UNIT 10 TRANSACTIONS AND CONCURRENCY MANAGEMENT

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 data amongst 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 this unit describes the term transaction. Concurrent execution of user programs is essential for better performance of DBMS, as concurrent running of several user programs keeps utilising CPU time efficiently, since disk accesses are frequent and are relatively slow in case of DBMS. This unit not only explains the issues of concurrent transaction but also explains algorithms to control those problems.

10.1 OBJECTIVES

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

- define the term database transactions and their properties;
- describe the issues of concurrent transactions;
- explain the mechanism to prevent issues arising due to concurrently executing transactions;
- describe the principles of locking and serialisability; and
- describe concepts of deadlock and its prevention.

10.2 THE TRANSACTIONS

A transaction is a unit of data processing. Database systems that deal with a large number of concurrent transactions are also called Transaction Processing Systems. A

transaction is a unit of work in a database system. For example, a database system for a bank may allow the transactions such as – withdrawal of money from an account; deposit of money to an account; transfer of money from A's account to B's account. A transaction would involve the manipulation of one or more data values in a database. Thus, it may require reading and writing of database values. The following are examples of transactions.

Example 1: A money withdrawal transaction of a bank can be written in pseudo-code as:
; Assume that you are doing this transaction for account number X.

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

Another similar example may be the transfer of money from account number X to account number Y. This transaction may be written as:

Example 2: Transaction to transfers some amount from account X to account Y.

```
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 ("INSUFFICIENT BALANCE IN X")
                  ROLLBACK
      ELSE DISPLAY ("ACCOUNT X OR Y DOES NOT EXIST")
End transaction
```

Please note the following:

The two keywords here `COMMIT` and `ROLLBACK` are:

`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.

In general, databases are stored in secondary storage as data blocks, whereas they can be processed in the main memory. The portion of main memory allotted for database processing is called a buffer. When a transaction desires to update a value X, a transaction program performs the following operations:

Read the block containing value X to memory buffer

Process the value of X in the memory buffer

Write the value of X to the data block

A block of data on secondary storage may contain many data items, as its size may be in MBs. Further, a transaction processing system may modify several data items simultaneously, therefore, an interesting question for a DBMS is: when to write a data block back to secondary storage? This process is called Buffer management. You may refer to further readings for details on this topic.

Transactions have certain desirable properties. Let us look into the properties of a transaction.

10.2.1 Properties of a Transaction

A transaction has four basic properties. These are:

- **A**tomicity
- **C**onsistency
- **I**solation
- **D**urability

These are also called the ACID properties of transactions.

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*. For example, in the transaction of Example 2, the transaction is reading and writing more than one data items. The atomicity property requires either operations on both the data item to be performed or not at all.

Consistency: This property ensures that 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, i.e., either to the database state that was before the start of the transaction or the database state after the end of the transaction.

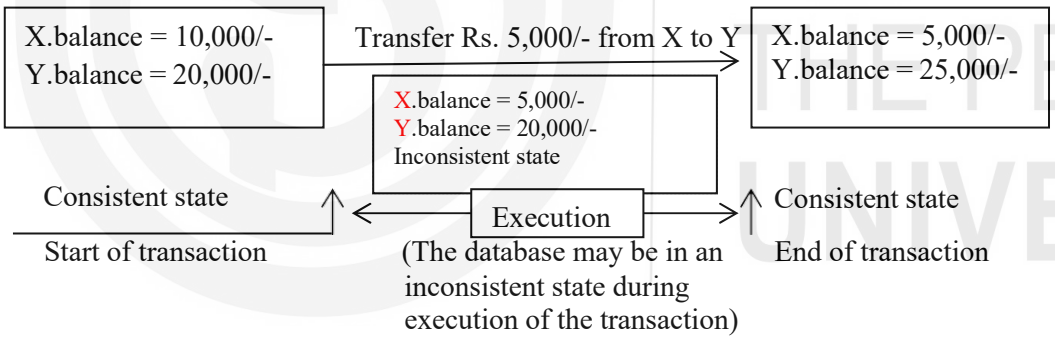


Figure 1: A Transaction execution

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 a transaction does not affect the outcome of another transaction. For example, if another transaction that is a withdrawal transaction which withdraws an amount of Rs. 5000 from X account is in progress, then failure or success of the transaction of example 2, should not affect the outcome of this transaction. Only the state of the data that has been read by the transaction should determine the outcome of this transaction.

Durability: This property necessitates that once a transaction has been 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.

10.2.2 States of a Transaction

A transaction can be in any one of the states during its execution. These states are displayed in *Figure 2*.

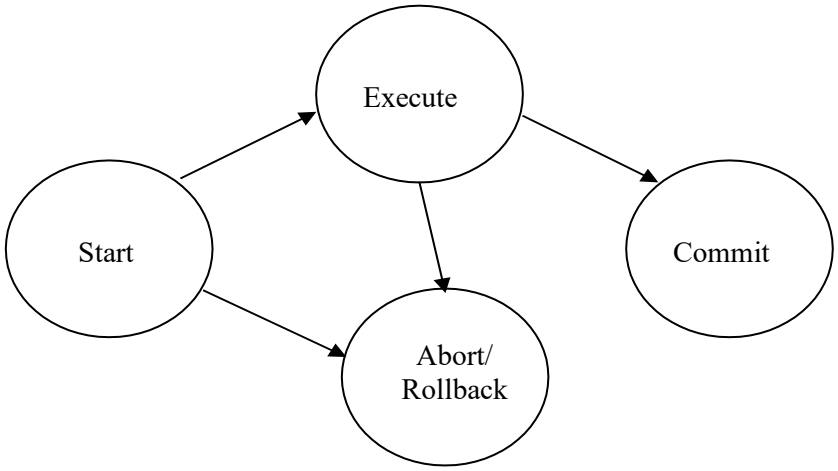


Figure 2: States of transaction execution

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

10.3 THE CONCURRENT TRANSACTIONS

Almost all commercial DBMSs support multi-user environment, 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., a transaction of one user does not affect the transactions of other users or even the other transactions issued by the same user. Please note that concurrency related problems 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 the same data item and no updates are performed on these values, then they do NOT cause any concurrency related problem. Now, let us first discuss why you need a mechanism to control concurrent transactions. This is explained next.

10.3.1 Transaction Schedule

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. S/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.

Consider that a database system has *n* active transactions, namely *T1, T2, ..., Tn*. Each of these transactions can be represented using a transaction program consisting of operations, as shown in Example 1 and Example 2. A schedule, say S, is defined as the sequential ordering of the operations of the '*n*' interleaved transactions, where the sequence or order of operations of an individual transaction is maintained.

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

(a) The two transactions

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

(b) One possible conflicting schedule for interleaved execution of TA and TB

Figure 3: A conflicting schedule

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.

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

a) Complete execution of T2 is 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
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**

b) Complete execution of T1 is before T2 starts, then sum X+Y will still show the correct 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 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

c) Part execution of transaction T1, followed by the complete execution of T2, followed by the remaining part of T1.

Figure 4: Three different possibilities of interleaved execution of T1 and T2

In this execution, an incorrect value is displayed. This is because Rs.100, although removed from X, has not been put in Y and is thus missing from the computation of T2. Please note that for the given transaction there are many more ways of interleaved instruction execution.

Thus, there can be a possibility of anomalies when the transactions T1 and T2 are allowed to execute in parallel. Let us define the anomalies of concurrent execution of transactions more precisely.

10.3.2 Anomalies of Concurrent Transactions

Let us assume the following transactions (assuming there will not be errors in datawhile 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

The possible problems in the concurrent execution of these transactions are:

1. **Lost Update Anomaly:** 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 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 partof some transactions T5 and T6; each of which increases the value of X by 1000.

T5	T6	Value of x originally 2000	
		T5	T6
Update X		3000	
	Update X		4000
ROLLBACK		2000	

Here, transactions T5 and T6 update the same item X (please note that an Update include Reading, Modifying and Write operations). Thereafter, T5 decides to undo its action and rolls back, causing the value of X to go back to the original value 2000. In this case, the update performed by T6 had got lost and a lost update is said to have occurred.

2. **Unrepeatable read Anomaly:** Suppose transaction 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, inconsistent values are read, and the results of the transaction may be in

error.

3. **Dirty Read Anomaly:** T10 reads a value which has been updated by T9. This update has not 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.

Please note that all three problems that we have discussed so far are primarily due to the violation of the Isolation property of the concurrently executing transactions that use the same data items.

In addition to the three anomalies, sometimes concurrent transactions may result in inconsistent analysis. For example, in the schedule of the two transactions T1 and T2, as shown in Figure 4(c), you may observe that one part of transaction T1 is executed prior to the execution of transaction T2 and the other part after the completion of Transaction T2. You may also note that transaction T2 produces an incorrect sum of the balance of accounts X and Y. The cause of the problem is that T2 transaction has accessed data items, which are still being modified by another transaction. The important thing to note is that the modifying transaction has been able to modify only some of the data values and some data values are yet to be modified. This resulted in a situation where the analysis transaction has read some modified values and some unmodified values, resulting in an inconsistent output.

As you can observe from the problems discussed in this section, concurrency-related problems occur when multiple transactions contend for a data item concurrently. But how do we ensure that the execution of two or more transactions does not result in concurrency-related problems?

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 are also 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 one data value? Can a transaction write a value without reading it? Give an example of a transaction.

……

……

……

2) What are the anomalies of concurrent transactions? Can these problems occur in transactions 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. A locked data item is available for the exclusive use of the transaction that has locked it. Other transactions are lockedout of that data item. When a transaction that has locked a data item does not desire to use it anymore, 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 manages the locking and unlocking of data items 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 it in more detail in the next section.

10.4.1 Serialisable Schedule

If the operations of two transactions conflict with each other, how to determine that no concurrency-related problems have occurred in the transaction execution? For this purpose, let us define the term – Schedule, Serial Schedule, interleaved schedule and Serializable Schedule.

A schedule can be defined as a sequence of actions/operations of one or more transactions, as explained in section 10.3.1. Figure 5 shows two schedules, viz. Schedule A and Schedule B.

A serial schedule is one in which the actions/operations of one transaction are performed at a time. This is followed by the actions/operations of the next transaction and so on. For example, Schedule A and Schedule B of Figure 5 are serial schedules, as in no schedule operations of transactions T1 and Transaction T2 interleave with each other.

An interleaved schedule allows the interleaving of actions/operations of different transactions. For example, the schedule in Figure 6 is an interleaved schedule. The basic question here is: How will you find out that a given interleaved schedule has not resulted in concurrency-related problems?

Schedule A: T2 followed by T1			**Schedule B: T1 followed by T2**		
Schedule A	**T1**	**T2**	**Schedule B**	**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 5: 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 6 (a): An Interleaved Schedule

Schedule D: A non-conflict equivalent schedule of Schedule C		
Schedule	**T1**	**T2**
READ X	READ X	
SUBTRACT 100	SUBTRACT 100	
WRITE X	WRITE X	
READ X		READ X
READ Y		READ Y
READ Y	READ Y	
ADD 100	ADD 100	
DISPLAY X+Y		DISPLAY X+Y
WRITE Y	WRITE Y	

Figure 6 (b): An Interleaved Schedule

You may observe that the Schedule C in Figure 6 is an interleaved schedule. It has conflicting operations, like reading and writing X and Y. There can be many interleaved schedules of transactions T1 and T2. How do you identify which two schedules are equivalent? Well, for that, let us define the term Conflict Equivalence.

Conflict Equivalence: Two schedules are defined as conflict equivalent if any two conflicting operations in the two schedules are in the same order. For example, Schedule D, as given below, is not a conflict equivalent schedule of Schedule C, as the sequence of WRITE X by T1 and READ X by T2 are not in the same order in the two interleaved schedules.

Conflict Serialisability or Serialisability

A schedule is called conflict serialisable if it is conflict equivalent to a serial schedule. In other words, for a given interleaved schedule, if the sequence of read and write is in the same order as that of any one of the serial schedules, then the interleaved schedule is called a conflict serialisable or serialsable schedule. In case an interleaved schedule is not serailsable, then it may result in problems due to concurrent execution of transactions.

Any schedule that produces the same results as a serial schedule is called a serialisable schedule. The basis of serialisability is taken from the notion of a serial schedule. Considering there are two concurrent transactions, T1 and T2, how many different serial schedules are possible for these two transactions? The possible serial schedules for two transactions, T1 and T2, are:

T1 followed by T2 OR T2 followed by T1.

Likewise, the number of possible serial schedules with three concurrent transactions would be defined by the number of possible permutations, which are:

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

But how can a schedule be determined to be serialisable or not? Is there any algorithmic way of determining whether a schedule is serialisable or not?

Using the notion of precedence graph, an algorithm can be devised to determine whether an interleaved schedule is serialisable 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. The basic principle for determining a serialisable schedule is that the precedence graph, which determines the sequences of occurrence of transactions, does not contain a cycle. Given a precedence graph with no cycles, then it must be equivalent to a serial schedule. 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 explain the algorithm with the help of the following example.

Example:

Let us use this algorithm to check whether the schedule given in Figure 6(a) is Serialisable. *Figure 7* 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 6*, 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 6* is **serialisable**. The equivalent serial schedule (as per step 3) would be T2 followed by T1, which is serial Schedule–A of Figure 5.

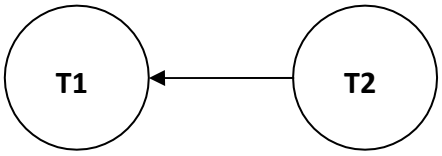


Figure 7: Test of Serialisability for the Schedule of Figure 6(a)

Please note that the schedule given in Figure 6(b) 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 T1 to T2.
- T2 reads Y 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:

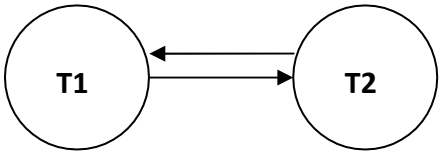


Figure 8: Test of Serialisability for the Schedule (c) of section 2.3

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

10.4.2 Locks

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

Types of Locks

There are two basic types of locks:

- Binary lock: This locking mechanism has two states for 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 and schedule given in Figure 4 (c) for this purpose; however, we will add 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 9: An incorrect locking implementation

Does the locking as done above solve the problem of concurrent transactions? No, the same problems remain. Try working with the old values given in figure 4(c). Thus, locking should be done with some logic 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 BYT2)	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	
	LOCK X request of T2 on X will be granted and transaction T2 resumes.	
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 10: A correct but restrictive locking implementation.

As shown in Figure 10, when you obtain all the locks at the beginning of the transaction and release them at the end, it 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 must 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 higher level of concurrency can be achieved compared to the binary locking scheme. The properties of shared and exclusive locks are summarised below:

a) Shared lock

- It is requested by a transaction that wants to just read the value of the 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.

We explain these locks with the help of an example. However, you may refer to further readings for more information on multiple-mode locking. 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 Shared 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 Shared mode. Thus, T1waits till both T2 and T11 will release the Shared 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 11: Example of Locking in multiple modes

As shown in Figure 11, locking can result in a serialisable schedule. Next, we discuss the concept of granularity of locking.

Granularity of Locking

In general, locks may be allowed on the following database items:

1) The complete database itself
2) A file of the database
3) A page or disk block of data
4) A record in a table
5) A data item of an attribute

Granularity of locking depends on the size of database item being locked. A coarse granularity locking means locking a larger data item, e.g. the complete database or file or page etc. The fine granularity locking is locking the database items of the smaller size, e.g. the record locking or the data item locking.

A lock can be implemented as a binary variable, which requires space. In addition, a lock requires locking and unlocking operations. Thus, coarse locking has low locking overheads, but it restricts the concurrency among transactions. For example, if a single transaction locks the complete database in an exclusive mode, all other transactions will wait for the completion of this transaction. On the other hand, the fine granularity of locking results in high locking overheads.

To make an efficient concurrent transaction processing system, multiple granularity locking is supported by a database system. These system support Intention mode locking. You may refer to the further readings for more details on such locking scheme.

In the next section, we discuss the locking protocol that ensures correct execution of concurrent transactions.

10.4.3 Two-Phase Locking (2PL)

In this section we try to answer the question: Can you release locks a bit early and still have no concurrency related problem? Yes, we can do it if we use two-phase locking protocol. 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 transaction T wants to modify an object, it needsto 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, any legal schedule of transactions that follows two-phase locking protocol is guaranteed to be serialisable. The two-phase locking protocol has been proven for its 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) Conservative 2PL
(2) Strict 2PL

The conservative 2PL allows the release of the lock at any time after all the locks have been acquired. For example, you can release the locks in the schedule of *Figure 10*, after you 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 conservative 2PL is shown graphically in *Figure 12.*

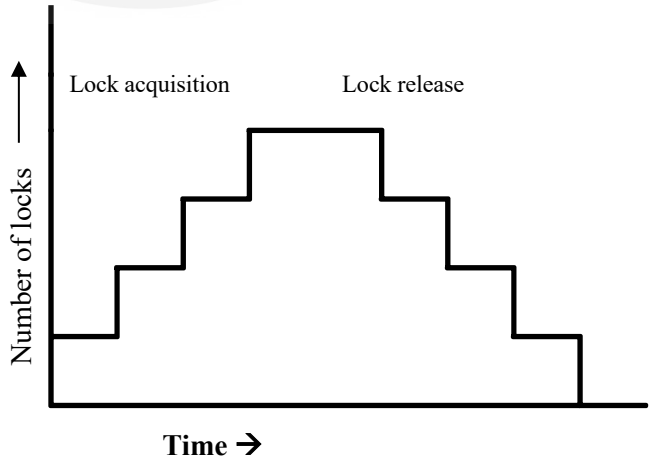


Figure 12: Conservative Two-Phase Locking

However, conservative 2PL suffers from the problem that it can result in loss of atomic or isolation property of transaction as theoretically speaking, once a lock is released on

a data item, it can be modified by another transaction before the first transaction commits or aborts.

To avoid such a situation, you use strict 2PL. The strict 2PL is graphically depicted in *Figure 13*. 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.

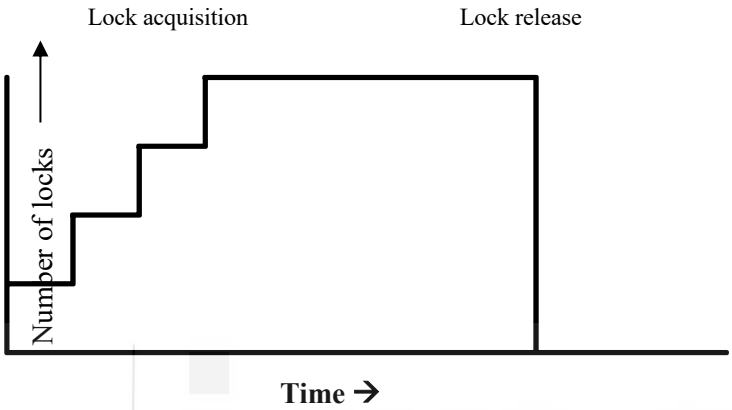


Figure 13: Strict Two-Phase Locking

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

☞ Check Your Progress 2

1) Let the transactions T1, T2, T3 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, and T3 are allowed to execute concurrently. If A has an initial value of zero, how many possible correct results are there? Enumerate them.

……

……

2) Consider the following two transactions on 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 HANDLING AND ITS PREVENTION

As seen earlier, though the 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 is unable to do so and 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 transactions are waiting for the other to unlock the locked resource (refer to figure 14).

Clearly, the schedule comes to a halt in its execution. The important thing to be seen here is that both TA and TB follow the 2PL, which guarantees serialisability. Whenever the situation, i.e. all the transactions are **waiting for a condition that will never occur,** arises, we say that a deadlock has occurred.

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 and a set of edges/arcs E. Each transaction is represented by node and an arc from Ti → Tj, if Tj holds a lock on data items that Ti is waiting for. 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 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 "wait-for" graph can be done. For example, the "wait-for" for the schedule of transactions TA and TB, as given earlier in this section, is shown in Figure 14.

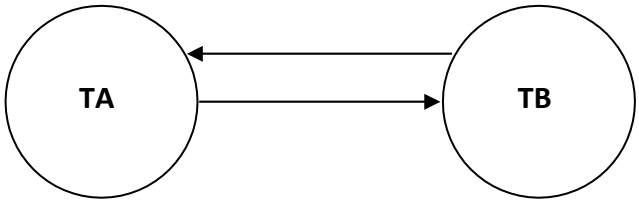


Figure 14: Wait For Graph of TA and TB

In Figure 14, 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 and causing a deadlock problem. The above case showsa direct cycle. However, in actual situations, more than two nodes may be there in a cycle.

A deadlock is a situation that can be created because of locks. It causes transactions to wait forever hence the name deadlock. A deadlock occurs becauseof 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 a database in a piecemeal way.
d) Circular waiting: Transactions lock part of the data resources needed and then wait indefinitely 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. You can refer to other methods from the further readings.

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. A deadlock prevention algorithm prevents a deadlock. It uses the approach: *do not allow circular wait*. 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 is based 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 T1 requests 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 that is currently held by transaction T2, then T3 is rolled back (die).

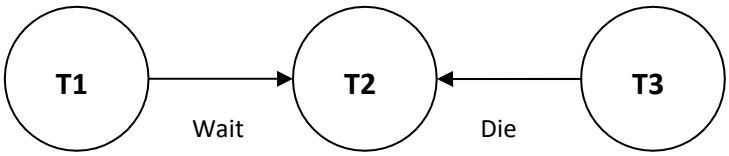


Figure 15: Wait-die Scheme of Deadlock Prevention.

"Wound-wait" scheme: It is based on a preemptive technique and is explained with the help of the following sequence of actions:

Resource Requesting Transaction	Resource is with the Transaction	Check Timestamp (TSP)	Action Needed
Ti	**Tj**	TSP (Ti) > TSP (Tj)	Ti is younger, therefore, can WAIT
		TSP (Ti) < TSP (Tj)	Ti is older, therefore, WOUND Ti

For example, consider three transactions T1, T2 and T3 with TSP(T1) < TSP(T2) < TSP(T3), i.e. T1 is the oldest and T3 is the youngest. For example, consider the following sequence of requests for a data item X:

T2 requests for X Action: Locks the Resource

T1 requests for X Action: Wound (Rollback) T1, as it is older.

T3 requests for X Action: T3 Waits for T2 to release X, as it is younger.

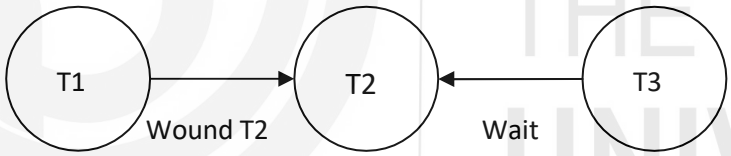


Figure 15: Wound-wait Scheme of Deadlock Prevention.

It is important to check that no transaction should get rolled back repeatedly such that it is never allowed to make progress. This is referred to starvation of a transaction. Also, both "wait-die" and "wound-wait" scheme should avoid starvation. The number of aborts and rollbacks will be higher in wait-die scheme than in the wound-wait scheme. But one major problem with both schemes is that these schemes may result in unnecessary rollbacks. You can refer to further readings for more details on deadlock-related schemes.

10.6 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.

In general, the concurrency-related problem does not occur frequently, therefore, an

optimistic concurrency control scheme allows database transactions to freely perform the update operations on data items. Further, in the validation phase a transaction cross-checks if any other transaction has modified the data items, which it had read or written. Therefore, the optimistic concurrency control algorithm has the following phases:

a) READ Phase: In this phase, a transaction makes a local copy of the data items needed by it in its own memory space. The transaction then makes changes in the local copies of data items.

b) VALIDATE Phase: In this phase, the transaction validates the values of data items read by the transaction during the READ phase. This validation ensures that no other transaction has changed the values of the data items, while this transaction was modifying the local copy of the data items. In case, the validation is unsuccessful, then this transaction is aborted, and the local updates of data items are discarded. However, in case of successful validation, the Write phase is performed.

c) WRITE Phase: This phase is performed if the Validate Phase is successful. In this phase, the transaction commits and all the data item updates made by the transaction in the local copies, are applied to the database.

To explain the optimistic concurrency control, the following terms are used:

- Write set (WS(T)): Given a transaction T, WS(T) is the set of data items that would be written by it.
- Read set (RS(T)): Given a transaction T, RS(T) is the set of data items that would be read by it.

More details on this scheme are available in further readings. But let us show this scheme here with the help of the following examples:Consider the set for transactions T1 and T2.

T1		**T2**	
Phase	**Operation**	**Phase**	**Operation**
-	-	Read	Reads the RS(T2), say variables X and Y and performs updating of local values
Read	Reads the RS(T1), say variable A and B and performs updating of local values	-	-
Validate	Validate the values of RS(T1)	-	-
-	-	Validate	Validate the values of RS(T2)
Write	Transaction Commits and writes the updated values WS(T1) in the database.	-	-
-	-	Write	Transaction Commits and writes the updated values WS(T2) in the database.

In this example, both T1 and T2 commit. Please note that read sets RS(T1) and RS(T2) are disjoint, also the Write sets are also disjoint, thus, no concurrency-related problem occurs.

Now let us consider another example, as given below:

T1	T2	T3
Read A	--	--
--	Read 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 commit point in a validation phase. The serialisation order is determined by the time of the transaction validation phase.

10.7 TIMESTAMP BASED PROTOCOLS

A timestamp is used to identify the sequence of transactions. This section explains the timestamp-based concurrent execution of transactions and the multi-version technique.

10.7.1 Timestamp-Based Concurrency Control

Timestamp-based protocols use the concept of a timestamp, which is a unique value assigned to a transaction that can determine the sequence or order of transactions. The timestamp of a transaction can either be the clock time at the start of a transaction or it can be an auto-incrementing counter. In addition, the timestamping-based protocol requires that each data item be associated with the following timestamps:

1. Read timestamp: Read timestamp of a data item, say X, is the highest timestamp of the transaction that has read that data item.
2. Write timestamp: Write timestamp of a data item is the highest timestamp of the transaction that has written that data item.

Assume that a data item Di has a read timestamp rDSi and write timestamp wDSi; and a transaction with timestamp trSi makes a request to READ Di, then:

READ operation on Di:

IF (trSi<wDSi) then REJECT READ and ROLLBACK Ti

//Why? The transaction is trying to read the data item, which has a lower timestamp than that of the highest timestamp of the transaction that has written the data item. Thus, this transaction is too late to read the data item, which is already written by a younger transaction. //

ELSE //trSi>= wDSi is TRUE//

so ALLOW READ and

SET rDSi = higher of (trSi, rDSi) //Set read timestamp of data item

Write operation on Di:

IF (trSi<rDSi) then REJECT WRITE and ROLLBACK Ti

//Why? The transaction is trying to write the data item, which is already read by a newer transaction. Therefore, this transaction is too late to write the data item.

IF (trSi<wDSi) then REJECT WRITE and ROLLBACK Ti

Why? The transaction is trying to write the data item, which is already written by a newer transaction. Therefore, this transaction is trying to write an old value to the data item.

ELSE //trSi>= rDSi and trSi>=wDSi //

so ALLOW WRITE and

SET wDSi = trSi

For example, consider the transactions T1 with timestamp 1, T2 with timestamp 2 and T3 with timestamp 3. A data item D1 is read by these transactions in the sequence T2, T1 and T3. In addition, they request to write the data item D1 in the sequence T3, T1, T2. Then how are these sequences allowed or rejected? The following Figure shows this sequence:

Request	Timestamp of the Transaction (trSi)	Action Performed	Read Timestamp of data item D1 (rDS1)	Write Timestamp of data item D1 (wDS1)
Initial State	-	-	0	0
`READ D1` by T2	2	Allowed	2	0
`READ D1` by T1	1	trS1<rDS1 Reject and Rollback T1	2	0
`READ D1` by T3	3	trS3>rDS1 Allowed	3	0
`WRITE D1` by T3	3	trS3>=rDS1 and trS3>wDS1 Allowed	3	3
`WRITE D1` by T1	-	This Transaction is already Rolled back	3	3
`WRITE D1` by T2	2	trS2<rDS1 Reject and Rollback T2	3	3

Thus, you can observe that most of the rules have been applied in the example, as given above.

10.7.2 Multi-version Technique

Multi-version technique, as the name suggests, allows the previous versions of data to be stored. Thus, this scheme avoids the rollback of transactions. This scheme also uses read timestamp and write timestamp for each version of the data item. This technique can be defined as:

Consider a data item Di and its version Di(Ver1), Di(Ver2), …, Di(Vern). For each Di(Veri), a read timestamp rDSi(Veri) and a write timestamp wDSi(Veri) are stored. The read timestamp for the multi-version technique is the highest timestamp of the

transactions that have read the i^{th} version; whereas the write timestamp is the timestamp of the transaction that has resulted in the creation of this version.

In this technique read operation reads the version whose timestamp is equal to or just less than the transaction. While the case of a write operation by a transaction with timestamp trSi, which is trying to write a data item Di that has k versions, the following three cases are possible:

- IF trSi<rDSi(Verk), then ROLLBACK transaction;
- IF trSi == wDSi(Verk), then REWRITE the version k of the data item;
- IF trSi>rDSi(Verk), then CREATE version k=1 of data item;

10.8 WEAK LEVEL OF CONSISTENCY AND SQL COMMANDS FOR TRANSACTIONS

In general, concurrent transactions are expected to be operated in serialisable mode of operation. However, in many applications, certain weaker consistency levels can also be used. These weaker consistency level puts lesser restrictions on the execution of transactions. In this section, we discuss the four different isolation levels supported by SQL, a few of which supports a weak level of consistency.

SERIALISABLE: In this isolation level, the transactions follow the ACID properties, therefore, eliminating problems due to concurrent execution of transactions, however, restricting the concurrency.

REPEATABLE READ: This isolation level allows the reading of data items only after the related transaction has been committed. However, in this isolation level a recently committed transaction may perform addition or removal of some rows.

READ COMMITTED: This isolation level also allows the reading of data items only after the related transaction has been committed. However, if a transaction repeats a read operation of a specific data item, then it is not guaranteed to obtain the same value for that data item.

READ UNCOMMITTED: This is the weakest isolation level and does not put any restriction on the concurrent execution of transactions. However, it may result in every concurrency-related problem.

The SQL command that supports transactions is:

To set a typical isolation level for transaction execution, you may set the transaction isolation level using the command:

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

You may also change the isolation level by using the alter command. You can commit a transaction by using COMMIT or roll it back by using ROLLBACK in the transaction.

You can refer to further readings for a detailed discussion on weak consistency levels and SQL commands related to transaction control.

☞ Check Your Progress 3

1) Draw a suitable graph for following locking requests and find whether the transactions are deadlocked or not.

T1: S_LOCK A	--	--
--	T2: X_LOCK B	--
--	T2: S_LOCK A	--
--	--	T3: X_LOCK C
--	T2: S_LOCK C	--
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?

……

……

3) What are the different types of timestamps for a data item? Why are they used?

……

……

4) What is the purpose of the multi-version technique? Does the multi-version technique also have rollback?

……

……

5) What are the different weak consistency levels?

……

……

10.9 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 transactions do not get into each other's way, i.e., updates of data items made by one transaction do not affect the updates made by the other transactions on the same data items.

Serialisability is the generally accepted criterion for the correctness of concurrency control. It is a concept related to concurrent schedules. It determines 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 source to be

unlocked. This unit also presents the Two-Phase Locking (2PL) protocol that ensures that transactions do not encounter concurrency-related problems. A system is in a deadlock state if there exists a set of transactions, which are waiting for each other to complete. We can use a deadlock prevention protocol to ensure that the system will never enter a deadlock state.

This unit also discusses optimistic concurrency control, timestamp-based concurrency control and multi-version technique for concurrency control. Finally, the unit discusses the different weak consistency levels and related SQL commands.

10.10 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 operations 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: Other transactions should not see uncommitted values
   d) Durability: Once committed the changes should be reflected in the database.

   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 item that has been issued.

2) The basic anomalies of concurrent transactions are:

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

   No, these problems cannot occur if the transactions do not perform interleaved read or write operations on the same data items.

3) The commit state arrives when a transaction has completed all the updates correctly, and all these changes are to be accepted. No, you cannot roll back after the commit.

Check Your Progress 2

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

Initially:	Sequence of operation as initially A=0	Final Value
T1-T2-T3:	A = 1; A=2; A is displayed as 2 and Set to 1	1
T1-T3-T2:	A = 1; A is displayed as 1 and set to 1; A=2	2
T2-T1-T3:	A = 0; A=1; A is displayed as 1 & set to 1	1

T2-T3-T1:	A = 0; A is displayed as 0 and set to 1; A=2	2
T3-T1-T2:	A is displayed as 0 and set to 1; A=2; A=4	4
T3-T2-T1:	A is displayed as 0 and set to 1; A=2; A=3	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) This is a schedule using conservative 2 PL.

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

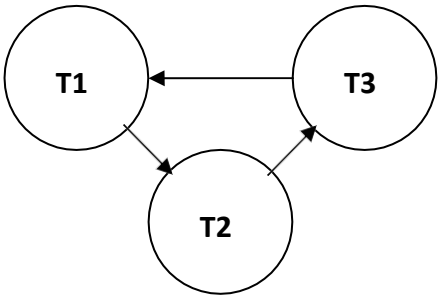
You must also make the schedules using read and exclusive lock and a schedule in strict 2PL.

Check Your Progress 3

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

- Now T2 requests for a shared lock on C, which is exclusively locked by T3, so this lock cannot 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. T1 request for a shared lock on C is not processed.
- Next, T3 requests for an exclusive lock on A, which is share locked by T1, so it cannot 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, you just validate your assumption before you make changes permanent. This is a good model for situations having a low rate of transactions.

3) Two basic timestamps are associated with a data item – Read timestamp and Write timestamp. They are primarily used to see that a transaction that is reading data should not be too old that some younger transaction has already written it, or a transaction that wants to write that data item should not be younger than the transaction that has read or written it.

4) The multi-version technique of concurrent transactions ensures that all the versions or changes in the value of a data item are duly recorded. Yes, even this scheme may have a rollback.

5) The four weak consistency levels are - SERIALISABLE, REPEATABLE READ, READ COMMITTED, READ UNCOMMITTED.

UNIT 11 DATABASE RECOVERY AND SECURITY

11.0 INTRODUCTION

In the previous unit of this block, you have gone through the details of transactions, their properties, and the management of concurrent transactions. In this unit, you will be introduced to two important issues relating to database management systems – how to deal with database failures and how to handle database security. 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 reliabilityof the database system is also linked to the reliability of the computer system on which it runs. 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. Thus, the basic objective of the recovery system is to resume the database system to the point of failure with almost no loss of information. Further, the recovery cost should be justifiable. In this unit, we will discuss various types of failures and some of the approaches to database recovery.

The second main issue that is being discussed in this unit is Database security. “Database security” is the 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. In addition, the unit discusses various access

control mechanisms for database access. Finally, the unit introduces the use of audit trails in a database system.

11.1 OBJECTIVES

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

- describe the terms recovery;
- explain log based recovery;
- explain the use of checkpoints in recovery;
- define the various levels of database security;
- define the access control mechanism;
- identify the use of audit trails in database security.

11.2 WHAT IS RECOVERY?

During the life of a transaction, i.e. after the start of a transaction but before the transaction commits, it makes several uncommitted changes in data items of a database. The database during this time may be in an inconsistent state. In practice, several things might happen to prevent a transaction from completing. Recovery techniques are designed to bring an inconsistent database, after a failure, into a consistent database state. If a transaction completes normally and commits, then all the changes made by that transaction on the database should get permanently registered in the database. They should not be lost (please recollect the durability property of transactions given in Unit 10). But if a transaction does not complete normally and terminates abnormally then all the changes made by it should be discarded.

11.2.1 Kinds of Failures

An abnormal termination of a transaction may occur 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 the 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 system restart. The DBMS should recover from any such failures. Let us first discuss the kinds of failure for ascertaining the approach of recovery.

A DBMS may encounter a failure. These failures may be of the following types:

1. Transaction failure: An ongoing transaction may fail due to:

- **Logical errors**: Transaction cannot be completed due to some internal error condition.
- **Database system errors**: A database system error can be caused by some failure at the level of a database. For example, a transaction deadlock may result in a database system error. This will result in the abrupt termination of some of the ongoing deadlocked transactions.

2. System crash: This kind of failure includes hardware/software failure of a computer system. In addition, a sudden power failure can also result in a system crash, which may result in the following:

- Loss or corruption of non-volatile storage contents.
- Loss of contents of the entire disk or parts of the disk. However, such loss is assumed to be detectable; for example, the checksums used on disk drives can detect this failure.

11.2.2 Storage Structures for Recovery

All these failures result in the inconsistent state of a database. Thus, we need a recovery scheme in a database system, but before we discuss recovery, let us briefly define the storage structure from the recovery point of view.

There are various ways for storing information for database system recovery. These are:

Volatile storage: Volatile storage does not survive system crashes. Examples of volatile storage are - the main memory or cache memory of a database server.

Non-volatile storage: The non-volatile storage survives the system crashes if it does not involve disk failure. Examples of non-volatile storage are - magnetic disk, magnetic tape, flash memory, and non-volatile (battery-backed) RAM.

Stable storage: This is a mythical form of storage structure that is assumed to survive all failures. This storage structure is assumed to maintain multiple copies on distinct non-volatile media, which may be independent disks. Further, data loss in case of disaster can be protected by keeping multiple copies of data at remote sites. In practice, software failures are more common than hardware failures. Fortunately, recovery from software failures is much quicker.

11.2.3 Recovery and Atomicity

The concept of recovery relates to the atomic nature of a transaction. Atomicity is the property of a transaction, which states that a transaction is a complete unit. Thus, the execution of a part transaction can lead to an inconsistent state of the database, which may require database recovery. 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 failed after account A has been written back to the 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 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 log files later in this unit when we define different methods of recovery. In the next section, we discuss the recovery of transactions in more detail.

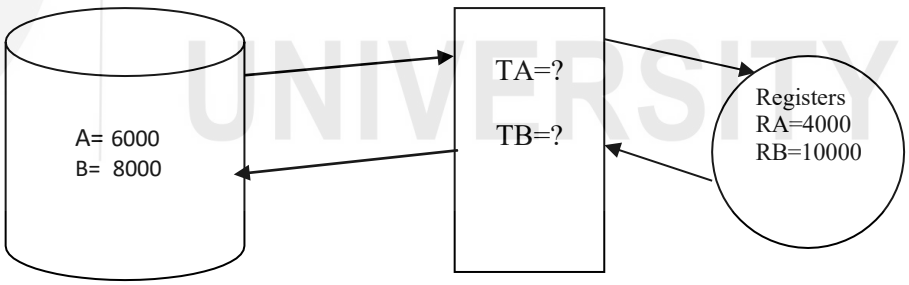
11.2.4 Transactions and Recovery

The basic unit of recovery is a transaction. But how are the transactions handled during recovery?

Consider the following two cases:

i. Consider that some transactions are deadlocked, then at least one of these transactions must be restarted to break the deadlock, and thus, the partial updates made by this restarted transaction program are to be undone to keep the database in a consistent state. In other words, you may ROLLBACK the effect of a transaction.

ii. A transaction has committed, but the changes made by the transaction have not been communicated to the physical database on the hard disk. 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 performs the changes made by the transaction again to bring the system to a consistent state. The database system can then be made available to the users. The point to be noted here is that such a situation has occurred as database updates are performed in the buffer memory. *Figure 1* shows cases of **undo** and **redo**.



Physical database RAM Buffers CPU

	Physical Database at the time of failure	RAM	Activity
Case 1	A=6000 B=8000	TA=4000 TB=8000	Transaction T1 has changed the value of A in register RA (4000) and RB (10000). RA is written to RAM (TA), but not updated in A. RB is not written back to RAM (TB). T1 did not COMMIT. **Now, T1 is aborted by the user.** The value of CPU registers and RAM

			are made irrelevant. You cannot determine if the TA, TB has been written back to A, B. **You must UNDO** the transaction.
Case 2	A=4000 B=8000	TA=4000 TB=8000	The value of A in the physical database has got updated due to buffer management. RB is not written back to RAM (TB). The transaction did not COMMIT so far. Now, the transaction T1 aborts. **You must UNDO** the transaction.
Case 3	A=6000 B=8000	TA=4000 TB=10000 COMMIT	The value B in the physical database has not got updated due to buffer management. T1 has raised the COMMIT flag. The changes of the transaction must be performed again. **You must REDO** the transaction. How? (Discussed in later sections).

Figure 1: Database Updates and Recovery

11.2.5 Recovery in Small Databases

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

- to regulate the power supply.
- to use a failsafe secondary storage system such as RAID.
- to take periodic database backups and keep track of transactions after each recorded state.
- to properly test transaction programs prior to use.
- to set important integrity checks in the databases as well as user interfaces.

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

11.3 TRANSACTION RECOVERY

As discussed in the previous section, a transaction is the unit of recovery. A commercial database system, like a banking system, may support many concurrent transactions at a time. A failure may affect multiple transactions in such systems. Several recovery techniques have been designed for commercial DBMSs. This section discusses some of the basic recovery schemes used in commercial DBMSs.

11.3.1 Log-Based Recovery

Let us first define the term transaction log in the context of DBMS. A transaction log, in DBMS, records information about every transaction that modifies any data values in the database. A log contains the following information about a transaction:

- A transaction BEGIN marker.
- Transaction identification - transaction ID, terminal ID, user ID, etc.
- The operations being performed by the transaction such as UPDATE, DELETE, INSERT.
- The data items or objects affected by the transaction - may include the table's name, row number and column number.
- 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 been updated 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 what 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 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 a log? Let us demonstrate this with the help of an example having three concurrent transactions that are active on ACCOUNTS relation:

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 structure):

Transaction Begin Marker	Transaction Id	Operation on ACCOUNTS table	UNDO values (assumed)	REDO values	Transaction Commit Marker
Yes	T1	SUB ON X ADD ON Y	X=500 Y=800	X=400 not done yet	No
Yes	T2	ADD ON A	A=1000	A=1200	No
Yes	T3	SUB ON Z	Z=900	Z=400	Yes

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.

Transaction /Values	Initial (UNDO value)	Just before the failure	Operation Required for recovery	DatabaseValues after Recovery
T1/X	500	400 (assuming update has been done in physical database also)	UNDO (As transaction did not COMMIT)	X = 500
T1/Y	800	800	UNDO	Y = 800

T2/A	1000	1000 (assuming update has not been done in physical database)	UNDO (As transaction did not COMMIT)	A = 1000
T3/Z	900	900 (assuming update has not been done in physical database)	REDO (As transaction *did* COMMIT)	Z = 400

Figure 4: The database recovery

The selection of REDO or UNDO for a transaction for the recovery is done based on 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 will be performed on T3.

Please note that in Figure 4, some of the values may not have yet been communicated to the database, yet we need to perform UNDO as we are not sure what values have been written back to the database. Similarly, you must perform REDO operations on committed transactions, such as Transaction T3 in Figure 3 and Figure 4.
But how will the system recover? Once the recovery operation has been specified, the system just determines the required REDO or UNDO values from the transaction log and changes the inconsistent state of the database to a consistent state. (Please refer to *Figure 3* and *Figure 4*).

13.3.2 Checkpoints in Recovery

Let us consider several transactions, which are shown on a timeline, with their respective START and COMMIT time (see Figure 5).

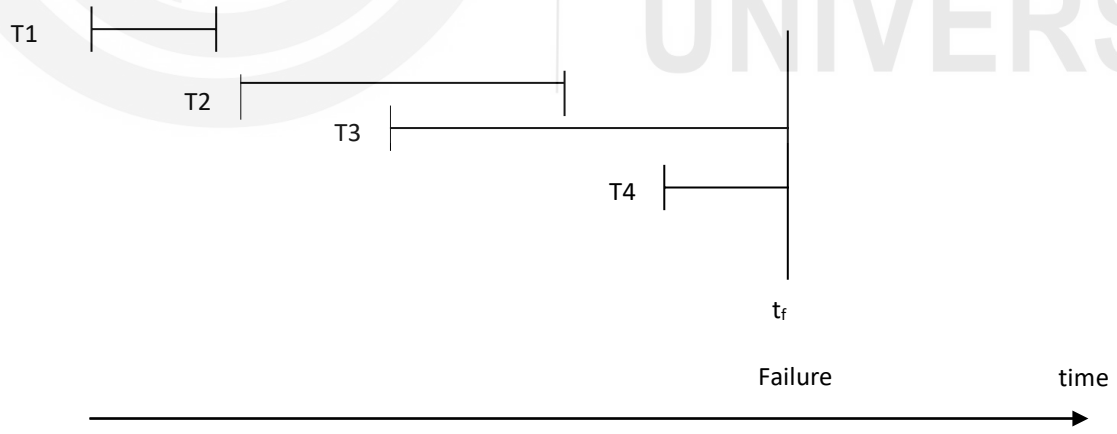


Figure 5: Execution of Concurrent Transactions

Consider that the transactions are allowed to execute concurrently, on encountering a failure at time t_f, the transactions T1 and T2 are to be REDONE and T3 and T4 will be UNDONE (Refer to Figure 5). Now, considering that a system allows 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 even those transactions that might have been committed several hours earlier. How can you improve this situation? You can remedy this problem by taking a checkpoint. *Figure 6* shows a checkpoint mechanism:

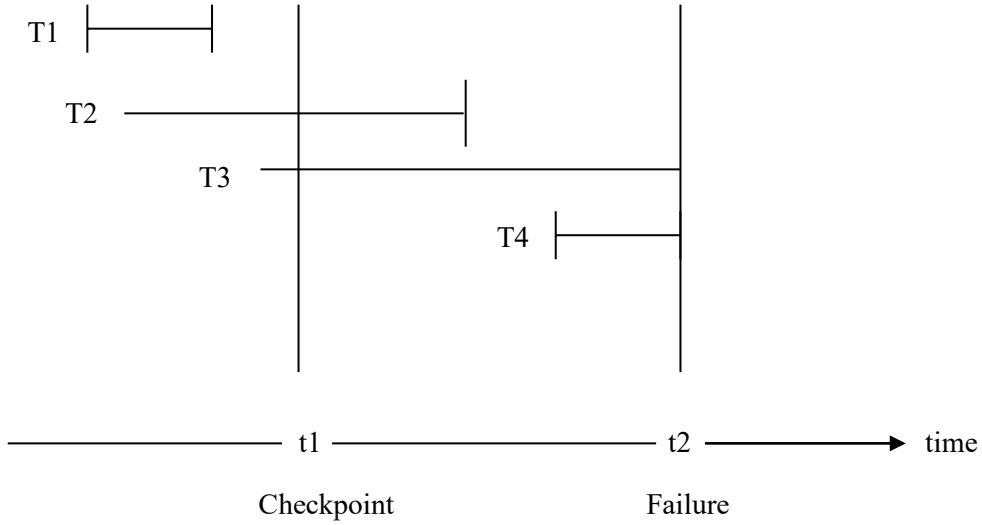


Figure 6: Checkpoint in Transaction Execution

A checkpoint is taken at time t1, and a failure occurs at time t2. Checkpoint transfers all the committed changes to the database and all the system logs to stable storage (the storage that would not be lost). On the restart of the system after the failure, the stable checkpointed state is restored. Thus, we need to REDO or UNDO only those transactions that have been completed or started after the checkpoint has been taken. A disadvantage of this scheme is that the database would not be available when the checkpoint is being taken. In addition, some of the uncommitted data values may be put in the physical database. To overcome the first problem, the checkpoints should be taken at times when the system load is low. To avoid the second problem, the system may allow the ongoing transactions to be complete while not starting any new transactions.

In the case of *Figure 6*, the recovery from failure at time t2 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 the physical database at checkpoint t1.
- The transaction T2 has not committed till checkpoint t1 but has committed before t2 will be REDONE.
- T3 must be UNDONE as the changes made by it before the checkpoint (we do not know for sure if any such changes were made prior to the 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 restart the transaction T4 with a new name.

The restart of a transaction requires the log to keep information on the new name of the transaction. This new transaction may be given higher priority.

But one question that remains unanswered is - during a failure, we lose database information in RAM buffers; we may also lose the content of the log as it is also stored in RAM buffers, so how does the log ensure recovery?

The answer to this question lies in the fact that for storing the log of the transaction, we follow a **Write Ahead Log Protocol.** As per this protocol, the transaction logs are written to stable storage as follows:

- **UNDO portion of the log is written to stable storage prior to any updates. and**
- **REDO portion of the log is written to stable storage prior to the 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 further readings for more details on recovery and its techniques.

☞ Check Your Progress 1

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

……

……

2) What is the log-based recovery?

……

……

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

……

……

11.3.3 Recovery Algorithms

As discussed earlier, a database failure may bring the database into an inconsistent state, as many ongoing transactions will simply abort. In order to bring such an inconsistent database to a consistent state, you are required to use recovery algorithms. Database recovery algorithms require basic data related to failed transactions. Thus, a recovery algorithm requires the following actions:

1) Actions are taken to collect the required information for recovery while the transactions are being processed prior to the failure.

2) Actions are taken after a failure to recover the database contents to a state that ensures atomicity, consistency and durability.

In the context of point 1 above, it may be noted that the information about the changes made by a transaction is recorded in a log file. In general, the sequence of logging during the transaction execution is as follows:

- A transaction, say T_i, announces its start by putting a log record consisting of $<T_i$ **start**>.
- *Before* T_i executes the **write**(X) (see Figure 7), put a log record $<T_i, X, V_1, V_2>$, where V_1 represented the older value of X, i.e., the value of X before the write (*undo* value), and V_2 is the updated value of X *(redo* value*)*.
- On successful completion of the last statement of the transaction T_i, a log

record consisting of <T_i **commit**> is put in the log. It may be noted that all these log records are put in stable storage media, that is, they are not buffered in the main memory).

Two approaches for recovery using logs are:

- Deferred database modification.
- Immediate database modification.

Deferred Database Modification

This scheme logs all the changes made by a transaction into a log file, which is stored on stable storage. Further, these changes are not made in the database till the transaction commits. Let us assume that transactions execute serially to simplify the discussion. Consider a transaction T_i, it performs the following sequence of log file actions:

Event	Record for Log file	Comments
Start of Transaction	<T_i ***start***>	
write(X)	<T_i, X, V >	Transaction writes a new value V for data item X in the database.
Transaction T_i commits	<T_i ***commit***>	The transaction has been committed, and now the deferred updates can be written to the database

- Please note that in this approach, the old values of the data items are not saved at all, as changes are being recorded in the log file and the database is not changed till a transaction commits. For example, the **write**(X), shown above, does not write the value of X in the database till the transaction commits. However, the record <T_i, X, V > is written to the log. All these updates are written to the database after the transaction commits.

How is the recovery performed for the deferred database modification scheme?

For the database recovery after the failure, the log file is checked. The transactions for which the log file contains the <T_i **start**> and the <T_i **commit**> records, the REDO operation is performed using log record <T_i, X, V >. Why? Because in the deferred update scheme we do not know if the changes made by this committed transaction have been carried out in the physical database or not. The Redo operation can be performed many times without the loss of information, so it will be applicable even if the crash occurs during recovery. The transactions for which <T_i **start**> is in the log file but not the <T_i **commit**>, the transaction UNDO is not required, as it is expected that the values are not yet written to the database.

Let us explain this scheme with the help of the transactions given in Figure 7.

```
T1: READ (X)                    T2 : READ (Z)
    X= X - 1000                     Z= Z - 1000
    WRITE (X)                       WRITE (Z)
    READ (Y)
    Y= Y + 1000
    WRITE (Y)
```

Sample Transactions T_1 and T_2 (T_1 executes before T_2)

Figure 7: Sample Transaction for demonstrating Recovery Algorithm

Figure 8 shows the state of a sample log file for three possible failure instances, namely

(a), (b) and (c). (Assuming that the initial balance in X is 10,000/- Y is 5,000/- and Z has 20,000/-):

<T_1 START> <T_1, X, 9000> <T_1, Y, 6000>	<T_1 START > <T_1, X, 9000> <T_1, Y, 6000> <T_1 COMMIT> <T_2 START > <T_2, Z, 19000>	<T_1 START> <T_1, X, 9000> <T_1, Y, 6000> <T_1 COMMIT > <T_2 START > <T_2, Z, 19000> <T_2 COMMIT >
(a)	(b)	(c)

Figure 8: Log Records for Deferred Database Modification for Transactions of Figure 7

Why do you store only the Redo value in this scheme? The reason is that No UNDO is required as updates are communicated to stable storage only after COMMIT or even later. The following REDO operations would be required if the log on stable storage at the time of the crash is as shown in Figure 8(a) 8(b) and 8(c).

(a) No REDO is needed, as no transaction has been committed in Figure 8(a).

(b) Perform REDO(T_1), as <T_1 **COMMIT**> is part of the log in Figure 8 (b).

(c) Perform REDO(T_1) and REDO(T_2), as <T_1 **COMMIT** > and <T_2 **COMMIT** > are in the log shown in Figure 8(c).

Please note that you can repeat this sequence of redo operation as suggested in Figure 8(c) any number of times, it will still bring the value of X, Y, and Z to consistent redo values. This property of the redo operation is called **idempotent.**

Immediate Database Modification

This recovery scheme allows the modification of the data items of the stored database by ongoing uncommitted transactions. Thus, the recovery database would require UNDO and REDO of updates made by these transactions. Therefore, the log file for this recovery scheme should include the UNDO value or the original value and the REDO value or the modified value. Further, the log records are assumed to be written to the stable storage.

The following log file shows the log of transactions given in Figure 7, for the case when immediate database modification is followed:

Log	**Write operation**	**Output**
<T_1 **START**>		
<T_1, X, 10000, 9000>	X = 9000	Output Block of X
<T_1, Y, 5000, 6000>	Y = 6000	Output Block of Y
<T_1 **COMMIT**>		
<T_2 **START**>		
<T_2, Z, 20,000, 19,000>	Z= 19000	Output Block of Z
<T_2 **COMMIT**>		

Figure 9: Log Records for Immediate Database Modification for Transactions of Figure 7

The UNDO and REDO operations for this recovery scheme are as follows:

- **UNDO**(T_i):
  - List all the transactions which have the start record in the log but no commit record. For all the uncommitted transactions in the list perform the following:
    - Find the last UNDO log record of a transaction.
    - Move backwards in the log file to find all the entries for UNDO of

that transaction.
    - For each UNDO entry, assign the UNDO value or original value to the data item value.

- **REDO**(T_i):
    - List all the transactions which have the start record and commit record in the log. For all the committed transactions in the list, perform the following:
        - Find the first REDO log record of a transaction.
        - Move forward in the log file to find all the entries for REDO of that transaction.
        - For each REDO entry, assign the REDO value or modified value to the data item value.

You may note that you may perform REDO/UNDO operations any number of times if needed. Further, UNDO operations are performed first, followed by the REDO operations.

Example:

Consider the log as it appears at three instances of time.

<T_1 start> <T_1, X 10000, 9000> <T_1, Y 5000, 6000> Failure (a)	<T_1 start> <T_1, X 10000, 9000> <T_1, Y 5000, 6000> <T_1, Commit> <T_2 start> <T_2, Z 20000, 19000> Failure (b)	<T_1 start> <T_1, X 10000, 9000> <T_1, Y 5000, 6000> <T_1, Commit> <T_2 start> <T_2, Z 20000, 19000> <T_2, Commit> Failure (c)

Figure 10: Recovery in Immediate Database Modification technique

For each of the failures as shown in Figure 10 (a), (b) and (c), the following recovery actions would be needed:

(a) UNDO (T_1):

X ← Undo value of X, i.e. 10000;

Y ← Undo value of Y, i.e. 5000.

(b) UNDO (T_2):

Z ← Undo value of Z, i.e. 20000;

REDO (T_1):

X ← Redo value of X, i.e. 9000;

Y ← Redo value of Y, i.e. 6000.

(c) REDO (T_1, T_2) by moving forward:

X ← Redo value of X, i.e. 9000;

Y ← Redo value of Y, i.e. 6000;

Z ← Redo value of Z, i.e. 19000;

Advanced Recovery Techniques

The recovery processes for DBMS have gone through several changes. One of the basic objectives of any recovery technique is to minimise the time that is required to perform the recovery. However, in general, this reduction in time results in increased complexity of the recovery algorithms. One of the recent popular log-based recovery techniques is named ARIES. You may refer to further reading for more details on ARIES. In general, several aspects that increase the speed of the recovery process are:

- Transaction log sequence numbers for log records may be assigned to log entries. This will help in identifying the related database log pages.
- Instead of deletion the records from the physical database, record the deletion in the log.
- Keep track of pages that have been updated in memory but have not been written back to the physical database. Perform Redo operations for only such pages. Also, keep track of updated pages during checkpointing.

You may refer to further readings for more details on newer methods and algorithms of recovery.

11.3.4 Recovery with Concurrent Transactions

In general, a commercial database management system, such as a banking system, has many users. These users can perform multiple transactions. Therefore, a centralised database system executes many concurrent transactions. As discussed in Unit 10, these transactions may experience concurrency-related issues and, thus, are required to maintain serialisability. In general, these transactions share a large buffer area and log files. Thus, a recovery scheme that allows better control of disk buffers and large log files should be employed for such systems. One such technique is called checkpointing. This changes the extent to which REDO or UNDO operations are to be performed in a database system. The concept of the checkpoint has already been discussed in section 11.3.2. How checkpoints can help in the process of recovery is explained below:

When you use the checkpoint mechanism, you also add records of checkpoints in the log file. A checkpoint record is of the form: <**checkpoint** *TL*>. In this case, *TL* is the list of transactions, which were started, but not yet committed at the time when the checkpoint was created. Further, we assume that at the time of creating a checkpoint record, no transaction was allowed to proceed.

On the restart of the database system after failure, the following steps are performed for the database recovery:

- Create two lists: UNDOLIST and REDOLIST. Initialise both lists to a NULL.
- Scan the log file for every log record, starting from the end of the file and moving backwards, till you locate the first checkpoint record <**checkpoint** *TL*>. Perform the following actions, for each of the log records found in this step:
    - Is the log record <T_i COMMIT>?
        - If yes, then add T_i to REDOLIST.
    - Is the log record <T_i START>?
        - If yes, then Is T_i NOT IN REDOLIST?
            - Add T_i to UNDOLIST, as it has not yet been committed.
    - For every T_i in *TL*: Is T_i in NOT IN REDOLIST?
            - add T_i to UNDOLIST, as this transaction was active at the time of checkpoint and has not been committed yet.

This will make the UNDOLIST and REDOLIST of the transactions. Now, you can perform the UNDO operations followed by REDO operations using the log file, as given in section 11.3.3.

11.3.5 Buffer Management

As discussed in the previous sections, the database transactions are executed in the memory, which contains a copy of the physical database. These database buffers are written back to stable storage from time to time. In general, it is the memory management service of the operating system that manages the buffers of a database system. However, several database management schemes require their own buffer management policies and, hence, the buffer management system. Some of these strategies are:

Log Record Buffering

The recovery process requires database transactions to write logs in stable storage. This is a very time-consuming process, as for a single transaction several log records are to be written to stable storage. Therefore, several commercial database management systems perform the buffering of the log file itself, which means that log records are kept in the blocks of the main memory allocated for this purpose. The logs are then written to the stable storage once the buffer becomes full or when a transaction commits. This log file buffering helps in reducing disk accesses, which are very expensive in terms of time of operation.

Database recovery requires that log records should be stored in stable storage. Therefore, log records should be transferred from memory buffers to the table storage as per the following scheme, called Write-Ahead logging.

- The sequence of log records in the memory buffer should be maintained in stable storage.
- A transaction should be moved to COMMIT state only if the <T_i **commit**> log record is written to stable storage.
- Prior to writing a database buffer to stable storage, related log records in the buffer should be moved to stable storage.

Database Buffering

The database updates are performed after moving database blocks on secondary storage to memory buffers. However, due to limited memory capacity, only a few database blocks can be kept in the memory buffers. Therefore, database buffer management consists of policies for deciding which blocks should be kept in database buffers and what blocks should be removed from the database buffers back to secondary storage. Removing a database block from the buffer requires that it is re-written to the secondary storage. In addition, the log is written to stable storage, as per the write-ahead logging.

11.3.6 Remote Backup Systems

Most of the techniques discussed above perform recovery in the context of a centralised system. However, present-day transaction processing systems require high availability. One of the ways of implementing a high availability system is to create a primary and a backup site for the transaction processing system. With fast, highly reliable networks, this backup site can be a remote backup site. This remote backup site may be very useful in case of disaster recovery. Some of the issues of the remote backup system include the following:

- The failure of the primary database site must be detected. It may be noted that this detection should not detect communication failure as a primary database failure. Thus, it may use a failsafe communication, which may have alternative communication links to the primary database site.
- The backup site should be capable enough to work as the primary database site at the time of failure of the primary site. In addition to recovery of ongoing transactions, once the primary site recovers it should get all the updates, which were performed while the primary site was down.

You may refer to the further readings for more details on this topic.

☞ Check Your Progress 2

1) What is deferred database modification in recovery algorithms?

……

……

2) How is log of the deferred database modification technique differ from the log of Immediate database modification?

……

……

3) Define buffer management and remote backup system in the context of recovery.

……

……

11.4 SECURITY IN COMMERCIAL DATABASES

You must realise that security is a journey, not the final destination. You cannot assume a product/technique is absolutely secure, as you may not be aware of fresh/new attacks on that product/technique. Many security vulnerabilities are not even published as attackers want to delay a fix, and manufacturers do not want negative publicity. There is an ongoing and unresolved discussion over whether highlighting security vulnerabilities in the public domain encourages or prevents further attacks.

The most secure database you can think of must be found in a most securely locked bank or nuclear-proof bunker, installed on a standalone computer without an Internetor network connection, and under guard for 24×7×365. However, that is not a likely scenario with which we would like to work. A database server maintains database services, which often contain security issues, and you should be realistic about possible threats. You must assume security failure at some point and never store truly sensitive data in a database that unauthorised users may easily infiltrate/access. A major point here is that most data loss occurs because of social exploits and not technical ones. Thus, the use of encryption algorithms for security may need to be looked into.

You will be able to develop effective database security if you realise that securing data is essential to the market reputation, profitability and business objectives. For example, personal information such as credit cards or bank account numbers are now commonly available in many databases; therefore, there are more opportunities for identity theft. As per an estimate, several identity theft cases are committed by employees who have access to large financial databases. Banks and companies that take credit card services externally must place greater emphasis on safeguarding and controlling access to this proprietary database information.

Securing the database is a fundamental tenet for any security personnel while developing his or her security plan. The database is a collection of useful data and can be treated as the most essential component of an organisation and its economic growth. Therefore, for any security effort, you must keep in mind that you need to provide the strongest level of control over the data of a database.

As is true for any other technology, the security of database management systems depends on many other systems. These primarily include the operating system, the applications that use the DBMS, services that interact with the DBMS, the web server that makes the application available to end users, etc. However, please note that most importantly, DBMS security depends on us, the users.

11.4.1 Common Database Security Failures

Database security is of paramount importance for an organisation, but many organisations do not take this fact into consideration till an eventual problem occurs. The common pitfalls that threaten database security are:

Weak User Account Settings: Many of the database user accounts do not contain the user settings that may be found in operating system environments. For example, the user accounts name and passwords, which are commonly known, are not disabled or modified to prevent access.

Insufficient Segregation of Duties: Several organisations have no established security administrator role. This results in database administrators (DBAs) performing both the functions of the administrator (for users' accounts), as well as the performance and operations expert. This may result in management inefficiencies.

Inadequate Audit Trails: The auditing capabilities of DBMS, since it requires keeping track of additional requirements, are often ignored for enhanced performance or disk space. Inadequate auditing results in reduced accountability. It also reduces the effectiveness of data history analysis. The audit trails record information about the actions taken on certain critical data. They log events directly associated with the data; thus, they are essential for monitoring the access and the activities on a database system.

Unused DBMS Security Features: The security of an individual application is usually independent of the security of the DBMS. Please note that security measures that are built into an application apply to users of the client software only. The DBMS itself and many other tools or utilities that can connect to the database directly through ODBC or any other protocol may bypass this application-level security completely. Thus, you must try to use security restrictions that are reliable, for instance, try using the security mechanisms that are defined within the DBMS.

11.4.2 Database Security Levels

Basically, database security can be broken down into the following levels:

- Server Security
- Database Connections
- Table Access Control

Server Security: Server security is the process of controlling access to the database server. This is the most important aspect of security and should be carefully planned. The basic idea here is "You cannot access what you do not see". For security purposes, you should never let your database server be visible to the world. If a database server is supplying information to a web server, then it should be configured in such a manner that it is allowed connections from that web server only. Such a connection would require a trusted IP address.

Trusted IP Addresses: To connect to a server through a client machine, you would need to configure the server to allow access to only trusted IP addresses. You should know exactly who should be allowed to access your database server. For example, if the database server is the backend of a local application that is running on the internal network, then it should only talk to addresses from within the internal network.

Database Connections: With the ever-increasing number of Dynamic Applications, an application may allow immediate unauthenticated updates to some databases. If you are going to allow users to make updates to a database via a web page, please ensure that you validate all such updates. This will ensure that all updates are desirable and safe. For example, you may remove any possible SQL code from user-supplied input if a normal user is not allowed to input SQL code.

Table Access Control: Table access control is probably one of the most overlooked but one of the very strong forms of database security because of the difficulty in applying it. Using a table access control properly would require the collaboration of both the system administrator as well as the database developer. In practice, however, such "collaboration" is relatively difficult to find.

By now, we have defined some of the basic issues of database security, let us now consider specifics of server security from the point of view of network access of the system. Internet-based databases have been the most recent targets of security attacks.

All web-enabled applications listen to a number of ports. Cyber criminals often perform a simple "port scan" to look for ports that are open from the popular default ports used by database systems. How can we address this problem? We can address this problem "by default", that is, we can change the default ports a database service would listen into. Thus, this is a very simple way to protect the DBMS from such criminals.

11.4.3 Relationship between Security and Integrity

Database security usually refers to the avoidance of unauthorised access and modification of data of the database, whereas database integrity refers to the avoidance of accidental loss of consistency of data. You may please note that data security deals not only with data modification but also access to the data, whereas data integrity, which is normally implemented with the help of constraints, essentially deals with data modifications. Thus, enforcement of data security, in a way, starts with data integrity. For example, any modification of data, whether unauthorised or authorised must ensure data integrity constraints. Thus, a very basic level of security may begin with data integrity but will require many more data controls. For example, SQL WRITE and UPDATE on specific data items or tables would be possible if it does not violate integrity constraints. Further, the data controls would allow only authorised WRITE and UPDATE on these data items.

11.4.4 Difference between Operating System and Database Security

Security within the operating system can be implemented at several levels ranging from passwords for access to the operating system to the isolation of concurrently executing processes within the operating system. However, there are a few differences between security measures taken at the operating system level compared to those 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 files, tuples, attribute values or indexes. 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. The operating system canprovide security by not allowing any operation to be performed on the database unless the user is authorised for the operation concerned.

After this brief introduction to different aspects of database security, let us discuss one of the important levels of database security, access control, in the next section.

11.5 ACCESS CONTROL

All relational database management systems provide some sort of intrinsic security mechanisms that are designed to minimise security threats, as stated in the previous sections. These mechanisms range from the simple password protection offered in Microsoft Access to the complex user/role structure supported by advanced relational databases like Oracle, MySQL, Microsoft SQL Server, IBM Db2 etc. But can we define access control for all these DBMS using a single mechanism? SQL provides that interface for access control. Let us discuss the security mechanisms common to all databases using the Structured Query Language (SQL).

An excellent practice is to create individual user accounts for each database user. If

users are allowed to share accounts, then it becomes very difficult to fix individual responsibilities. Thus, it is important that we provide separate user accounts for separate users. Does this mechanism have any drawbacks? If the expected number of database users is small, then it is all right to give them individual usernames and passwords and all the database access privileges that they need to have on the database items.
However, consider a situation where there are a large number of users. Specification of access rights to all these users individually will take a long time. That is still manageable as it may be a one-time effort; however, the problem will be compounded if we need to change the access rights for a particular user. Such an activity would require huge maintenance costs. This cost can be minimised if we use a specific concept called "Roles". A database may have hundreds of users, but their access rights may be categorised in specific roles, for example, teacher and student in a university database. Such roles would require the specification of access rights only oncefor each **role.** The users can then be assigned usernames, passwords, and specific roles. Thus, the maintenance of user accounts becomes easier as now we have limited roles to be maintained. You may study these mechanisms in the context of specific DBMS. A role can be defined using a set of data item/object authorisations. In the next sections, we define some of the authorisations in the context of SQL.

11.5.1 Authorisation of Data Items

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 objects, but not modification, deletion, or insertion of a data object.
2) **INSERT**: allows insertion of new data,for example, insertion of a tuple in a relation, but it does not allow the modification of existing data.
3) **UPDATE**: allows modification of data, but not its deletion. However, 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 authorisations, which are broadly called access authorisations.

In addition to these manipulation operations, a user may be granted control operations such as:

1) **ADD**: allows adding new objects such as new relations.
2) **DROP**: allows the deletion of relations in a database.
3) **ALTER**: allows the addition of new attributes in a relation or deletion of existing attributes in a relation.
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 user, for example, if user A has access right R over a relation S and if s/he has right to propagate access control, then s/he can propagate her/his access right R over relation S to another user B either fully or partially. In SQL, you can use WITH GRANT OPTION for this right.

The ultimate form of authority is given to the database administrator. S/he is the one who may authorise new users, restructure the database and so on. The process of authorisation involves supplying information only to the person who is authorised to access that information.

11.5.2 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:

Example

Consider the relation:

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

Assume there are two types of users: The personnel manager and the general user. What access rights may be granted to each user? One extreme possibility is to grant unconstrained access or to have 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 object entities to which access must be controlled.
2) A set of subject entities that request access to objects.
3) A set of access rules in the form of an authorisation matrix, as given in Figure 11 for the relation of the example.

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

Figure 11: Authorisation Matrix for Employee relation.

As the above matrix shows, Personnel Manager and General User are the two subjects. Objects of the database are Empno, Name, Address, Deptno, Salary and Assessment. As per the access matrix, the personnel manager can perform any operation on the database of an employee except for updating the Empno and Name, which may be created once and 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 can be written using SQL statements, which are given in the next subsection.

11.5.3 SQL Support for Security and Recovery

You would need to create the users or roles before you grant them various permissions. The permissions then can be granted to a created user or role. This can be done with the use of the SQL GRANT statement.

The syntax of this statement is:

GRANT <permissions> [ON <table/view>] TO <user/role>

[WITH GRANT OPTION]

Let us define this statement line-by-line. The first line, GRANT <permissions>, allows you to specify the specific permissions on a table or a database view. These can be either relation-level data manipulation permissions (such as SELECT, INSERT, UPDATE and DELETE) or data definition permissions (such as CREATE TABLE, ALTER DATABASE and GRANT). More than one permission can be granted in a single GRANT statement, but data manipulation permissions and data definition permissions may not be combined in a single statement.

The second line, ON <table/view>, is used to specify the table or a view on which permissions are being given. This line is not needed if we are granting data definition permissions.

The third line specifies the user(s) or role(s) that is/are being granted permissions.

Finally, the fourth line, WITH GRANT OPTION, is optional. If this line is included *in the statement, the user* is also permitted to grant the same permissions that s/he has received to other users. Please note that the WITH GRANT OPTION cannot be specified when permissions are assigned to a *role.*

Let us look at a few examples of the use of this statement.

Example 1: Assume that you have recently hired a group of 25 data entry operators who will be adding and maintaining student records in a University database system. They need to be able to access information in the STUDENT table, modify this information and add new records to the table. However, they should not delete a record from the database.

Solution: First, you should create user accounts for each operator and then add them to a new role - DataEntry. Next, you will grant them the appropriate permissions, as given below:

```
GRANT SELECT, INSERT, UPDATE
ON STUDENT
TO DataEntry
```

And that is all that you need to do. The following example assigns data definition permissions to a role.

Example 2: You want to allow members of the DBA role to add new tables to your database. Furthermore, you want DBA to be able to grant permission to other users.

Solution: The SQL statement to do so is:

```
GRANT CREATE TABLE
TO DBA
WITH GRANT OPTION
```

Notice that we have included the WITH GRANT OPTION line to ensure that our DBAs can assign this permission to other users.

Let us now look at the commands for removing permissions from users.

Removing Permissions

Once we have granted permissions, it may be necessary to revoke them at a later date. SQL provides us with the REVOKE command to remove granted permissions. The following is the syntax of this command:

```
REVOKE [GRANT OPTION FOR] <permissions>
ON <table>
FROM <user/role>
```

Please notice that the syntax of this command is almost similar to that of the GRANT command. Please also note that the WITH GRANT OPTION is specified on the REVOKE command line and not at the end of the command as was the case in GRANT. As an example, let us imagine we want to revoke a previously granted permission to the user Usha, such that she is not able to remove records from the STUDENT database. The following SQL command will be able to do so:

```
REVOKE DELETE
ON STUDENT
FROM Usha
```

The access control mechanisms supported by the SQL is a good starting point, but you must look into the DBMS documentation to locate the enhanced security measures supported by your system. You will find that many DBMS support more advanced access control mechanisms, such as granting permissions on specific attributes.

SQL does not have very specific commands for recovery but, it allows explicit COMMIT, ROLLBACK and other related commands.

11.6 AUDIT TRAILS IN DATABASES

One of the key issues to consider while procuring a database security solution is making sure you have a secure audit trail. An audit trail tracks and reports activities around confidential data. Many companies have not realised the potential amount of risk associated with sensitive information within databases unless they run an internal audit which details who has access to sensitive data and have assessed it. Consider the situation in which a DBA who has complete control of database information may conduct asecurity breach with respect to business details and financial information. This will cause tremendous loss to the company. In such a situation database audit helps in locating the source of the problem. The database audit process involves a review of log files to find and examine all reads and writes to database items during a specific time period to ascertain mischief, if any. A banking database is one such database which contains very critical data and should have the security feature of auditing. An audit trail is a log that is used for the purpose of security auditing.

Database auditing is one of the essential requirements for security, especially for companies in possession of critical data. Such companies should define their auditing strategy based on their knowledge of the application or database activity. Auditing need not be of the type "all or nothing". One must do intelligent auditing to save time and reduce performance concerns. This also limits the volume of logs and also causes more critical security events to be highlighted.

More often than not, it is the insiders who make database intrusions as they often have network authorisation, knowledge of database access codes and the idea about the value of data they want to exploit. Sometimes, despite having all the access rights and policies in place, database files may be directly accessible (either on the server or from backup media) to such users. Most database applications store information in 'form text' that is completely unprotected and viewable.

As huge amounts are at stake, incidents of security breaches will increase and continue to be widespread. For example, a large global investment bank conducted an audit of its proprietary banking data. It was revealed that more than ten DBAs had unrestricted access to their key sensitive databases, and over a hundred employees had administrative access to the operating systems. The security policy that was in place was that proprietary information in the database should be denied to employees who did not require access to such information to perform their duties. Further, the bank's database internal audit also reported that the backup data (which is taken once every day) also caused concern as backup media could get stolen. Thus, the risk to the database was high and real and the bank needed to protect its data.

However, a word of caution, while considering ways to protect sensitive database information, please ensure that the privacy protection process should not prevent authorised personnel from obtaining the right data at the right time.

Credit card information is the single most common financially traded information that is desired by database attackers. The positive news is that database misuse or unauthorised access can be prevented with currently available database security products and audit procedures.

☞ Check Your Progress 3

1) On what systems does the security of a Database Management System depend?

……

……

2) Write the syntax for granting permission to alter the database.

……

……

3) Write the syntax for 'Revoke Statement' that revokes the grant option.

……

……

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

……

……..……..

11.7 SUMMARY

In this unit, we have discussed the recovery of the data contained in a database system after failure. 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 no loss of information and at an economically justifiable cost. The basic technique to implement database recovery is to use 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. 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.

Security constraints guard against accidental or malicious tampering with data; integrity constraints ensure that any properly authorised access, alteration, deletion, orinsertion 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. The unit also discussed the use of audit trails.

11.8 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. The transaction is the basic unit of recovery.
2) All recovery processes require redundancy. Log-based recovery process records the consistent state of the database and all the modifications made by a transaction into a log on the stable storage. In case of any failure, the stable log and the database states are used to create a consistent database state.
3) 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. The checkpoint helps in recovery, as in case of a failure all the committed transactions prior to the checkpoint are NOT to be redone. Only non-committed transactions at the time of checkpoint or transactions that started after the checkpoint are required to be REDONE or UNDONE based on the log.

Check Your Progress 2

1) The deferred database modification recovery algorithm postpones, as far as possible, the writing of updates into the physical database. Rather, the modifications are recorded in the log file, which is written into stable storage. In case of a failure, the log can be used to REDO the committed transactions. In this process UNDO operation is not performed.
2) The log of deferred database modification technique just stores the REDO information. UNDO information is not required as updates are performed in the main memory buffers only, while the immediate modification scheme maintains both the UNDO and REDO information in the log
3) Buffer management is important from the point of view of recovery, as it may determine the time taken in recovery. It is also used for implementing different recovery algorithms.
   Remote backup is very useful in the case of disaster recovery. It may also make the database available even if one site of the database fails.

Check Your Progress 3

1) The database system security may depend on the security of the Operating system, including the network security; and the application that uses the database and services that interact with the web server.
2) GRANT ALTER DATABASE TO UserName
3) REVOKE GRANT OPTION
   FOR <permissions> ON <table> FROM <user/role>
4) Data security is the protection of information that is maintained in the 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.

UNIT 12 QUERY PROCESSING AND EVALUATION

Structure **Page Nos.**

12.0 INTRODUCTION

The Query Language – SQL is one of the main reasons for the success of RDBMS. A userjust needs to write the query in SQL that is close to the English language and does not need to say how such a 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 introduces you to the complexity of query evaluation in DBMS.

12.1 OBJECTIVES

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

- Explain the measure of 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. *Figure 1* shows the steps of query processing.

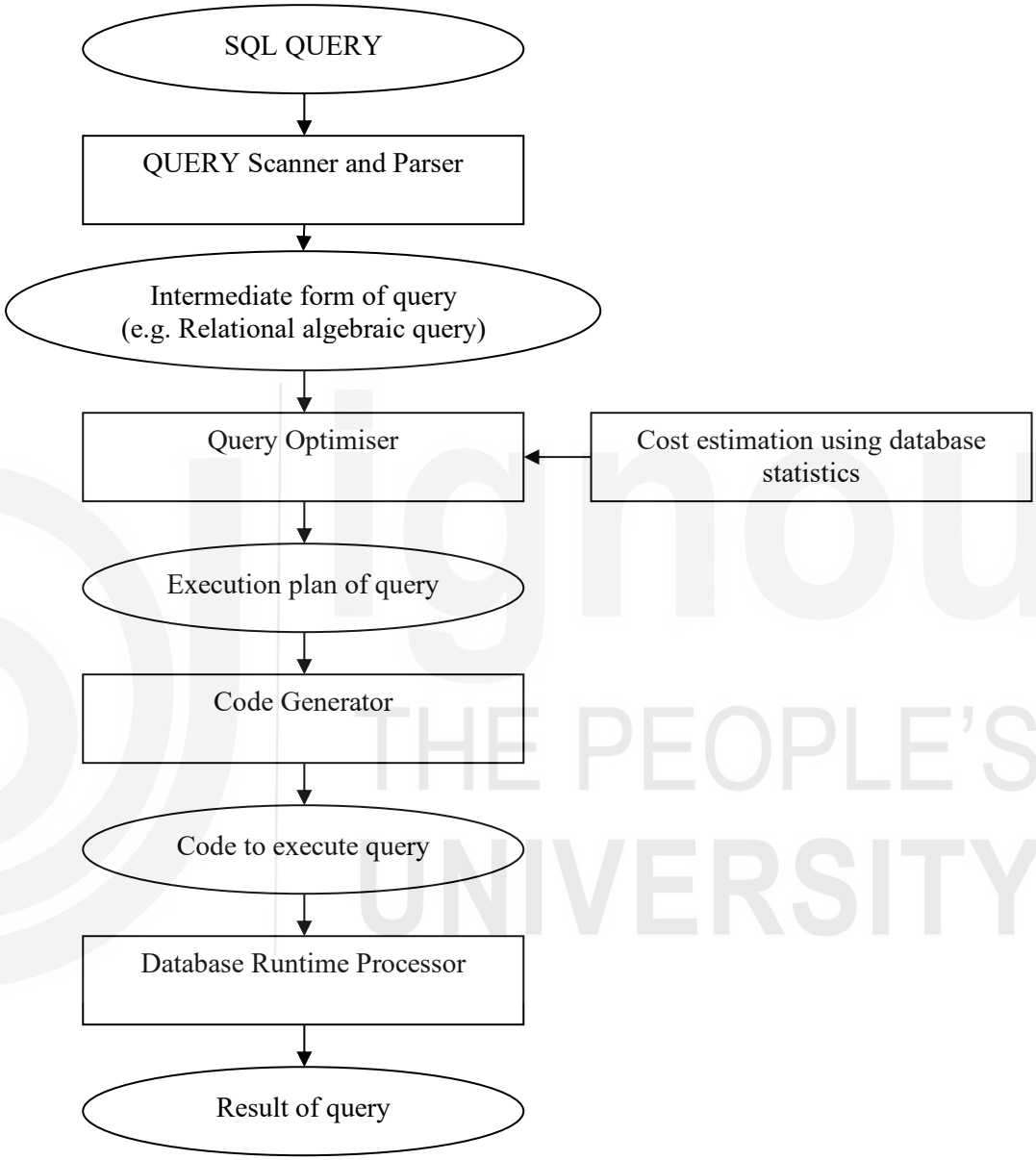


Figure 1: Query processing

In the first step, a query is scanned, parsed, validated and translated into an internal form. This internal form may be the relational algebra (an intermediate query form). The parser checks the syntax of the query and verifies the relations. Next, the query is optimised, and a query execution plan is generated, which is then compiled into a code that can be executed by the database runtime processor. The query processing involves the study of the following concepts:

- measures to find query cost.
- algorithms for evaluating relational algebraic operations.

- evaluating a complete expression using algorithms on individual operations.

12.2.1 Role of Relational Algebra in Query Optimisation

In order to optimise the evaluation of a query, first, you must define the query using relational algebra. A relational algebra expression may have many equivalent expressions. For example, the relational algebraic expression $\sigma_{(salary < 5000)}(\pi_{salary}(EMP))$ is equivalent to $\pi_{salary}(\sigma_{salary < 5000}(EMP))$. This may result in generating many alternative ways of evaluating the query.

Further, a relational algebraic expression can be evaluated in many different ways. A detailed evaluation strategy for an expression is known as an evaluation plan. For example, you can use an index on *salary* to find employees with *salary* < *5000*, or you can perform a complete relation scan and discard employees with *salary* ≥ *5000.* Both of these are separate evaluation plans. The basis of the selection of the best evaluation plan is the cost of these evaluation plans.

Query Optimisation: The query optimisation selects the query evaluation plan with the lowest cost among the equivalent query evaluation plans. Cost is estimated using the statistical information obtained from the database catalogue, viz., the number of tuples in each relation, the size of tuples, different values of an attribute, etc. The cost estimation is made on the basis of heuristic rules.

What is the basis for measuring the query cost? The next section addresses this question.

12.2.2 Using Statistics and Stored Size for Cost Estimation

The query cost is generally measured as the total elapsed time for answering the query. There are many factors that contribute to the cost in terms of elapsed time. These are the time of *disk accesses*, *CPU time*, and *data communication time on the network*. However, these times can be measured when the query is being executed. Therefore, you may use statistics, like the number of records, number of blocks, number of attributes, possible number of different values for each attribute, etc., to estimate the cost. However, disk access is typically the predominant cost as disk transfer is very slow. In addition, disk accesses are relatively easier to estimate. Therefore, the following disk access cost can be used to estimate the query cost:

Number of seeks × average-seek-time; and
Number of blocks read × average-block-read-time; and
Number of blocks written × average-block-write-time.

Please note that the cost of writing a block is higher than the cost of reading a block. This is due to the fact that the data is read back after being written to ensure that the write operation was successful. However, for the sake of simplicity, we will just use the *number of block transfers from the disk as the cost measure.* We will also ignore the difference in cost between sequential and random Input/Output, which **depends** on the search criteria, such as point/range query on an ordering field vs other fields; and the file structures: heap, sorted, hashed. In addition, the number of block transfer is also dependent on the use of indices, such as primary, clustering, secondary, B+ tree, etc. Other cost factors may include buffering, disk placement, view materialisation, overflow / free space management, etc. Please also note that CPU time and communication time are also not being considered for cost computation.

In the subsequent section, let us try to find the cost estimates of various operations. Please note that the stored size of a relation is an important cost estimate, if the entire database-related cost is to be estimated. The other statistics like the number of tuples or the number of attributes etc. are useful when only part of the database is to be considered for query processing.

12.3 COST OF SELECTION OPERATION

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

12.3.1 File scan

File scan algorithms locate and retrieve records that fulfil a selection condition in a file.

The following are the two basic file scan algorithms for selection operation:

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

   The cost of this algorithm (in terms of block transfer): This algorithm would require reading all the blocks of the file, as it must test all the records for the specific condition.

   $$\boldsymbol{Cost}_{\boldsymbol{To\ find\ reords\ \ that\ match\ a\ given\ criteria}} = Size\ of\ database\ in\ terms\ of\ Number\ of\ blocks = N_b.$$

   $$\boldsymbol{Cost}_{\boldsymbol{Finding\ a\ specific\ value\ of\ key\ attribute}} = Average\ number\ of\ block\ transfer\ for\ locating\ the\ value\ (on\ an\ average\ half\ of\ the\ file\ needs\ to\ be\ traversed)\ so\ the\ cost\ is = N_b/2.$$

   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 an attribute on which the file is ordered. Assume that the blocks of a relation are stored continuously then, the cost can be estimated as:

   $$Cost_{binary\ search} = Cost_{Block\ based\ Binary\ search\ to\ find\ first\ tuple\ that\ matches\ the\ search\ criteria} + Cost_{fetching\ contiguous\ blocks\ \ till\ the\ records\ keep\ matching\ the\ criteria}$$

   $$= \lceil log_2\ (N_b) \rceil + \frac{Mean\ of\ the\ number\ of\ tuples\ with\ similar\ attribute\ values}{Size\ of\ a\ Block\ of\ data\ in\ terms\ of Number\ of\ tuples}$$

You may observe that the cost computation is based on the database statistics.

12.3.2 Index scan

The index scan can be used for cases where the database contains an index on an attribute set that forms the search key.

1) (a) *Scanning for equality condition on a Primary index:* These kinds of searches try to find a specific key value using the primary index of a database system. Since the search criteria include equality on the primary key, therefore, the output of this search would be just a single record or no record at all. The cost of the scan is defined as:

   Cost = The depth traversed in the index to locate the block pointer +
   1 (for transfer of block consisting of desired primary key value).

   (b) *Hash key:* It retrieves a single block directly, thus, the cost in the hash key organisation is given as:

   =Block transfer needed for finding hash target +1

2) *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 backwards as the condition may be, displaying all the records. Thus, the cost, in this case, would be:

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

Please note you can roughly compute the number of blocks satisfying the condition as:

$$Number\ of\ attribte\ values\ that\ satisfy\ the\ condition \times average\ number\ of\ tuples\ per\ attribute\ value / blocking\ factor\ of\ the\ relation.$$

3) (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 the search key is not a candidate key.

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

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 the comparison on *secondary index value* > *value searched*, the index is used to find the first index entry which is greater than the *value searched*, thereafter, the indexed is scanned sequentially till the end, finding the pointers to records.

For the ≤ type query, just scan the leaf pages of the index to find the pointers to the records until the first entry that satisfies the condition is found. Thereafter, the index can be scanned till the records satisfy the condition to obtain pointers to the records.

You may please note that after you have found the pointers to the records using the index scan, retrieving those pointed records may require one block transfer for each record. Please note that linear file scans may be cheaper if many records are to be fetched.

12.3.3 Implementation of Complex Selections

Conjunction: Conjunction is basically a set of AND conditions.

Conjunctive selection using one index: In such case, select any algorithm given earlier on one or more conditions and then test remaining conditions on the selected tuples after fetching them into the memory buffer.

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

Disjunction: Disjunctions are basically a set of OR conditions.

Disjunction using the union of identifiers is applicable if *all* conditions have available indices, otherwise, use linear scan. Use the 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 an attribute, which is being negated, then find the satisfying records using the index and fetch them from the file.

12.4 COST OF SORTING

This section introduces the cost of query evaluation when it requires sorting of records. 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 and 2 may lead to one block access per tuple).

3) Techniques like *quicksort* can be used for relations that fit in the memory.

4) *External sort-merge* is a good choice for relations that do not fit in the memory.

Once you decide on the sorting technique, you can find the cost of these algorithms to find the sorted file. You may refer to further readings for more details.

☞ Check Your Progress 1

1) What are the basic steps in query processing?

……

……

2) How can the cost of a query be measured?

……

……

3) What are the various methods adopted for performing selection operation?

……

……

12.5 COST OF JOIN OPERATION

There are several algorithms that can be used to implement joins:

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

The choice of join algorithm is based on the cost estimates. We will elaborate on only a few of these algorithms in this section. The following relations and related statistics will be used to elaborate those algorithms.

MARKS (enrollno, subjectcode, marks): 20000 rows, 500 blocks
STUDENT (enrollno, name, dob): 5000 rows, 200 blocks.

12.5.1 Block Nested-Loop Join

In this join approach, a complete block of the outer loop is joined with the complete block of the inner loop. We have chosen STUDENT as the outer relation, as it is smaller in size and, therefore, will result in a smaller number of overall block transfers.

The algorithm for this may be written as:

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

In the worst case, when only one block of both the relations can be stored in RAM, the estimation of block accesses is:

= Number of Blocks of outer relation (STUDENT) × Number of blocks of inner relation (MARKS) + Number of blocks of outer relation (STUDENT).

= 200×500 + 200 = 100200

In the best case, when the smaller relation can be completely in RAM, the number of block accesses would be = Blocks of STUDENT + Blocks of MARKS

= 200 + 500 = 700

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 terms of the number of blocks.
- Use one buffer block to buffer the inner relation.
- Use one buffer block to buffer the output.

This method minimises the number of iterations.

12.5.2 Merge-Join

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

1) Sort both relations on the joining attribute (if not already sorted).
2) Merge the sorted relations to Join them. In this step, every pair with the same value on the joining attribute must be matched.

For example, consider the instances of STUDENT and MARKS relations, as given in Figure 2.

STUDENT

enrolno	Name	-----
1001	Ajay	……
1002	Aman	…...
1005	Rakesh	…...
1100	Raman	…...

Block 1 of STUDENT relation

MARKS

enrolno	subjectcode	Marks
1001	MCS-211	55
1001	MCS-212	75
1002	MCS-212	90
1005	MCS-215	75

Block 1of MARKS relation

Figure 2: Sample Relations for Computing Join

Computation of the number of block accesses:

Join operation on enrolment number would be as follows:

i) The enrolment number 1001 of STUDENT relation will join with two tuples of MARKS relation as:

1001 …. 1001 MCS-211
1001 …. 1001 MCS-212

ii) Joining on the key 1001 will be over as soon as you find 1002 in MARKS relations, as MARKS relation is also sorted on enrolno. Thus, the join will continue as Merge-Join operations and the following tuples will be output after the output of the first two tuples for 1001.

This will be followed by output.

1002 ….. 1002 MCS-212
1005 ….. 1005 MCS-215

You may observe that the cost of the Merge-Join operation can be computed based on the assumption that a block that is part of the merge is read only once. The basic underlying assumption is that the main memory is sufficiently large to accommodate all the tuples of a specific attribute join value. Therefore, the number of block accesses for Merge-Join is:

=Blocks of STUDENT + Blocks of MARKS + the cost of sorting on enrolno (if relations are unsorted)

12.5.3 Hash-Join

Hash-join can be performed for both the equijoin and natural join operations. A hash function *h* is applied on joining attributes to partition tuples of both relations. In the case of STUDENT and MARKS relations, the hash function *h* maps joining attribute (enrolno in our example case) 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 the mod 5 function for hashing, therefore, *n* = 05.

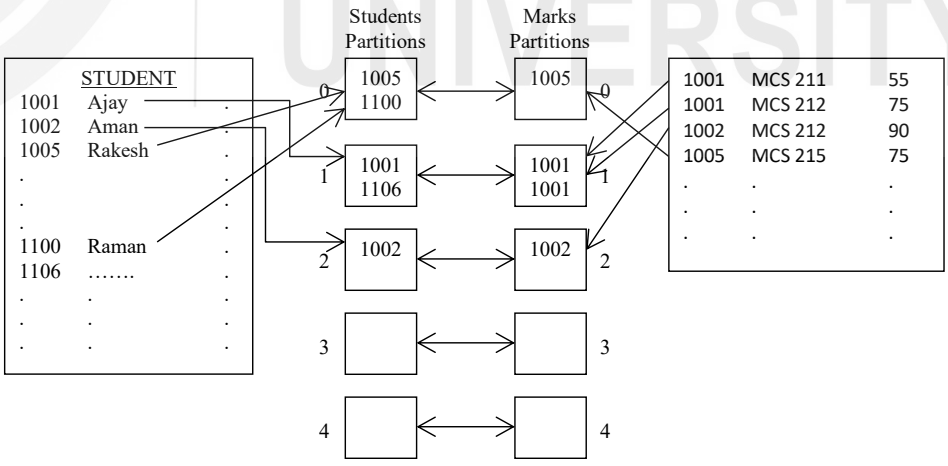


Figure 3: A hash-join example

Once the partition tables of STUDENT and MARKS are made on the enrolno, 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 an equivalent partition and, thus, can be joined easily. For example, STUDENT partition 1 consists of tuples of STUDENT with enrolno 1001, 1006; whereas

MARKS partition 1 consists of marks of the student enrolno 1001 in two different subjects. The join can now simply be performed for enrolno 1001. There is no joining MARKS tuple for STUDENT 1006.

Algorithm for Hash-Join

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

- Partition the relation *r* and *s* using the same hash function *h*. (When partitioning a relation, one block of memory is reserved as the output buffer for each partition). It may be noted that the tuples in i^{th} partition of *r* may join only with the tuples of i^{th} partition of *s*.

- For each partition s_i of *s,* load the partition into memory and build an in-memory hash index using the joining attribute. Why is this hash index needed? As several joining attributes may get mapped into the same partition. For example, partition 0 and partition 1 in Figure 3.
- Read the partition *i* of *r* into memory block by block. For each tuple in r_i, find the tuples s_i in the hash index of s_i , which would join with that tuple in r_i. Output the joined results.

The value *n* (the number of partitions) and the hash function *h are* chosen in such a manner that each s_i should fit into the memory. Typically, *n* is chosen as:

[*the Number of blocks of s/Number of memory buffers*] × *f*
where *f* is typically around 1.2.

The partition of *r* can be large, as r_i need not fit in memory.

You may refer to the further readings for more details on hash join.

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 to partitions, so
cost1 = 2 (*blocks of r + blocks of s*).

(ii) The cost of performing the hash-join using the hash index on *s* will require at least one block transfer for reading the partitions.
cost2 = (*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 low in comparison to *cost1* and *cost2*.
Thus, the total *cost* = *cost1* + *cost2*
= 3 (*blocks of r + blocks of s*)

Even if *s* is recursively partitioned, *hash-table overflow* can occur, i.e., some partition s_i may not fit in the memory. This may happen if many tuples in *s* have the same value for join attributes or the chosen hash function is bad. Partitioning is said to be *skewed* if some partitions have significantly more tuples than others. This is the overflow condition. The overflow can be handled in a variety of ways; however, they are beyond the scope of this unit.

Let us explain the hash join and its cost for the natural join $STUDENT \bowtie MARKS$
Assume a memory size of 25 blocks $\Rightarrow M$=25.

SELECT *s* as STUDENT as it has a smaller number of blocks (200 blocks) and *r* as MARKS (500 blocks).

Number of partitions to be created for STUDENT = (*blocks of* STUDENT/*M*) × 1.2
= (200/25) × 1.2 = 9.6 ≈10

Thus, the STUDENT relation will be partitioned into 10 partitions of 20 blocks each. MARKS will also have 10 partitions of 50 blocks each. The 25 buffers will be used as:

20 blocks for one complete partition of STUDENT,
01 block will be used for input of MARKS partitions, and
The remaining 4 blocks may be used for storing the results.

The total cost = 3(200+500) = 2100 as no recursive partitioning is needed.

12.6 OTHER OPERATIONS

There are many other operations that are performed in database systems. Let us introduce these operations 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 the deletion of duplicates during generation as well as at intermediate merge steps in an 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 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. For count, min, max, and sum, you may add up the aggregates. For calculating the average, take the sum of the aggregates and count the number of aggregates; and then divide the sum with the count at the end.

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

Using the Hashing:

1) Partition both relations using the same hash function, thereby creating partitions consisting of tuples of r and s, as:

   $r_0, r_1 \ldots r_{n-1}$ and $s_0, s_1 \ldots s_{n-1}$

2) Process each partition i as follows:

   Using a different hashing function, build an in-memory hash index on r_i after it is brought into the memory.

   $r \cup s$: Add tuples in s_i to the hash index if they are not already in it. Output the hash index.

   $r \cap s$: For each tuple s_i of the relation s, check if it is in the hash index, if yes, output this tuple.

   $r - s$: For each tuple in s_i of the relation s, if the similar r_j is part of the hash index, delete r_j from the hash index. Once the entire s is processed, output the remaining hash index tuples.

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) What is the cost of Hash-Join?

.....

.....

3) What are the other operations that may be part of query evaluation?

.....

.....

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 result of the student(s) whose phone number is '1129250025', the following SQL query may be written:

```
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 would be:

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

This is a very good internal representation; however, it may be a good idea to represent the relational algebraic expression as 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:

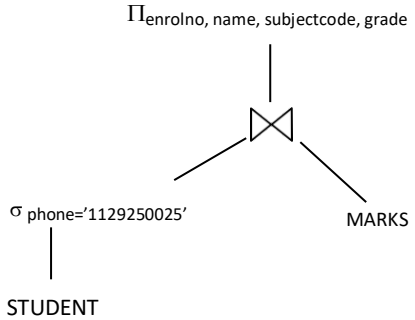


Figure 4: A Sample query tree

12.7.1 Evaluating a Query Tree.

In this section, let us examine the methods for evaluating a query expression that is expressed as a query tree. In general, we use two methods:

- Materialisation
- Pipelining.

Materialisation: Evaluates a relational algebraic expression in a bottom-up approach by explicitly generating and storing the results of each operation of expression. For example, for *Figure 4*, first selection operation on the STUDENT relation would be performed. Next, the result of the selection operation would be joined with the MARKS relation. Finally, the projection operation will be performed. Materialised evaluation is always possible even though the cost of writing/reading results to/from disk can be quite high.

Pipelining: Evaluates 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 (or materialise) results instead, it passes the tuples of the selection operation directly to the join operation. Similarly, it does not store the results of the join operation and passes the tuples of join operations directly to the projection operation. Thus, there is no need to store temporary relations on a disk for each operation. Pipelining may not always be possible or easy if sort or hash-join operations are used.

The pipelining execution method may involve a buffer, which is being filled by the result tuples of a lower-level operation, while records may be picked up from the buffer by a higher-level operation.

12.7.2 Evaluating Complex Joins

When an expression involves three relations, then you 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, and *join* the result with SUBJECTS.

Strategy 2: Compute MARKS ⋈ SUBJECTS first, and then *join* the result with STUDENT.

Strategy 3: Perform the pair of joins at the same time. This can be done by building an index of enrolno in STUDENT and on subjectcode 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 evaluation in the earlier section of this unit, but can something be done during these two stages that optimise the query evaluation? This section deals with this process in detail.

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

1) Generating logically equivalent expressions using **equivalence rules**
2) Generate alternative query plans.
3) Choose the cheapest plan based on the 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 need to estimate the cost of operations and statistical information about relations. For example, the number of tuples, the number of distinct values for an attribute, etc., helps estimate the size of the intermediate results, and information like available indices may help in estimating the cost of complex expressions. Let us discuss all the steps in query-evaluation plan development in more detail.

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 (the order of tuples is irrelevant). "Appendix-A", given at the end of this unit, lists the rules that may be used to generate equivalent relational expressions. However, the rules given in Appendix A are too general, and a few heuristics rules generated from these rules can be used to transform the relational expressions. These rules are:

(1) Combine a cascade of selections into a conjunction and test all the predicates on the tuples in a single iteration:
Expression like: $\sigma_{\theta 2} (\sigma_{\theta 1} (E))$
Can be converted to: $\sigma_{\theta 2 \wedge \theta 1} (E)$

(2) Combining a cascade of projections into a single outer projection (please note that the final projection would be the outer projection.
Expression like: $\pi_4 (\pi_3 (\ldots(E)))$
Can be converted to: $\pi_4 (E)$

(3) Commutate the selection and projection or vice-versa. This commutation may sometimes reduce query cost.

(4) Use associative or commutative rules for the Cartesian product or join operation to find various alternative paths for query evaluation.

(5) Move the selection and projection (projection may be expanded to include join condition) before Join operations. The selection and projection result in the reduction of the number of tuples and, therefore, may reduce the cost of joining.

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

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

STUDENT (enrolno, name, phone)
MARKS (enrolno, subjectcode, grade)
SUBJECT (subjectcode, sname)

Example 1: 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:

$\pi_{\text{enrolno, name, grade}} (\sigma_{(\text{sname = 'DBMS' } \wedge \text{ grade = 'A'})} ((\text{STUDENT} \bowtie \text{MARKS}) \bowtie \text{SUBJECT})$

The query tree for this would be:

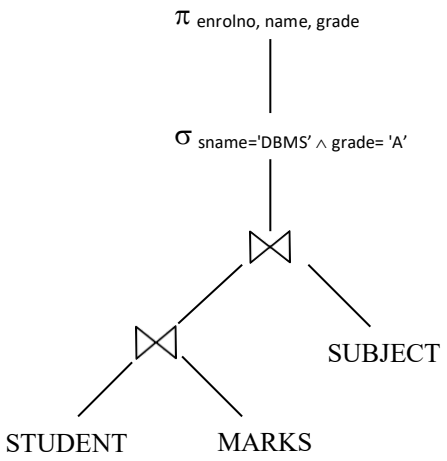


Figure 5: The query tree of example 1

As per the suggested rules, the selection condition may be moved before the join operation. The selection condition given in *Figure 5* above is: *sname = 'DBMS' and grade = 'A'*. Both of these conditions belong to different tables, as *sname* is available only in the SUBJECT table and *grade* in the MARKS table. Thus, the selection conditions will be mapped accordingly, as shown in *Figure 6*. Thus, the equivalent expression will be:

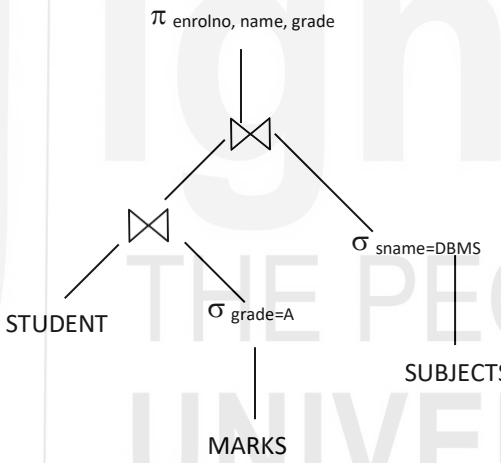


Figure 6: A modified tree

Further, the expected size of SUBJECT and MARKS after selection will be small, so it may be a good idea to join MARKS with SUBJECT first, and thereafter, the resultant relation is joined with the STUDENT relation. Hence, the associative law of JOIN may be applied.

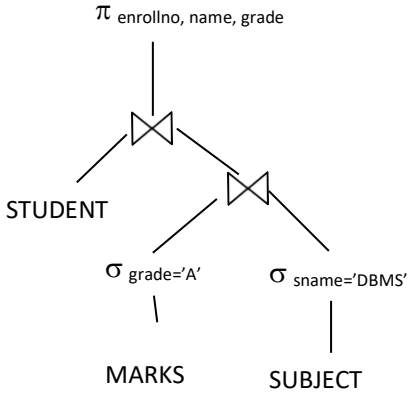


Figure 7: Modified query tree using associativity of join.

Even moving the projection before possible join, wherever possible, may optimise the query processing, as projection may also reduce the size of the intermediate result. Thus, you can move projection of outer join (refer to Figure 7) to inner join (Refer Figure 8).

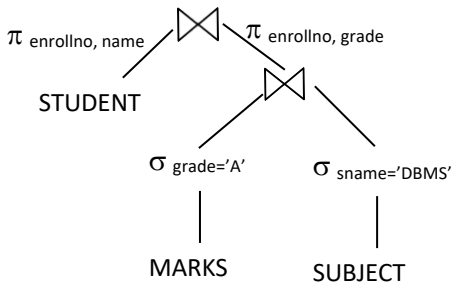


Figure 8: Moving the projection.

The final equivalent relation algebraic query expression of query of example 1 is:

$$(\pi_{\text{enrolno, name}}(\text{STUDENT})) \bowtie (\pi_{\text{enrolno, grade}}((\sigma_{\text{(grade ='A')}}\text{MARKS}) \bowtie (\sigma_{\text{(sname = 'DBMS')}}\text{SUBJECT})))$$

Alternative Query Expressions

A relational algebraic query can be optimised by the query optimiser, which creates several relational algebraic expressions that are equivalent to the query expression using the equivalence rules of relational algebra. One of the techniques to do so is to keep applying the equivalence rules to the relational algebraic query to generate a new set of expressions. However, this is a very time-consuming process. Therefore, in general, a set of heuristics, which are based on certain criteria like "apply those transformation rules that result in a reduction in the size of intermediate results", can be used with the objective to produce more efficient equivalent query expressions.

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 operationand how the execution of the operation is coordinated. For example, *Figure 9* shows the query tree with an evaluation plan.

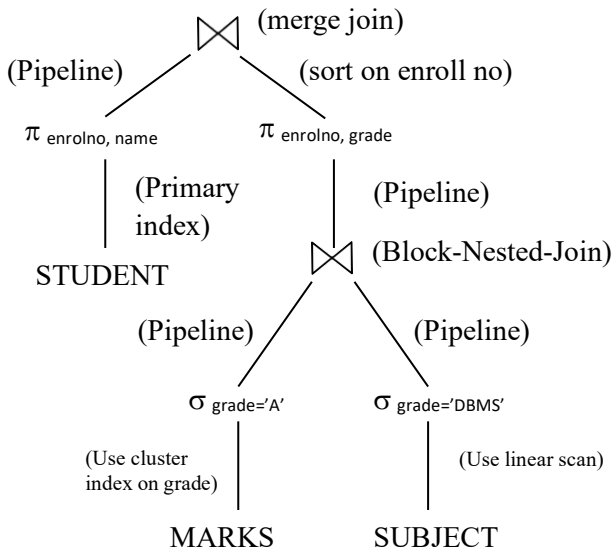


Figure 9: A sample query evaluation plan

12.8.3 Choosing an Optimal Evaluation Plan

The output of the previous step of query evaluation is the number of query evaluation plans. Which of these plans should be chosen for the final query evaluation? This decision is normally based on the database statistics. In addition, it is not necessary that individual best plans for each part of query evaluation may make the best query evaluation algorithm. For example, to join two relations, hash join is computationally less expensive than the merge join; however, the merge join produces sorted output. Thus, for the cases where sorted output may be useful for further processing of part results, merge join may be preferred. Similarly, the use of the nested loop method of joining may allow pipelining of the results of the join operation for further operations. In general, for choosing an optimal evaluation plan, you may perform cost-based optimisation to choose an optimal query evaluation plan.

12.8.4 Cost and Storage-Based Query 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 section 12.3 with respect to the method and operation (JOIN, SELECT, etc.).

Storage and Query Optimisation

Cost calculations are primarily based on disk access; thus, storage has an important role to play in cost computation. In addition, some of the operations also require intermediate storage; thus, the cost is further enhanced in such cases. The cost of finding an optimal query plan is offset by the savings in terms of the query-execution time, particularly by reducing the number of slow disk accesses.

12.9 VIEWS AND QUERY PROCESSING

A view is defined as a query. 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.

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 computing the view again.

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 re-computation on every update. A better option is to use incremental view maintenance, i.e., where only the affected part of the view is modified. View maintenance, in general, can be performed using triggers, which can be written for any data manipulation operation on the relations that are part of a view definition.

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 you 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) List the methods used for the evaluation of expressions.

……

……

2) How do you define cost-based optimisation?

……

……

……

3) Define the term "Evaluation plan".

……

……

……

12.10 SUMMARY

This Unit introduces you to the basic concepts of query processing and evaluation. A query is to be represented in a standard form before it can be processed. In general, a query is evaluated after representing it using relational algebra. Thereafter, several query evaluation plans are generated using query transformations and an optimal plans is chosen for query evaluation. To find the cost of a query evaluation plan, you may use database statistics. The unit also defines various algorithms and the cost of these algorithms for evaluating various operations like select, project, join etc. You may refer to database textbooks for more details on query evaluation.

12.11 SOLUTIONS/ANSWERS

Check Your Progress 1

1) The steps of query evaluation are as follows:
   a. In the first step, query scanning, parsing and validating is done.
   b. Next, translate the query into a relational algebraic expression.
   c. Next, the syntax is checked along with the names of the relations.

d. Finally, the optimal query evaluation plan is executed and the
e. answers to the query are returned.

2) Query cost is measured by considering the following activities:
- Number of disk-head seeks.
- Number of blocks of tables.
- Number of blocks of the results to be written

Please note that the cost measures above are based on the *number of disk blocks to memory buffer transfe*r. We generally ignore the difference in cost between sequential and random I/O, CPU and communication costs.

3) The selection operation can be performed in several ways, such as:
Using File Scan: Using linear search or using the Binary search
Using Index Scan: Using the primary index (for equality) or the Hash key. You can also use a clustering index or a secondary index.

Check Your Progress 2

1) For each tuple in block B_r of r {
For each tuple in block B_s of s {
Test pair(t_i,s_i) to see if they satisfy the join condition
If they do, add the joined tuple to the result.
};
};

2) The cost of Hash-join is.
3(*Blocks of r + blocks of s*)

3) Some of the other operations in query evaluation are:
- Duplicate elimination
- Projection
- Aggregate functions.
- Set operations.

Check Your Progress 3

1) Methods used for evaluation of expressions:

(a) Materialisation
(b) Pipelining

2) Cost based optimisation consists of the following steps:

(a) Generating logically equivalent expressions using equivalence rules
(b) Generate alternative query plans.
(c) Choose the cheapest plan based on the estimated cost.

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

"APPENDIX-A"

Equivalence Rules

1) The conjunctive selection operations can be equated to a sequence of individual selections. 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 conditions:

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

7) The selection operation distributes over the theta join operation under conditions when all the attributes in the 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 difference is not commutative.

$E1 \cup E2 = E2 \cup E1$ 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.

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