Database transaction

A **database transaction** is a unit of work performed against a database management system or similar system that is treated in a coherent and reliable way independent of other transactions.

A transaction must satisfy the following: -

- It should either be done entirely or not at all
- If it succeeds, the effects of write operations persist (commit); if it fails, no effects of write operations persist (abort)
- These guarantees are made despite concurrent activity in the system, and despite failures that may occur

A **Transaction** can also be defined as an action or a series of actions, carried out by a single user or application program which reads or updates the contents of the database. It is a logical unit of work on the database.

A database transaction, by definition, must be **atomic, consistent, isolated and durable.** These properties of database transactions are often referred to by the acronym **ACID.**

Transactions provide an "all-or-nothing" proposition stating that work units performed in a database must be completed in their entirety or take no effect whatsoever. Further, transactions must be isolated from other transactions, results must conform to existing constraints in the database and transactions that complete successfully must be committed to durable storage.

In some systems, transactions are also called **LUWs** for *Logical Units of Work.*

Purpose of transaction

Databases and other data stores in which the integrity of data is of great importance often include the ability to handle transactions to ensure the integrity of data is maintained. A single transaction is composed of one or more independent units of work, each reading and/or writing information to a database or other data store. When this happens it is often important to ensure that the database or data store is left in a consistent state.

The concept of transactions is often demonstrated via double-entry accounting systems. In double-entry accounting every debit requires an associated credit be recorded. If you run a small kiosk in buruburu and you bought Kshs. 5000/= worth of Unga to add to your stock a transactional double-entry accounting system would record the following two entries to cover the single transaction:

1) Deduct Kshs. 5000/= from "cash at hand".
2) Add Kshs. 5000/= to the value of stock.

In a transactional system **both** entries would be made, or **both** entries would fail. By treating the recording of multiple entries as an atomic transactional unit of work you maintain the integrity of the data recorded. In other words, you will not end up with a situation in which a debit is recorded but no associated credit is recorded, or vice versa.

Properties of Transactions

There are four properties that all transactions should posses. The four basic properties, also called **ACID** properties of a transaction are Atomicity, Consistency, !solation and Durability.

Atomicity-Atomicity refers to the ability of the DBMS to guarantee that either all of the tasks of a transaction are performed or none of them are. For example, the transfer of funds can be completed or it can fail for a multitude of reasons, but atomicity guarantees that one account won't be debited if the other is not credited. Atomicity is the "all nothing" property. It is the responsibility of the recovery subsystem of the DBMS to ensure atomicity.

Consistency - Consistency property ensures that the database remains in a consistent state before the start of the transaction and after the transaction is over (whether successful or not). A transaction must transform the database from one consistent state to another consistent state. It is the responsibility of both the DBMS and the application developers to ensure consistency. The DBMS can ensure consistency by enforcing all the constraints that have been specified and the programmers must ensure the correctness of the code.

Isolation - Isolation refers to the ability of the application to make operations in a transaction appear isolated from all other operations. This means that no operation outside the transaction can ever see the data in an intermediate state; for example, a bank manager can see the transferred funds on one account or the other, but never on both-even if he runs his query while the transfer is still being processed. In other words, the partial effects of incomplete transactions should not be visible to other transactions. It is the responsibility of the concurrency control subsystem to ensure isolation.

Durability - The effects of a successfully completed (committed) transaction are permanently recorded in the database and must not be lost because of a subsequent failure. Durability refers to the guarantee that once the user has been notified of success, the transaction will persist, and not be undone. This means it will survive system failure, and that the database system has checked the integrity constraints and won't need to abort the transaction. Many databases implement durability by writing all transactions into a log that can be played back to recreate the system state right before the failure. A transaction can only be deemed committed after it is safely in the log. It is the responsibility of the recovery subsystem to ensure durability.

Concurrency Control

Definition of Concurrency

Concurrency in terms of databases means allowing multiple users to access the data contained within a database at the same time. If concurrent access is not managed by the Databas Management System (DBMS) so that simultaneous operations don't interfere with one another problems can occur when various transactions interleave, resulting in an inconsistent database.

Concurrency 1s achieved by the DBMS, which interleaves actions (reads/writes of database objects) of various transactions. Each transaction must leave the database in a consistent state if the database is consistent when the transaction begins.

Concurrent execution of user programs is essential for good DBMS performance. Because disk accesses are frequent, and relatively slow, it is important to keep the CPU busy by working on several user programs concurrently. Interleaving actions of different user programs can lead to inconsistency: e.g., check is cleared while account balance is being computed. DBMS ensures such problems don't arise: users can pretend they are using a single-user system.

Definition of Concurrency Control

Concurrency control is the process of managing simultaneous operations on the database without having them interfere with one another.

Concurrency control is needed to handle problems that can occur when transactions execute concurrently.

The following are the concurrency issues (Potential problems caused by concurrency):-

1) Lost Update Problem - An update to an object by some transaction is overwritten by another interleaved transaction without knowledge of the initial update.
2) Uncommitted Dependency - A transaction reads an object/data updated by another transaction that later fails.
3) Inconsistent Analysis Problem - Occurs when a transaction reads several values from the database but a second transaction updates some of them during the execution of the first.

Reasons (Need) for allowing Concurrent Execution

1) Improved throughput and resource utilization.

(Throughput- Number of Transactions executed per unit ohime.)

The CPU and the Disk can operate in parallel. While one Transaction is performing an 1/0 [Input/Output] operation, the other transaction can be running in the CPU. When the one running in the CPU reaches an I/0 operation, the CPU suspends it and executes commands of the other transaction until it reaches another 1/0 operation. In this way, the operations of the two transactions are **interleaved** to achieve concurrent execution. This increases the CPU and Disk (I/0 subsystem) utilization ultimately increasing throughput.

2) Reduced waiting time.

In a serial processing a short Transaction may have to wait for a long transaction to complete. Concurrent execution reduces the average response time; the average time for a Transaction to be completed.

Potential problems caused by concurrency

As listed above, the potential problems caused by concurrency are: Lost update problem, Uncommitted dependency and Inconsistent analysis problem. These issues are discussed further below.

Lost Update Problem.

An apparently .successfully completed update operation by one user can be overridden by another user. This is known as the lost update problem.

Example 1

Time	T,	T2	R
t1	Read(R)		500
t2	Add 100 to R		500
t3		Read (R)	500
t4		Add 50 to R	500
ts	Write(R)		600
t6		Write(R)	550

Transactions T1's update is lost.

Example 2

Time	T,	T2	balx
t1		begin transaction	10,000
12	begin transaction	Read(balx)	10,000
t3	Read(balx)	balx = balx + 5000	10,000
t4	balx = balx - 1000	Write(balx)	15,000
ts	Write(balx)	Commit	9,000
t6	commit		9,000

In the above example (example 2), transaction T_1 is executing concurrently with transaction T_2. T_1 is withdrawing kshs. 1000/= from an account with balance balx, initially kshs. 10,000/= and $T_2$is depositing kshs. 5000/= into the same account. If these transactions are executed serially, one after the other with no interleaving of operations, the final balance would be kshs. 14,000/= no matter what transaction is performed first.

Transaction $T_1$and T_2 start at nearly the same time, and both read the balance as kshs. 10,000/=. T_2 increases balx by 5,000 to kshs. 15,000/= and stores the update in the database. Meanwhile, transaction T1decrements it's copy of balx by 1000 to kshs. 9,000/= and stores this value in the database, overwriting the previous update, and thereby 'loosing' the kshs. 5,000/= previously added to the balance.

The loss of T2's update is avoided by preventing T1from reading the value of balx until after T_2' s update has been completed.

Uncommitted Dependency Problem (Dirty Read Problem).

The uncommitted dependency problem occurs when one transaction is allowed to see the intermediate results of another transaction before it has committed. The transaction whose intermediate results have been read might abort and rolled back (values restored back to their original values).

Example 1

Time	T3	T4	R
t1	Read(R)		800
t2	Subtract 200 from R		800
t3	Write(R)		600
t4	---	Read (R)	600
ts	Rollback	Add 50 to R	800

- 1... – t6_ _,	..._1_w_r_it_e( )	_,	6_5_0_ ___,
. t7	. Commit		650

Example 2

Time	T3	T4	balx
t1		begin transaction	2000
ti		Read(balx)	2000
t3		balx= balx+ 1000	2000
t4	begin transaction	Write(balx)	3000
t5	Read(balx)	---	3000
t6	balx= balx- 500	Rollback	2000
t7	Write(balx)		2500
ts	commit		2500

In the above example, transaction T4 updates balx to 3000 but it aborts the transaction so that balx is restored to it's original value of 2000. However, by this time, transaction T3 has already read the new value of balx (3000) and is using this value as the basis of the kshs. 500/= reduction, giving a new incorrect balance of kshs. 2500 instead of kshs. 1500/=. The value of balx read by T3 is called *dirty data*, giving rise to the alternative name, *the dirty read problem.*

The reason for the rollback is not important; it maybe that the transaction was in error, perhaps crediting the wrong account. The effect is the assumption by T3 that T/s update completed successfully although the update was subsequently rolled back. This problem is avoided by preventing T3 from reading balx until after the decision has been made to either commit or abort T/s effects.

Inconsistent Analysis Problem

Occurs when a transaction reads several values from the database but a second transaction updates some of them during the execution of the first. For example, a transaction that is summarizing data in the database (for example totaling balances) will obtain inaccurate results if, while it is executing, other transactions are updating the database.

Time	Ts	T6	balx	balv	bal_2	Sum
t1		begin transaction	10,000	5,000	2,500	
t2	begin transaction	Sum=0	10,000	5,000	2,500	0
t3	Read(balx)	Read(balx)	10,000	5,000	2,500	0
t4	balx= balx- 1000	Sum= Sum+ balx	10,000	5,000	2,500	10,000
ts	Write(balx)	Read(balv)	9,000	5,000	2,500	10,000
t6	Read(balz)	Sum= Sum+ balv	9,000	5,000	2,500	15,000
t7	bal_2 = balz + 1000		9,000	5,000	2,500	15,000
ts	Write(bal_2)		9,000	5,000	3,500	15,000
t9	Commit	Read(balz)	9,000	5,000	3,500	15,000
t10		Sum= Sum+ bal_2	9,000	5,000	3,500	18,500
t1I		Commit	9,000	5,000	3,500	18,500

Serializability

In databases and transaction processing, a schedule (transaction history) is **serializable,** i.e. it has the **Serializability** property, if its outcome (the resulting database state, the values of the database's data) is equal to the outcome of its transactions executed sequentially without overlapping in time. Transactions are normally executed concurrently (they overlap), since this is the most efficient way. Serializability is the major correctness criterion for concurrent transactions' executions. It is considered the highest level of isolation between transactions, and plays an essential role in concurrency control. As such it is supported in all general purpose database systems.

*A **schedule** is a list of actions, (i.e. reading, writing, aborting, committing), from a set of transactions. OR*

*A **schedule** is a sequence of operations by a set of concurrent transactions that preserves the order of the operations in each of the individual transactions.*

Correctness

Correctness - Serializability

Due to the fact that *Serializability* is the major criterion for the correctness of concurrent transactions' executions, it is supported in all general purpose database systems. The rationale behind it is the following:

> If each transaction is correct by itself, then any serial execution (no transaction overlap, but at any order) of these transactions is correct. As a result, any execution that is equivalent (in its outcome) to a serial execution is correct.

Schedules that are not serializable are likely to generate erroneous outcomes. Well known examples are with transactions that debit and credit accounts with money. If the related schedules are not serializable, then the total sum of money may not be preserved. Money could disappear, or be generated from nowhere.

Correctness - Recoverability

In all real systems transactions can abort, and serializability by itself is not sufficient for correctness. Schedules also need to possess the *recoverability* property. Recoverability means that committed transactions have not read data written by aborted transactions (whose effects do not exist in the resulting database states). While serializability is currently compromised in many applications, compromising recoverability would quickly violate the database's integrity, as well as that of transactions' results.

Recoverability - Transactions commit only after all transactions whose changes they read commit.
If a transaction Tl aborts, and a transaction T2 commits, but T2 relied on Tl, we have an unrecoverable schedule.

elaxing serializability

In many appli ations absolute correctness is not needed. For example, when retrieving a list of products according to specification, in most cases it does not matter much if a product, whose data was updated a short time ago, does not appear in the list, even if it meets the specification. It will typically appear in such a list when tried again a short time

l ter. Commercial databases provide concurrency control with a whole range of isolation levels which are in fact (controlled) serializability violations in order to achieve higher performance. Higher performance means better transaction execution rate and shorter transaction response time (transaction duration).

View and conflict serializability

Two major types of serializability exist: *view serializability*, and *conflict serializability*. Any schedule that is conflict-serializable is also view-serializable. Conflict serializability is widely utilized because it is easier to determine.

Conflict serializability is defined by equivalence to a serial schedule (no overlapping transactions) with the same transactions, such that both schedules have the same sets of respective chronologically-ordered pairs of conflicting operations (same precedence relations of respective conflicting operations).

Time	T1	T2		T1	T2		T1	T2
11	**Begin_ Transaction**			**Begin_ Transaction**			**Begin_ Transaction**	
12	Read(balx)			Read(balx)			Read(balx)	
t3	Write(balx)			Write(balx)			Write(balx)	
t4		**Begin_ Transaction**			**Begin_ Transaction**		Read(baly)	
ts		Read(balx)			Read(balx)		Write(balv)	
t6		Write(balx)		Read(balv)			**Commit**	
t7	Read(baly)				Write(balx)			**Begin_ Transaction**
tg	Write(balv)			Write(balv)				Read(balx)
t9	**Commit**			**Commit**				Write(balx)
t10		Read(balv)			Read(balv)			Read(balv)
t1 I		Write(balv)			Write(balv)			Write(balv)
112		**Commit**			**Commit**			**Commit**

Schedule S3 is a serial schedule and, since S_I and S_2 are equivalent to S_3, S_I and S_2 are serializable schedules. This type of serializability is known as **conflict serializability.** A conflict serializable schedule orders any conflicting operations in the same way as some serial execution.

View serializability of a schedule is defined by equivalence to a serial schedule (no overlapping transactions) with the same transactions, such that respective transactions in the two schedules read and write the same data values ("view" the same data values).
Two schedules S_I and S2consisting of the same operations from n transactions T_1, T_2, ... , T_n are view equivalent if the following three conditions hold:

- For each data item x, if transaction Ti reads the initial value of x in schedule S_1, then transaction Ti must also read the initial value of x in schedule S_2.
- For each read operation on data item x by transaction Ti in schedule S_1, if the value read by Ti has been written by transaction Tj, then transaction Ti must also read the value of x produced by transaction Tj in schedule S2.

- _For each data item x, if the last write operation on x was performed by transaction Ti in schedule S_1, the same transaction must perform the final write on data item x in schedule S2

*A **recoverable schedule** is a schedule whereby for each pair of transactions T; and l';* , *if 1j reads a data item previously written by T;, then the commit operation of T; precedes the commit operation of 1j.*

Reading Exercise

Read and make notes on:-

a) Testing for conflict serializability.
b) Testing for view serializability.

Concurrency Control Techniques

Serializability is achieved in several ways. There are two main concurrency control techniques that allow transactions to execute safely in parallel subject to certain constraints: locking and timestarnp methods.

Locking and time stamping are essentially **conservative** (or **pessimistic)** approaches in that they cause transactions to be delayed in case they conflict with other transactions some time in the future. **Optimistic**-methods on the other hand are based on the premise that conflict is rare so they allow transactions to proceed unsynchronized and only check for conflicts at the end, when a transaction commits.

Locking Methods

Locking - A procedure used to control concurrent access to data. When one transaction is accessing the database, a lock may deny access to other transactions to prevent incorrect results.

Locking methods are the most widely used approach to ensure serializabilty of concurrent transactions. There are several variations but all share the same fundamental characteristic; A transaction must claim a **shared** (read) or **exclusive** (write) lock on a data item before the corresponding database read or write operation. The **lock** prevents another transaction from modifying the item or even reading it, in the case of an exclusive lock. Data items of various sizes ranging from an entire database down to a field may be locked. The size of the item determines the fineness or **granularity,** of the lock. The actual lock may be implemented by setting a bit in the data item to indicate that portion of the database is locked, or by keeping a list of locked parts of the database or by other means.

The following are basic rules for locking.

Shared lock - If a transaction has a shared lock on a data item it can read the item but not update it.

Exclusive lock - If a transaction has an exclusive lock on a data item, it can both read and update the item.

Since read operations cannot conflict, it is permissible for more than one transaction to hold shared locks simultaneously on the same item. On the other hand, an exclusive lock gives a transaction exclusive access to that item. Thus, as long as a transaction holds the exclusive lock on the item, **no** other transactions can read or update that data item.

Locks are used in the following way:

- Any transaction that needs to access a data item must first lock the item, requesting a shared lock for read only access or an exclusive lock for both read and write access.
- If the item is not already locked by another transaction, the lock will he granted.
- If the item is currently locked, the DBMS determines whether the request is compatible with the existing lock. If a shared lock is requested on an item that already has a shared lock on it, the request will be granted; otherwise, the transaction must wait until the existing lock is released.
- A transaction continues to hold a lock until it explicitly releases it either during execution or when it terminates (aborts or commits). It is only when the exclusive lock has been released that the effects of the write operation will be made visible to other transactions.

In addition to these rules, some systems permit a transaction to issue a shared lock on an item and then later to upgrade the lock to an exclusive lock. This in effect allows a transaction to examine the data first and then decide whether it wishes to update it. If upgrading is not supported, a transaction must hold exclusive locks on all data items that it may update at some time during the execution of the transaction, thereby potentially reducing the level of concurrency in the system. For the same reason, some systems also permit a transaction to issue an exclusive lock and then later to downgrade the lock to a shared lock.

Using locks in transactions, as described above, does not guarantee serializability of schedules by themselves.

Incorrect locking schedule

An example of an incorrect locking schedule is one in which the schedule releases the locks that are held by a transaction as soon as the associated read/write is executed and that lock item (say balx) no longer needs to be accessed. However, the transaction itself is locking other items (baly), after it releases its lock on balx, Although this may seem to allow greater concurrency, it permits transactions to interfere with one another, resulting in the loss of total isolation and atomicity.

To guarantee serializability, we must follow an additional protocol concerning the positioning of the lock and unlock operations in every transaction. The best-known protocol is two-phase locking (2PL).

Two-phase locking (2PL) - A transaction follows the two-phase locking protocol if all locking operations precede the first unlock operation in the transaction.

According to the rules of this protocol, every transaction can be divided into two phases: first a **growing phase,** in which it acquires all the locks needed but cannot release any locks, and then a **shrinking phase,** in which it releases its locks but cannot acquire any

new locks, There is no requirement that all locks be obtained simultaneously. Normally, the transaction acquires some locks, does some processing, and goes on to acquire additional locks as needed. However, it never releases any lock until it has reached a stage where no new locks are needed. The rules are:

- A transaction must acquire a lock on an item before operating on the item. The lock may be read or write, depending on the type of access needed.
- Once the transaction releases a lock, it can never acquire any new locks. If upgrading of locks is allowed, upgrading can take place only during the growing phase and may require that the transaction wait until another transaction releases a shared lock on the item, Downgrading can take place only during the shrinking phase.

Two-phase locking can be used to resolve the three potential problems (Lost update problem, Uncommitted dependency problem and the Inconsistent analysis problem) associated with concurrency.

This is illustrated below.

Preventing the Lost Update problem using Two Phase Locking (2PL)

A solution to the lost update problem is shown in the figure below. To prevent the lost update problem occurring, T_2 first requests an exclusive lock on balx. It can then proceed to read the value of balx from the database, increment it by Kshs. 1,000, and write the new value back to the database. When T1starts, it also requests an exclusive lock on balx. However, because the data item balx, is currently exclusively locked by T_2, the request is not immediately granted and T_1 has to wait until the lock is released by T_2. This occurs only once the commit of T_2 has been completed.

Time	T1	T2	balx
t1		begin transaction	10,000
t2	begin transaction	write **lock(balx)**	10,000
t3	Write **lock(balx)**	**read(balx)**	10,000
t4	WAIT	**balx = balx** + 1000	10,000
ts	WAIT	write **(balx)**	11,000
t6	WAIT	**Commit/unlock(balx)**	11,000
t7	**read(balx)**		11,000
tg	**balx = balx** - 100		11,000
t9	**write(balx)**		11,900
t10	**Commit/unlock(balx)**		11,900

Preventing the lost update problem.

Preventing the Uncommitted Dependency problem using Two Phase Locking (2PL)

A solution to the uncommitted dependency problem is shown in the figure below. To prevent this problem occurring, T_4 first requests an exclusive lock on balx. It can then proceed to read the value of balx, from the database, increment it by Kshs. 1,000, and write the new value back to the database. When the rollback is executed, the updates of transaction T4 are undone and the value of balx in the database is returned to its original value of Kshs. 10,000. When T_3 starts, it also requests an exclusive lock on balx. However, because the data item balx, is currently exclusively locked by T_4, the request is

not immediately granted and T3 has to wait until the lock is released by T4. This occurs only once the rollback of T4 has been completed.

Time	T3	T4	balx
t1		begin_transaction	10,000
t2		write_**lock(balx)**	10,000
t3		**read(balx)**	10,000
t4	begin_transaction	**balx = balx +** 1000	10,000
ts	write_**lock(balx)**	**write(balx)**	11,000
t6	WAIT	ro11back/unlock(balx)	10,000
t7	**read(balx)**		10,000
tg	**balx = balx -** 100		10,000
t9	**write(balx)**		10,900
t10	commit/unlock **(balx)**		10,900

Preventing the uncommitted dependency problem.

Preventing the Inconsistent Analysis problem using Two Phase Locking (2PL)

A solution to the inconsistent analysis problem is shown in figure below. To prevent this problem occurring, T_5 must precede its reads by exclusive locks, and T_6 must precede its reads with shared locks. Therefore, when T_5 starts it requests and obtains ari exclusive lock on balx, Now, when T_6 tries to share lock balx, the request is not immediately granted and T_6 has to wait until the lock is released, which is when T_5 commits.

Time	Ts	T6	balx	balv	balz	sum
t1		begin transaction	100	50	25	
t2	begin transaction	sum=0	100	50	25	0
t3	write **lock(balx)**		100	50	25	0
t4	**read(balx)**	read **lock(balx)**	100	50	25	0
ts	**balx = balx -** 10	**WAIT**	100	50	25	0
t6	**write(balx)**	**WAIT**	90	50	25	0
t7	write **lock(balz)**	**WAIT**	90	50	25	0
tg	**read(balz)**	**WAIT**	90	50	25	0
t9	**balz= balz+** 10	**WAIT**	90	50	25	0
t10	**write(balz)**	**WAIT**	90	50	35	0
t1ı	**comrnit/unlock(balx,balzi**	**WAIT**	90	50	35	0
t12		**read(balx)**	90	50	35	0
t13		sum = sum **+ balx**	90	50	35	90
t14		read **lock(balv)**	90	50	35	90
t15		**read(balv)**	90	50	35	90
t16		sum = sum **+ balv**	90	50	35	140
t(7		read **lock(balz)**	90	50	35	140
tis		**read(balz)**	90	50	35	140
t19		sum = sum **+ balz**	90	50	35	175
t20		**comrnit/unlock(balx,balz,halz)**	90	50	35	175

It can be proved that if *every* transaction in a schedule follows the two-phase locking protocol, then the schedule is guaranteed to be conflict serializable. However, while the two-phase locking protocol guarantees serializability, problems can occur with the interpretation of when locks can be released, as the next example shows.

Cascading rollback

Consider a schedule consisting of the three transactions shown in the figure below, which conforms to the two-phase locking protocol. Transaction T14 obtains an exclusive lock on balx then updates it using baly, which has been obtained with a shared lock, and writes the value of balx back to the database before releasing the lock on balx. Transaction TI s, then obtains an exclusive lock on balx, reads the value of balx, from the database, updates it, and writes the new value back to the database before releasing the lock. Finally, T16 share locks balx, and reads it from the database. By now, T14 has failed and has been rolled back. However, since T15 is dependent on T14 (it has read an item that has been updated by T14), T15 must also be rolled back. Similarly, T16 is dependent on Tis, so it too must be rolled back. This situation, in which a single transaction leads to a series of rollbacks, is called **cascading rollback.**

Time	T14	Tis	T16
t1	begin transaction		
t2	write **lock(balx)**		
t3	**read(balx)**		
t4	read lock(balv)		
ts	read (balv)		
t6	**balx=balv +balx**		
t7	write(balx)		
tg	unlock(balx)	begin transaction	
t9	:	write lock(balx)	
t10	:	read(balx)	
11I	:	**balx=balv+100**	
t12	:	**write(balx)**	
t13	:	**unlock(balx)**	
t14	:	:	
tis	**rollback**	:	
116		:	begin transaction
t17		:	read **lock(balx)**
t1g		**rollback**	:
t19			**rollback**

Cascading rollback with 2PL

Cascading rollbacks are undesirable since they potentially lead to the undoing of a significant amount of work. Clearly, it would be useful if we could design protocols that prevent cascading rollbacks. One way to achieve this with two-phase locking is to leave the release of ***all*** locks until the end of the transaction, as in the previous examples. In this way, the problem illustrated here would not occur, as T15 would not obtain its exclusive lock until after T14 had completed the rollback. This is called **rigorous two-**

phase locking (rigorous 2PL). It can be shown that with rigorous 2PL, transactions can be serialized in the order in which they commit. Another variant of 2PL, called **strict two-phase locking (strict 2PL),** only holds ***exclusive locks*** until the end of the transaction. Most database systems implement one of these two variants of 2PL.

Another problem with two-phase locking, which applies to all locking-based schemes, is that it can cause **deadlock,** since transactions can wait for locks on data items. If two transactions wait for locks on items held by the other, a deadlock will occur and a deadlock detection and recovery will be needed. It is also possible for transactions to be in **livelock,** that is, left in a wait state indefinitely, unable to acquire any new locks, although the DBMS is not in deadlock. This can happen if the waiting algorithm for transactions is unfair and does not take account of the time that transactions have been waiting. To avoid livelock, a priority system can be used, whereby the longer a transaction has to wait, the higher its priority, for example, *a first-come-first- served* queue can be used for waiting transactions.

Deadlock

A **deadlock** is an impasse that may result when two (or more) transactions are each waiting for locks to be released that are held by the other.

The figure below shows two transactions, T_{17} and T1s, that are deadlocked because each is waiting for the other to release a lock on an item it holds. At time t_2, transaction T_{17}, requests and obtains an exclusive lock on item balx, and at time t3transaction T,s obtains an exclusive lock on item baly, Then at t_6, T_{17} requests an exclusive lock on item baly, Since T18 holds a lock on baly, transaction T17 waits. Meanwhile, at time t7, T1s requests a - lock on item balx, which is held by transaction T_{17}. Neither transaction can continue because each is waiting for a lock it cannot obtain until the other completes. Once deadlock occurs, the applications involved cannot resolve the problem. Instead, the DBMS has to recognize that deadlock exists and break the deadlock in some way.

Unfortunately, there is only one way to break deadlock: abort one or more of the transactions. This usually involves undoing all the changes made by the aborted transaction(s). In the figure below, we may decide to abort transaction T18. Once this is complete, the locks held by transaction T1s are released and T_{17} is able to continue again. Deadlock should be transparent to the user, so the DBMS should automatically restart the aborted transaction(s).

Time	T17	Tis
t,	begin transaction	
t2	write lock(balx)	begin transaction
t3	read(balx)	write lock(balv)
t4	balx= balx- 100	read(balv)
ts	write(balx)	balv= balv- 100
16	write lock(balv)	write(balv)
t7	WAIT	write lock(balx)
tg	WAIT	WAIT
t9	WAIT	WAIT
t,o		WAIT

Deadlock between two transactions.

There are three general techniques for handling deadlock:

i.) Timeouts.
ii.) Deadlock prevention.
iii.)Deadlock detection and recovery.

With timeouts, the transaction that has requested a lock waits for at most a specified period of time. Using *deadlock prevention,* the DBMS looks ahead to determine if a transaction would cause deadlock, and never allows deadlock to occur. Using *deadlock detection and recovery,* the DBMS allows deadlock to occur but recognizes occurrences of deadlock and breaks them. Since it is more difficult to prevent deadlock than to use timeouts or testing for deadlock and breaking it when it occurs, systems generally avoid the deadlock prevention method.

Timeouts

A simple approach to deadlock prevention is based on *lock timeouts.* With this approach, a transaction that requests a lock will wait for only a system-defined period of time. If the lock has not been granted within this period, the lock request times out. In this case, the DBMS assumes the transaction may be deadlocked, even though it may not be, and it aborts and automatically restarts the transaction. This is a very simple and practical solution to deadlock prevention and is used by several commercial DBMSs.

Deadlock Prevention

Another possible approach to deadlock prevention is to order transactions using transaction timestamps. Two algorithms have been proposed. One algorithm, *Wait-Die,* allows only an older transaction to wait for a younger one, otherwise the transaction is aborted *(dies)* and restarted with the same timestamp, so that eventually it will become the oldest active transaction and will not die. The second algorithm, *Wound-Wait,* uses a symmetrical approach: only a younger transaction can wait for an older one. If an older transaction requests a lock held by a younger one, the younger one is aborted *(wounded).*

A variant of 2PL, called **conservative 2PL,** can also be used to prevent deadlock. Using conservative *2PL,* a transaction obtains all its locks when it begins or it waits until all the locks are available. This protocol has the advantage that if lock contention is heavy, the time that locks are held is reduced because transactions are never blocked and therefore never have to wait for locks. On the other hand, if lock contention is low then locks are held longer under this protocol. Further, the overhead for setting locks is high because all the locks must be obtained and released all at once. Thus, if a transaction fails to obtain one lock it must release all the current locks it has obtained and start the lock process again. From a practical perspective, a transaction may not know at the start which locks it may actually need and, therefore, may have to set more locks than is required. This protocol is not used in practice.

Deadlock detection

Deadlock detection is usually handled by the construction of a **wait-for graph** (WFG) that shows the transaction dependencies; that is, transaction Ti, is dependent on Tj if

trc1.nsaction Tj holds the lock on a data item that Ti is waiting for. The WFG is a directed graph G = (N, E) that consists of a set of nodes N and a set of directed edges E, which is constructed as follows:

- Create a node for each transaction.
- Create a directed edge Ti -> Tj, if transaction Ti is waiting to lock an item that is currently locked by Tj,

Deadlock exists if and only if the WFG contains a cycle. The figure above shows the WFG for the transactions met earlier (involving T_{17} and T_{18}). Clearly, the graph has a cycle in it (T_{I7}-> T_{I8} -> T_{17}), so we can conclude that the system is in deadlock.

Frequency of deadlock detection

Since a cycle in the wait-for graph is a necessary and sufficient condition for deadlock to exist, the deadlock detection algorithm generates the WFG at regular intervals and examines it for a cycle. The choice of time interval between executions of the algorithm is important. If the interval chosen is too small, deadlock detection will add considerable overhead; if the interval is too large, deadlock may not be detected for a long period. Alternatively, a dynamic deadlock detection algorithm could start with an initial interval size. Each time no deadlock is detected, the detection interval could be increased, for example, to twice the previous interval, and each time deadlock is detected, the interval could be reduced, for example, to half the previous interval, subject to some upper and lower limits.

Recovery from deadlock detection

As mentioned earlier, once deadlock has been detected the DBMS needs to abort one or more of the transactions. There are several issues that need to he considered:

1) *Choice of deadlock victim*. In some circumstances, the choice of transactions to abort may be obvious. However, in other situations, the choice may not be so clear. In such cases, we would want to abort the transactions that incur the minimum costs. This may take into consideration:

   a) How long the transaction has been running. It may be better to abort a transaction that has just started rather than one that has been running for some time.

   b) How many data items have been updated by the transaction. It would be better to abort a transaction that has made little change to the database rather than one that has made significant changes to the database.

   c) How many data items the transaction is still to update. It would be better to abort a transaction that has many changes still to make to the database rather than one that has few changes to make. Unfortunately, this may not be something that the DBMS would necessarily know.

2)_ *How far to roll a transaction back.* Having decided to abort a particular transaction, we have to decide how far to roll the transaction back. Clearly, undoing all the changes made by a transaction is the simplest solution, although not necessarily the most efficient. Hmay be possible to resolve the deadlock by rolling back only part of the transaction.

3) *Avoiding starvation.* Starvation occurs when the same transaction is always chosen as the victim (to be aborted), and the transaction can never complete. Starvation is very similar to livelock mentioned earlier, which occurs when the concurrency control protocol never selects a particular transaction that is waiting for a lock. The DBMS can avoid starvation by storing a count of the number of times a tn;msaction has been selected as the victim and using a different selection criterion once this count reaches some upper limit.

Timestamping Methods

The use of locks, combined with the two-phase locking protocol, guarantees serializability of schedules. The order of transactions in the equivalent serial schedule is based on the order in which the transactions lock the items they require. If a transaction needs an item that is already locked, it may be forced to wait until the item is released. A different approach that also guarantees serializability uses transaction timestamps to order transaction execution for an equivalent serial schedule.

Timestamp methods for concurrency control are quite different from locking methods. No locks are involved, and therefore there can be no deadlock. Locking methods generally prevent conflicts by making transactions wait. With timestamp methods, there is no waiting: transactions involved in conflict are simply rolled back and restarted.

Timestamp - A timestamp is a unique identifier created by the DBMS that indicates the relative starting time of a transaction.

Timestamps can be generated by simply using the system clock at the time the transaction started, or, more normally, by incrementing a logical counter every time a new transaction starts.

Timestamping - Timestamping is a concurrency control protocol that orders transactions in such a way that older transactions, transactions with *smaller* timestamps, get priority in the event of conflict.

With timestamping, if a transaction attempts to read or write a data item, then the read or write is only allowed to proceed if the *last update* on *that data item* was carried out by an older transaction. Otherwise, the transaction requesting the read/write is restarted and given a new timestamp. New timestamps must be assigned to restarted transactions to prevent their being continually aborted and restarted. Without new timestamps, a transaction with an old timestamp might not he able to commit owing to younger transactions having already committed.

Besides timestamps for transactions, there are timestamps for data items. Each data item contains a **read_timestamp,** giving the timestamp of the last transaction to read the item, and a **write_timestamp,** giving the timestamp of the last transaction to write (update) the item.

For a transaction T with timestamp ts(T), the timestamp ordering protocol works as follows.

1) *Transaction Tissues a read(x)*

   a) Transaction Tasks to read an item (x) that has already been updated by a younger (later) transaction, that is ts(T) < write_timestamp(x). This means that an earlier transaction is trying to read a value of an item that has been updated by a later transaction. The earlier transaction is too late to read the previous outdated value, and any other values it has acquired are likely to be inconsistent wit_h the updated value of the data item. In this situation, transaction T must be· aborted and restarted with a new (later) timestamp

   b) Otherwise, ts(T) >= write_timestamp(x), and the read operation can proceed. We set read_timestamp(x) = max (ts(T), read_timestamp(x)).

2) *Transaction Tissues a write(x)*

   a) Transaction T asks to write an item (x) whose value has already been read by a younger transaction, that is ts(T) < read_timestamp(x). This means that a later transaction is already using the current value of the item and it would be an error to update it now. This occurs when a transaction is late in doing a write and a younger transaction has already read the old value or written a new one. In this case, the only solution is to roll back transaction T and restart it using a later timestamp.

   b) Transaction Tasks to write an item (x) whose value has already been written by a younger transaction, that is ts(T) < write_timestamp(x). This means that transaction T is attempting to write an obsolete value of data item x. Transaction T should be rolled back and restarted using a later timestamp.

   c) Otherwise, the write operation can proceed. We set write_timestamp(x) = ts(T).

This scheme, called basic timestamp ordering, guarantees that transactions are conflict serializable, and the results are equivalent to a serial schedule in which the transactions are executed in chronological order of the timestamps. In other words, the results will be as if all of transaction 1 were executed, then all of transaction 2, and so on, with no interleaving. However, basic timestamp ordering does not guarantee recoverable schedules. Before we show how these rules can be used to generate a schedule using timestamping, we first examine a slight variation to this protocol that provides greater concurrency.

Thomas's write rule

A modification to the basic timestamp ordering protocol that relaxes conflict serializability can be used to provide greater concurrency by rejecting obsolete write operations (Thomas, 1979). The extension, known as **Thomas's write rule,** modifies the checks for a write operation by transaction T as follows:

a) Transactiqn T asks to write an item (x) whose value has already been read by a younger transaction, that is ts(T) < read_timestamp(x). As before, roll back transaction T and restart it using a later timestamp.

b)_ Transaction T asks to write an item (x) whose value has already been written by a younger transaction, that is ts(T) < write_timestamp(x). This means that a later transaction has already updated the value of the item, and the value that the older transaction is writing must he based on an obsolete value of the item. In this case, the write operation can safely be ignored. This is sometimes known as the **ignore obsolete write rule,** and allows greater concurrency.

c) Otherwise, as before, the write operation can proceed. We set write_timestamp(x) = ts(T).

The use of Thomas's write rule allows schedules to be generated that would not have been possible under the other concurrency protocols.

Example: Basic timestamp ordering

Three transactions are executing concurrently, as illustrated in the figure. Transaction T_{19} has a timestamp of ts(T_{19}), T_{20} has a timestamp of ts(T_{20}), and T_{21}has a timestamp of ts(T_{21}), such that ts(T_{19}) < ts(T_{20}) < ts(T_{21}). At time ts, the write by transaction T_{20} violates the first write rule and so $T_{20}$is aborted and restarted at time 114. Also at time 114, the write by transaction T_{19} can safely be ignored using the ignore obsolete write rule, as it would have been overwritten by the write of transaction T_{21}at time 112-

Time	Op	T19	T20	T21
11		begin transaction		
12	**read(balx)**	read(balx)		
t3	**balx** = **balx** + 10	**balx** = **balx** + 10		
4	write(balx)	write(balx)	begin transaction	
ts	read(balv)		read(balv)	
16	balv = balv + 20		**balv** = **balv** + 20	begin transaction
t7	read(balv)			read(balv)
tg	write(balv)		write(balv)*	
t9	balv= balv+ 30			**balv** = **balv** + 30
110	write(balv)			write(balv)
111	**balz** = 100			**balz** = 100
112	write(balz)			write(balz)
113	**balz** = 50	**balz** = 50		commit
114	write(balz)	write(balz)**	begin transaction	
tis	read(balv)	commit	read(balv)	
t16	**balv** = **balv** + 20		**balv** = **balv**+ 20	
t17	**write(balv)**		**write(balv)**	
tis			commit	

* At time ts, the write transaction T_{20} violates the first timestamping write rule described above and therefore is aborted and restarted at time t_{14}.

** At time 114, the write transaction T_{19} can safely be ignored using the ignore obsolete write rule, as it would have been overwritten by the write of transaction T_{21}at time t_{12}.

Optimistic Techniques.

In some environments, conflicts between transactions are rare, and the additional processing required by locking or timestamping protocols is unnecessary for many of the transactions. Optimistic techniques are based on the assumption that conflict is rare, and that it is more efficient to allow transactions to proceed without imposing delays to ensure serializability. When a transaction wishes to commit, a check is performed to determine whether conflict has occurred. If there has been a conflict, the transaction must be rolled back and restarted. Since the premise is that conflict occurs very infrequently, rollback will be rare. The overhead involved in restarting a transaction may be considerable, since it effectively means redoing the entire transaction. This could be tolerated only if it happened very infrequently, in which case the majority of transactions will be processed without being subjected to any delays. These techniques potentially allow greater concurrency than traditional protocols since no locking is required.

There are two or three phases to an optimistic concurrency control protocol, depending on whether it is a read-only or an update transaction:

- *Read phase* - This extends from the start of the transaction until immediately before the commit. The transaction reads the values of all data items it needs from the database and stores them in local variables. Updates are applied to a local copy of the data, not to the database itself.
- *Validation phase* - This follows the read phase. Checks are performed to ensure serializability is not violated if the transaction updates are applied to the database. For a read-only transaction, this consists of checking that the data values read are still the current values for the corresponding data items. If no interference occurred, the transaction is committed. If interference occurred, the transaction is aborted and restarted. For a transaction that has updates, validation consists of determining whether the current transaction leaves the database in a consistent state, with serializability maintained. If not, the transaction is aborted and restarted.
- *Write phase* - This follows the successful validation phase for update transactions. During this phase, the updates made to the local copy are applied to the database.

The validation phase examines the reads and writes of transactions that may cause interference. Each transaction T is assigned a timestamp at the start of its execution, *start(T),* one at the start of its validation phase, *validation(T),* and one at its finish time, *finish(T),* including its write phase, if any. To pass the validation test, one of the following must be true:

1) All transactions S with earlier timestamps must have finished before transaction T started; that *is,finish(S) < start(T).*
2) If transaction T starts before an earlier one S finishes, then:
   a) The set of data items written by the earlier transaction are not the ones read by the current transaction; *and*
   b) The eat lier transaction completes its write phase before the current transaction enters its validation phase, that is *start(T) < finish(S) < validation(T).*

Rqle 2(a) guarantees that the writes of an earlier transaction are not read by the current transaction; rule 2(6) guarantees that the writes are done serially, ensuring no conflict.

Although optimistic techniques are very efficient when there are few conflicts, they can result in the rollback of individual transactions. Note that the rollback involves only a local copy of the data so there are no cascading rollbacks, since the writes have not actually reached the database. However, if the aborted transaction is of a long duration, valuable processing time will be lost since the transaction must be restarted. If rollback occurs often, it is an indication that the optimistic method is a poor choice for concurrency control in that particular environment.

Granularity of Data Items

Granularity is the size of data items chosen as the ***unit of protection*** by a concurrency control protocol.

All the concurrency control protocols discussed earlier assume that the database consists of a number of 'data items', without explicitly defining the term. Typically, a data item is chosen to be one of the following, ranging from coarse to fine, where fine granularity refers to small item sizes and coarse granularity refers to large item sizes:

- The entire database.
- A file.
- A page (sometimes called an area or database space - a section of physical disk in which relations are stored).
- A record.
- A field value of a record.

The size or granularity of the data item that can be locked in a single operation has a significant effect on the overall performance of the concurrency control algorithm. However, there are several tradeoffs that have to be considered in choosing the data item size. The discussion on these tradeoffs in this section is in context of locking, although similar arguments can be made for other concurrency control techniques.

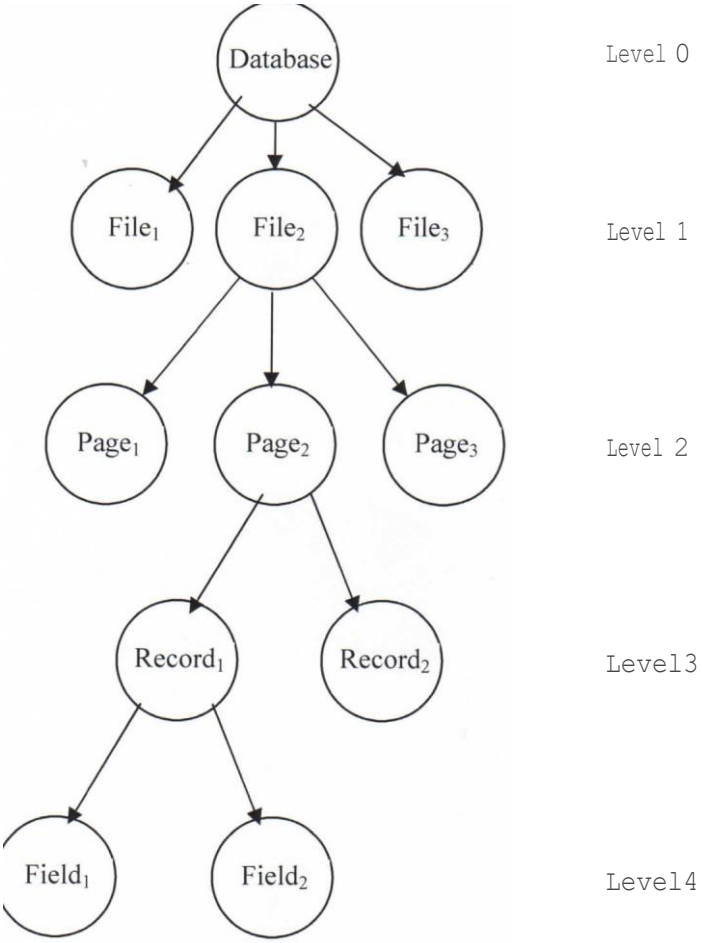
Consider a transaction that updates a single tuple (record) of a relation. The concurrency control algorithm might allow the transaction to lock only that single tuple (record), in which case the granule size for locking is a single record. On the other hand, it might lock the entire database, in which case the granule size is the entire database. In the second case, the granularity would prevent any other transactions from executing until the lock is released. This would clearly be undesirable. On the other hand, if a transaction updates 95% of the records in a file, then it would be more efficient to allow it to lock the entire file rather than to force it to lock each record separately. However, escalating the granularity from field or record to file may increase the likelihood of deadlock occurring.

Thus, the coarser the data item size, the lower the degree of concurrency permitted. On the other hand, the finer the item size; the more locking information that needs to be stored. The best item size depends upon the nature of the transactions. If a typical transaction accesses a small number of records, it is advantageous to have the data item granularity at the record level. On the other hand, if a transaction typically accesses many records of the same file, it may be better to have page or file granularity so that the transaction considers all those records as one (or a few) data items.

Some techniques have been proposed that have dynamic data item sizes. With these techniques, depending on the types of transaction that are currently executing, the data item size may be changed to the granularity that best suits these transactions. Ideally, the DBMS should support mixed granularity with record, page, and file level locking. Some systems automatically upgrade locks from record or page to file if a particular transaction is locking more than a certain percentage of the records or pages in the file.

Hierarchy of granularity

We could represent the granularity of locks in a hierarchical structure where each node represents data items of different sizes, as shown in the figure below. Here, the root node represents the entire database, the level 1 nodes represent files, the level 2 nodes represent pages, the level 3 nodes represent records, and the level 4 leaves represent individual fields.



Levels of Locking

Whenever a node is locked, all its descendants are also locked. For example, if a transaction locks a page, *Page2,* all its records *(Record,* and *Record2)* as well as all their fields *(Field,* and *Field2)* are also locked. If another transaction requests an incompatible lock on the *same* node, the DBMS clearly knows that the lock cannot be granted.

If another transaction requests a lock on any of the *descendants* of the locked node, the DBMS checks the hierarchical path from the root to the requested node to determine if any of its ancestors are locked before deciding whether to grant the lock. Thus, if the request is for an exclusive lock on record $Record_1$, the DBMS checks its parent $(Page_2)$, its grandparent $(File_1)$, and the database itself to determine if any of them are locked. When it finds that $Page_2$ is already locked, it denies the request.

Additionally, a transaction may request a lock on a node and a descendant of the node is already locked. For example, if a lock is requested on $File_2$, the DBMS checks every page in the file, every record in those pages, and every field in those records to determine if any of them are locked.

Multiple-granularity locking

To reduce the searching involved in locating locks on descendants, the DBMS can use another specialized locking strategy called **multiple-granularity** locking. This strategy uses a new type of lock called an **intention lock.** When any node is locked, an intention lock is placed on all the ancestors of the node. Thus, if some descendant of $File_2$ (in our example, $Page_2$) is locked and a request is made for a lock on *File2*, the presence of an intention lock on *File2* indicates that some descendant of that node is already locked.

To ensure serializability with locking levels, a two-phase locking protocol is used as follows:

- No lock can be granted once any node has been unlocked.
- No node may be locked until its parent is locked by an intention lock.
- No node may be unlocked until all its descendants are unlocked.

In this way, locks are applied from the root down using intention locks until the node requiring an actual read or exclusive lock is reached, and locks arc released from the bottom up. However, deadlock is still possible and must he handled as discussed previously.

Database Recovery

Database recovery is the process of restoring the database to a correct state in the event of a failure.

The concept of database recovery is a service that should he provided by the DBMS to ensure that the database is reliable and remains in a consistent state in the presence of failures. In this context, reliability refers to both the resilience of the DBMS to various types of failure and its capability to recover from them. This section considers how this service can be provided.

The Need for Recovery

The storage of data generally includes four different types of media with an increasing degree of reliapility: main memory, magnetic disk, magnetic tape, and optical disk. Main memory is **volatile** storage that usually does not survive system crashes. Magnetic disks provide online non-volatile storage. Compared with main memory, disks are more reliable and much cheaper, but slower by three to four orders of magnitude. Magnetic

tape is an **offline non-volatile** storage medium, which is far more reliable than disk and fairly inexpensive, but slower, providing only sequential access. Optical disk is more reliable than tape, generally cheaper, faster, and providing random access. Main memory is also referred to as **primary storage** and disks and tape as **secondary storage. Stable storage** represents information that has been replicated in several non-volatile storage media (usually disk) with independent failure modes. For example, it may he possible to simulate stable storage using RAID (Redundant Array of Independent Disks) technology, which guarantees that the failure of a single disk, even during data transfer, does not result in loss of data.

There are many different types of failure that can affect database processing, each of which has to he dealt with in a different manner. Some failures affect main memory only, while others involve non-volatile (secondary) storage. Among the causes of failure are:

- **System crashes due** to hardware or software errors, resulting in loss of main memory.
- **Media failures,** such as head crashes or unreadable media, resulting in the loss of parts of secondary storage.
- **Application software errors,** such as logical errors in the program that is accessing the database, which cause one or more transactions to fail.
- **Natural physical disasters,** such as fires, floods, earthquakes, or power failures.
- **Carelessness** or unintentional destruction of data or facilities by operators or users.
- **Sabotage,** or intentional corruption or destruction of data, hardware, or software facilities.

Whatever the cause of the failure, there are two principal effects that we need to consider: the loss of main memory, including the database buffers, and the loss of the disk copy of the database.

Transactions and Recovery

Transactions represent the basic *unit of recovery* in a database system. It is the role of the recovery manager to guarantee two of the four *ACID* properties of transactions, namely *atomicity* and *durability,* in the presence of failures. The recovery manager has to ensure that, on recovery from failure, either all the effects of a given transaction are permanently recorded in the database or none of them are. The situation is complicated by the fact that database writing is not an atomic (single-step) action, and it is therefore possible for a transaction to have committed but for its effects not to have been permanently recorded in the database, simply because they have not yet reached the database.

Consider the example below, in which the salary of a member of staff is being increased.

ead(staffNo = x,salary)
salary = salary * 1.1
write(staffNo = x,salary)

To implement the read operation, the DBMS carries out the following steps:

- _ Find the address of the disk block that contains the record with primary key value x.
- Transfer the disk block into a database buffer in main memory.
- Copy the salary data from the database buffer into the variable *salary*.

For the write operation, the DBMS carries out the following steps:

- Find the address of the disk block that contains the record with primary key value x.
- Transfer the disk block into a database buffer in main memory.
- Copy the salary data from the variable *salary* into the database buffer.
- Write the database buffer back to disk.

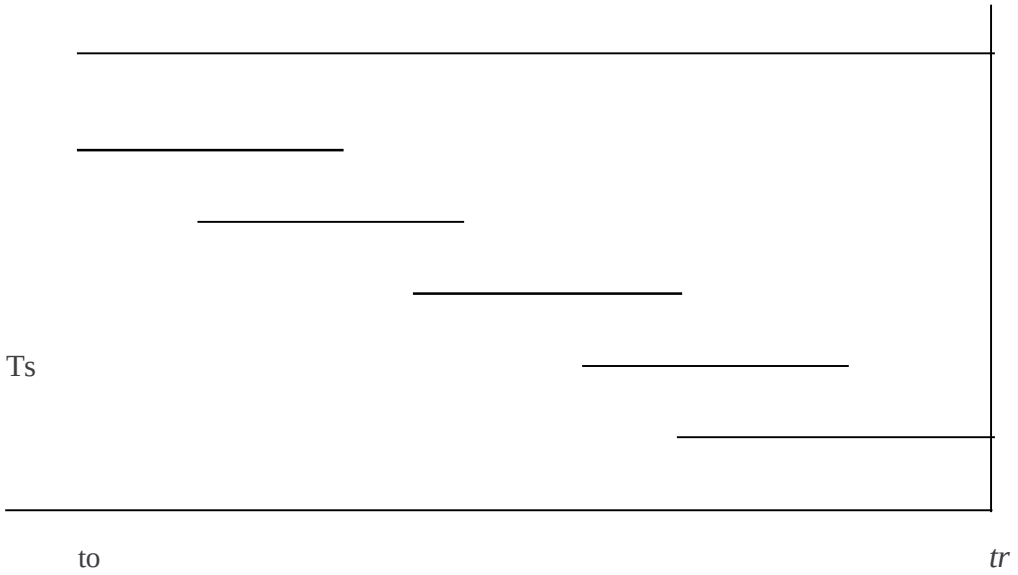
The database buffers occupy an area in main memory from which data is transferred to and from secondary storage. It is only once the buffers have been **flushed** to secondary storage that any update operations can be regarded as permanent. This flushing of the buffers to the database can be triggered by a specific command (for example, transaction commit) or automatically when the buffers become full. The explicit writing of the buffers to secondary storage is known as **force-writing.**

If a failure occurs between writing to the buffers and flushing the buffers to secondary storage, the recovery manager must determine the status of the transaction that performed the write at the time of failure. If the transaction had issued its commit, then to ensure durability the recovery manager would have to **redo** that transaction's updates to the database (also known as **rollforward).**

On the other hand, if the transaction had not committed at the time of failure, then the recovery manager would have to **undo (rollback)** any effects of that transaction on the database to guarantee transaction atomicity. If only one transaction has to be undone, this is referred to as **partial undo.** A partial undo can be triggered by the scheduler when a transaction is rolled back and restarted as a result of the concurrency control protocol, as described earlier. A transaction can also be aborted unilaterally, for example, by the user or by an exception condition in the application program. When all active transactions have to be undone, this is referred to as **global undo.**

Example: Use of UNDO/REDO

The figure below illustrates a number of concurrently executing transactions T_1, ..•, T_6. The DBMS staits at time to but fails at time tr. We assume that the data for transactions T_2 and T3 has been written to secondary storage before the failure. Clearly T_1 and T_6 had not committed at the point of the crash, therefore at restart the recovery manager must ***undo*** transactions TI and T_6. However, it is not clear to what extent the changes made by the other (committed) transactions T_4 and T_5 have been propagated to the database on non-volatile storage. The reason for this uncertainty is the fact that the volatile database buffers may or may not have been written to disk. In the absence of any other i.nformation, the recovery manager would be forced to ***redo*** transactions T_2, T3, T_4 and T_5.



Example of UNDO/REDO

Buffer management

The management of the database buffers plays an important role in the recovery process and this section briefly discusses their management.

The buffer manager is responsible for the efficient management of the database buffers that are used to transfer pages to and from secondary storage. This involves reading pages from disk into the buffers until the buffers become full and then using a ***replacement strategy*** to decide which buffer(s) to force-write to disk to make space for new pages that need to be read from disk. Example replacement strategies are *first-in-first-out* (FIFO) and *least recently used* (LRU). In addition, the buffer manager should not read a page from disk if it is already in a database buffer.

One approach is to associate two variables with the management information for each database buffer: ***pinCount*** and ***dirty***, which are initially set to zero for each database buffer. When a page is requested from disk, the buffer manager will check to see whether the page is already in one of the database buffers. If it is not, the buffer manager will:

1) Use the replacement strategy to choose a buffer for replacement (which we will call the *replacement buffer)* and increment its pinCount. The requested page is now **pinned** in the database buffer and cannot be written back to disk yet. The replacement strategy will not choose a buffer that has been pinned.
2) If the dirty variable for the replacement buffer is set, it will write the buffer to disk.
) Read the page from disk into the replacement buffer and reset the buffer's dirty variable to zero.

If the same page is requested again, the appropriate pinCount is incremented by 1. When the system informs the buffer manager that it has finished with the page, the appropriate pinCount is decremented by 1. At this point, the system will also inform the buffer

manager if it has modified the page and the dirty variable is set accordingly. When a pinCount reaches zero, the page is unpinned and the page can be written back to disk if it has been modified (that is, if the dirty variable has been set).

The following terminology is used in database recovery when pages are written back to disk:

- A **steal policy** allows the buffer manager to write a buffer to disk before a transaction commits (the buffer is unpinned). In other words, the buffer manages *'steals'* a page from the transaction. The alternative policy is **no-steal.**
- A **force policy** ensures that all pages updated by a transaction are immediately written to disk when the transaction commits. The alternative policy is **no-force.**

The, simplest approach from an implementation perspective is to use a no-steal, force policy: with *no-steal* we do not have to undo changes of an aborted transaction because the changes will not have been written to disk, and *withforce* we do not have to redo the changes of a committed transaction if there is a subsequent crash because all the changes will have been written to disk at commit.

On the other hand, the *steal* policy avoids the need for a very large buffer space to store all updated pages by a set of concurrent transactions (which in practice may be unrealistic). In addition, the *no-force* policy has the distinct advantage of not having to rewrite a page to disk for a later transaction that has been updated by an earlier committed transaction and may still be in a database buffer. For these reasons, most DBMSs employ a steal, no-force policy.

Recovery Facilities

A DBMS should provide the following facilities to assist with recovery:

- A backup mechanism, which makes periodic backup copies of the database.
- Logging Facilities, which keep track of the current state of transactions and database changes.
- A checkpoint facility, which enables updates to the database that are in progress to be made permanent.
- A recovery manager, which allows the system to restore the database to a consistent state following a failure.

Backup mechanism

The DBMS should provide a mechanism to allow backup copies of the database and the *log file* (discussed below) to be made at regular intervals without necessarily having to stop the system first. The backup copy of the database can be used in the event that the database has been damaged or destroyed. A backup can be a complete copy of the entire database or an incremental backup, consisting only of modifications made since the last complete or incremental backup. Typically, the backup is stored on offline storage, such as magnetic tape.

Log file

To keep track of database transactions, the DBMS maintains a special file called a **log** (or **journal)** that contains information about all updates to the database. The log may contain the following data:

- **Transaction records,** containing:
  - Transaction identifier.
  - Type of log record (transaction start, insert, update, delete, abort, commit).
  - Identifier of data item affected by the database action (insert, delete, and update operations).
  - **Before-image** of the data item, that is, its value before change (update and delete operations only).
  - **After-image** of the data item, that is, its value after change (insert and update operations only).
  - Log management information, such as a pointer to previous and next log records for that transaction (all operations).
- **Checkpoint records** (Described later).

The log is often used for purposes other than recovery (for example, for performance monitoring and auditing). In this case, additional information may be recorded in the log file (for example, database reads, user logons, logoffs, and etc.). These are not relevant to recovery though.

The figure below illustrates a segment of a log file that shows three concurrently executing transactions TI, T2, and T3. The columns pPtr and nPtr represent pointers to the previous and next log records for each transaction.

Tid	Time	Operation	Object	Before Image	After Image	pPtr	nPtr
Tl	10:12	START				0	2
Tl	10:13	UPDATE	STAFF SL21	(old value)	(new value)	1	8
T2	10:14	START				0	4
T2	10:16	INSERT	STAFF SG37		(new value)	3	5
T2	10:17	DELETE	STAFF SLA9	(old value)		4	6
T2	10:17	UPDATE	PROPERTY PG16	(old value)	(new value)	5	9
T3	10:18	START				0	11
Tl	10:18	COMMIT				2	0
	10:19	CHECKPOINT	T2, T3				
T2	10:19	COMMIT				6	1
T3	10:20	INSERT	PROPERTY PG4		(new value)	7	12
T3	10:21	COMMIT				11	0

A segment of a log file

Owing to the importance of the transaction log file in the recovery process, the log may be duplexed or triplexed (that is, two or three separate copies are maintained) so that if one copy is damaged, another can be used. In the past, log files were stored on magnetic tape because tape was more reliable and cheaper than magnetic disk. However, nowadays

DBMSs are expected to be able to recover quickly from minor failures. This requires that the log file be stored online on a fast direct-access storage device.

In some environments where a vast amount of logging information is generated every day (a daily logging rate of 10^4 megabytes is not uncommon), it is not possible to hold all this data online all the time. The log file is needed online for quick recovery following minor failures (for example, rollback of a transaction following deadlock). Major failures, such as disk head crashes, obviously take longer to recover from and may require access to a large part of the log. In these cases, it would be acceptable to wait for parts of the log file to be brought back online from offline storage.

One approach to handling the offlining of the log is to divide the online log into two separate random access files. Log records are written to the first file until it reaches a high water mark, for example 70% full. A second log file is then opened and all log records for *new* transactions are written to the second file. *Old* transactions continue to use the first file until they have finished, at which time the first file is closed and transferred to offline storage. This simplifies the recovery of a single transaction as all the log records for that transaction are either on offline or online storage. It should be noted that the log file is a potential bottleneck and the speed of the writes to the log file can be critical in determining the overall performance of the database system.

Checkpointing

The information in the log file is used to recover from a database failure. One difficulty with this scheme is that when a failure occurs we may not know how far back in the log to search and we may end up redoing transactions that have been safely written to the database. To limit the amount of searching and subsequent processing that we need to carry out on the log file, we can use a technique called checkpointing.

Checkpoint - A checkpoint is the point of synchronization between the database and the transaction log file. All buffers are force-written to secondary storage.

Checkpoints are scheduled at predetermined intervals and involve the following operations:

- Writing all log records in main memory to secondary storage.
- Writing the modified blocks in the database buffers to secondary storage.
- Writing a checkpoint record to the log file. This record contains the identifiers of all transactions that are active at the time of the checkpoint.

If transactions are performed serially, then, when a failure occurs, we check the log file to find the last transaction that started before the last checkpoint. Any earlier transactions would have committed previously and would have been written to the database at the checkpoint. Therefore, we need only redo the one that was active at the checkpoint and any subsequent transactions for which both start and commit records appear in the log. If a transaction is active at the time of failure, the transaction must be undone. If transactions are performed concurrently, we redo all transactions that have committed since the checkpoint and undo all transactions that were active at the time of the crash.

Example: Use of UNDO/REDO with checkpointing

See the previous example on use of UNDO/REDO (Page 25). If we assume that a checkpoint occurred at point tc then we would know that the changes made by transactions T_2 and T_3 had been written to secondary storage. In this case, the recovery manager would be able to omit the redo for these two transactions. However, the recovery manager would have to redo transactions T_4 and T_5, which have committed since the checkpoint, and undo transactions T, and T_6, which were active at the time of the crash.

Generally, checkpointing is a relatively inexpensive operation, and it is often possible to take three or four checkpoints an hour. In this way, no more than 15 - 20 minutes of work will need to be recovered.

Recovery Techniques

The particular recovery procedure to be used is dependent on the extent of the damage that has occurred to the database. We consider two cases:

- If the database has been extensively damaged, for example a disk head crash has occurred and destroyed the database, then it is necessary to restore the last backup copy of the database and reapply the update operations of committed transactions using the log file. This assumes, of course, that the log file has not been damaged as well. It is recommended that, where possible, the log file be stored on a disk separate from the main database files. This reduces the risk of both the database files and the log file being damaged at the same time.
- If the database has not been physically damaged but has become inconsistent, for example the system crashed while transactions were executing, then it is necessary to undo the changes that caused the inconsistency. It may also be necessary to redo some transactions to ensure that the updates they performed have reached secondary storage. Here, we do not need to use the backup copy of the database but can restore the database to a consistent state using the **before-** and **after-images** held in the log file.

We now look at two techniques for recovery from the latter situation, that is, the case where the database has not been destroyed but is in an inconsistent state. The techniques, known as **deferred update** and **immediate update,** differ in the way that updates are written to secondary storage. We also look briefly at an alternative technique called **shadow paging.**

Recovery techniques using deferred update

Using the ***deferred update*** recovery protocol, updates are not written to the database until after a transaction has reached its commit point. If a transaction fails before it reaches this point (commit), it will not have modified the database and so no undoing of changes will be necessary. However, it may he necessary to redo the updates of committed transactions, as their effect may not have reached the database. In this case, we use the log file to protect against system failures in the following way:

- When a transaction starts, write a *transaction start* record to the log.

- When any write operation is performed, write a log record containing all the log data specified previously (excluding the before-image of the update). Do not actually write the update to the database buffers or the database itself.
- When a transaction is about to commit, write a *transaction commit* log record, write all the log records for the transaction to disk, and then commit the transaction. Use the log records to perform the actual updates to the database.
- If a transaction aborts, ignore the log records for the transaction and do not perform the writes.

Note that we write the log records to disk before the transaction is actually committed, so that if a system failure occurs while the actual database updates are in progress, the log records will survive and the updates can be applied later. In the event of a failure, we examine the log to identify the transactions that were in progress at the time of failure. Starting at the last entry in the log file, we go back to the most recent checkpoint record:

- Any transaction with *transaction start* and *transaction commit* log records should be **redone.** The redo procedure performs all the writes to the database using the after-image log records for the transactions, *in the order in which they were written to the log* If this writing has been performed already, before the failure, the write has no effect on the data item, so there is no damage done if we write the data again (that is, the operation is **idempotent).** However, this method guarantees that we will update any data item that was not properly updated prior to the failure.
- For any transactions with *transaction start* and *transaction abort* log records, we do nothing since no actual writing was done to the database, so these transactions do not have to be undone.

If a second system crash occurs during recovery, the log records are used again to restore the database. With the form of the write log records, it does not matter how many times we redo the writes.

Recovery techniques using immediate update

Using the ***immediate update*** recovery protocol, updates are applied to the database as they occur without waiting to reach the commit point. As well as having to redo the updates of committed transactions following a failure, it may now be necessary to undo the effects of transactions that had not committed at the time of failure, In this case, we use the log file to protect against system failures in the following way:

- When a transaction starts, write a *transaction start* record to the log.
- When a write operation is performed, write a record containing the necessary data to the log file.
- Once the log record is written, write the update to the database buffers.
- The updates to the database itself are written when the buffers are next flushed to secondary storage.
- When the transaction commits, write a *transaction commit* record to the log.

It_is essential that log records (or at least certain parts of them) are written *before* the corresponding write to the database. This is known as the **write-ahead log protocol.** If updates were made to the database first, and failure occurred before the log record was written, then the recovery manager would have no way of undoing (or redoing) the operation. Under the write-ahead log protocol, the recovery manager can safely assume that, if there is no *transaction commit* record in the log file for a particular transaction then that transaction was still active at the time of failure and must therefore be undone.

If a transaction aborts, the log can be used to undo it since it contains all the old values for the updated fields. As a transaction may have performed several changes to an item, the wri"tes are undone *in reverse order*. Regardless of whether the transaction's writes have been applied to the database itself, writing the before-images guarantees that the database is restored to its tate prior to the start of the transaction.

If the system fails, recovery involves using the log to undo or redo transactions:

- For any transaction for which both a *transaction start* and *transaction commit* record appear in the log, we r edo using the log records to write the after-image of updated fields, as described above. Note that if the new values have already been written to the database, these writes, although unnecessary, will have no effect. However, any write that did not actually reach the database will now be performed.
- For any transaction for which the log contains a *transaction start* record but not a *transaction commit* record, we need to undo that transaction. This time the log records are used to write the before-image of the affected fields, and thus restore the database to its state prior to the transaction's start. The undo operations are performed *in the reverse order to which they were written to the log.*

Shadow paging

An alternative to the log-based recovery schemes described above is shadow paging. This scheme maintains two-page tables during the life of a transaction: a *current* page table and a *shadow* page table. When the transaction starts, the two-page tables are the same. The shadow page table is never changed thereafter, and is used to restore the database in the event of a system failure. During the transaction, the current page table is used to record all updates to the database. When the transaction completes, the current page table becomes the shadow page table. Shadow paging has several advantages over the log-based schemes: the overhead of maintaining the log file is eliminated, and recovery is significantly faster since there is no need for° undo or redo operations. However, it has disadvantages as well, such as data fragmentation and the need for periodic garbage collection to reclaim inaccessible blocks.

Recovery in a Distributed DBMS

Distributed Database Management Systems (Distributed DBMS or DDBMS) consists of a logically interrelated collection of databases physically distributed over a computer network, each under the control of a local DBMS. In a DDBMS, **distributed transactions** (transactions that access data at more than one site) are divided into a number of suo- transactions, one for each site that has to be accessed. In such a system, atomicity has to be maintained for both the subtransactions and the overall (global) transaction. The techniques described above can be used to ensure the atomicity of

subtransactions. Ensuring atomicity of the global transaction means ensuring that the subtransactions either all commit or all abort. The two common protocols for distributed recovery are known as two-phase commit (2PC) and three-phase commit (3PC).

NB: Distributed DBMSs will be discussed later.

Database Security

This section describes the scope of database security and discusses why organizations must take potential threats to their computer systems seriously. The range of threats and their consequences on computer systems is also identified.

Database Security - Database security is the mechanisms that protect the database against intentional or accidental threats.

Security considerations apply not only to the data..held in a database: breaches of security may affect other parts of the system, which may in turn affect the database. Consequently, database security encompasses hardware, software, people, and data. To effectively implement security requires appropriate controls, which are defined in specific mission objectives for the system. This need for security, while often having been neglected or overlooked in the past, is now increasingly recognized by organizations. The reason for this turnaround is the increasing amounts of crucial corporate data being stored on computer and the acceptance that any loss or unavailability of this data could prove to be disastrous.

A database represents an essential corporate resource that should be properly secured using appropriate controls. We consider database security in relation to the following situations:

1) Theft and fraud.
2) Loss of confidentiality (secrecy).
3) Loss of privacy.
4) Loss of integrity.
5) Loss of availability.

These situations broadly represent areas in which the organization should seek to reduce risk, that is the possibility of incurring loss or damage. In some situations, these areas are closely related such that an activity that leads to loss in one area may also lead to loss in another. In addition, events such as fraud or loss of privacy may arise because of either intentional or unintentional acts, and do not necessarily result in any detectable changes to the database or the computer system.

Theft and fraud affect not only the database environment but also the entire organization. As it is people who perpetrate such activities, attention should focus on reducing the opportunities for this occurring. Theft and fraud do not necessarily alter data, as is the case for activities that result in either loss of confidentiality or loss of privacy.

Confidentiality refers to the need to maintain secrecy over data, usually only that which is critical to the organization, whereas privacy refers to the need to protect data about individuals. B·reaches of security resulting in loss of confidentiality could, for instance, lead to loss of competitiveness, and loss of privacy could lead to legal action being taken against the organization.

LQss of data integrity results in invalid or corrupted data, which may seriously affect the operation of an organization. Many organizations are now seeking virtually continuous operation, the so-called 24/7 availability (that is, 24 hours a day, 7 days a week). Loss of availability means that the data, or the system, or both cannot be accessed, which can seriously affect an organization's financial performance. In some cases, events that cause a system to be unavailable may also cause data corruption.

Database security aims to minimize losses caused by anticipated events in a cost-effective manner without unduly constraining the users. In recent times, computer-based criminal activities have significantly increased and are forecast to continue to rise over the next few years.

Threats

Threat - A threat is any situation or event, whether intentional or accidental, that may adversely affect a system and consequently the organization.

A threat may be caused by a situation or event involving a person, action, or circumstance that is likely to bring harm to an organization. The harm may be tangible, such as loss of hardware, software, or data, or intangible, such as loss of credibility or client confidence. The problem facing any organization is to identify all possible threats. Therefore, as a minimum, an organization should invest time and effort in identifying the most serious threats.

While some types of threat can be either intentional or unintentional, the impact remains the same. Intentional threats involve people and may be perpetrated by both authorized users and unauthorized users, some of whom may be external to the organization.

Any threat must be viewed as a potential breach of security, which if successful, will have a certain impact. The table below presents examples of various types of threat, listed under the area on which they may have an impact. For example, 'viewing and disclosing unauthorized data' as a threat may result in theft and fraud, loss of confidentiality, and loss of privacy for the organization.

Threat	Theft and Fraud	Loss of Confidentiality	Loss of Privacy	Loss of Inte2ritv	Loss of Availability
Using another person's means of access					
Unauthorized amendment or copying of data					
Program alteration				1	
Inadequate policies and procedures that allow a mix of confidential and normal output			1		
Wire tapping					
Illegal entry by hacker					
Blackmail					
Creating "trap door" irito system					
Theft of data, programs & equipment					

Failure of security mechanism, giving greater access than normal					
Staff shortage or strike					
Inadequate staff training.					
Viewing and disclosing unauthorized data.					
Electronic interference & radiation					
Data corruption owing to power loss or surge.					
Fire (electrical fault, lightning strike, arson), flood, bomb.					
Physical damage to equipment					
Breaking cables or disconnection of cables.					
Introduction of viruses.					

The extent that an organization suffers as a result of a threat's succeeding depends upon a number of factors, such as the existence of countermeasures and contingency plans. For example, if a hardware failure occurs corrupting secondary storage, all processing activity must cease until the problem is resolved. The recovery will depend upon a number of factors, which include when the last backups were taken and the time needed to restore the system.

An organization needs to identify the types of threat it may be subjected to and initiate appropriate plans and countermeasures, bearing in mind the costs of implementing them. Obviously, it may not be cost-effective to spend considerable time, effort, and money on potential threats that may result only in minor inconvenience. The organization's business may also influence the types of threat that should be considered, some of which may be rare. However, rare events should be taken into account, particularly if their impact would be significant.

Countermeasure - Computer-Based Controls

The types of countermeasure to threats on computer systems range from physical controls to administrative procedures. Despite the range of computer-based controls that are available, it is worth noting that, generally, the security of a DBMS is only as good as that of the operating system, owing to their close association. In this section we focus on the following computer-based security controls for a multi-user environment (some of which may not be available in the PC environment):

1) Authorization.
2) Access controls.
3) Views.
4) Backup and recovery.
5) Integrity.
6) Encryption.
7) RAID technology.

1) Authorization

Authorization - Authorization is the granting of a right or privilege that enables a subject to have legitimate access to a system or system's object.

Authorization controls can be built into the software, and govern not only what system or object a specified user can access, but also what the user may do with it. The process of authorization involves authentication of subjects requesting access to objects, where 'subject' represents a user or program and 'object' represents a database table, view, procedure, trigger, or any other object that can be created within the system.

Authentication - Authentication is a mechanism that determines whether a user is who he or she claims to be.

A system administrator is usually responsible for allowing users to have access to a computer system by creating individual user accounts. Each user is given a unique identifier, which is used by the operating system to determine who they are. Associated with each identifier is a password, chosen by the user and known to the operating system, which must be supplied to enable the operating system to verify (or authenticate) who the user claims to be.

This procedure allows authorized use of a computer system but does not necessarily authorize access to the DBMS or any associated application programs. A separate, similar procedure may have to be undertaken to give a user the right to use the DBMS. The responsibility to authorize use of the DBMS usually rests with the Database Administrator (DBA), who must also set up individual user accounts and passwords using the DBMS itself.

Some DBMSs maintain a list of valid user identifiers and associated passwords, which can be distinct from the operating system's list. However, other DBMSs maintain a list whose entries are validated against the operating system's list based on the current user's login identifier. This prevents a user from logging on to the DBMS with one name, having already logged on to the operating system using a different name.

2) Access Controls

The typical way to provide access controls for a database system is based on the granting and revoking of privileges. A **privilege** allows a user to create or access (that is read, write, or modify) some database object (such as a relation, view, or index) or to run certain DBMS utilities. Privileges are granted to users to accomplish the tasks required for their jobs. As excessive granting of unnecessary privileges can compromise security: a privilege should only be granted to a user if that user cannot accomplish his or her work without that privilege. A user who creates a database object such as a relation or a view automatically gets all privileges on that object. The DBMS subsequently keeps track of how these privileges are granted to other users, and possibly revoked, and ensures that at all times only users with necessary privileges can access an object.

Discretionary Access Control (DAC)

Most commercial DBMSs provide an approach to managing privileges that uses SQL called **Discretionary Access Control (DAC).** The SQL standard supports DAC through the GRANT and REVOKE commands. The GRANT command gives privileges to users, and the REVOKE command takes away privileges. In discretionary access control, each user is given appropriate access rights (privileges) on specific database objects.

Typically, users obtain privileges when they create an object and can pass some or all of these privileges to other users at their discretion.

Discretionary access control, while effective, has certain weaknesses. In particular, an unauthorized user can trick an authorized user into disclosing sensitive data. For example, an unauthorized user such as an Assistant in a company can create a relation (table) to capture new client details and give access privileges to an authorized user such as a Manager without their knowledge. The Assistant can then alter some application programs that the Manager uses to include some hidden instruction to copy sensitive data from the Client relation that only the Manager has access to, into the new relation created by the Assistant. The unauthorized user, namely the Assistant, now has a copy of the sensitive data, namely new clients of company, and to cover up his or her actions now modifies the altered application programs back to the original form.

Clearly, an additional security approach is required to remove such loopholes, and this requirement is met in an approach called Mandatory Access Control (MAC), which is discussed in detail below. Although discretionary access control is typically provided by most commercial DBMSs, only some also provide support for mandatory access control.

Mandatory Access Control (MAC)

Mandatory Access Control (MAC) is based on system-wide policies that cannot be changed by individual users. In this approach each database object is assigned a *security class* and each user is assigned a *clearance* for a security class, and *rules* are imposed on reading and writing of database objects by users. The DBMS determines whether a given user can read or write a given object based on certain rules that involve the security level of the object and the clearance of the user. These rules seek to ensure that sensitive data can never be passed on to another user without the necessary clearance. The SQL standard does not include support for MAC.

A popular model for MAC is called Bell-LaPadula model (Bell and LaPadula, 1974), which is described in terms of **objects** (such as relations, views, tuples, and attributes), **subjects** (such as users and programs), **security classes,** and **clearances.** Each database object is assigned a *security class,* and each subject is assigned a *clearance* for a security class. The security classes in a system are ordered, with a most secure class and a least secure class. For our discussion of the model, we assume that there are four classes: *top secret (TS), secret (SJ, confidential (CJ,* and *unclassified (U),* and we denote the class of an object or subject A as *class (A).* Therefore for this system, *TS*> S> C > *U,* where A> B means that class A data has a higher security level than class B data.

The Bell-LaPadula model imposes two restrictions on all reads and writes of database objects:

i.) **Simple Security Property:** Subject S is allowed to read object O only if class (S) >= class *(OJ.* For example, a user with *TS* clearance can read a relation with C clearance, but a user with C clearance cannot read a relation with *TS* classification.

ii.) ***_Property:** Subject S is allowed to write object *O* only if class (S) <= class *(OJ.* For example, a user with S clearance can only write objects with Sor *TS* classification.

If discretionary access controls are also specified, these rules represent additional restrictions. Thus to read or write a database object, a user must have the necessary

privileges provided through the SQL GRANT command and the security classes •of the user and the object must satisfy the restrictions given above.

Multilevel Relations and Polyinstantiation

In order to apply mandatory access control policies in a relational DBMS, a security class must be assigned to each database object. The objects can be at the granularity of relations (tables), tuples (records), or even individual attribute values. Assume that each tuple is assigned a security class. This situation leads to the concept of a **multilevel relation,** which is a relation that reveals different tuples to users with different security clearances.

For example, the Client relation with an additional attribute displaying the security class for each tuple is shown in the figure below.

Users with *S (Secret)* and *TS (Top Secret)* clearance will see all tuples in the Client relation. However, a user with C *(Confidential)* clearance will only see the first two tuples and a user with *U (Unclassified)* clearance will see no tuples at all. Assume that a user with clearance C wishes to enter a tuple (CR74, Wilberforce Wanyama) into the *Client* relation, where the primary key of the relation is *clientNo.* This insertion is disallowed because it violates the primary key constraint for this relation. However, the inability to insert this new tuple informs the user with clearance C that a tuple exists with a primary key value of CR74 at a higher security class than C. This compromises the security requirement that users should not be able to infer any information about objects that have a higher security classification.

This problem of inference can be solved by including the security classification attribute as part of the primary key for a relation. In the above example, the insertion of the new tuple into the Client relation is allowed, and the relation instance is modified as shown in figure (b). Users with clearance C see the first two tuples and the newly added tuple, but users with clearance Sor *TS* see all five tuples. The result is a relation with two tuples with a clientNo of CR74, which can be confusing. This situation may be dealt with by assuming that the tuple with the higher classification takes priority over the other, or by only revealing a single tuple according to the user's clearance. The presence of data objects that appear to have different values to users with different clearances is called **polyinstantiation.**

ClientNo	FName	LName	telNo	preType	max.Rent	securityClass
CR76	Peter	Kioko	0722-457834	Flat	12,000	C
CR56	Mary	Sanaipei	0735-986898	Flat	25,000	C
CR74	Alice	Atieno	0721-897822	House	20,000	S
CR62	Simon	Makonde	0733-493478	Flat	6,500	S

(a) The client relation with an additional attribute displaying the security class for each tuple

ClientNo	FName	LName	telNo	preType	maxRent	sectrrityClass
CR76	Peter	Kioko	0722-457834	Flat	12,000	C
CR56	Mary	Sanaipei	0735-986898	Flat	25,000	C
CR74	Alice	Atieno	0721-897822	House	20,000	S
CR62	Simon	Makonde	0733-493478	Flat	6,500	S
CR74	Wilberforce	Wanyama	0712-897623	House	8,200	C

(b) The client relation with two tuples displaying clientNo as CR74. The primary keys for this relation is (ClientNo,securityClass).

Although mandatory access control does address a major weakness of discretionary access control, a major disadvantage of MAC is the rigidity of the MAC environment. For example, MAC policies are often established by database or systems administrators, and the classification mechanisms are sometimes considered to be inflexible.

3) Views

A **View** is the dynamic result of one or more relational operations operating on the base relations (tables) to produce another relation. A view is a *virtual relation* that does not necessarily exist in the database but can be produced upon request by a particular user, at the time of request. To the database user, a view appears just like a real table, with a set of named columns and rows of data. However, unlike a base table, a view does not necessarily exist in the database as a stored set of data values. Instead, a view is defined as a query on one or more base tables or views.

The view mechanism provides a powerful and flexible security mechanism by hiding parts of the database from certain users. The user is not aware of the existence of any attributes or rows that are missing from the view. A view can be defined over several relations with a user being granted the appropriate privilege to use it, but not to use the base relations. In this way, using a view is more restrictive than simply having certain privileges granted to a user on the base relation(s).

4) Backup and Recovery

Backup - The process of periodically taking a copy of the database and log file (and possibly programs) on to offline storage media.

A DBMS should provide backup facilities to assist with the recovery of a database following failure. It is always advisable to make backup copies of the database and log file at regular intervals and to ensure that the copies are in a secure location. In the event of a failure that renders the database unusable, the backup copy and the details captured in the log file are used to restore the database to the latest possible consistent state.

Journaling - The process of keeping and maintaining a log file (or journal) of all changes made to the database to enable recovery to be undertaken effectively in the event of a failure.

A DBMS should provide logging facilities, sometimes referred to as journaling, which keep track of the current state of transactions and database changes, to provide support for recovery procedures. The advantage of journaling is that, in the event of a failure, the database can be recovered to its last known consistent state using a backup copy of the database and the information contained in the log file. If no journaling is enabled on a

failed system, the only means of recovery is to restore the database using the latest backup version of the database. However, without a log file, any changes made after the last backup to the database will be lost.

5) Integrity

Integrity Constraints are a set of rules that the database is not permitted to violate.

Integrity constraints also contribute to maintaining a secure database system by preventing data from becoming invalid, and hence giving misleading or incorrect results.

6) Encryption

If a database system holds particularly sensitive data, it may be deemed necessary to encode it as a precaution against possible external threats or attempts to access it. Some DBMSs provide an encryption facility for this purpose. The DBMS can access the data (after decoding it), although there is a degradation in performance because of the time taken to decode it. Encryption also protects data transmitted over communication lines. There are a number of techniques for encoding data to conceal the information; some are termed 'irreversible' and others 'reversible'. Irreversible techniques, as the name implies, do not permit the original data to be known. However, the data can be used to obtain valid statistical information. Reversible techniques are more commonly used. To transmit data securely over insecure networks requires the use of a cryptosystem, which includes:

- An *encryption key* to encrypt the data (plaintext).
- An *encryption algorithm* that, with the encryption key, transforms the plaintext into *ciphertext.*
- A *decryption key* to decrypt the ciphertext.
- A *decryption algorithm* that, with the decryption key, transforms the ciphertext back into plaintext.

One technique, called **symmetric encryption,** uses the same key for both encryption and decryption and relies on safe communication lines for exchanging the key. However, most users do not have access to a secure communication line and, to be really secure, the keys need to be as long as the message. However, most working systems are based on user keys shorter than the message. One scheme used for encryption is the **Data Encryption Standard** (DES), which is a standard encryption algorithm developed by IBM. This scheme uses one key for both encryption and decryption, which must be kept secret, although the algorithm need not be. The algorithm transforms each 64-bit block of plaintext using a 56-bit key. The DES is not universally regarded as being very secure, and some authors maintain that a larger key is required. For example, a scheme called PGP (Pretty Good Privacy) uses a 128-bit symmetric algorithm for bulk encryption of the data it sends.

Keys with 64 bits are now probably breakable by major governments with special hardware, albeit at substantial cost. However, this technology will be within the reach of organized criminals, major organizations, and smaller governments in a few years. While it is envisaged that keys with 80 bits will also become breakable in the future, it is probable that keys with 128 bits will remain unbreakable for the foreseeable future. The terms 'strong authentication' and 'weak authentication' are sometimes used to distinguish

between algorithms that, to all intents and purposes, cannot be broken with existing technologies and knowledge (strong) from those that can be (weak).

Another type of cryptosystem uses different keys for encryption and decryption, and is referred to as **asymmetric encryption.** One example is **public key** cryptosystems, which use two keys, one of which is public and the other private. The encryption algorithm may also be public, so that anyone wishing to send a user a message can use the user's publicly known key in conjunction with the algorithm to encrypt it. Only the owner of the private key can then decipher the message. Public key cryptosystems can also be used to send a 'digital signature' with a message and prove that the message came from the person who claimed to have sent it. The most well known asymmetric encryption is **RSA** (the name is derived from the initials of the three designers of the algorithm).

Generally, symmetric algorithms are much faster to execute on a computer than those that are asymmetric. However, in practice, they are often used together, so that a public key algorithm is used to encrypt a randomly generated encryption key, and the random key is used to encrypt the actual message using a symmetric algorithm.

7) RAID (Redundant Array of Independent Disks)

The hardware that the DBMS is running on must be *fault-tolerant,* meaning that the DBMS should continue to operate even if one of the hardware components fails. This suggests having redundant components that can be seamlessly integrated into the working system whenever there is one or more component failures. The main hardware components that should be fault-tolerant include disk drives, disk controllers, CPU, power supplies, and cooling fans. Disk drives are the most vulnerable components with the shortest times between failure of any of the hardware components.

One solution is the use of **Redundant Array of Independent Disks (RAID)** technology. RAID originally stood for *Redundant Array of Inexpensive Disks,* but more recently the 'I' in RAID has come to stand for *Independent.* RAID works on having a large disk array comprising an arrangement of several independent disks that are organized to improve reliability and at the same time increase performance.

Performance is increased through *data striping:* the data is segmented into equal-size partitions (the *striping unit)* which are transparently distributed across multiple disks. This gives the appearance of a single large, fast disk where in actual fact the data is distributed across several smaller disks. Striping improves overall 1/0 performance by allowing multiple I/Os to be serviced in parallel. At the same time, data striping also balances the load among disks.

Reliability is improved through storing redundant information across the disks using a *parity* scheme or an *error-correcting* scheme, such as Reed-Solomon codes. In a parity scheme, each byte may have a parity bit associated with it that records whether the number of bits in the byte that are set to 1 is even or odd. If the number of bits in the byte becomes corrupted, the new parity of the byte will not match the stored parity. Similarly, if the stored parity bit becomes corrupted, it will not match the data in the byte. Error-correcting schemes store two or more additional bits, and can reconstruct the original data if a single bit becomes corrupt. These schemes can be used through striping bytes across disks.

Reading Exercise.

Read and make notes on the different disk configurations with RAID (RAID *levels)* i.e. RAID 0, RAID 1... RAID 5. For each of these configurations, you should have a detailed explanation accompanied by a (well labeled) diagrammatic representation of the configuration.

DBMSs and Web Security

Internet communication relies on TCP/IP as the underlying protocol. However, TCP/IP and HTTP were not designed with security in mind. Without special software, all Internet traffic travels 'in the clear' and anyone who monitors traffic can read it. This form of attack is relatively easy to perpetrate using freely available 'packet sniffing' software, since the Internet has traditionally been an open network. Consider, for example, the implications of credit card numbers being intercepted by unethical parties during transmission when customers use their cards to purchase products over the Internet. The challenge is to transmit and receive information over the Internet while ensuring that:

- lt is inaccessible to anyone but the sender and receiver (privacy).
- lt has not been changed during transmission (integrity).
- The receiver can be sure it came from the sender (authenticity).
- The sender can be sure the receiver is genuine (non-fabrication).
- The sender cannot deny he or she sent it (non-repudiation).

However, protecting the transaction only solves part of the problem. Once the information has reached the Web server, it must also be protected there. With the three-tier architecture that is popular in a Web environment, we also have the complexity of ensuring secure access to, and of, the database. Today, most parts of such architecture can be secured, but it generally requires different products and mechanisms.

One other aspect of security that has to be addressed in the Web environment is that information transmitted to the client's machine may have executable content. For example, HTML pages may contain ActiveX controls, JavaScript/VB Script, and/or one or more Java applets. Executable content can perform the following malicious actions, and measures need to be taken to prevent them:

- Corrupt data or the execution state of programs.
- Reformat complete disks.
- Perform a total system shutdown.
- Collect and download confidential data, such as files or passwords, to another site.
- Usurp identity and impersonate the user or user's computer to attack other targets on the network.
- Lock up resources making them unavailable for legitimate users and programs.
- Cause non:fatal but unwelcome effects, especially on output devices.

Earlier sections identified general security mechanisms for database systems. However, the increasing accessibility of databases on the public Internet and private intranets

requires a re-analysis and extension of these approaches. This section addresses some of the issues associated with database security in these environments.

1) Proxy Servers

In a Web environment, a proxy server is a computer that sits between a Web browser and a Web server. It intercepts all requests to the Web server to determine if it can fulfill the requests itself. If not, it forwards the requests to the Web server. Proxy servers have two main purposes: to improve performance and filter requests.

Improve performance

Since a proxy server saves the results of all requests for a certain amount of time, it can significantly improve performance for groups of users. For example, assume that user A and user B access the Web through a proxy server. First, user A requests a certain Web page and, slightly later, user B requests the same page. Instead of forwarding the request to the Web server where that page resides, the proxy server simply returns the cached page that it had already fetched for user A. Since the proxy server is often on the same network as the user, this is a much faster operation.

Filter requests

Proxy servers can also be used to filter requests. For example, an organization might use a proxy server to prevent its employees from accessing a specific set of Web sites.

2) Firewalls

The standard security advice is to ensure that Web servers are unconnected to any in-house networks and regularly backed up to recover from inevitable attacks. When the Web server has to be connected to an internal network, for example to access the company database, firewall technology can help to prevent unauthorized access, provided it has been installed and maintained correctly.

A **firewall** is a system designed to prevent unauthorized access to or from a private network. Firewalls can be implemented in both hardware and software, or a combination of both. They are frequently used to prevent unauthorized Internet users from accessing private networks connected to the Internet, especially intranets. All messages entering or leaving the intranet pass through the firewall, which examines each message and blocks those that do not meet the specified security criteria. There are several types of firewall technique:

i.) **Packet filter** - A packet filter looks at each packet entering or leaving the network and accepts or rejects it based on user-defined rules. Packet filtering is a fairly effective mechanism and transparent to users, but can be difficult to configure. In addition, it is susceptible to IP spoofing. (IP spoofing is a technique used to gain unauthorized access to computers, whereby the intruder sends messages to a computer with an IP address indicating that the message is coming from a trusted port.).

ii.) **ApplicatiQn gateway** - An application gateway applies security mechanisms to specific applications, such as FTP and Telnet servers. This is a very effective mechanism, but can degrade performance.

iii.) **Circuit-level gateway** - a Circuit-level gateway applies security mechanisms when a TCP or UDP (User Datagram Protocol) connection is established. Once the connection has been made, packets can flow between the hosts without further checking.

iv.) **Proxy server** - A proxy server intercepts all messages entering and leaving the network. The proxy server in effect hides the true network addresses.

In practice, many firewalls provide more than one of these techniques. A firewall is considered a first line of defense in protecting private information. For greater security, data can be encrypted.

3) Message Digest Algorithms and Digital Signatures

A message digest algorithm, or one-way hash function, takes an arbitrarily sized string (the *message)* and generates a fixed-length string (the *digest* or *hash).* A digest has the following characteristics:

i.) It should be computationally infeasible to find another message that will generate the same digest.

ii.) The digest does not reveal anything about the message.

A digital signature *(not to be confused with a digital certificate)* is an electronic signature that can be used to authenticate the identity of the sender of a message or the signer of a document, and possibly to ensure that the original content of the message or document that has been sent is unchanged. Digital signatures are easily transportable, cannot be imitated by someone else, and can be automatically time-stamped. The ability to ensure that the original signed message arrived means that the sender cannot easily repudiate it later.

A digital signature can be used with any kind of message, whether it is encrypted or not, simply so that the receiver can be sure of the sender's identity and that the message arrived intact.

Like a handwritten signature, a digital signature has many useful properties:

i.) Its authenticity can be verified, using a computation based on the corresponding public key.

ii.) It cannot be forged (assuming the private key is kept secret).

iii.) It is a function of the data signed and cannot be claimed to be the signature for any other data.

iv.) The signed data cannot be changed, otherwise the signature will no longer verify the data as being authentic.

Some digital signature algorithms use message digest algorithms for parts of their computations; others, for efficiency, compute the digest of a message and digitally sign the digest rather than signing the message itself.

How Digital Signatures Work

Assume you were going to send you Advanced Database Systems assignment to your lecturer. You want to give the lecturer the assurance that it was unchanged from what you sent and that it is really from you.

i.) You copy-and-paste the assignment into an e-mail note.

ii.) Using special software, you obtain a message hash (mathematical summary) of the assignment.

iii.) You then use a private key that you have previously obtained from a public-private key authority to encrypt the hash.

iv.) The encrypted hash becomes your digital signature of the message. (Note that it will be different each time you send a message.)

At the other end, your lawyer receives the message.

i.) To make sure it's intact and from you, the lecturer makes a hash of the received message.

ii.) The lecturer then uses your public key to decrypt the message hash or summary.

iii.) If the hashes match, the received message is valid.

4) Digital Certificates

A digital certificate is an attachment to an electronic message used for security purposes, most commonly to verify that a user sending a message is who he or she claims to be, and to provide the receiver with the means to encode a reply.

An individual wishing to send an encrypted message applies for a digital certificate from a Certificate Authority (CA). The CA issues an encrypted digital certificate containing the applicant's public key and a variety of other identification information. The CA makes its own public key readily available through printed material or perhaps on the Internet.

The recipient of an encrypted message uses the CA's public key to decode the digital certificate attached to the message, verifies it as issued by the CA, and then obtains the sender's public key and identification information held within the certificate. With this information, the recipient can send an encrypted reply.

Clearly, the CA's role in this process is critical, acting as a go-between for the two parties. In a large, distributed complex network like the Internet, this third-party trust model is necessary as clients and servers may not have an established mutual trust yet both parties want to have a secure session. However, because each party trusts the CA, and because the CA is vouching for each party's identification and trustworthiness by signing their certificates, each party recognizes and implicitly trusts each other.

5) Kerberos

Kerberos is a server of secured user names and passwords (named after the three-headed monster in Greek mythology that guarded the gate of hell). The importance of Kerberos is that it provides one centralized security server for all data and resources on the network. Database access, login, authorization control, and other security features are

centralized on trusted Kerberos servers. Kerberos has a similar function to that of a Certificate server: to identify and validate a user.

6) Secure Sockets Layer and Secure HTTP

Many large Internet product developers agreed to use an encryption protocol known as Secure Sockets Layer (SSL) developed by Netscape for transmitting private documents over the Internet. SSL works by using a private key to encrypt data that is transferred over the SSL connection. Both Netscape Navigator and Internet Explorer support SSL, and many Web sites use this protocol to obtain confidential user information, such as credit card numbers. The protocol, layered between application-level protocols such as HTTP and the TCP/IP transport-level protocol, is designed to prevent eavesdropping, tampering, and message forgery. Since SSL is layered under application-level protocols, it may be used for other application-level protocols such as FTP and NNTP.

Another protocol for transmitting data securely over the Web is Secure HTTP (S-HTTP), a modified version of the standard HTTP protocol. S-HTTP was developed by Enterprise Integration Technologies (EIT), which was acquired by Verifone, Inc. in 1995. Whereas SSL creates a secure connection between a client and a server, over which any amount of data can be sent securely, S-HTTP is designed to transmit individual messages securely. SSL and S-HTTP, therefore, can be seen as complementary rather than competing technologies.

By convention, Web pages that require an SSL connection start with **https:** instead of **http:.** Not all Web browsers and servers support SSL/S-HTTP.

Basically, these protocols allow the browser and server to authenticate one another and secure information that subsequently flows between them. Through the use of cryptographic techniques such as encryption, and digital signatures, these protocols:

i.) Allow Web browsers and servers to authenticate each other.

ii.) Permit Web site owners to control access to particular servers, directories, files, or services.

iii.) Allow sensitive information (for example, credit card numbers) to be shared between browser and server, yet remain inaccessible to third parties.

iv.) Ensure that data exchanged between browser and server is reliable, that is, cannot be corrupted either accidentally or deliberately, without detection.

A key component in the establishment of secure Web sessions using the SSL or S-HTTP protocols is the digital certificate. Without authentic and trustworthy certificates, protocols like SSL and S-HTTP offer no security at all.

7) Secure Electronic Transactions and Secure Transaction Technology

The Secure Electronic Transactions (SET) protocol is an open, interoperable standard for processing credit card transactions over the Internet, created jointly by Netscape, Microsoft, Visa, Mastercard, GTE, SAIC, Terisa Systems, and VeriSign. SET's goal is to allow credit card transactions to be as simple and secure on the Internet as they are in retail stores. To address privacy concerns, the transaction is split in such a way that the merchant has access to information about what is being purchased, how much it costs,

and whether the payment is approved, but no information on what payment method the customer is using. Similarly, the card issuer (for example, Visa) has access to the purchase price but no information on the type of merchandise involved.

Certificates are heavily used by SET, both for certifying a cardholder and for certifying that the merchant has a relationship with the financial institution.

Distributed DBMSs - Concepts and Design

Database technology has moved from a paradigm of data processing in which each application defined and maintained its own data, to one in which data is defined and administered centrally. During recent times, there have been rapid developments in network and data communication technology, epitomized by the Internet, mobile and wireless computing, intelligent devices, and grid computing. Now with the combination of these two technologies, distributed database technology may change the mode of working from centralized to decentralized. This combined technology is one of the major developments in the database systems area.

The previous sections have concentrated on centralized database systems, that is systems with a single logical database located at one site under the control of a single DBMS. This section discusses the concepts and issues of the **Distributed Database Management System** (DDBMS), which allows users to access not only the data at their own site but also data stored at remote sites.

Introduction

A major motivation behind the development of database systems is the desire to integrate the operational data of an organization and to provide controlled access to the data. Although integration and controlled access may imply centralization, this is not the intention. In fact, the development of computer networks promotes a decentralized mode of work. This decentralized approach mirrors the organizational structure of many companies, which are logically distributed into divisions, departments, projects, and so on, and physically distributed into offices, plants, factories, where each unit maintains its own operational data. The shareability of the data and the efficiency of data access should be improved by the development of a distributed database system that reflects this organizational structure, makes the data in all units accessible, and stores data close to the location where it is most frequently used.

Distributed DBMSs should help resolve the *islands of information* problem. Databases are sometimes regarded as electronic islands that are distinct and generally inaccessible places, like remote islands. This may he a result of geographical separation, incompatible computer architectures, incompatible communication protocols etc. Integrating the databases into a logical whole may prevent this way of thinking.

Concepts

Definitions:

Distributed database - A distributed database is a logically interrelated collection of Shared data (and a description of this data) physically distributed aver a computer network.

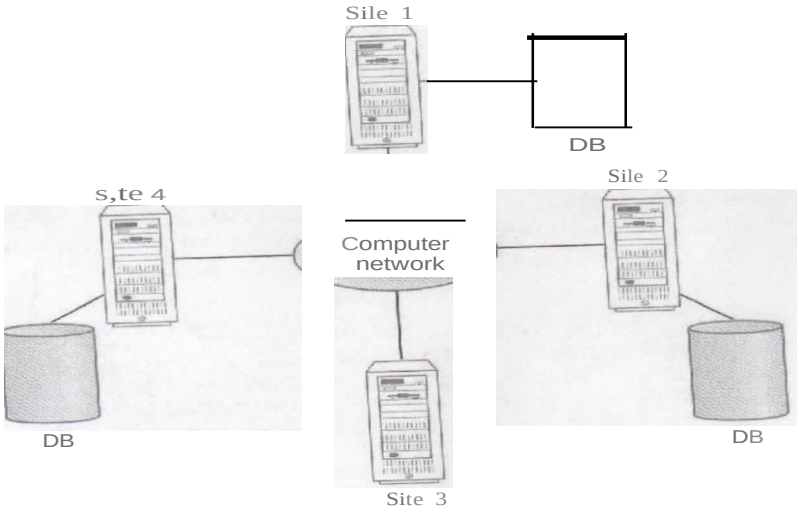
Distributed DBMS - A Distributed DBMS is the, software system that permits the management of the distributed database and makes the distribution transparent to users.

A **Distributed Database Management System** (DDBMS) consists of a single logical database that is split into a number of **fragments.** Each fragment is stored on one or more computers under the control of a separate DBMS, with the computers connected by a communications network. Each site is capable of independently processing user requests that require access to local data (that is, each site has some degree of local autonomy) and is also capable of processing data stored on other computers in the network.

Users access the distributed database via applications, which are classified as those that do not require data from other sites (local applications) and those that do require data from other sites (global applications). We require a DDBMS to have at least one global application. A DDBMS therefore has the following characteristics:

- A collection of logically related shared data.
- The data is split into a number of fragments.
- Fragments may be replicated.
- Fragments/replicas are allocated to sites.
- The sites are linked by a communications network.
- The data at each site is under the control of a DBMS.
- The DBMS at each site can handle local applications, autonomously.
- Each DBMS participates in at least one global application.

It is not necessary for every site in the system to have its own local database.



A distributed database management system

From the definition of the DDBMS, the system is expected to make the distribution **transparent (invisible)** to the user. Thus, the fact that a distributed database is split into fragments that can he stored on different computers and perhaps replicated, should be hidden from the user. The objective of transparency is to make the distributed system appear like a centralized system. This is sometimes referred to as the **fundamental principle** of distributed DBMSs. This requirement provides significant functionality for

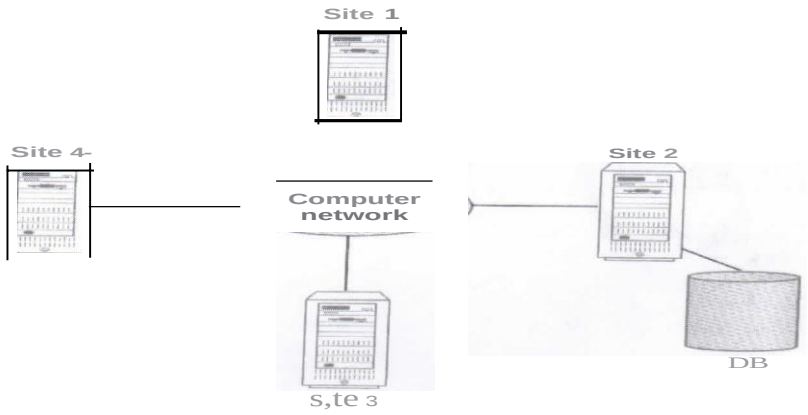
tht end-user but, unfortunately, creates many additional problems that have to be handled by the DDBMS.

Distributed processing

It is important to make a distinction between a distributed DBMS and distributed processmg.

Distributed processing - A centralized database that can be accessed over a computer network (Distributed Processing refers to any of a variety of computer systems that use more than one computer, or processor, to run an application).

The key point with the definition of a distributed DBMS is that the system consists of data that is physically distributed across a number of sites in the network. If the data is centralized, even though other users may be accessing the data over the network, we do not consider this to be a distributed DBMS, simply distributed processing. We illustrate the topology of distributed processing in figure below. Compare this figure, which has a central database at site 2, with the previous figure, which shows several sites each with their own database (DB).



Distributed Processing

Parallel DBMSs

We also make a distinction between a distributed DBMS and a parallel DBMS.

Parallel DBMS - A DBMS running across multiple processors and disks that is designed to execute operations in parallel, whenever possible, in order to improve performance.

Parallel processing - *Parallel processing is the simultaneous use of more than one CPU to execute a program. Ideally, parallel processing makes a program run faster because there are more CPUs running it. In practice, it is often difficult to divide a program in such a way that separate CPUs can execute different portions without interfering with each other.*

Parallel DBMSs are based on the premise that single processor systems can no longer meet the growmg requirements for cost-effective scalability, reliability, and performance. A powerful and financially attractive alternative to a single-processor-driven DBMS is a parallel DBMS driven by multiple processors. Parallel DBMSs link multiple, smaller

machines to achieve the same throughput as a single, larger machine, often with greater scalability and reliability than single-processor DBMSs.

To provide multiple processors with common access to a single database, a parallel DBMS must provide for shared resource management. Which resources are shared and how those shared resources are implemented, directly affects the performance and seal-ability of the system which, in tum, determines its appropriateness for a given application/environment.

The three main architectures for parallel DBMSs are:

i.) Shared Memory.

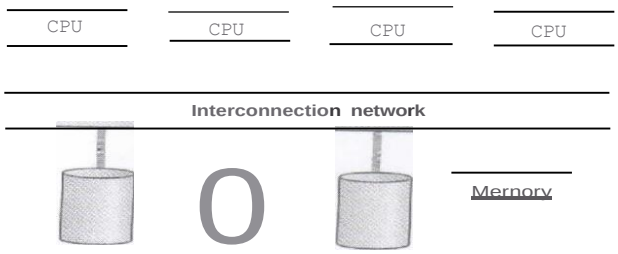
ii.) Shared Disk.

iii.) Shared Nothing.

Parallel Database Architectures.

i.) **Shared Memory**

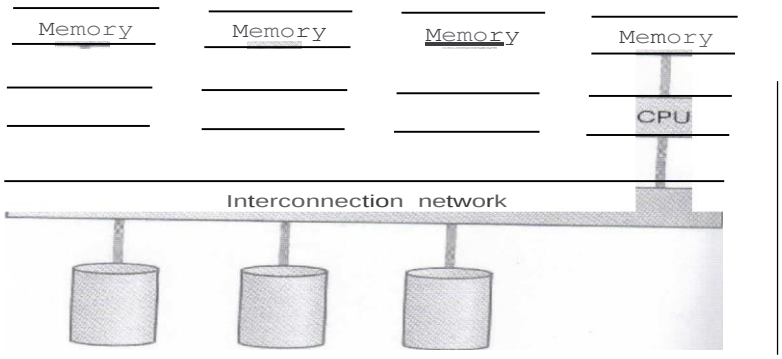
Shared memory is a tightly coupled architecture in which multiple processors within a single system share system memory. Known as symmetric multiprocessing (SMP), this approach has become popular on platforms ranging from personal workstations that support a few microprocessors in parallel, to large RISC (Reduced Instruction Set Computer) based machines, all the way up to the largest mainframes. This architecture provides high-speed data access for a limited number of processors, but it is not scalable beyond about 64 processors.



Shared memory architecture

ii.) **Shared Disk**

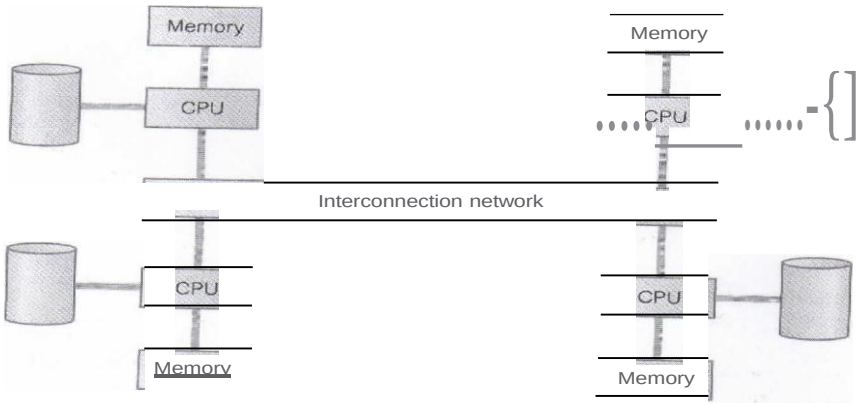
Shared disk is a loosely-coupled architecture optimized for applications that are inherently centralized and require high availability and performance. Each processor can access all disks directly, but each has its own private memory. Like the shared nothing architecture, the shared disk architecture eliminates the shared memory performance bottleneck. Unlike the shared nothing architecture, however, the shared disk architecture eliminates this bottleneck without introducing the overhead associated with physically partitioned data. Shared disk systems are sometimes referred to as **clusters.**



Shared disk architecture

iii.) **Shared Nothing**

Shared nothing, often known as massively parallel processing (MPP), is a multiple processor architecture in which each processor is part of a complete system, with its own memory and disk storage. The database is partitioned among all the disks on each system associated with the database, and data is transparently available to users on all systems. This architecture is more scalable than shared memory and can easily support a large number of processors. However, performance is optimal only when requested data is stored locally.



Shared nothing architecture

While the shared nothing definition sometimes includes distributed DBMSs, the distribution of data in a parallel DBMS is based solely on performance considerations. Further, the nodes of a DDBMS are typically geographically distributed, separately administered, and have a slower interconnection network, whereas the nodes of a parallel DBMS are typically within the same computer or within the same site.

Parallel technology is typically used for very large databases, or systems that have to process thousands of transactions per second. These systems need access to large volumes of data and must provide timely responses to queries. A parallel DBMS can use the underlying architecture to improve the performance of complex query execution using parallel scan, join, and sort techniques that allow multiple processor nodes automatically to share the processing workload.

Advantages and Disadvantages of DDBMSs

The distribution of data and applications has potential advantages over traditional centralized database systems. Unfortunately, there are also disadvantages. This section reviews the advantages and disadvantages of the DDBMS.

Advantages of Distributed Database Management Systems.

1) Reflects organizational structure

Many organizations are naturally distributed over several locations. For example, Kenya Commercial Bank has branches in different parts of Kenya (and some East African countries) and even different branches in the same towns. It is natural for databases used in such an institution to be distributed over these locations. Kenya Commercial Bank may keep a database at each branch containing details of account holders at that branch. The staff at a branch office will make local inquiries of the database. The company headquarters may wish to make global inquiries involving the access of data at all or a number of branches.

2) Improved shareability and local autonomy

The geographical distribution of an organization can reflected in the distribution of the data; users at one site can access data stored at other sites. Data can be placed at the site close to the users who normally use that data. In this way, users have local control of the data and they can consequently establish and enforce local policies regarding the use of this data. A global database administrator (DBA) is responsible for the entire system. Generally, part of this responsibility is devolved to the local level, so that the local DBA can manage the local DBMS. In the example of Kenya Commercial Bank, there could be a distributed database system in which there is a database for each branch but the database for a specific branch can be accessed from any other branch.

3) Improved availability

In a centralized DBMS, a computer failure terminates the operations of the DBMS. However, a failure at one site of a DDBMS, or a failure of a communication link making some sites inaccessible, does not make the entire system inoperable. Distributed DBMSs are designed to continue to function despite such failures. If a single node fails, the system may be able to reroute the failed node's requests to another site.

4) Improved reliability

As data may be replicated so that it exists at more than one site, the failure of a node or a communication link does not necessarily make the data inaccessible.

5) Improved performance

As the data is located near the site of 'greatest demand', and given the inherent parallelism of distributed DBMSs, speed of database access may be better than that achievable from a remote centralized database. Furthermore, since each site handles only a part of the entire database, there may not be the same contention for CPU and *VO* services as characterized by a centralized DBMS.

6)_ Cost Effectiveness (Economics)

It (now) generally costs much less to create a system of smaller computers with the equivalent power of a single large computer. This makes it more cost-effective for corporate divisions and departments to obtain separate computers. It is also much more cost-effective to add workstations to a network than to update a mainframe system.

The second potential cost saving occurs where databases are geographically remote and the applications require access to distributed data. In such cases, owing to the relative expense of data being transmitted across the network as opposed to the cost of local access, it may be much more economical to partition the application and perform the processing locally at each site.

7) Modular growth

In a distributed environment, it is much easier to handle expansion. New sites can be added to the network without affecting the operations of other sites. This flexibility allows an organization to expand relatively easily. Increasing database size can usually be handled by adding processing and storage power to the network. In a centralized DBMS, growth may entail changes to both hardware (the procurement of a more powerful system) and software (the procurement of a more powerful or more configurable DBMS).

8) Integration

Integration is one of the key advantages of the DBMS approach. No one package can provide all the functionality that an organization requires nowadays and therefore it is important for organizations to be able to integrate software components from different vendors to meet their specific requirements. DDBMSs provide for easy intergration.

9) Remaining competitive

There are a number of relatively recent developments that rely heavily on distributed database technology such as e-Business, computer-supported collaborative work, and workflow management. Many enterprises have had to reorganize their businesses and use distributed database technology to remain competitive. For example, while more people may not buy products from a certain company just because the Internet exists, the company may lose some of its market share if it does not allow clients to view and purchase products from it's different branches online now.

Disadvantages of Distributed Database Management Systems.

1) Complexity

A distributed DBMS that hides the distributed nature from the user and provides an acceptable level of performance, reliability, and availability is inherently more complex than a centralized DBMS. The fact that data can be replicated also adds an extra level of complexity to the distributed DBMS. If the software does not handle data replication adequately, there will be degradation in availability, reliability, and performance compared with the centralized system, and the advantages cited above will become disadvantages

2). Cost

Increased complexity means that we can expect the procurement and maintenance costs for a DDBMS to be higher than those for a centralized DBMS. Furthermore, a distributed DBMS requires additional hardware to establish a network between sites. There are ongoing communication costs incurred with the use of this network. There are also additional labor costs to manage and maintain the local DBMSs and the underlying network.

3) Security

In a centralized system, access to the data can be easily controlled. However, in a distributed DBMS not only does access to replicated data have to be controlled in multiple locations, but the network itself has to be made secure.

4) Integrity control more difficult

Enforcing integrity constraints generally requires access to a large amount of data that defines the constraint but which is not involved in the actual update operation itself. Also, in a distributed DBMS, the communication and processing costs that are required to enforce integrity constraints may be prohibitive

Database integrity refers to the validity and consistency of stored data. Integrity is usually expressed in terms of constraints, which are consistency rules that the database is not permitted to violate.

5) .Lack of standards

Although distributed DBMSs depend on effective communication, we are only now starting to see the appearance of standard communication and data access protocols. This lack of standards has significantly limited the potential of distributed DBMSs. There are also no tools or methodologies to help users convert a centralized DBMS into a distributed DBMS.

6) Lack of experience

General-purpose distributed DBMSs have not been widely accepted, although many of the protocols and problems are well understood. Consequently, we do not yet have the same level of experience in industry as we have with centralized DBMSs. For a prospective adopter of this technology, this may be a significant deterrent.

7) Database design more complex

It is more complex to design a distributed system than a centralized database. Besides the normal difficulties of designing a centralized database, the design of a distributed database has to take account of fragmentation of data, allocation of fragments to specific sites, and data replication.

Homogeneous and Heterogeneous DDBMSs

A DDBMS may be classified as homogeneous or heterogeneous. In a **homogeneous** system, all sites use the same DBMS product. In a **heterogeneous system,** sites may run different DBMS products, which need not be based on the same underlying data model,

and so the system may be composed of relational, network, hierarchical, and object oriented DBMSs.

Homogeneous systems are much easier to design and manage. This approach provides incremental growth, making the addition of a new site to the DDBMS easy, and allows increased performance by exploiting the parallel processing capability of multiple sites.

Heterogeneous systems usually result when individual sites have implemented their own databases and integration is considered at a later stage. In a heterogeneous system, translations are required to allow communication between different DBMSs. To provide DBMS transparency, users must be able to make requests in the language of the DBMS at their local site. The system then has the task of locating the data and performing any necessary translation. Data may be required from another site that may have:

i.) Different hardware.

ii.) Different DBMS products.

iii.) Different hardware and different DBMS products.

If the hardware is different but the DBMS products are the same, the translation is straightforward, involving the change of codes and word lengths. If the DBMS products are different, the translation is complicated involving the mapping of data structures in one data model to the equivalent data structures in another data model. For example, relations in the relational data model are mapped to records and sets in the network model. It is also necessary to translate the query language used (for example, SQL SELECT statements are mapped to the network FIND and GET statements). If both the hardware and software are different, then both these types of translation are required. This makes the processing extremely complex.

The typical solution used by some relational systems that are part of a heterogeneous DDBMS is to use gateways, which convert the language and model of each different DBMS into the language and model of the relational system. However, the gateway approach has some serious limitations. First, it may not support transaction management, even for a pair of systems; in other words, the gateway between two systems may be only a query translator. For example, a system may not coordinate concurrency control and recovery of transactions that involve updates to the pair of databases. Second, the gateway approach is concerned only with the problem of translating a query expressed in one language into an equivalent expression in another language. As such, generally it does not address the issues of homogenizing the structural and representational differences between different schemas.

Open database access and interoperability

The Open Group formed a Specification Working Group (SWG) to respond to a White Paper on open database access and interoperability. The goal of this group was to provide specifications or to make sure that specifications exist or are being developed that will create a database infrastructure environment where there is:

i.)- A common and powerful SQL Application Programming Interface (API) that allows client applications to be written that do not need to know the vendor of the DBMS they are accessing.

ii.) A common database protocol that enables a DBMS from one vendor to communicate directly with a DBMS from another vendor without the need for a gateway.

iii.) A common network protocol that allows communications between different DBMSs.

Multidatabase systems

Multidatabase system (MDBS) - A multidatabase system is a distributed DBMS in which each site maintains complete autonomy.

In recent years, there has been considerable interest in MDBSs, which attempt to logically integrate a number of independent DDBMSs while allowing the local DBMSs to maintain complete control of their operations. One consequence of complete autonomy is that there can be no software modifications to the local DBMSs. Thus, an MDBS requires an additional software layer on top of the local systems to provide the necessary functionality.

A multidatabase system (MDBS) allows users to access and share data without requiring full database schema integration ***(NB: The schema is the structure of the database).*** However, it still allows users to administer their own databases without centralized control, as with true DDBMSs. The DBA of a local DBMS can authorize access to particular portions of his or her database by specifying an *export schema,* which defines the parts of the database that may be accessed by non-local users. There are **unfederated** (where there are no local users) and **federated** MDBSs. A federated system is a cross between a distributed DBMS and a centralized DBMS; it is a distributed system for global users and a centralized system for local users.

In simple terms, a multidatabase system (MDBS) is a DBMS that resides transparently on top of existing database and file systems, and presents a single database to its users. An MDBS maintains only the global schema against which users issue queries and updates and the local DBMSs themselves maintain all user data. The global schema is constructed by integrating the schemas of the local databases. The MDBS fust translates the global queries and updates into queries and updates on the appropriate local DBMSs. It then merges the local results and generates the final global result for the user. Furthermore, the MDBS coordinates the commit and abort operations for global transactions by the local DBMSs that processed them, to maintain consistency of data within the local databases. An MDBS controls multiple gateways and manages local databases through these gateways.

Distributed Relational Database Design

This section examines the additional factors (to the conceptual and logical design of a centralized relational database) that have to be considered for the design of a distributed relational database. The specific areas examined are:

i.) *Fragmentation* A relation may be divided into a number of subrelations, called **fragments,** which are then distributed. There are two main types of fragmentation:

_ **horizontal** and **vertical.** Horizontal fragments are subsets of tuples (records) and vertical fragments are subsets of attributes (fields).

ii.) *Allocation* Each fragment is stored at the site with 'optimal' distribution.

iii.) Replication The DDBMS may maintain a copy of a fragment at several different sites.

The definition and allocation of fragments must be based on how the database is to be used. This involves analyzing transactions. Generally, it is not possible to analyze all transactions, so we concentrate on the most important ones. It has been suggested that the most active 20% of user queries account for 80% of the total data access, and this 80/20 rule may be used as a guideline in carrying out the analysis.

The design should be based on both quantitative and qualitative information. Quantitative information is used in allocation; qualitative information is used in fragmentation. The quantitative information may include:

i.) The frequency with which a transaction is run.
ii.) The site from which a transaction is run.
iii.) The performance criteria for transactions.

The qualitative information may include information about the transactions that are executed, such as:

i.) The relations, attributes, and tuples accessed.
ii.) The type of access (read or write);
iii.) The predicates of read operations.

The definition and allocation of fragments are carried out strategically to achieve the following objectives:

i.) *Locality of reference* Where possible, data should be stored close to where it is used. If a fragment is used at several sites, it may be advantageous to store copies of the fragment at these sites.

ii.) *Improved reliability and availability* Reliability and availability are improved by replication: there is another copy of the fragment available at another site in the event of one site failing.

iii.) Acceptable performance Bad allocation may result in bottlenecks occurring, that is a site may become inundated (flooded) with requests from other sites, perhaps causing a significant degradation in performance. Alternatively, bad allocation may result in underutilization of resources.

iv.) *Balanced storage capacities and costs* Consideration should be given to the availability and cost of storage at each site so that cheap mass storage can be used, where possible. This must be balanced against *locality of reference.*

v.) *Minimal communication costs* Consideration should be given to the cost of remote requests. Retrieval costs are minimized when *locality of reference* is maximized or when each site has its own copy of the data. However, when replicated data is updated, the update has to be performed at all sites holding a duplicate copy, thereby increasing communication costs.

Data Allocation

There are four alternative strategies regarding the placement of data: centralized, fragmented, complete replication, and selective replication.

i.) **Centralized**

This strategy consists of a single database and DBMS stored at one site with users distributed across the network (referred to as distributed processing). Locality of reference is at its lowest as all sites, except the central site, have to use the network for all data accesses. This also means that communication costs are high. Reliability and availability are low, as a failure of the central site results in the loss of the entire database system.

ii.) **Fragmented (partitioned)**

This strategy partitions the database into disjoint fragments, with each fragment assigned to one site. If data items are located at the site where they are used most frequently, locality of reference is high. As there is no replication, storage costs are low; similarly, reliability and availability are low, although they are higher than in the centralized case as the failure of a site results in the loss of only that site's data. Performance should be good and communications costs low if the distribution is designed properly.

iii.) Complete replication

This strategy consists of maintaining a complete copy of the database at each site. Therefore, locality of reference, reliability and availability, and performance are maximized. However, storage costs and communication costs for updates are the most expensive. To overcome some of these problems, snapshots are sometimes used. A snapshot is a copy of the data at a given time. The copies are updated periodically, for example, hourly or weekly, so they may not be always up to date. Snapshots are also sometimes used to implement views in a distributed database to improve the time it takes to perform a database operation on a view.

iv.) **Selective replication**

This strategy is a combination of fragmentation, replication, and centralization. Some data items are fragmented to achieve high locality of reference and others, which are used at many sites and are not frequently updated, are replicated; otherwise, the data items are centralized. The objective of this strategy is to have all the advantages of the other approaches but none of the disadvantages. This is the most commonly used strategy because of its flexibility.

	Locality Reference	**Reliability & Availability**	**Performance**	**Storage Cost**	**Communication Costs**
Centralized	Lowest	Lowest	Unsatisfactory	Lowest	Highest
Fragmented	High*	Low for item; high for system	Satisfactory*	Lowest	Low*
Complete Replication	Highest	Highest	Best for read	Highest	High for update; Low for read
Selective Replication	High*	Low for item; high for system	Satisfactory*	Average	Low*

Tl)e table above compares the different strategies for data allocation.
* Indicates subject to good design i.e. what is stated only applies if the design is good.

Fragmentation

Reasons for fragmentation

There are four main reasons for fragmenting a relation (table):

i.) *Usage* - In general, applications work with views rather than entire relations. Therefore, for data distribution, it seems appropriate to work with subsets of relations (tables) as the unit of distribution.

ii.) *Efficiency* - Data is stored close to where it is most frequently used. In addition, data that is not needed by local applications is not stored.

iii.) Parallelism - With fragments as the unit of distribution, a transaction can be divided into several subqueries that operate on fragments. This should increase the degree of concurrency, or parallelism, in the system thereby allowing transactions that can do so safely to execute in parallel.

iv.) Security- Data not required by local applications is not stored and consequently not available to unauthorized users.

Fragmentation has two primary disadvantages:

i.) *Performance* - The performance of global applications that require data from several fragments located at different sites may be slower.

ii.) *Integrity*- Integrity control may be more difficult if data and functional dependencies are fragmented and located at different sites.

Correctness of fragmentation

Fragmentation cannot be carried out haphazardly. There are three rules that must be followed during fragmentation:

i.) *Completeness* - If a relation instance *R* is decomposed into fragments R_1, R_2, ... ,*Rn* each data item that can be found in *R* must appear in at least one fragment. This rule is necessary to ensure that there is no loss of data during fragmentation.

ii.) *Reconstruction* - It must be possible to define a relational operation that will reconstruct the relation *R* from the fragments. This rule ensures that functional dependencies are preserved.

iii.) Disjointness - If a data item *di* appears in fragment *Ri* then it should not appear in any other fragment. Vertical fragmentation is the exception to this rule, where primary key attributes must he repeated to allow reconstruction. This rule ensures minimal data redundancy.

In the case of horizontal fragmentation, a data item is a tuple (record); for vertical fragmentation, a data item is an attribute (field).

Reading Exercise

Read and make notes on **relational algebra**

Types of fragmentation

There are two main types of fragmentation: **horizontal** and **vertical.** Horizontal fragments are subsets of tuples and vertical fragments are subsets of attributes, as illustrated in the figure below. There are also two other types of fragmentation: **mixed** and derived a type of horizontal fragmentation.

Horizontal fragmentation

Horizontal fragment - A horizontal fragment consists of a subset of the tuples (records) of a relation (table).

.Horizontal Fragmentation

Vertical Fragmentation

(a) (b)

Mixed Fragmentation

(a) Vertical fragments, horizontally fragmented.
(b) Horizontal fragments, vertically fragmented.

Horizontal fragmentation groups together the tuples (records) in a relation (table) that are collectively used by the important transactions. A horizontal fragment is produced by specifying a predicate that performs a restriction on the tuples in the relation, it is defined using the *Selection* operation of the relational algebra. The selection operation groups together tuples that have some common property; for example, the tuples are all used by the same application or at the same site.

Horizontal fragmentation example

Assuming that there are only two categories of students, Diploma and Degree students, the horizontal fragmentation of a table called tblStudents by *course applied for* can be obtained as follows:

P1: DegreeStudents
P2: DiplomaStudents

This produces two fragments (P1and P2), one consisting of those tuples where the value of the *course applied for* attribute is 'Degree' and the other consisting of those tuples where the value of the *course applied for* attribute is 'Diploma', as shown in the figure below. This particular fragmentation strategy may be advantageous if there are separate applications dealing with Degree Students and Diploma Students.

sNo	First Name	Second Name	Gender	Religion	Course Applied For	Mode of Study
1	Jane	Waithira	Female	Christian	Diploma	Full Time
2	Peter	Mutua	Male	Muslim	Degree	Full Time
3	Alice	Atieno	Female	Christian	Diploma	Evening
4	Simon	Wetangula	Male	Muslim	Degree	Full Time
5	Michelle	Obama	Female	Christian	Degree	Evening
6	Jacob	Nzuma	Male	Christian	Diploma	Evening

tblStudents

The figure above shows tblStudents (before fragmentation). The figures below show the table after horizontal fragmentation based on the *course applied for*

sNo	First Name	Second Name	Gender	Religion	Course Applied For	Mode of Study
1	Jane	Waithira	Female	Christian	Diploma	Full Time
3	Alice	Atieno	Female	Christian	Diploma	Evening
6	Jacob	Nzuma	Male	Christian	Diploma	Evening

Fragment P1

sNo	First Name	Second Name	Gender	Religion	Course Applied For	Mode of Study
2	Peter	Mutua	Male	Muslim	Degree	Full Time
4	Simon	Wetangula	Male	Muslim	Degree	Full Time
5	Michelle	Obama	Female	Christian	Degree	Evening

Fragment P2

The fragmentation schema satisfies the correctness rules:

i.) *Completeness* - Each tuple (record) appears in either fragment P1or P2

ii.) *Reconstruction* - tblStudents can be reconstructed from the fragments using the union operation as shown below:

PI U P, = tblStudents

iii.) Disjointness - The fragments are disjoint; there can't be a student applying for a diploma and degree course (for the same application).

Vertical fragmentation

Vertical fragment-A vertical fragment consists of a subset of the attributes (fields) of a relation (table).

V rtical fragmentation groups together the attributes in a relation that are used jointly by the important transactions. A vertical fragment is defined using the *Projection* operation of the relational algebra.

Horizontal fragmentation example

The table below shows a table called tblEmployees for a certain organization

staffNo	First Name	Second Name	Gender	Position	Employment Date	BranchNo	Salary
HQ32	Jane	Waithira	Female	Assistant	1/1/08	NRB00l	21,000
HQ16	Peter	Mutua	Male	Supervisor	31/11/01	MSA003	32,000
HQ12	Alice	Atieno	Female	Manager	12/1/04	NRB00l	43,000
HQ47	Simon	Wetangula	Male	Assistant	1/10/06	MSA003	20,500
HQ30	Michelle	Obama	Female	Supervisor	21/3/99	NRB00l	36,000
HQ18	Jacob	Nzuma	Male	Manager	5/1/07	MSA002	45,000

tblEmployees

The accounts depaitment (and hence the payroll software application) requires staffNo, First Name, Second Name, Position, Employment Date and Salary

The human resource department requires staffNo, First Name, Second Name, Gender and BranchNo.

The table tblEmployees can therefore be fragmented into two as shown below:

staffNo	First Name	Second Name	Position	Employment Date	Salary
HQ32	Jane	Waithira	Assistant	1/1/08	21,000
HQ16	Peter	Mutua	Supervisor	31/11/01	32,000
HQ12	Alice	Atieno	Manager	12/1/04	43,000
HQ47	Simon	Wetangula	Assistant	1/10/06	20,500
HQ30	Michelle	Obama	Supervisor	21/3/99	36,000
HQ18	Jacob	Nzuma	Manager	5/1/07	45,000

Fragment S1

staffNo	First Name	Second Name	Gender	BranchNo
HQ32	Jane	Waithira	Female	NRB00l
HQ16	Peter	Mutua	Male	MSA003
HQ12	Alice	Atieno	Female	NRB00l
HQ47	Simon	Wetangula	Male	MSA003
HQ30	Michelle	Obama	Female	NRB00l
HQ18	Jacob	Nzuma	Male	MSA002

Fragment S2

NB: Note that both fragments contain the primary key, *stafJNo,* to enable the original relation to be reconstructed. The advantage of vertical fragmentation is that the fragments can be stored at the sites that need them. In addition, performance is improved as the fragment is smaller than the original base relation. This fragmentation schema satisfies the correctness rules:

i.) *Completeness* - Each attribute (field) appears in either fragment S_I or S_2

ii.) *Reconstruction* - tblEmployees can be reconstructed from the fragments S_1 and S_2

iii.) Disjointness - The fragments are disjoint except the primary key, which is necessary for reconstruction.

Mixed fragmentation

For some applications horizontal or vertical fragmentation of a database schema by itself is insufficient to adequately distribute the data. Instead, mixed or hybrid fragmentation is required.

Mixed Fragment - A mixed fragment consists of a horizontal fragment that is subsequently vertically fragmented, or a vertical fragment that is then horizontally fragmented.

A mixed fragment is defined using the *Selection* and *Projection* operations of the relational algebra.

Transparencies in a DDBMS

The definition of a DDBMS given earlier states that the system should make the distribution transparent to the user. Transparency hides implementation details from the user. For example, in a centralized DBMS data independence is a form of transparency - It hides changes in the definition and organization of the data from the user. A DDBMS may provide various levels of transparency. However, they all participate in the same overall objective: to make the use of the distributed database equivalent to that of a centralized database.

There are four main types of transparency in a DDBMS:

i.) Distribution transparency.
ii.) Transaction transparency.
iii.) Performance transparency.
iv.) DBMS transparency.

Full transparency is not a universally accepted objective. For example, Gray (1989) argues that full transparency makes the management of distributed data very difficult and that applications coded with transparent access to geographically distributed databases have poor manageability, poor modularity, and poor message performance. Note, rarely are all the transparencies discussed here met by a single system.

1) **Distribution Transparency**

Distribution transparency allows the user to perceive the database as a single, logical entity. If a DDBMS exhibits distribution transparency, then the user does not need to know the data is fragmented **(fragmentation transparency)** or the location of data items **(location transparency).**

If the user needs to know that the data is fragmented and the location of fragments then this is called local mapping transparency. These transparencies are ordered as discussed below.

Fragmentation Transparency

Fragmentation is the highest level of distribution transparency. If fragmentation transparency is provided by the DDBMS, then the user does not need to know that the data is fragmented. As a result, database accesses are based on the global schema, so the user does not need to specify fragment names or data locations. For example, to retrieve the names of all Managers, with fragmentation transparency one could write:

```
SELECT [fName], [sName]
FROM tblStaff
WHERE position = "Manager";
```

This is the same SQL statement as we would write in a centralized system.

Location transparency

Location is the middle level of distribution transparency. With location transparency, the user must know how the data has been fragmented but still does not have to know the location of the data.

When rewriting the above SQL statement in a situation where there was location transparency but no fragmentation transparency, one would have to specify the names of the fragments in the query as opposed to the name of the table (tblStaff). One would also have to use a join (or subquery) because the attributes position and fName and sName appear in different vertical fragments. The main advantage of location transparency is that the database may be physically reorganized without impacting on the application programs that access them.

Replication transparency

Closely related to location transparency is replication transparency, which means that the user is unaware of the replication of fragments. Replication transparency is implied by location transparency. However, it is possible for a system not to have location transparency but to have replication transparency.

Local mapping transparency

This is the lowest level of distribution transparency. With local mapping transparency, the user needs to specify both fragment names and the location of data items, taking into consideration any replication that may exist. This type of query is more complex and time consuming for the user to enter and it is unlikely that a system that only provides this level of transparency would be acceptable to end-users.

Naming transparency

As a result to the above distribution transparencies, we have **naming transparency.** Just like in a centralized database, each item in a distributed database must have a unique name. Therefore, the DDBMS must ensure that no two sites create a database object with the same name. One solution to this problem is to create a central name server, which has the responsibility for ensuring uniqueness of all names in the system. However, this approach results in:

i.) Loss of some local autonomy.
ii.) Performance problems, if the *central* site becomes a bottleneck.

iiiJLow availability; if the central site fails, the remaining sites cannot create any new database objects.

An alternative solution is to prefix an object with the identifier of the site that created it. For example, the relation Branch created at site S_1 might be named Sl.Branch. Similarly, we need to be able to identify each fragment and each of its copies. Thus, copy 2 of fyagment 3 of the Branch relation created at site S_1 might be referred to as Sl.Branch.F3.C2. However, this results in loss of distribution transparency.

An approach that resolves the problems with both these solutions uses aliases (sometimes called synonyms) for each database object. Thus, Sl.Branch.F3.C2 might he known as LocalBranch by the user at site S1. The DDBMS has the task of mapping an alias to the appropriate database object.

2) **Transaction Transparency**

Transaction transparency in a DDBMS environment ensures that all distributed transactions maintain the distributed database's integrity and consistency. A **distributed transaction** accesses data stored at more than one location. Each transaction is divided into a number of **subtransactions,** one for each site that has to be accessed.

The atomicity of the distributed transaction is still fundamental to the transaction concept but in addition the DDBMS must also ensure the atomicity of each subtransaction. Therefore, not only must the DDBMS ensure synchronization of subtransactions with other local transactions that are executing concurrently at a site, but it must also ensure synchronization of subtransactions with global transactions running simultaneously at *the same* or different sites. Transaction *transparency* in a distributed DBMS is complicated by the fragmentation, allocation, and replication schemas. We consider two further aspects of transaction transparency: **concurrency transparency** and **failure transparency.**

Concurrency transparency

Concurrency transparency is provided by the DDBMS if the results of all concurrent transactions (distributed and non-distributed) execute *independently* and are logically *consistent* with the results that are obtained if the transactions are executed one at a time, in some arbitrary serial order. These are the same fundamental principles as those for the centralized DBMS. However, there is the added complexity that the DDBMS must ensure that both global and local transactions do not interfere with each other. Similarly, the DDBMS must ensure the consistency of all subtransactions of the global transaction.

Replication makes the issue of concurrency more complex. If a copy of a replicated data item is updated, the update must eventually be propagated to all copies. An obvious strategy is to propagate the changes as part of the original transaction, making it an atomic operation. However, if one of the sites holding a copy is not reachable when the update being processed, either because the site or the communication link has failed, then the transaction is delayed until the site is reachable. If there are many copies of the data item, the probability of the transaction succeeding decreases exponentially. An alternative strategy is to limit the update propagation to only those sites that are currently available. The remaining sites must be updated when they become available again. A further

strategy would be to allow the updates to the copies to happen *asynchronously*, sometime after original update. The delay in regaining consistency may range from a few seconds to several hours.

Failure transparency

In earlier sections it was stated that a centralized DBMS must provide a recovery mechanism that ensures that, in the presence of failures, transactions are **atomic:** either all the operations of the transaction are carried out or none at all. Furthermore, once a transaction has committed the changes are **durable.** The types of failure that could occur in a centralized system such as system crashes, media failures, software errors, carelessness, natural physical disasters and sabotage were also examined. In the distributed environment, the DDBMS must also cater for:

i.) The loss of a message.
ii.) The failure of a communication link.
iii.) The failure of a site.
iv.) Network partitioning.

The **DDBMS** must ensure the atomicity of the global transaction, which means ensuring that subtransactions of the global transaction either all commit or all abort Thus, the DDBMS must synchronize the global transaction to ensure that all subtransactions have completed successfully before recording a final COMMIT for the global transaction. For example, consider a global transaction that has to update data at two sites, S_I and S_2. The subtransaction at site S_I completes successfully and commits, but the subtransaction at site S_2 is unable to commit and rolls back the changes to ensure local consistency. The distributed database is now in an inconsistent state: we are unable to *uncommit* the data at site S_1, owing to the durability property of the subtransaction at S_1.

3) **Performance Transparency**

Performance transparency requires a DDBMS to perform as if it were a centralized DBMS. In a distributed environment, the system should not suffer any performance degradation due to the distributed architecture, for example the presence of the network. Performance transparency also requires the DDBMS to determine the most cost-effective strategy to execute a request.

In a centralized DBMS, the query processor (QP) must evaluate every data request and find an optimal execution strategy, consisting of an ordered sequence of operations on the database. In a distributed environment, the distributed query processor (DQP) maps a data request into an ordered sequence of operations on the local databases. It has the added complexity of taking into account the fragmentation, replication, and allocation schemas. The DQP has to decide:

i.) Which fragment to access.
ii.) Which copy of a fragment to use, if the fragment is replicated.
iii.) Which location to use.

The *DQP* produces an execution strategy that is optimized with respect to some cost function. Typically, the costs associated with a distributed request include:

i.) The access time (1/0) cost involved in accessing the physical data on disk.

ii) The CPU time cost incurred when performing operations on data in main memory.
iii.)The communication cost associated with the transmission of data across the network.

The first two factors are the only ones considered in a centralized system. In a distributed environment, the DDBMS must take account of the communication cost, which may be the most dominant factor in WANs with a bandwidth of a few kilobytes per second. In such cases, optimization may ignore 1/0 and CPU costs. However, LANs have a bandwidth comparable to that of disks, so in such cases optimization should not ignore 1/0 and CPU costs entirely.

4) **DBMS Transparency**

DBMS transparency hides the knowledge that the local DBMSs may be different, and is therefore only applicable to heterogeneous DDBMSs. It is one of the most difficult transparencies to provide as a generalization. Problems associated with the provision of heterogeneous systems were discussed earlier.

Summary of Transparencies in a DDBMS

As mentioned earlier, complete transparency is not a universally agreed objective. Transparency is not an 'all or nothing' concept, but it can be provided at different levels. Each level requires a particular type of agreement between the participant sites. For example, with complete transparency the sites must agree on such things as the data model; the interpretation of the schemas, the data representation, and the functionality provided by each site. At the other end of the spectrum, in a non-transparent system there is only agreement on the data exchange format and the functionality provided by each site.
From the user's perspective, complete transparency is highly desirable. However, from the local DBA's perspective fully transparent access may be difficult to control.

Date's Twelve Rules for a DDBMS

Below is a list of Date's twelve rules (or objectives) for DDBMSs. The basis for these rules is that a distributed DBMS should feel like a non-distributed DBMS to the user.

Fundamental principle

To the user, a distributed system should look exactly like a non-distributed system.

1) Local autonomy

The sites in a distributed system should be autonomous. In this context, autonomy means that:

i.) Local data is locally owned and managed.
ii.) Local operations remain purely local.
iii.)All operations at a given site are controlled by that site.

2) No reliance on a central site.

There should be no one site without wruch the system cannot operate. This implies that there should he no central servers for services such as transaction management, deadlock detection, query optimization, and management of the global system catalog.

3). Continuous operation

Ideally, there should never be a need for a planned system shutdown, for operations such as:

i.) Adding or removing a site from the system
ii.) The dynamic creation and deletion of fragments at one or more sites.

4) Location independence

Location independence is equivalent to location transparency. The user should be able to access the database from any site. Furthermore, the user should be able to access all data as if it were stored at the user's site, no matter where it is physically stored.

5) Fragmentation independence

The user should be able to access the data, no matter how it is fragmented.

6) Replication independence

The user should be unaware that data has been replicated. Thus, the user should not be able to access a particular copy of a data item directly, nor should the user have to specifically update all copies of a data item.

7) Distributed query processing

The system should be capable of processing queries that reference data at more than one site.

8) Distributed transaction processing

The system should support the transaction as the unit of recovery. The system should ensure that both global and local transactions conform to the ACID rules for transactions, namely: atomicity, consistency, isolation, and durability.

9) Hardware independence

It should be possible to run the DDBMS on a variety of hardware platforms.

10) Operating system independence

As a result to the previous rule, it should be possible to run the DDBMS on a variety of operating systems.

11) Network independence

It should be possible to run the DDBMS on a variety of dissimilar communication networks.

12) Database independence

It should be possible to have a DDBMS made up of different local DBMSs, perhaps supporting different underlying data models, i.e. the system should support heterogeneity.

Reading Exercise

Read and makes notes on Database Replication, including but not limited to:

- Definition.

- Benefits of database replication.
- Applications of replication.
- Basic components of database replication.
- Database replication environments.
- Replication servers - Functionality and implementation issues.