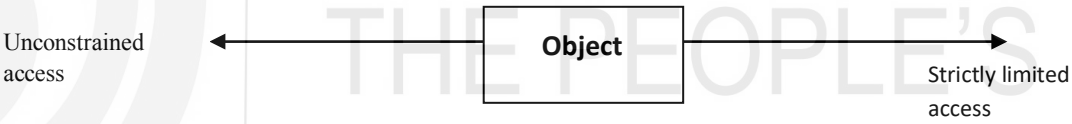
A basic model of Database Access Control

Models of database access control have grown out of earlier work on protection in operating systems. Let us discuss one simple model with the help of the following example:

Security problem: Consider the relation:

Employee (Empno, Name, Address, Deptno, Salary, Assessment)

Assuming there are two users: Personnel manager and general user . What access rights may be granted to each user? One extreme possibility is to grant an unconstrained access or to have a limited access.



One of the most influential protection models was developed by Lampson and extended by Graham and Denning. This model has 3 components:

1) A set of objects: where objects are those entities to which access must be controlled.

2) A set of subjects: where subjects are entities that request access to objects.

3) A set of all access rules: which can be thought of as forming an access (often referred to as authorisation matrix).

Let us create a sample authorisation matrix for the given relation:

Object / Subject	Empno	Name	Address	Deptno	Salary	Assessment
Personnel Manager	Read	All	All	All	All	All
General User	Read	Read	Read	Read	Not accessible	Not accessible

As the above matrix shows, Personnel Manager and general user are the two subjects. Objects of database are Empno, Name, Address, Deptno, Salary and Assessment. As per the access matrix, Personnel Manager can perform any operation on the database of an employee except for updating the Empno that may be self-generated and once given can never be changed. The general user can only read the data but cannot update, delete or insert the data into the database. Also the information about the salary and assessment of the employee is not accessible to the general user.

In summary, it can be said that the basic access matrix is the representation of basic access rules. These rules may be implemented using a view on which various access rights may be given to the users.

Check Your Progress 2

1) What are the different types of data manipulation operations and control operations?

……

……..……

……..……..……

……..……..……..……

2) What is the main difference between data security and data integrity?

……

……..……

……..……..……

……..……..……..……

3) What are the various aspects of security problem?

……

……..……

……..……

……..……..……

4) Name the 3 main components of Database Access Control Model?

……

……..……

……..……

……..……..……

11.6 SUMMARY

In this unit we have discussed the recovery of the data contained in a database system after failures of various types. The types of failures that the computer system is likely to be subject to include that of components or subsystems, software failures, power

outages, accidents, unforeseen situations, and natural or man-made disasters.
Database recovery techniques are methods of making the database fault tolerant. The aim of the recovery scheme is to allow database operations to be resumed after a failure with the minimum loss of information and at an economically justifiable cost.

The basic technique to implement database recovery is by using data redundancy in the form of logs, and archival copies of the database. Checkpoint helps the process of recovery.

Security and integrity concepts are crucial since modifications in a database require the replacement of the old values. The DBMS security mechanism restricts users to only those pieces of data that are required for the functions they perform. Security mechanisms restrict the type of actions that these users can perform on the data that is accessible to them. The data must be protected from accidental or intentional (malicious) corruption or destruction. In addition, there is a privacy dimension to data security and integrity.

Security constraints guard against accidental or malicious tampering with data; integrity constraints ensure that any properly authorized access, alteration, deletion, or insertion of the data in the database does not change the consistency and validity of the data. Database integrity involves the correctness of data and this correctness has to be preserved in the presence of concurrent operations, error in the user's operation and application programs, and failures in hardware and software.

11.7 SOLUTIONS/ANSWERS

Check Your Progress 1

1) Recovery is needed to take care of the failures that may be due to software, hardware and external causes. The aim of the recovery scheme is to allow database operations to be resumed after a failure with the minimum loss of information and at an economically justifiable cost. One of the common techniques is log-based recovery. A transaction is the basic unit of recovery.

2) A checkpoint is a point when all the database updates and logs are written to stable storage. A checkpoint ensures that not all the transactions need to be REDONE or UNDONE. Thus, it helps in faster recovery from failure. You should create a sample example, for checkpoint having a sample transaction log.

3) The following properties should be taken into consideration:

- Loss of data should be minimal
- Recovery should be quick
- Recovery should be automatic
- Affect small portion of database.

Check Your Progress 2

1) Data manipulation operations are:

- Read

- Insert
- Delete
- Update

Data control operations are:

- Add
- Drop
- Alter
- Propagate access control

2) Data security is the protection of information that is maintained in database against unauthorised access, modification or destruction. Data integrity is the mechanism that is applied to ensure that data in the database is correct and consistent.

3)

- Legal, social and ethical aspects
- Physical controls
- Policy questions
- Operational problems
- Hardware control
- Operating system security
- Database administration concerns

4) The three components are:

- Objects
- Subjects
- Access rights

UNIT 12 QUERY PROCESSING AND EVALUATION

(Adopted from MCS-043 Block-2 Unit-1)

Structure **Page Nos.**

12.0 INTRODUCTION

The Query Language – SQL is one of the main reasons of success of RDBMS. A user just needs to specify the query in SQL that is close to the English language and does not need to say how such query is to be evaluated. However, a query needs to be evaluated efficiently by the DBMS. But how is a query-evaluated efficiently? This unit attempts to answer this question. The unit covers the basic principles of query evaluation, the cost of query evaluation, the evaluation of join queries, etc. in detail.
It also provides information about query evaluation plans and the role of storage in query evaluation and optimisation. This unit thus, introduces you to the complexity of query evaluation in DBMS.

12.1 OBJECTIVES

After going through this unit, you should be able to:

- measure query cost;

- define algorithms for individual relational algebra operations;
- create and modify query expression;
- define evaluation plan choices, and
- define query processing using views.

12.2 QUERY PROCESSING: AN INTRODUCTION

Before defining the measures of query cost, let us begin by defining query processing.Let us take a look at the *Figure 1*.

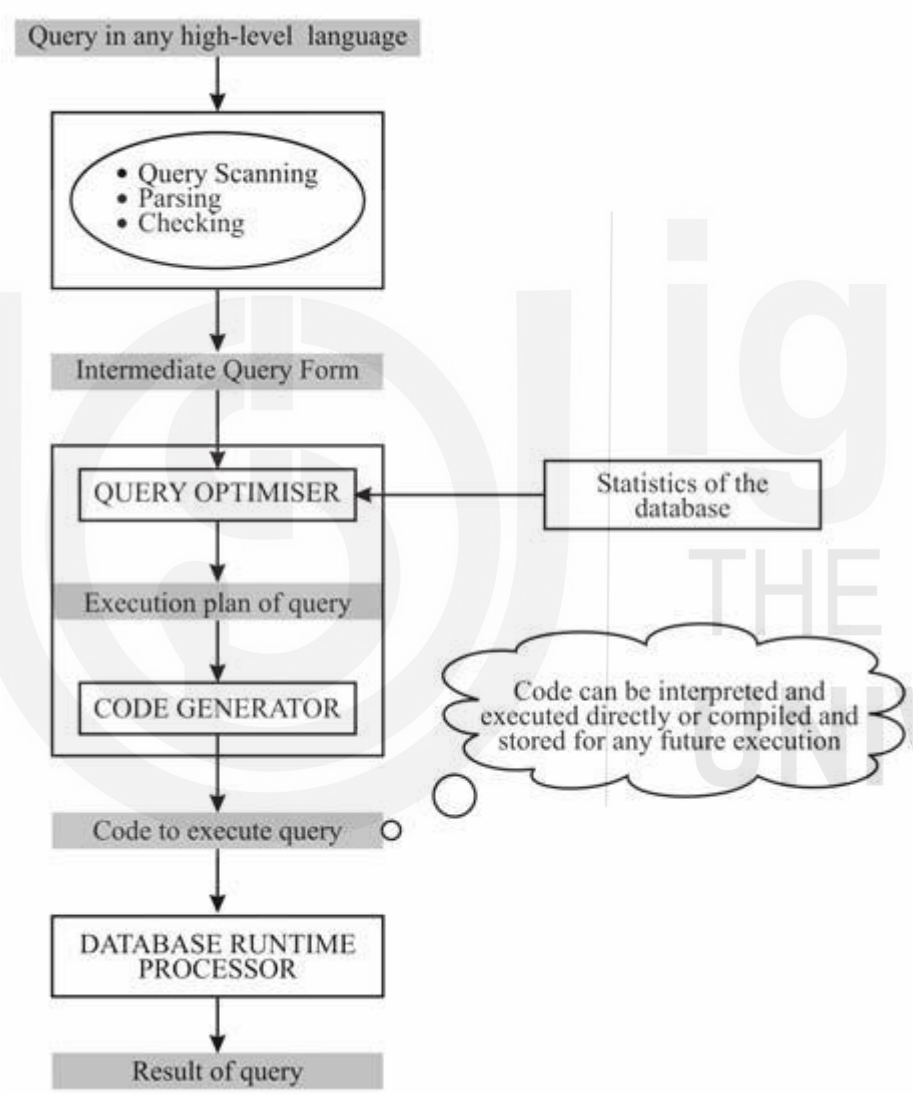


Figure 1: Query processing

In the first step Scanning, Parsing, and Validating is done to translate the query into its internal form. This is then further translated into relational algebra (an intermediate query form). Parser checks syntax and verifies relations. The query then is optimised with a query plan, which then is compiled into a code that can be executed by the database runtime processor.

We can define query evaluation as the query-execution engine taking a query-

evaluation plan, executing that plan, and returning the answers to the query. The query processing involves the study of the following concepts:

- how to measure query costs?
- algorithms for evaluating relational algebraic operations.
- how to evaluate a complete expression using algorithms on individual operations?

12.2.1 Optimisation

A relational algebra expression may have many equivalent expressions. For example, $\sigma_{(salary < 5000)}(\pi_{salary}(EMP))$ is equivalent to $\pi_{salary}(\sigma_{salary < 5000}(EMP))$.

Each relational algebraic operation can be evaluated using one of the several different algorithms. Correspondingly, a relational-algebraic expression can be evaluated in many ways.

An expression that specifies detailed evaluation strategy is known as evaluation-plan, for example, you can use an index on *salary* to find employees with *salary < 5000* or we can perform complete relation scan and discard employees with salary ≥ 5000. The basis of selection of any of the scheme will be the cost.

Query Optimisation: Amongst all equivalent plans choose the one with the lowest cost. Cost is estimated using statistical information from the database catalogue, for example, number of tuples in each relation, size of tuples, etc.

Thus, in query optimisation we find an evaluation plan with the lowest cost. The cost estimation is made on the basis of heuristic rules.

12.2.2 Measure of Query Cost

Cost is generally measured as total elapsed time for answering the query. There are many factors that contribute to time cost. These are *disk accesses*, CPU time, or even network *communication*.

Typically disk access is the predominant cost as disk transfer is a very slow event and is also relatively easy to estimate. It is measured by taking into account the following activities:

Number of seeks	×	average-seek-cost
Number of blocks read	×	average-block-read-cost
Number of blocks written	×	average-block-written-cost.

Please note that the cost for writing a block is higher than the cost for reading a block. This is due to the fact that the data is read back after being written to ensure that the write was successful. However, for the sake of simplicity we will just use *number of block transfers from disk as the cost measure.* We will also ignore the difference in cost between sequential and random I/O, CPU and communication costs. The I/O cost **depends** on the search criteria i.e., point/range query on an ordering/other fields and the file structures: heap, sorted, hashed. It is also dependent on the use of indices such as primary, clustering, secondary, B+ tree, multilevel, etc. There are other cost factors also, these may include buffering, disk placement, materialisation, overflow / free space management etc.

In the subsequent section, let us try to find the cost of various operations.

12.3 SELECT OPERATION

The selection operation can be performed in a number of ways. Let us discuss the algorithms and the related cost of performing selection operations.

File scan: These are the search algorithms that locate and retrieve records that fulfil a selection condition in a file. The following are the two basic files scan algorithms for selection operation:

1) *Linear search:* This algorithm scans each file block and tests all records to see whether they satisfy the selection condition.

The cost of this algorithm (in terms of block transfer) is defined as:

Cost of searching records satisfying a condition = Number of blocks in a database = N_b.

Cost for searching a key attribute value = Average number of block transfer for locating the value (on an average, half of the file needs to be traversed) so it is = $N_b/2$.

Linear search can be applied regardless of selection condition or ordering of records in the file, or availability of indices.

2) *Binary search:* It is applicable when the selection is an equality comparison on the attribute on which file is ordered. Assume that the blocks of a relation are stored continuously then, cost can be estimated as:

Cost = Cost of locating the first tuple by a binary search on the blocks + sequence of other blocks that continue to satisfy the condition.

$$= \lceil log_2 (N_b) \rceil + \frac{\text{average number of tuples with the same value}}{\text{Blocking factor (Number of tuples in a block) of the relation}}$$

These two values may be calculated from the statistics of the database.

Index scan: Search algorithms that use an index are restricted because the selection condition must be on the search-key of the index.

3) (a) *Primary index-scan for equality:* This search retrieves a single record that satisfies the corresponding equality condition. The cost here can be calculated as:

Cost = Height traversed in index to locate the block pointer +1(block of the primary key is transferred for access).

(b) *Hash key:* It retrieves a single record in a direct way thus, cost in hash key may also be considered as Block transfer needed for finding hash target +1

4) *Primary index-scan for comparison:* Assuming that the relation is sorted on the attribute(s) that are being compared, (< , > etc.), then we need to locate the

first record satisfying the condition after which the records are scanned forward or backward as the condition may be, displaying all the records. Thus cost in this case would be:

Cost = Number of block transfer to locate the value in index + Transferring all the blocks of data satisfying that condition.

Please note we can calculate roughly (from the cost point of view) the number of blocks satisfying the condition as:

Number of values that satisfy the condition × average number of tuples per attribute value/blocking factor of the relation.

5) *Equality on clustering index* to retrieve multiple records: The cost calculations in this case are somewhat similar to that of algorithm (4).

6) (a) *Equality on search-key of secondary index:* Retrieves a single record if the search-key is a candidate key.

Cost = cost of accessing index + 1.

It retrieves multiple records if search-key is not a candidate key.

Cost = *cost of accessing index + number of records retrieved* (It can be veryexpensive).

Each record may be on a different block, thus, requiring one block access for each retrieved record (this is the worst case cost).

(b) *Secondary index, comparison:* For the queries of the type that use comparison on secondary index value ≥ a value, then the index can be used to find first index entry which is greater than that value, scan index sequentially from there till the end and also keep finding the pointers to records.

For the ≤ *type query* just scan leaf pages of index, also keep finding pointers to records, till first entry is found satisfying the condition.

In either case, retrieving records that are pointed to, may require an I/O for each record. Please note linear file scans may be cheaper if many records are to be fetched.

Implementation of Complex Selections

Conjunction: Conjunction is basically a set of AND conditions.

7) *Conjunctive selection using one index:* In such a case, select any algorithm given earlier on one or more conditions. If possible, test other conditions on these tuples after fetching them into memory buffer.

8) *Conjunctive selection using multiple-key index:* Use appropriate composite (multiple-key) index if they are available.

9) *Conjunctive selection by intersection of identifiers* requires indices with record pointers. Use corresponding index for each condition, and take the intersection

of all the obtained sets of record pointers. Then fetch records from file if, some conditions do not have appropriate indices, test them after fetching the tuple from the memory buffer.

Disjunction: Specifies a set of OR conditions.

10) *Disjunctive selection by union of identifiers* is applicable if *all* conditions have available indices, otherwise use linear scan. Use corresponding index for each condition, take the union of all the obtained sets of record pointers, and eliminate duplicates, then fetch data from the file.

Negation: Use linear scan on file. However, if very few records are available in the result and an index is applicable on attribute, which is being negated, then find the satisfying records using index and fetch them from the file.

12.4 SORTING

Now we need to take a look at sorting techniques that can be used for calculating costing. There are various methods that can be used in the following ways:

1) Use an existing applicable ordered index (e.g., B+ tree) to read the relation in sorted order.

2) Build an index on the relation, and then use the index to read the relation in sorted order. (Options 1&2 may lead to one block access per tuple).

3) For relations that fit in the memory, techniques like quicksort can be used.

4) For relations that do not fit in the memory, *external sort-merge* is a good choice.

Let us go through the algorithm for External Sort-Merge.

i) **Create Sorted Partitions:**

Let *i* be 0 initially.

Repeat steps (a) to (d) until the end of the relation:

(a) Read *M* blocks of relation into the memory. (Assumption M is the numberof available buffers for the algorithm).

(b) Sort these buffered blocks using internal sorting.

(c) Write sorted data to a temporary file – temp (i)

(d) *i = i + 1;*

Let the final value of *i* be denoted by *N;*

Please note that each temporary file is a sorted partition.

ii) **Merging Partitions (N-way merge):**

// We assume (for now) that $N < M$.
// Use N blocks of memory to buffer temporary files and 1 block to buffer output.

Read the first block of each temporary file (partition) into its input buffer block;

Repeat steps (a) to (e) until all input buffer blocks are empty;

(a) Select the first record (in sort order) among all input buffer blocks;

(b) Write the record to the output buffer block;

(c) **If** the output buffer block is full then write it to disk and empty it for the next set of data. This step may be performed automatically by the OperatingSystem;

(d) Delete the record from its input buffer block;

(e) **If** the buffer block becomes empty **then** read the next block (if any) of thetemporary file into the buffer.

If $N \geq M$, several merge *passes* are required, in each pass, contiguous groups of $M - 1$ partitions are merged and a pass reduces the number of temporary files temp (i) by a factor of $M - 1$. For example, if M=11 and there are 90 temporary files, one pass reduces the number of temporary files to 9, each temporary file begin 10 times the size of the earlier partitions.

Repeated passes are performed till all partitions have been merged into one.

Figure 2 shows an example for external sorting using sort-merge.

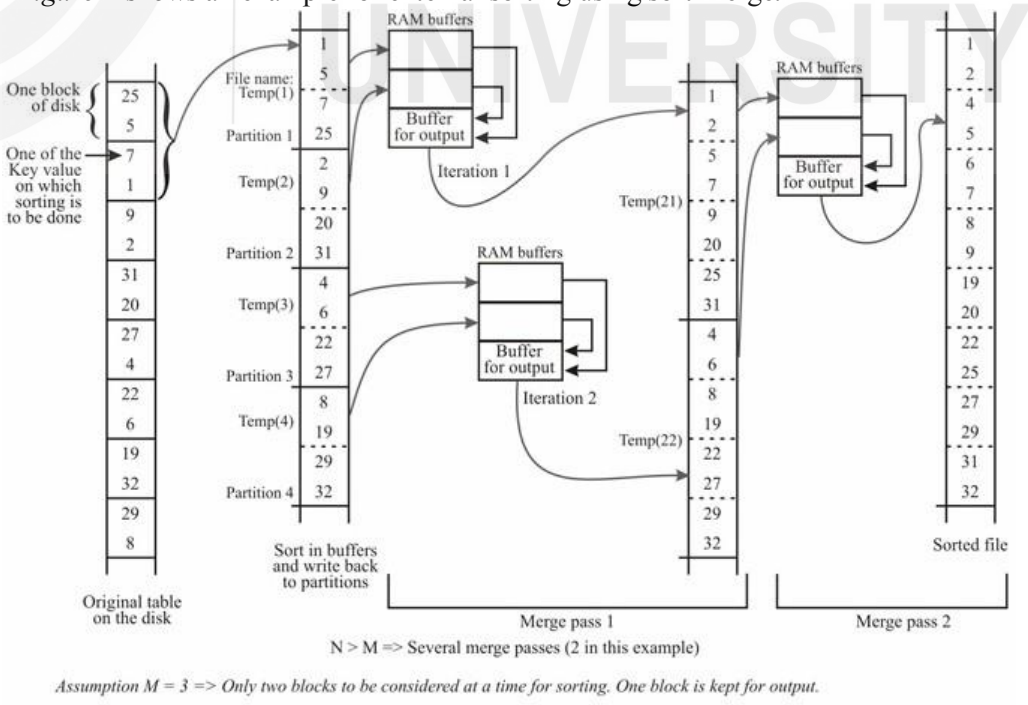


Figure 2: External merge-sort example

Cost Analysis:

Cost Analysis is may be performed, according to the following activities:

- Assume the file has a total of Z blocks.
- Z block input to buffers + Z block output – for temporary file creation.
- Assuming that N ≥ M, then a number of merge passes are required
- Number of merge passes = $\lceil \log_{M-1} (Z/M) \rceil$. Please note that of M buffers 1 is used for output.
- So number of block transfers needed for merge passes $=2\times Z (\lceil \log_{M-1} (Z/M) \rceil)$ as all the blocks will be read and written back of the buffer for each merge pass.
- Thus, the total number of passes for sort-merge algorithm = $2Z + 2Z (\lceil \log_{M-1} (Z/M) \rceil) = 2Z \times (\lceil \log_{M-1} (Z/M) \rceil + 1)$.

☞ Check Your Progress 1

1) What are the basic steps in query processing?

……

……

……

……

2) How can the cost of query be measured?

……

……

……

……

3) What are the various methods adopted in select operation?

……

……

……

……

4) Define External-Sort-Merge algorithm.

……

……

……

……

12.5 JOIN OPERATION

There are number of algorithms that can be used to implement joins:

- Nested-loop join
- Block nested-loop join
- Indexed nested-loop join
- Merge-join
- Hash-join
- Complex join.

The choice of join algorithm is based on the cost estimate. Let us use the following information to elaborate the same:

MARKS (enroll no, subject code, marks):10000 rows, 500 blocks

STUDENT (enroll no, name, dob): 2000 rows, 100 blocks

12.5.1 Nested-Loop Join

Let us show the algorithm for the given relations.

To compute the theta or equi-join

```
for each tuple s in STUDENT
{
        for each tuple m in MARKS
        {
                test s.enrolno = m.enrolno to see if they satisfy the
                joincondition if they do, output joined tuple to the
                result.
        };
};
```

- In the nested loop join there is one *outer* relation and one *inner* relation.
- It does not use or require indices. It can be used with any kind of join condition.However, it is expensive as it examines every pair of tuples in the two relations.
- If there is only enough memory to hold one block of each relation, the number of disk accesses can be calculated as:

  For each tuple of STUDENT, all the MARKS tuples (blocks) that need to be accessed.

However, if both or one of the relations fit entirely in the memory, block transfer will be needed only once, so the total number of transfers in such a case, may be calculated as:

= Number of blocks of STUDENT + Number of blocks of MARKS
= 100 + 500 = 600.

If only the smaller of the two relations fits entirely in the memory then use that as the inner relation and the bound still holds.

Cost for the worst case:

Number of tuples of outer relation × Number of blocks of inner relation + Number of blocks of outer relation.

2000 ∗ 500 + 100 = 1,000,100 with STUDENT as outer relation.

There is one more possible bad case when MARKS is on outer loop and STUDENT in the inner loop. In this case, the number of Block transfer will be:

10000 ∗ 100+500 = 1,000,500 with MARKS as the outer relation.

12.5.2 Block Nested-Loop Join

This is a variant of nested-loop join in which a complete block of outer loop is joined with the block of inner loop.

The algorithm for this may be written has:

```
for each block s of  STUDENT
{
    for each block m of MARKS
    {
        for each tuple si in s
        {
            for each tuple mi in m
            {
                Check if (si and mi) satisfy the join condition
                        if they do output joined tuple to the result
            };
        };
    };
};
```

Worst case of block accesses in this case = Number of Blocks of outer relation (STUDENT) ×Number of blocks of inner relation (MARKS) + Number of blocks of outer relation (STUDENT).

Best case: Blocks of STUDENT + Blocks of MARKSNumber of block transfers

assuming worst case:

100*500 + 100 = 50,100 (much less than nested-loop join)

Number of block transfers assuming best case:

400 + 100 = 500 (same as with nested-loop join)

Improvements to Block Nested-Loop Algorithm

The following modifications improve the block Nested method:

Use $M - 2$ disk blocks as the blocking unit for the outer relation, where M = memory size in blocks.

Use one buffer block to buffer the inner relation.

Use one buffer block to buffer the output.

This method minimizes the number of iterations.

12.5.3 Indexed Nested-Loop Join

Index scans can replace file scans if the join is an equi-join or natural join, and an index is available on the inner relation's join attribute.

For each tuple *si* in the outer relation STUDENT, use the index to look up tuples in MARKS that satisfy the join condition with tuple *si.*

In a worst case scenarios, the buffer has space for only one page of STUDENT, and, for each tuple in MARKS, then we should perform an index lookup on *MARKS index.*

Worst case: Block transfer of STUDENT+ number of records in STUDENT * cost of searching through index and retrieving all matching tuples for each tuple of STUDENT.

If a supporting index does not exist than it can be constructed as and when needed.

If indices are available on the join attributes of both STUDENT and MARKS, then use the relation with fewer tuples as the outer relation.

Example of Index Nested-Loop Join Costs

Compute the cost for STUDENT and MARKS join, with STUDENT as the outer relation. Suppose MARKS has a primary B+-tree index on enroll no, which contains 10 entries in each index node. Since MARKS has 10,000 tuples, the height of the tree is 4, and one more access is needed to the actual data. The STUDENT has 2000 tuples. Thus, the cost of indexed nested loops join as:

100 + 2000 * 5 = 10,100 disk accesses

12.5.4 Merge-Join

The merge-join is applicable to equi-joins and natural joins only. It has the following process:

1) Sort both relations on their join attribute (if not already sorted).

2) Merge the sorted relations to join them. The join step is similar to the merge stage of the sort-merge algorithm, the only difference lies in the manner in which duplicate values in join attribute are treated, i.e., every pair with same value on join attribute must be matched.

STUDENT

Enrol no	Name	-----
1001	Ajay	……
1002	Aman	…...
1005	Rakesh	…...
1100	Raman	…...
……	…….	…...

MARKS

Enrol no	subject code	Marks
1001	MCS-011	55
1001	MCS-012	75
1002	MCS-013	90
1005	MCS-015	75
….....	…....	….....

Figure 3: Sample relations for computing join

The number of block accesses:

Each block needs to be read only once (assuming all tuples for any given value of the

join attributes fit in memory). Thus number of block accesses for merge-join is:

Blocks of STUDENT + Blocks of MARKS + the cost of sorting if relations areunsorted.

Hybrid Merge-Join

This is applicable only when the join is an equi-join or a natural join and one relation is sorted and the other has a secondary B+-tree index on the join attribute.

The algorithm is as follows:

Merge the sorted relation with the leaf entries of the B+-tree. Sort the result on the addresses of the unsorted relation's tuples. Scan the unsorted relation in physical address order and merge with the previous results, to replace addresses by the actual tuples. Sequential scan in such cases is more efficient than the random lookup method.

12.5.5 Hash-Join

This is applicable to both the equi-joins and natural joins. A hash function *h* is used to partition tuples of both relations, where *h* maps joining attribute (enroll no in our example) values to {0, 1, ..., *n-1*}.

The join attribute is hashed to the join-hash partitions. In the example of *Figure 4* we have used mod 100 function to hashing, and n = 100.

Error!

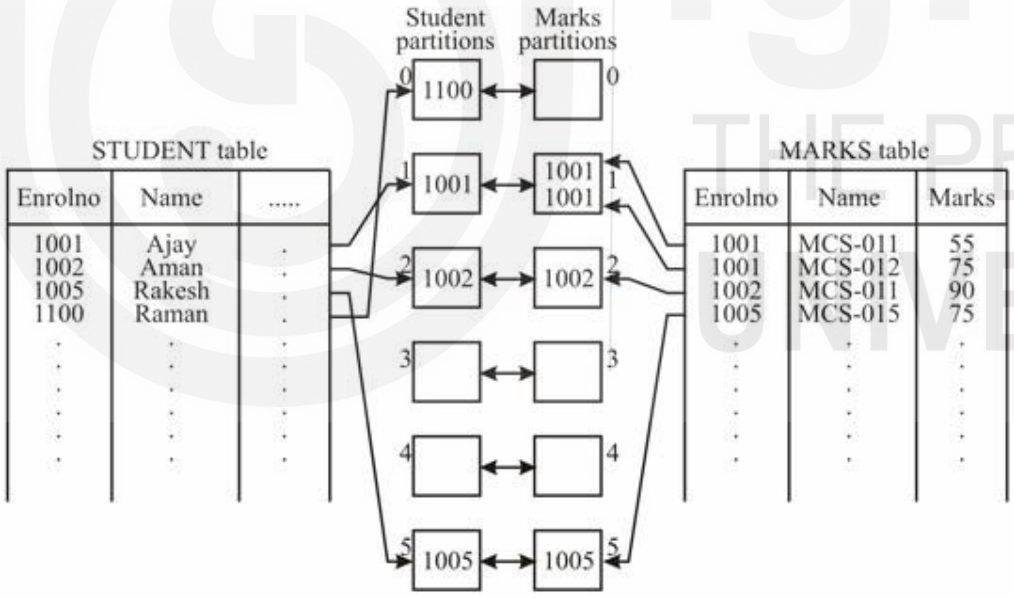


Figure 4: A hash-join example

Once the partition tables of STUDENT and MARKS are made on the enrolment number, then only the corresponding partitions will participate in the join as:

A STUDENT tuple and a MARKS tuple that satisfy the join condition will have the same value for the join attributes. Therefore, they will be hashed to equivalent partition and thus can be joined easily.

Algorithm for Hash-Join

The hash-join of two relations *r* and *s* is computed as follows:

- Partition the relation *r* and *s* using hashing function *h*. (When partitioning a relation, one block of memory is reserved as the output buffer for each partition).

- For each partition *si* of *s,* load the partition into memory and build an in-memory hash index on the join attribute.
- Read the tuples in *ri* from the disk, one block at a time. For each tuple in *ri* locate each matching tuple in *si* using the in-memory hash index and output theconcatenation of their attributes.

In this method, the relation *s* is called the *build* relation and *r* is called the *probe* relation. The value *n* (the number of partitions) and the hash function *h* is chosen in such a manner that each *si* should fit in to the memory. Typically *n* is chosen as Number of blocks of s/Number of memory buffers] *f (M) where f is a "fudge factor", typically around 1.2. The probe relation partitions *ri* need not fit in memory.

Average size of a partition *si* will be less than *M* blocks using the formula for *n* as above thereby allowing room for the index. If the build relation *s* is very huge, then the value of *n* as given by the above formula may be greater than *M– 1* i.e., the number of buckets is > the number of buffer pages. In such a case, the relation *s* can be recursively partitioned, instead of partitioning *n* ways, use *M* – 1 partitions for *s and* further partition the *M* – 1 partitions using a different hash function. You should use the same partitioning method on *r.* This method is rarely required, as recursive partitioning is not needed for relations of 1GB or less for a memory size of 2MB, with block size of 4KB.

Cost calculation for Simple Hash-Join

(i) Cost of partitioning *r* and *s*: all the blocks of *r* and *s* are read once and after partitioning written back, so cost 1 = 2 (blocks of *r* + blocks of *s*).

(ii) Cost of performing the hash-join using build and probe will require at least oneblock transfer for reading the partitions
Cost 2 = (blocks of *r* + blocks of *s*)

(iii) There are a few more blocks in the main memory that may be used for evaluation, they may be read or written back. We ignore this cost as it will be too less in comparison to cost 1 and cost 2.
Thus, the total cost = cost 1 + cost 2
= 3 (blocks of *r* + blocks of *s*)

Cost of Hash-Join requiring recursive partitioning:

(i) The cost of partitioning in this case will increase to number of recursion required, it may be calculated as:

Number of iterations required = ([$\log_{M-1}$ (blocks of s] –1)

Thus, cost 1 will be modified as:

= 2 (blocks of r + blocks of s) × ([$\log_{M-1}$ (blocks of s)] −1)

(ii) The cost for step (ii) and (iii) here will be the same as that given in steps and (iii) above.

Thus, total cost = 2*(*blocks of r + blocks of s*)* ([log_{M-1}(blocks of s) – 1]) + (blocks of r + blocks of s*)*.

Because s is in the inner term in this expression, it is advisable to choose the smaller relation as the build relation. If the entire build input can be kept in the main memory, n can be set to 1 and the algorithm need not partition the relations but may still build an in-memory index, in such cases the cost estimate goes down to (Number of blocks r + Number of blocks of s).

Handling of Overflows

Even if s is recursively partitioned *hash-table overflow* can occur, i.e., some partition si may not fit in the memory. This may happen if there are many tuples in s with the same value for join attributes or a bad hash function.

Partitioning is said to be *skewed* if some partitions have significantly more tuples than the others. This is the overflow condition. The overflow can be handled in a variety of ways:

Resolution (during the build phase): The overflow partition s is further partitioned using different hash function. The equivalent partition of r must be further partitioned similarily.

Avoidance (during build phase): Partition build relations into many partitions, then combines them.

However, such approaches for handling overflows fail with large numbers of duplicates. One option of avoiding such problems is to use block nested-loop join on the overflowed partitions.

Let us explain the hash join and its cost for the natural join STUDENT ⋈ MARKS
Assume a memory size of 25 blocks ⇒ M=25;

SELECT build s as STUDENT as it has less number of blocks (100 blocks) and r probe as MARKS (500 blocks).

Number of partitions to be created for STUDENT = (block of STUDENT/M)* fudge factor (1.2) = (100/25) × 1.2 = 4.8

Thus, STUDENT relation will be partitioned into 5 partitions of 20 blocks each. MARKS will also have 5 partitions of 100 blocks each. The 25 buffers will be used as –20 blocks for one complete partition of STUDENT plus 4 more blocks for one block of each of the other 4 partitions. One block will be used for input of MARKS partitions.

The total cost = 3(100+500) = 1800 as no recursive partitioning is needed.

Hybrid Hash-Join

This is useful when the size of the memory is relatively large, and the build input is larger than the memory. Hybrid hash join keeps the first partition of the build relation in the memory. The first partition of STUDENT is maintained in the first 20 blocks of the buffer, *and not written to the disk*. The first block of MARKS is used right away

for probing, instead of being written and read back. Thus, it has a cost of
3(80 + 400) + 20 +100 = 1560 block transfers for hybrid hash-join, instead of 1800 with plain hash-join.

Hybrid hash-join is most useful if *M* is large, such that we can have bigger partitions.

12.5.6 Complex Joins

A join with a conjunctive condition can be handled, either by using the nested loop or block nested loop join, or alternatively, the result of one of the simpler joins (on a few conditions) can be computed and the final result may be evaluated by finding the tuples that satisfy the remaining conditions in the result.

A join with a disjunctive condition can be calculated either by using the nested loop or block nested loop join, or it may be computed as the union of the records in individual joins.

12.6 OTHER OPERATIONS

There are many other operations that are performed in database systems. Let us introduce these processes in this section.

Duplicate Elimination: Duplicate elimination may be implemented by using hashing or sorting. On sorting, duplicates will be adjacent to each other thus, may be identified and deleted. An optimised method for duplicate elimination can be deletion of duplicates during generation as well as at intermediate merge steps in external sort-merge. Hashing is similar – duplicates will be clubbed together in the same bucket and therefore may be eliminated easily.

Projection: It may be implemented by performing the projection process on each tuple, followed by duplicate elimination.

Aggregate Function Execution: Aggregate functions can be implemented in a manner similar to duplicate elimination. Sorting or hashing can be used to bring tuples in the same group together, and then aggregate functions can be applied to each group. An optimised solution could be to combine tuples in the same group during part time generation and intermediate merges, by computing partial aggregate values. For count, min, max, sum, you may club aggregate values on tuples found so far in the group. When combining partial aggregates for counting, you would need to add up the aggregates. For calculating the average, take the sum of the aggregates and the count/number of aggregates, and then divide the sum with the count at the end.

Set operations ($\cup$, $\cap$ and —) can either use a variant of merge-join after sorting, or a variant of hash-join.

Hashing:

1) Partition both relations using the same hash function, thereby creating *r0, .., rn – 1* and *s0,.., sn – 1*

2) Process each partition *i* as follows: Using a different hashing function, build anin-memory hash index on *ri* after it is brought into the memory.

   $r \cup s$: Add tuples in *si* to the hash index if they are not already in it. At the end of *si* add the tuples in the hash index to the result.

   $r \cap s$: Output tuples in *si* to the result if they are already there in the hash index.

r – s: For each tuple in *si,* if it is there in the hash index, delete it from the index.

At end of *si* add remaining tuples in the hash index to the result.

There are many other operations as well. You may wish to refer to them in further readings.

☞ Check Your Progress 2

1) Define the algorithm for Block Nested-Loop Join for the worst-case scenario.

.....

.....

.....

2) Define Hybrid Merge-Join.

.....

.....

.....

4) Give the method for calculating the cost of Hash-Join?

.....

.....

.....

5) Define other operations?

.....

.....

.....

12.7 REPRESENTATION AND EVALUATION OF QUERY EXPRESSIONS

Before we discuss the evaluation of a query expression, let us briefly explain how a SQL query may be represented. Consider the following student and marks relations:

STUDENT (enrolno, name, phone)

MARKS (enrolno, subjectcode, grade)

To find the results of the student(s) whose phone number is '1129250025', the following query may be given.

```
SELECT enrolno, name, subjectcode, grade
FROM STUDENT s, MARKS m
WEHRE s.enrolno=m.enrolno AND phone= '1129250025'
```

The equivalent relational algebraic query for this will be:

$\pi_{enrolno, name, subjectcode}$ ($\sigma_{phone='1129250025'}$ (STUDENT) $\bowtie$ MARKS)

This is a very good internal representation however, it may be a good idea to represent the relational algebraic expression to a query tree on which algorithms for query optimisation can be designed easily. In a query tree, nodes are the operators and relations represent the leaf. The query tree for the relational expression above would be:

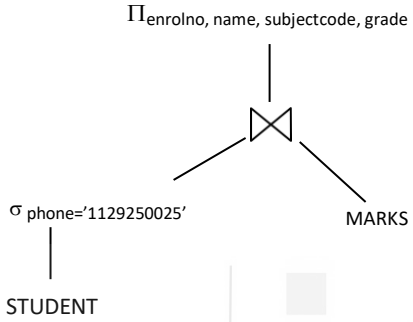


Figure 5: A Sample query tree

In the previous section we have seen the algorithms for individual operations. Now let us look at the methods for evaluating an entire expression. In general we use two methods:

- Materialisation
- Pipelining.

Materialisation: Evaluate a relational algebraic expression from the bottom-up, by explicitly generating and storing the results of each operation in the expression. For example, in *Figure 5* compute and store the result of the selection operation on STUDENT relation, then take the join of this result with MARKS relation and then finally compile the projection operation.

Materialised evaluation is always possible though; the cost of writing/reading results to/from disk can be quite high.

Pipelining

Evaluate operations in a multi-threaded manner, (i.e., passes tuples output from one operation to the next parent operation as input) even as the first operation is being executed. In the previous expression tree, it does not store (materialise) results instead, it passes tuples directly to the join. Similarly, does not store results of join, and passes tuples directly to the projection. Thus, there is no need to store a temporary relation on a disk for each operation. Pipelining may not always be possible or easy for sort, hash-join.

One of the pipelining execution methods may involve a buffer filled by the result tuples of lower level operation while, records may be picked up from the buffer by the higher level operation.

Complex Joins

When an expression involves three relations then we have more than one strategy for the evaluation of the expression. For example, join of relations such as:

STUDENT ⋈ MARKS ⋈ SUBJECTS
may involve the following three strategies:

Strategy 1: Compute STUDENT ⋈ MARKS*; use* result to computeresult ⋈ SUBJECTS

Strategy 2: Compute MARKS ⋈ SUBJECTS first, and then join the resultwith STUDENT

Strategy 3: Perform the pair of joins at the same time. This can be done by building an index of enroll no in STUDENT and on subject code in SUBJECTS.

For each tuple *m* in MARKS, look up the corresponding tuples in STUDENT and the corresponding tuples in SUBJECTS. Each tuple of MARKS will be examined only once.

Strategy 3 combines two operations into one special-purpose operation that may be more efficient than implementing the joins of two relations.

12.8 CREATION OF QUERY EVALUATION PLANS

We have already discussed query representation and its final evaluation in earlier section of this unit, but can something be done during these two stages that optimises the query evaluation? This section deals with this process in detail.

Generation of query-evaluation plans for an expression involves several steps:

1) Generating logically equivalent expressions using **equivalence rules**
2) Annotating resultant expressions to get alternative query plans
3) Choosing the cheapest plan based on **estimated cost.**

The overall process is called **cost based optimisation.**

The cost difference between a good and a bad method of evaluating a query would be enormous. We would therefore, need to estimate the cost of operations and statistical information about relations. For example a number of tuples, a number of distinct values for an attribute etc. Statistics helps in estimating intermediate results to compute the cost of complex expressions.

Let us discuss all the steps in query-evaluation plan development in more details next.

12.8.1 Transformation of Relational Expressions

Two relational algebraic expressions are said to be **equivalent** if on every legal database instance the two expressions generate the same set of tuples (order of tuples is irrelevant).

Let us define certain equivalence rules that may be used to generate equivalent relational expressions.

Equivalence Rules

1) The conjunctive selection operations can be equated to a sequence of individualselections. It can be represented as:

$$\sigma_{\theta_1 \wedge \theta_2}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$$

2) The selection operations are commutative, that is,

$$\sigma_{\theta_1}(\sigma_{\theta_2}(E)) = \sigma_{\theta_2}(\sigma_{\theta_1}(E))$$

3) Only the last of the sequence of projection operations is needed, the others can be omitted.

$$\pi_{attriblist1}(\pi_{attriblist2}(\pi_{attriblist3} \ldots..(E)\ ..) = \pi_{attriblist1}(E)$$

4) The selection operations can be combined with Cartesian products and theta join operations.

$\sigma_{\theta 1}(E_1 \times E_2) = E_1 \bowtie_{\theta 1} E_2$

and

$\sigma_{\theta 2}(E_1 \bowtie_{\theta 1} E_2) = E_1 \bowtie_{\theta 2 \wedge \theta 1} E_2$

5) The theta-join operations and natural joins are commutative.

$E1 \bowtie_{\theta} E2 = E2 \bowtie_{\theta} E1$

6) The Natural join operations are associative. Theta joins are also associative but with the proper distribution of joining condition:

$(E1 \bowtie E2) \bowtie E3 = E1 \bowtie (E2 \bowtie E3)$

7) The selection operation distributes over the theta join operation, under conditions when all the attributes in selection predicate involve only the attributes of one of the expressions ($E1$) being joined.

$\sigma_{\theta 1}(E1 \bowtie_{\theta} E2) = (\sigma_{\theta 1}(E1)) \bowtie_{\theta} E2)$

8) The projections operation distributes over the theta join operation with only those attributes, which are present in that relation.

$\pi_{attrlist1 \cup attrlist2}(E1 \bowtie_{\theta} E2) = (\pi_{attrlist1}(E1) \bowtie_{\theta} \pi_{attrlist2}(E2))$

9) The set operations of union and intersection are commutative. But set differenceis not commutative.

$E1 \cup E2 = E2 \cup E$ and similarly for the intersection.

10) Set union and intersection operations are also associative.

$(E1 \cup E2) \cup E3 = E1 \cup (E2 \cup E3)$ and similarly for intersection.

11) The selection operation can be distributed over the union, intersection, and set-differences operations.

$\sigma_{\theta 1}(E1 - E2) = ((\sigma_{\theta 1}(E1) - (\sigma_{\theta 1}(E2))$

12) The projection operation can be distributed over the union.

$\pi_{attriblist1}(E1 \cup E2) = \pi_{attriblist1}(E1) \cup \pi_{attriblist1}(E2)$

The rules as above are too general and a few heuristics rules may be generated from

these rules, which help in modifying the relational expression in a better way. These rules are:

(1) Combining a cascade of selections into a conjunction and testing all the predicates on the tuples at the same time:

$\sigma_{\theta 2}(\sigma_{\theta 1}(E))$ convert to $\sigma_{\theta 2 \wedge \theta 1}(E)$

(2) Combining a cascade of projections into single outer projection:

$\pi_4(\pi_3(\ldots\ldots(E))) = \pi_4(E)$

(3) Commutating the selection and projection or vice-versa sometimes reduces cost

(4) Using associative or commutative rules for Cartesian product or joining to findvarious alternatives.

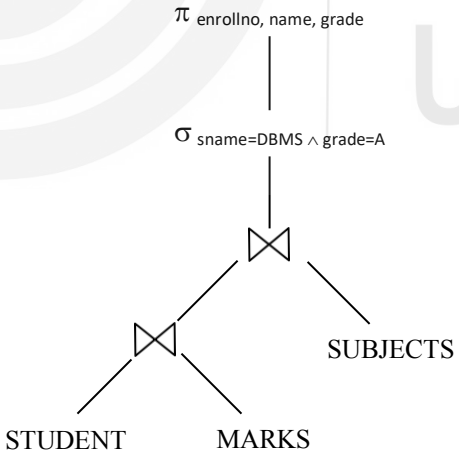
(5) Moving the selection and projection (it may have to be modified) before joins. The selection and projection results in the reduction of the number of tuples andtherefore may reduce cost of joining.

(6) Commuting the projection and selection with Cartesian product or union.

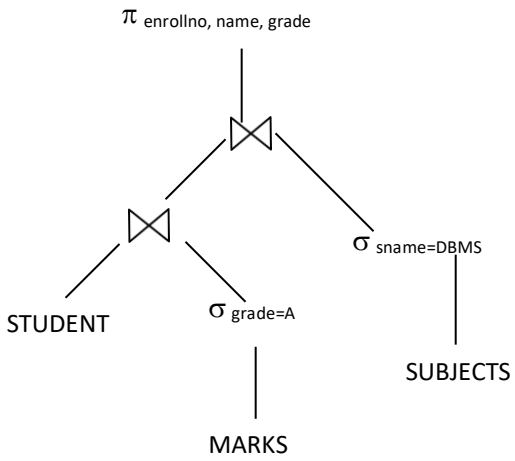
Let us explain use of some of these rules with the help of an example. Consider the query for the relations:

STUDENT (enrollno, name, phone)
MARKS (enrollno, subjectcode, grade)
SUBJECTS (subjectcode, sname)

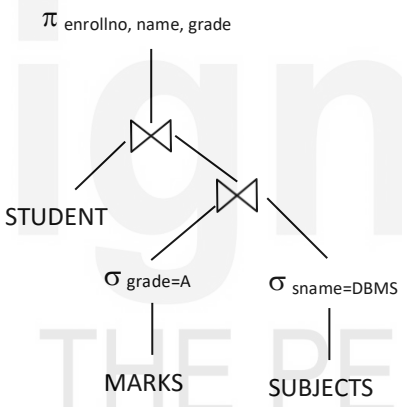
Consider the query: Find the enrolment number, name, and grade of those students who have secured an A grade in the subject DBMS. One of the possible solutions to this query may be:



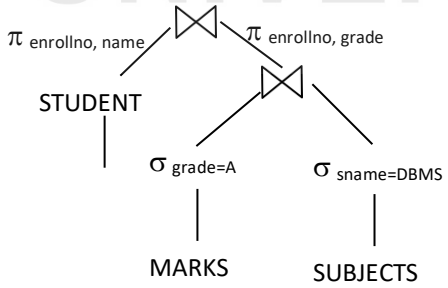
The selection condition may be moved to the join operation. The selection condition given in the *Figure* above is: *sname = DBMS and grade = A*. Both of these conditions belong to different tables, as *s name* is available only in the SUBJECTS table and *grade* in the MARKS table. Thus, the conditions of selection will be mapped accordingly as shown in the *Figure* below. Thus, the equivalent expression will be:



The expected size of SUBJECTS and MARKS after selection will be small so it may be a good idea to first join MARKS with SUBJECTS. Hence, the associative law of JOIN may be applied.



Finally projection may be moved inside. Thus the resulting query tree may be:



Please note the movement of the projections.

Obtaining Alternative Query Expressions

Query optimisers use equivalence rules to systematically generate expressions equivalent to the given expression. Conceptually, they generate all equivalent expressions by repeatedly executing the equivalence rules until no more expressions are to be found. For each expression found, use all applicable equivalence rules and

add newly generated expressions to the set of expressions already found. However, the approach above is very expensive both in time space requirements. The heuristics rules given above may be used to reduce cost and to create a few possible but good equivalent query expression.

12.8.2 Query Evaluation Plans

Let us first define the term Evaluation Plan.

An evaluation plan defines exactly which algorithm is to be used for each operation, and how the execution of the operation is coordinated. For example, *Figure 6* shows the query tree with evaluation plan.

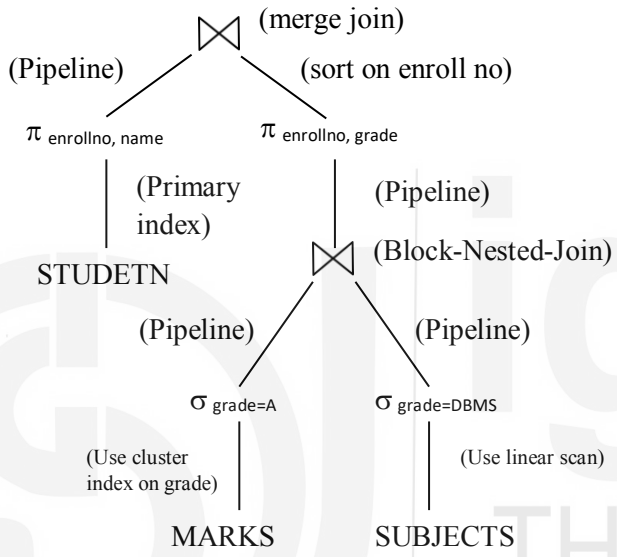


Figure 6: Query evaluation plan

12.8.3 Choice of Evaluation Plans

For choosing an evaluation technique, we must consider the interaction of evaluation techniques. Please note that choosing the cheapest algorithm for each operation independently may not yield best overall algorithm. For example, merge-join may be costlier than hash-join, but may provide a sorted output which reduces the cost for a higher level aggregation. A nested-loop join may provide opportunity for pipelining. Practical query optimisers incorporate elements of the following two broad approaches:

1) Searches all the plans and chooses the best plan in a cost-based fashion.
2) Uses heuristic rules to choose a plan.

12.8.4 Cost Based Optimisation

Cost based optimisation is performed on the basis of the cost of various individual operations that are to be performed as per the query evaluation plan. The cost is calculated as we have explained in the earlier section with respect to the method and operation (JOIN, SELECT, etc.).

12.8.5 Storage and Query Optimisation

Cost calculations are primarily based on disk access, thus, storage has an important

role to play in cost. In addition, some of the operations also require intermediate storage thus; cost is further enhanced in such cases. The cost of finding an optimal query plan is usually more than offset by savings at query-execution time, particularly by reducing the number of slow disk accesses.

12.9 VIEWS AND QUERY PROCESSING

A view must be prepared, passing its parameters, which describe the query and the manner in which tuples should be evaluated. This takes the form of a pre-evaluation window, which gives the user of the program the ability to trade off memory usage for faster navigation, or an attempt to balance both of these resources.

The view may maintain the complete set of tuples following evaluation. This requires a lot of memory space, therefore, it may be a good idea to partially pre-evaluate it.
This is done by hinting at how the number of that should be present in the evaluated view tuples before and after the current tuple. However, as the current tuple changes, further evaluation changes, thus, such scheme is harder to plan.

12.9.1 Materialised View

A materialised view is a view whose contents are computed and stored.

Materialising the view would be very useful if the result of a view is required frequently as it saves the effort of finding multiple tuples and totalling them up.

Further the task of keeping a materialised view up-to-date with the underlying data is known as materialised view maintenance. Materialised views can be maintained by recomputation on every update. A better option is to use incremental view maintenance, that is, where only the affected part of the view is modified. View maintenance can be done manually by defining triggers on insert, delete, and update of each relation in the view definition. It can also be written manually to update the view whenever database relations are updated or supported directly by the database.

12.9.2 Materialised Views and Query Optimisation

We can perform query optimisation by rewriting queries to use materialised views. For example, assume that a materialised view of the join of two tables *b* and *c* is available as:

$$a = b \ \ NATURAL\ JOIN \ \ c$$

Any query that uses natural join on b and c can use this materialised view '*a*' as:

Consider you are evaluating a query:

$$z = r\ NATURAL\ JOIN \ \ b \ \ NATURAL\ JOIN \ \ c$$

Then this query would be rewritten using the materialised view 'a' as:

$$z = r \ \ NATURAL\ JOIN \ \ a$$

Do we need to perform materialisation? It depends on cost estimates for the two alternatives viz., use of a materialised view by view definition, or simple evaluation.

Query optimiser should be extended to consider all the alternatives of view evaluation

and choose the best overall plan. This decision must be made on the basis of the system workload. Indices in such decision-making may be considered as specialised views. Some database systems provide tools to help the database administrator with index and materialised view selection.

☞ Check Your Progress 3

1) Define methods used for evaluation of expressions?

……

……

……

……

2) How you define cost based optimisation?

……

……

……

……

2) How you define evaluation plan?

……

……

……

……

12.10 SUMMARY

In this unit we have discussed query processing and evaluation. A query in a DBMS is a very important operation, as it needs to be efficient. Query processing involves query parsing, representing query in alternative forms, finding the best plan of evaluation of a query and then actually evaluating it. The major query evaluation cost is the disk access time. In this unit, we have discussed the cost of different operations in details. However, an overall query cost will not be a simple addition of all such costs.

12.11 SOLUTIONS/ANSWERS

Check Your Progress 1

1) (a) In the first step scanning, parsing & validating is done
(b) Translation into relational algebra
(c) Parser checks syntax, verifies relations
(d) Query execution engine take a query evaluation plan, executes it andreturns answer to the query.

2) Query cost is measured by taking into account following activities:

* Number of seeks * Number of blocks * Number of block written

For simplicity we just use number of block transfers from disk as the costmeasure. During this activity we generally ignore the difference in cost between sequential and random I/O and CPU/communication cost.

3) Selection operation can be performed in a number of ways such as:

	File Scan	**Index Scan**
1.	Linear Search	1. (a) Primary index, equality (b) Hash Key
2.	Binary Search	2. Primary index, Comparison 3. Equality on clustering index 4. (a) Equality on search key of secondary index (b) Secondary index comparison

4) Algorithm for External Sort-Merge

1. Create sorted partitions
   (a) Read M blocks of relation into memory
   (b) Write sorted data to partition R_i
2. Merge the partitions (N-way Merge) until all input buffer blocks areempty.

Check Your Progress 2

1)
```
For each block Br of r {
    For each block Bs of s {
        For each tuple ti in Br {
            For each tuple ti in Bs {
            Test pair(ti,si) to see if they satisfy the join condition
             If they do, add the joined tuple to result
                };
            };
        };
    };
```

2) Hybrid Merge-Join is applicable when the join is equi-join or natural join and one relation is sorted.

Merge the sorted relation with leaf of B+tree.

(i) Sort the result on address of sorted relations tuples.
(ii) Scan unsorted relation and merge with previous result.

3) Cost of Hash-join.

(i) If recursive partitioning not required
3(Blocks of r +blocks of s)

(ii) If recursive partitioning required then
2(blocks of r + blocks of s $(\lceil \log_{m-1}(\text{blocks of s}) \rceil - 1)$ + blocks of r+blocks of s

4) Other operations

(a) Duplicate elimination
(b) Projection
(c) Aggregate functions.

Check Your Progress 3

1) Methods used for evaluation of expressions:

   (a) Materialisation
   (b) Pipelining

2) Cost based optimisation

   (a) Generalising logically equivalent expressions using equivalence rules
   (b) Annotating resultant expressions to get alternative query plans.
   (c) Choosing the cheapest plan based on estimated cost.

3) Evaluation plan defines exactly what algorithms are to be used for each operation and the manner in which the operations are coordinated.